Important note:
This manual describes HOL-OCL 0.9.0 with referential universes and smashed collection types.
The manual of version 0.9.0 is also available as technical report number 525 from the department of computer science, ETH Zurich.
Contents

I. Motivation and Introduction 9

1. Introduction 11
   1.1. Motivation 11
   1.2. How to Read This Document 13
   1.3. Typographic Conventions 14
   1.4. Acknowledgements 15

2. Background 17
   2.1. Higher-order Logic 17
   2.2. Concepts and Use of Isabelle/HOL 19
   2.3. A Short Introduction into UML/OCL 20
   2.4. A Note About Standards 22

II. Theory 25

3. Foundations 27
   3.1. An Overview of Embedding Techniques 27
   3.2. Embeddings of Specification Languages 28
   3.3. Challenges of a Shallow Embedding of UML/OCL 29
   3.4. An Overview over Embedding Techniques 31

4. Faithfully Representing UML/OCL 33
   4.1. A Note About Standard Compliance 33
   4.2. Building-Blocks of the HOL-OCL Architecture 35
   4.3. Formal Preliminaries for Representing Semantics in HOL 37
   4.4. Encoding Standard OCL Operations 39
   4.5. Textbook vs. Combinator Style Semantics of Operations 40
   4.6. Compliance to the Standards OCL Requirements 42
   4.7. Organizing the Embedding into Layer 43
      4.7.1. Datatype Adaption 45
      4.7.2. Functional Adaption 46
      4.7.3. Embedding Adaption for Shallow Embedding 48
   4.8. Extensible Universes in Typed Meta-Language 49
   4.9. Encoding Object Structures 50
## 7. Installing HOL-OCL

### 7.1. Prerequisites

### 7.2. Installation

#### 7.2.1. Installation from Source

#### 7.2.2. Using Debian Packages

### 7.3. Starting

## 8. Limitations of HOL-OCL

### 8.1. The Supported UML Subset

### 8.2. The Supported OCL Subset

#### 8.2.1. OCL Syntactical Variants

#### 8.2.2. OCL context declarations

#### 8.2.3. OCL Types

#### 8.2.4. Predefined Properties

### 8.3. Reporting Bugs


### 9.1. The Basic Workflow

### 9.2. The Modelling Phase

### 9.3. Using HOL-OCL: The Verification Phase

#### 9.3.1. Getting Started

#### 9.3.2. Loading XMI files

#### 9.3.3. Canonizing Hypotheses

#### 9.3.4. One-Step-Rewriting

#### 9.3.5. Automated Proof-Procedures

### 9.4. Extending HOL-OCL

## 10. Case Studies

### 10.1. Encoding a Stack in UML/OCL

## IV. Appendix

### A. The Syntax of OCL

### B. Isabelle Theories

#### B.1. Introduction

#### B.1.1. HOL-OCL Configurations

#### B.1.2. Notational Remarks

#### B.2. Overview

#### B.3. Foundations

#### B.3.1. The Theory of Lifting and its Combinators

#### B.3.2. The OCL Kernel

#### B.3.3. Type Definition for OclUndefined

#### B.3.4. Type Definition for Boolean
Contents

C.3.3. Encoding Level 2: ......................................................... 456

D. The GNU General Public Licence 473
   D.1. Preamble ................................................................. 473
   D.2. Terms and conditions for copying, distribution and modification .................................................. 474
   D.3. Appendix: How to Apply These Terms to Your New Programs ......................................................... 478

V. Bibliography and Index .................................................. 481
   Bibliography .................................................................. 483
   List of Glitches ............................................................. 491
   List of Extensions .......................................................... 493
   List of Figures ................................................................ 495
   List of Tables .................................................................. 497
   List of Definitions .......................................................... 499
   Listings ........................................................................... 501
Contents
Part I.

Motivation and Introduction
Chapter 1.

Introduction

1.1. Motivation

Building safe and secure software systems requires engineering techniques. Just as using “blueprints” in common engineering practice, the development of complex software systems requires a detailed specification describing its data structures and its desired behavior. Specification documents can vary in its precision from textual descriptions over structured text up to formal specification using a language based on mathematical logic such as Z [26] or VDM [27]. Depending on the precision (or formality), different computer-supported techniques can be applied that assure the consistency of specification documents. For example, type-checking or wellformedness-checking can be used to find contradictions in requirement and design documents of a system and thus provide an a-priori analysis. Moreover, more involved techniques can assure the correct transition from a specification document to its implementation in “real code”; such an a-posteriori analysis techniques is called validation. These techniques and their computer-support are summarized under the term formal methods. Obviously, the power of these techniques crucially depends on the degree of formality.

Producing formal specifications and maintaining their consistency during system development is a task that requires substantial efforts and training. This holds to an even larger extent for the validation phase, where techniques such as model-based testing, model-checking or even interactive theorem proving were applied. For this reason, the acceptance of formal methods has been very reluctant in industrial practice so far, although it is meanwhile widely accepted that specification and testing activities outweigh by far the costs of the implementation phase of a large system and that formal methods have a positive effect here. Rather, the overwhelming need for specification led to the development of semi-formal specification documents, that have their roots in light-weighed graphical notations. These semi-formal kinds of documents became eventually a more and more defined semantics and were annotated by programs or abstract test code. Thus, instead of using mathematical notation such as Z or the Hoare Calculus, there is a trend to introduce formal specification techniques such as pre and post conditions through the backdoor, leading to a more gradual transition of the industrial practice by bridging the gap to software developers by languages and notations they are familiar with.

The need for computer-support of these “light-weighed” specification methods led to a standardization effort and the Unified Modelling Language (UML), which achieved
remarkable acceptance in the industry. The UML offers an integrated object-oriented development methodology ranging from “object-oriented modeling” to code-generation in “model-based architecture” development. The UML is continuously defined by the Object Management Group (OMG) [41], an open standardization committee. The UML provides standards for graphical notations, their representation in abstract syntax (called meta-models) and partly also their semantics. These notations comprise among others activity charts, sequence charts, class diagrams and state charts. The latter two notations are of particular interest from the perspective of formal methods, since they represent forms of data-oriented and behavioral modeling, thus well-known concepts in a new shape. Moreover, since UML 1.3, a logical annotation formalism, called the Object Constraint Language (OCL) is a part of the UML and is heavily used in the specification documents of the meta-models of the UML itself.

To sum up, from the perspective of formal methods, the success of CASE tools supporting the UML [40] opens a door for bringing formal methods a step closer to industry. However, to turn this vision into reality, several challenges need to be faced: first, the semantics of UML/OCL is conceptually much closer to an object-oriented programming language than to a traditional logic, although OCL comprises a version of predicate logic and arithmetic. Second, considerable effort has to be invested to cope with the object-oriented features of OCL in a logically clean way, allowing for adequate symbolic computations. Third, a considerable research effort has to be invested to develop an adequate proof methodology for object-oriented modeling as it is meanwhile established in the user community of UML/OCL.

As a technical basis for a long-term effort to meet these challenges, we developed HOL-OCL, an interactive proof environment for acsocl. The main design goals of HOL-OCL are summarized as follows:

- HOL-OCL follows semantically the OCL 2.0 standard [41];
- HOL-OCL provides technical support for importing UML/OCL models (i.e., class-diagrams at the moment, start charts are under preparation) in form of XMI-formats generated by conventional UML/OCL modeling tools, and
- HOL-OCL is intended to analyze UML/OCL models with all the means provided by an up-to-date interactive, tactic-controlled theorem proving environment such as Isabelle [3].

In itself, HOL-OCL also provides achievements for the definition and development of the language UML/OCL itself:

- It defines and supports several syntactic formats for OCL (one more text-oriented for programmers and software developers, another more mathematical one for proof engineers);
- It defines a machine-checked formalization of the semantics as described in the standard 2.0. This is implemented as a conservative, shallow embedding consisting of OCL into the higher-order logic (HOL) instance of the interactive theorem
1.2. How to Read This Document

This book has three different aspects and describes them at various levels of detail. Therefore, we divided the material into three parts which can be read largely independently from each other:

- **consistent checking** of UML models (e.g., is a class-diagram implementable, i.e., do states exist that satisfy the invariants? Contradict pre-conditions and post-conditions to system invariants?),
- **proving refinements** between UML models,
- **code verification** of concrete programs with respect to a UML model by means of a Hoare-Calculus,
- **proving temporal properties** of UML models (e.g., do all reachable states satisfy a security-property?),
- **automated test-case generation** out of UML models for concrete code.
Chapter 1. Introduction

- Part I gives an overall introduction and motivation. In particular, it introduces into the foundational notations and the target language.

- Part II presents the concepts of the underlying ideas and formalization techniques behind HOL-OCL.

- Part III explains the HOL-OCL system from a users perspective and can be seen as a kind of "users manual."

Further, in the appendix a complete companion of the Isabelle theory files is provided, including the technical details of HOL-OCL and reference material such as, e.g., tables that compare different OCL syntaxes.

Depending on your background knowledge and interests, we recommend kind of cherry-picking strategy for reading this book. For example,

- If you are a HOL-OCL user and want to get a quick “hands-on” experience you may start with the installation instructions (chapter 7) and start reading with chapter 1 and chapter 2. You can then continue directly with chapters 9 and 10.

- If you are interested in the ideas and techniques of HOL-OCL we recommend (after reading the introductory chapters 1 and 2) an in depth study of chapters 3 to 5.

- If you are an Isabelle hacker and are interested in the details necessary for theorem proving work you can directly start with the theory-files in the appendix, i.e., chapter B. However, we advise to start with the more conceptual descriptions in 5.

1.3. Typographic Conventions

The following typographic conventions appear in this book:

- Pure OCL specifications are either written inline like self.s->includes(5) or together with their context specification:

  \begin{center}
  \begin{tabular}{l}
  context A:
  \hline
  inv: self.s->includes(5)
  \end{tabular}
  \end{center}

  Keywords are printed in a blue typeface.

- OCL formulae that are interpreted within HOL-OCL, i.e., are written inline as self.s->includes(5), or alternatively in mathematical syntax as: $5 \in (\text{self.s})$. As you can see, we use a (shorter) mathematical notation for OCL expressions. This notation is introduced as an alternative concrete syntax (see Table A.2 for a syntax comparison). Overall, HOL-OCL supports both notations, but we prefer the mathematical one for semantic definitions and proof work.
We use a color coding to distinguish [OCL] and [HOL] sub-expressions in formulae containing both, e.g.:

\[ \cup \equiv \text{lift}_2(\text{strictify}(\lambda X. \text{strictify}(\lambda Y. \text{AbsSet}_{\text{RepSet } X \cup \text{RepSet } Y}))), \]

or

\[- + \equiv \text{lift}_2(\text{strictify}(\lambda x. \text{strictify}(\lambda y. [x] + [y])),) \].

Overall, [HOL] expressions are printed using the default color. We resolve ambiguities between the underlying mathematical syntax (i.e., [HOL]) and the [OCL] level by using colors: Expressions that are internally used within HOL-OCL, like the lifting operator \[\_\]\_ are printed in a green typeface. Using our mathematical [OCL] syntax, expressions on the [OCL] level, like \[- \land -\], are written in a magenta typeface. For the concrete syntax presented in the standard, e.g., \[- + -\], we use a magenta typeface. In rare cases, notable for the arithmetic operators like \[- + -\], we deviate from this scheme out of technical reasons.

[HOL] formulae are written using the usual mathematical notion, i.e., \(s \in S\). Theory files for HOL-OCL and Isabelle/HOL are printed as follows:

```plaintext
theory royals_and_loyals
imports OCL
begin
load_xmi "royals_and_loyals_ocl.xmi"
end
```

Keywords are printed in a green typeface.

SML code fragments are written inline like \(\text{fn } x => 2 \cdot x\) or in display style:

```plaintext
datatype OclType = Integer | Real | String | Boolean
| OclAny | Set of OclType (* ... *)
```

Keywords are printed in a blue typeface.

Further, we mark problems and extensions to the [OCL] standard as follows:

- Errors in the standard are marked by with a danger sign on the margin throughout this document, and our definitions can be seen as our proposal for repair.

- In some cases we see our proposals as an extension of the standard, these cases are marked with an exclamation sign on the margin.

### 1.4. Acknowledgements

- HOL-OCL uses the component [su4smi] which is developed by Achim D. Brucker, Jürgen Doser, and Burkhart Wolff as the underlying [UML] repository.
Chapter 1. Introduction

- Internally, su4sml uses the XML parser Functional Extensible Markup Language (XML) Parser (FXP) [2].
- During his semester thesis, Simon Meier checked the library for completeness and substantially extended the theory.
Chapter 2.

Background

2.1. Higher-order Logic

Higher-order Logic (HOL) [14, 8] is a classical logic with equality enriched by total polymorphic higher-order functions. It is more expressive than first-order logic, e.g., induction schemes can be expressed inside the logic. Pragmatically, HOL can be viewed as a combination of a typed functional programming language like SML or Haskell extended by logical quantifiers.

HOL is based on the typed \( \lambda \)-calculus—i.e., the terms of HOL are \( \lambda \)-expressions. The application is written by juxtaposition \( E \, E' \), and the abstraction is written \( \lambda x. \, E \). Types may be built from type variables (like \( \alpha \), \( \beta \), . . . , optionally annotated by type classes as in \( \alpha :: \text{order} \) or \( \alpha :: \text{bot} \)) or type constructors (like bool or nat). Type constructors may have arguments (as in \( \alpha \) list or \( \alpha \) set). The type constructor for the function space \( \Rightarrow \) is written infix: \( \alpha \Rightarrow \beta \); multiple applications like \( \tau_1 \Rightarrow (\ldots \Rightarrow \tau_n \Rightarrow \tau_{n+1} \ldots) \) have the alternative syntax \([\tau_1, \ldots, \tau_n] \Rightarrow \tau_{n+1}\). HOL is centered around the extensional logical equality \( = \) with type \([\alpha, \alpha] \Rightarrow \text{bool}\), where bool is the fundamental logical type. We use infix notation: instead of \((=_E) \, E_1 \, E_2\) we write \(E_1 = E_2\). The logical connectives \( \wedge, \lor, \rightarrow \) of HOL have type \([\text{bool}, \text{bool}] \Rightarrow \text{bool}\), \(\neg \) has type bool \(\Rightarrow\) bool. The quantifiers \(\forall \) and \(\exists \) have type \([\alpha \Rightarrow \text{bool}] \Rightarrow \text{bool}\). The quantifiers may range over types of higher order, i.e., functions.

The type discipline rules out paradoxes such as Russel’s paradox in untyped set theory. Sets of type \( \alpha \) set can be defined isomorphic to functions of type \( \alpha \Rightarrow \text{bool} \); the elementhood \( \in \) has then type \([\alpha, \alpha \text{ set}] \Rightarrow \text{bool} \) and corresponds basically to the application; in contrast, the set comprehension \([\_ = \_] \) has type \([\alpha \Rightarrow \text{bool}] \Rightarrow \alpha \) set and corresponds to the \( \lambda \)-abstraction. The definition of \( \cup, \cap \) and set complement is standard, as well as bounded versions of the quantifiers \(\forall \) and \(\exists \) of type \([\alpha \Rightarrow \text{bool}] \Rightarrow \text{bool}\).

Thus, HOL allows for specifications in a very natural (mathematical) way, similar to naive set theory. In contrast to the latter, however, HOL is a consistent logical system (relative to ZFC) provided that all formulae can be type-checked, which we always assume throughout this text and which is technically done behind the scenes.

The modules of larger logical systems built on top of HOL are Isabelle theory files (or just: theories, if no confusion arises). Among many other constructs, they con-

\(^1\)to be more specific: parametric polymorphism
tain type and constant declarations as well as axioms. Since stating arbitrary axioms in a theory is extremely error-prone and should be avoided, only very limited forms of axioms should be admitted and the constraints (both syntactical and semantical) checked by machine. These fixed blocks of declarations and axioms described by a syntactic scheme were called conservative theory extensions since and extended theory is consistent ("has models") provided the original theory was. Four different conservative extensions were discussed in the literature: constant definition, type definition, constant specification, type specification (see [17] for an in depth discussion, which also includes the correctness proofs). For example, the mostly used constant definition consists of a constant declaration
\[ c :: \tau \]
and an axiom of the form:
\[ c = E \]

where \( c \) has not been previously declared, the axiom is well-typed, \( E \) is a closed expression (i.e., does not contain free variables) and \( E \) does not contain \( c \) (no recursion).

A further restriction forbids type variables in the types of constants in \( E \) that do not occur in the type \( \tau \). As a whole, a constant definition can be seen as an "abbreviation" which makes the conservativity of the construction plausible (but see [17] for the hairy details), and the syntactic side-conditions are checked by Isabelle automatically.

The idea of an "abbreviation" is also applied for the conservative type definition of a type \((\alpha_1, \ldots, \alpha_n)T\) from a set \( \{ x \mid P(x) \} \).

In this case, the set of type constructors is extended by the constructor \( T \) of arity \( n \). The predicate \( P \) of type \( \tau \Rightarrow \text{bool} \) for a base type \( \tau \) constructs a set of elements \( \tau \) set; the new type is defined to be isomorphic to this set. Technically, this isomorphism is stated by the declaration of two constants representing the abstraction and the representation function and by two axioms over them. More precisely, the constant \( \text{Abs}_T \) of type \( \tau \Rightarrow (\alpha_1, \ldots, \alpha_n)T \) and the constant \( \text{Rep}_T \) of type \((\alpha_1, \ldots, \alpha_n)T \Rightarrow \tau \) were declared. The two isomorphism axioms have the form:
\[ \text{Abs}_T(\text{Rep}_T(x)) = x \]
and
\[ P(x) \Rightarrow \text{Rep}_T(\text{Abs}_T(x)) = x \]

where the precise meaning of the \( \_ \Rightarrow \_ \)-arrow will become apparent in the next section; for the moment, we can consider it equivalent to the logical implication \( \_ \rightarrow \_ \).

The Isabelle/HOL library (and the libraries of similar HOL systems) prove, that the typed set theory including least fixedpoint theory, order theory including well-founded recursion, number theory including real number theory and theories for data-structures like pairs, type sums and lists can be build on top of the HOL core-language entirely by these conservative definitions. A large part of these theories consists in deriving
2.2. Concepts and Use of Isabelle/HOL

Isabelle [37] is a generic theorem prover of the Logic for Computable Functions (LCF) prover family implemented in Standard Meta Language (SML). We heavily use the possibility to build SML programs performing symbolic computations over formulae in a logically safe way on top of the logical core engine: this is how the proof procedures of HOL-OCL are built technically. Isabelle/HOL offers support for checks for conservativity of definitions, data types, primitive and well-founded recursion, and powerful generic proof engines based on rewriting and tableaux provers.

Isabelle’s proof engine can directly process natural deduction rules: $A_1 \implies \ldots \implies A_n \implies A_{n+1}$, also written $\llbracket A_1; \ldots; A_n \rrbracket \implies A_{n+1}$, is viewed as a rule of the form “from assumptions $A_1$ to $A_n$, infer conclusion $A_{n+1}$”. Here, $\implies$ denotes the built-in meta-implication of Isabelle. With meta-implications, also more complex rules like: if assumption $B$ can be inferred from assumption $A$, infer $A \rightarrow B$ (also called: implication introduction) can exactly be expressed in Isabelle by: $(A \implies B) \implies A \rightarrow B$. In the mathematical literature, such natural deduction rules were also written as:

$$
\begin{array}{c}
A_1 \\
\vdots \\
A_n \\
\hline
A_{n+1}
\end{array}
\quad
\begin{array}{c}
\llbracket A \rrbracket \\
\vdots \\
B \\
\hline
A \rightarrow B
\end{array}
$$

A proof state in Isabelle contains an implicitly conjoint sequence of Horn-clause-like rules called subgoals $\phi_1, \ldots, \phi_n$ and a goal $\phi$. Logically, subgoals and the goal are connected to a theorem of the form $\llbracket \phi_1; \ldots; \phi_n \rrbracket \implies \phi$. In order to cope with quantifiers naturally occurring in logic, subgoals have a slightly more general format than just Horn-clauses: variables may be bound by a built-in meta-quantifier:

$$
\bigwedge x_1, \ldots, x_m. \llbracket A_1; \ldots; A_n \rrbracket \implies A_{n+1}
$$

The meta-quantifier $\bigwedge$ helps to capture the usual side-constraints “$x$ must not occur free in the assumptions” for quantifier rules; meta-quantified variables can be logically considered as free variables. Further, Isabelle supports meta-variables (written $?x, $?y, . . .), which can be seen as “holes in a term” that can still be substituted. Meta-variables are instantiated by Isabelle’s built-in higher-order unification and occur only inside proofs.

The initial proof state is built from the trivially true theorem $\phi \implies \phi$ for any (typed) formula $\phi$. A theorem is proven if a final proof state of the form $\phi$ could be reached by tactics i.e., SML functions allowing for the transformation of proof states. It is a key feature of Isabelle’s design that all tactics are based on a few operations provided by the logical core engine of Isabelle. Moreover, these core operations log all logical operations in a derivation tree called proof-object; thus, if someone has serious doubts on the correct implementation of Isabelle’s proof procedures (i.e., tactics),
he/she may generate the proof objects and check the derivations by an independent program following the rules of the logic.

Besides the SML-based programming interface, there is also an own input language to Isabelle theories, called Intelligible semi-automated reasoning (Isar). Isar allows for writing specifications consisting of definitions, proofs, and technical setups for the prover in a fairly readable way. In particular, Isabelle can generate LaTeX documentation while checking the entire hierarchy of theories (this is what this document is ...).

Isabelle theories written in Isar are supported by a fairly powerful Emacs-based front-end called ProofGeneral. HOL-OCL is in fact integrated and customized for this front-end, such that HOL-OCL specific Isar commands and HOL-OCL specific fonts can be used.

2.3. A Short Introduction into \texttt{UML/OCL}

Now we introduce the concepts of the target of our work, namely Unified Modelling Language (UML) and Object Constraint Language (OCL). As mentioned, Unified Modelling Language (UML) provides a variety of diagram types for describing dynamic (e.g., state charts, activity diagrams) and static (e.g., class diagrams) system properties.

One of the more prominent diagram types of the UML is the \textit{class diagram} for modelling the underlying object-oriented data model of a system. The class diagram in Figure 2.1 illustrates a simple accounting scenario where customers can own different kinds of accounts and transfer money between them. In more detail: customers are modelled as a \textit{class}. There are further classes modelling the different account types, a checkbook with checks and also transactions. A class does not only describe a set of \textit{object instances}, i.e., record-like data consisting of \textit{attributes} such as \texttt{balance}, but also \textit{operations} defined over them. The class \texttt{Customer} in our example has only attributes, namely \texttt{name}, \texttt{address}, \texttt{gender}, and \texttt{title}. The different account types are organized in a hierarchy of subtypes denoted by an arrow, i.e., between \texttt{Account} and \texttt{BankAccount}. Such an subtype relation is called \textit{inheritance} in the object-oriented paradigm. Sometimes this is also called \textit{specialization}, i.e, a \texttt{CurrencyTrading} specializes a \texttt{BankAccount}. Further, it is characteristic for the object-oriented paradigm that the functional behavior of a class and all its methods are also accessible for all subtypes. A class is allowed to redefine an inherited method, as long as the method interface does not change; this is called \textit{overriding}, as it is done in the example for the operation \texttt{makeWithdrawal}.

Of course, there is a relation between customers and accounts. This is modelled in UML by an \textit{association} \texttt{belongsTo}. An association can be constrained by \textit{multiplicities}. In Figure 2.1 the multiplicities of the association \texttt{belongsTo} requires that every object instance of \texttt{Account} is associated with exactly one object instance of \texttt{Bank}. This captures the requirement that every account belongs to a unique bank. In the other direction, the association models that an instance of class \texttt{Bank} is related to a (non-empty) set of instances of class \texttt{Account} or its subtypes.

20
2.3. A Short Introduction into **UML**

A **Transaction** can be modeled as:

```plaintext
t1
belongsTo
owner
1..*
accounts
includes
20..50
checks
checkbook
checkbook
0..25
book
0..1
checkAccount
```

**Account** includes:

```plaintext
makeDeposit(in amount : Real) : Boolean
makeWithdrawal(in amount : Real) : Boolean
getBalance() : Real
```

**Customer** includes:

```plaintext
name : String
address : String
gender : Boolean
title : String
```

**BankAccount** includes:

```plaintext
accountNumber : String
currency : String
balance : Real
getBalance() : Real
makeDeposit(in amount : Real) : Boolean
makeWithdrawal(in amount : Real) : Boolean
getCurrency() : String
```

**CurrencyTradingAccount** includes:

```plaintext
tradingCurrency : String
price : Real
buy(in amount : Real) : Boolean
sell(in amount : Real) : Boolean
getBalance() : Real
getNormBalance() : Boolean
```

**CreditAccount** includes:

```plaintext
makeDeposit(in amount : Real) : Boolean
makeWithdrawal(in amount : Real) : Boolean
creditLimit : Real
maxAmount : Real
```

**Checkbook** includes:

```plaintext
checkbook
book : 0.25
includes
checkAccount
```

**Check** includes:

```plaintext
newOperation() : void
```

**Figure 2.1.** Modeling a simple banking scenario with **UML**
Chapter 2. Background

Understanding OCL as a data-oriented specification formalism, it seems natural to refine class diagrams using OCL for specifying invariants, pre-conditions and post-conditions of operations. By specifying the following OCL constraints one can make the specification of the class Account much more precise, e.g., we can describe that accountNumbers are unique:

```
context BankAccount
inv: BankAccount.allInstances ->forall (a1,a2 | a1<>a2 implies a1.accountNumber <> a2.accountNumber)
```

Or constrain the effects of operations:

```
context BankAccount::getBalance():Integer
post: result=balance
context BankAccount::getCurrency():String
post: result=currency
context BankAccount::makeDeposit(amount:Integer):Boolean
post: balance = balance@pre + amount
context BankAccount::makeWithdrawal(amount:Integer):Boolean
post: balance = balance@pre - amount and currency = currency@pre
```

where in post-conditions @pre allows one to access the previous state.

In UML, class members can contain attributes of the type of the defining class. Thus, UML can represent (mutually) recursive data types. Moreover, OCL introduces also recursively specified methods [41]; however, at present, a dynamic semantics of a method call is missing ([12] gives a short discussion of the resulting problems).

Note, that many diagrammatic UML-features can be translated to OCL-expression without losing any information, e.g., associations can be represented by introducing implicit set-valued attributes into the objects with a suitable data invariant describing the multiplicity. These transformations are already described in the UML-standard [40]. Some of these transformation were done during the process of loading an UML/OCL specification into HOL-OCL, e.g., we have not to provide a special treating for associations in our proof environment.\(^2\)

### 2.4. A Note About Standards

OMG standards are developed in an open process leading to a variety of “standardization” documents. Especially for UML and OCL which have a long history and our particular choice needs some explanation.

\(^2\)Direct support for associations is planned for the future.
OCL was introduced as an OMG specification language as additional document [38] completing the UML 1.1 standard [39]. In later releases of the UML standards of the version 1.x series the OCL standard was a chapter of the UML specification, e.g., [40, Chapt. 6]. As several shortcomings of the OCL 1.x specifications were discussed [50, 32, 21] the development of OCL 2.0 led to many documents describing the different intermediate states.

For our work we have chosen the following two documents as our main references:

1. **OMG Unified Modeling Language Specification** (excluding Chapt. 6 which describes OCL 1.5) [40] which is publicly available at [http://www.omg.org/cgi-bin/apps/doc?formal/03-03-01.pdf](http://www.omg.org/cgi-bin/apps/doc?formal/03-03-01.pdf)


This combination is also used by a many recent tools, especially the OCL type-checker [1] from the Dresden University which is integrated into our tool chain.

---

3 also known as OMG document “formal/03-03-01”

4 also known as OMG document “ptc/03-10-14”
Chapter 2. Background
Part II.

Theory
Chapter 3.

Foundations

3.1. An Overview of Embedding Techniques

A theory representing syntax and semantics of a programming or specification language in another specification language is called an embedding. While the underlying techniques are known since the invention of typed $\lambda$-calculus, it was not before the late seventies that the overall importance of higher-order abstract syntax (HOAS) [43] for the representation of binding in logical rules and program transformations [24] and for implementations [43] was recognized.

As an example, we revise the universal quantifier of HOL already introduced in Section 2.1: it is represented in higher-order abstract syntax (HOAS) by a constant $\text{All :: (} \alpha \Rightarrow \text{bool) \Rightarrow bool}$, where the term $\text{All}(\lambda x. P(x))$ is paraphrased by the usual notation $\forall x. P(x)$. This is in contrast to the usual textbook definition for predicate logic, where a free datatype for terms and predicates, explicit substitution and well-typedness functions over them is provided. This conventional representation requires explicit side-conditions in logical rules over quantifiers preventing variable-clashes and variable capture. The representation using higher-order abstract syntax (HOAS) has two advantages:

1. the substitution required by logical rules like $\forall x. P(x) \Rightarrow P(t)$ can be directly implemented by the $\beta$-reduction underlying the $\lambda$-calculus, and
2. the typing discipline of the typed $\lambda$-calculus can be used to represent the typing of the represented language. For example, a multi-sorted first-order logic (having syntactic categories for arithmetic terms, list terms, etc.) is immediately possible by admitting expressions of type nat and $\alpha$ list.

In short, HOAS has the advantage of “internalization” of substitution and typing into the meta-language, which can therefore be handled significantly more general and substantially more efficient, which is a prerequisite for using Isabelle as an implementation platform.

When using HOAS-style semantic definitions, the technique is extended into a “shallow embedding” [9] of an object-language, a technique which is opposed to a “deep embedding” which treats syntax by free datatypes and semantics via semantic interpretation functions as common in logical textbooks.

A shallow embedding definition of the universal quantifier is, for example, $\text{All P} \equiv (P = \lambda x. \text{true})$ (the propositional function of the “body” of the quantifier must be
equal to the function that yields true for any argument), a deep representation follows usual textbooks:

$$\text{Sem}[\forall x. P(x)] \equiv \begin{cases} 
\text{true} & \text{if } \text{Sem}[P(x)] \mid [x := d] \text{ for all } d \\
\text{false} & \text{otherwise}
\end{cases}$$

where we assume a meta-language with well-defined concepts such as if, otherwise, and for all, e.g., Zermelo-Fränkel-set theory.

As can be seen, shallow embeddings can have a remarkably different flavor in their semantic presentation, in particular when striving for conservativity as in the example above. However, since the usual inference rules were derived from these definitions (like the rule $\forall x. P(x) \Rightarrow P(t)$ from above), they are finally proven as semantically equivalent.

If a shallow embedding is built entirely by conservative theory extensions (which implies that the language definition is consistent if the meta-language is; cf. Section 2.1), we also speak of a conservative embedding.

3.2. Embeddings of Specification Languages

Since shallow embeddings in itself are fairly well-known in principle, the question arises, what the fundamental problems and technical challenges exist for representing “real world” specification languages. The main difficulty stems from the fact, that the most successful shallow-embeddings (as, for example, HOL encoded in typed $\lambda$-calculus) are designed to fit to the underlying meta-language. In contrast, “Real world” languages are typically conceived solely on a kind of mathematical notation based on naïve set theory and no specific experience in theorem proving in mind; in some cases, the definition process of a “real” specification language takes place in a “development by committee” process prone to all sorts of arcane compromises. While a deep embedding is always possible whenever a formal semantics exists, a shallow embedding may conflict with certain constraints the shallow technique imposes, be it on the representability of the binding structure, the type discipline and the semantics of the language.

As an example for a different outcome of language design decisions, one might consider the following pathological case: In standard logic textbooks (like [31]), the notion of a model also admits empty carrier sets for sorts or types. Therefore, in many textbook-driven research papers, a lot of consideration of these pathological cases is made. In a tool-oriented research approach, it suffices to see that $\beta$-reduction is simply not sound.\footnote{Consider the two types $a$ and $b$ with some empty carrier set $A$ and a non-empty carrier $B$. Since $B^A$ is the empty set, $(\lambda x :: a \Rightarrow b. C)$ is an uninterpretable term and $(\lambda x :: a \Rightarrow b. C) = C$ does not hold for some constant $C$ interpretable in $B$ and some variable $X$} Given the fact that $\beta$-reduction is vital for higher-order logic provers and for the issue of internalization of substitution, this led to a modification of the model notion, i.e., to ruling out non-empty carrier sets and living with the methodological consequences such as side conditions in type definitions.

1\footnotemark
Thus, even fairly worked out semi-formal semantic descriptions of a specification language may be quite distant to a formal (i.e., machine-checkable) representation, and even a formal semantic representation may be quite distant to a set of derived rules that allow for the formal support of a particular method of this language. Providing paradigmatical embeddings and new techniques helpful for bridging this gap is the main goal of this work. To describe the problem domain in more detail, we use the following classification: Specification languages may be:

1. *process oriented*, i.e., their focus of description is the *behavior*, usually described as possible sequences (traces) of states or communication events,

2. *data oriented*, i.e., their focus of description is the structure of data a system processes, its states, and individual steps the system performs when making a transition from one state to another, and

3. *object-oriented*, which is essentially a data-oriented specification approach where object-oriented data structuring techniques such as inheritance and subtyping are emphasized.

Examples for process-oriented specification languages are temporal or modal logics (such as LTL, CTL), \(\mu\)-calculus [15], or process-algebras (such as CCS [34] or CSP [45]). Examples for data-oriented specification languages are VDM [27] or Z [47], examples for the object-oriented language class are Object-Z [46] or UML/OCL [40, 41].

### 3.3. Challenges of a Shallow Embedding of UML/OCL

Producing a shallow embedding of the key features of object-oriented modeling in HOL is the technically most challenging enterprise in this work, but also the one with the greatest potential of use in software engineering. A shallow embedding in HOL must capture the essence of OCL with respect to subtyping and inheritance in a type system that does not provide a notion of subtypes, and provide means for extensibility and reuse to capture the pragmatics of object orientation. Moreover, a formalization must capture that object-oriented specification is deeply intertwined with the notion of *state*, i.e., a map of object-references to objects (abstractions of pieces of memory) and its possible *state transitions*. Technically, these constraints have the form of class invariants or pre and postconditions of methods; OCL in class diagram is thus a typical data-oriented formal modelling technique.

We consider the following example: Since path expressions resulting from UML class-diagrams are basic elements of the OCL language, one can state that the attribute \(a\) of an object \(self\) must be 5 in the pre state: \(\text{self} @ \text{pre}.a = 5\). With respect to a state transition, we may state that \(a\) increases during a transition: \(\text{self} @ \text{pre}.a = \text{self}.a + 1\).

Thus, since expressions in OCL do not only compute values, but also contain operations that refer to the pre and post state \((\sigma, \sigma_{\text{pre}})\), the semantics of an expression \(\phi\) depends on such state pairs: \(I[\phi]_{\sigma, \sigma_{\text{pre}}} = 11\), Definition A.30]. The standard also introduces \(I[\phi]_{\sigma}\), i.e., a semantic interpretation on only one state.
version is used for precondition and invariant expressions. For simplification reasons, we only use the former version; invariant and precondition expressions are sufficiently characterized syntactically by the requirement that they must not contain any \$pre in paths. For our purpose of a machine-representable, shallow embedding for OCL, we make a further simplification and omit an explicit semantic function I. This can be done by viewing \( \phi \) as a function from \((\sigma, \sigma_{\text{pre}})\) to the semantic value of \( I[\phi](\sigma, \sigma_{\text{pre}}) = \phi(\sigma, \sigma_{\text{pre}}) \). Technically, this means that we have to lift over the context any semantic function occurring in \( \phi \), e.g., \( \_ + \_ \) or \( \_ = \_ \) in the example above, over the states \( \tau = (\sigma, \sigma_{\text{pre}}) \). More precisely, the addition \( \_ + \_ \) of type \([\text{Integer}, \text{Integer}] \Rightarrow \text{Integer}\) in the sense of the standard must be lifted over the context \( \tau \), i.e., the corresponding operation \( \_ + \_ \) in HOL-OCL must have the type \( (\tau \Rightarrow \text{Integer}) \Rightarrow (\tau \Rightarrow \text{Integer}) \Rightarrow \tau \Rightarrow \text{Integer} \). When introducing the type synonym \( V_\tau(\alpha) \) for \( \tau \Rightarrow \alpha \), this type transformation can be made explicit: the HOL-OCL type for the OCL type above is \([V_\tau(\text{Integer}), V_\tau(\text{Integer})] \Rightarrow V_\tau(\text{Integer}) \). We call \( \tau \) also the context of this type transformation, which is classified as “semi-canonical” translation (see Section 3.4). It turns out that the complete set of OCL operators, be it from the core logic or be it from the quite rich library, can be syntactically constructed as semi-canonical translations. This includes operators with non-trivial binding-structure such as the \texttt{let}-construct, or the \texttt{iterators} (corresponding to \texttt{fold}-like operators known from functional languages).

A further complication of a shallow embedding of OCL is due to the fact, that paths into the state may be undefined, and therefore all functions in expressions may have undefined results. Technically, this means that we have to extend all types by an additional \( \perp \) element—which leads to another “semi-canonical” type transformation—, but for the expression semantics, the logic, and the calculi, this has dramatic consequences: Many rules can only be applied for “defined” values, which results in particular case splits and side-conditions in thousands of lemmas and rules. These side-conditions must be established by more or less complex side-calculi; for the example of undefinedness, for example, we need rules that infer facts like “if \( a + b \) is defined, \( a \) and \( b \) must be defined.” Thus, although considered semantically fairly trivial in many research communities, a proper treatment of definedness in the sense of the standard is usually avoided in deduction systems over OCL-like or VDM-like languages because of the size and complexity of the side-calculi.

Beyond the typical problems concerning undefinedness and states, the most challenging part of HOL-OCL as a shallow embedding consists in a particular semantic understanding of the key-notions UML state diagrams. This implies a notion of typed state, covering concepts such as static and dynamic types in systems, and combining this with the practical need of extensibility of the object-type system and modular theories over it. In the end, the theory also offers a new perspective on methods and late-binding method invocation. However, a modular treatment of state diagrams requires a kind of “semantic compiler” converting it into a semantic theory over the types, their relation, their properties with respect to object construction and destruction, etc.
3.4. An Overview over Embedding Techniques

To put HOL-OCL as an embedding into perspective, we define several notions for their classification. As already mentioned, we call a syntax translation:

- **canonical**, if and only if the non-terminals (or types) of the object language can be mapped one-to-one to type constructors in the \( \text{HOL} \) representation,

- **semi-canonical**, if and only if the non-terminals (or types) of the object language can be mapped to type expressions (possibly containing free type variables),

- **pre-compiled**, if and only if the non-terminals (or types) of the object language were mapped to type expressions according to their context in the term-language.

As an example for a canonical logical embedding, consider \( \text{HOL-CSP} \) [49]. Here, the syntactical category of a “process over an alphabet \( \Sigma \)” can be directly represented by a type constructor \((\alpha)\text{process} \), which is represented by the semantic domains of \( \text{CSP} \) (traces, failures, divergences). Binding of summation operators can be represented directly by higher-order abstract syntax (HOAS), as well as the fixedpoint operator. Derived calculi allow for equivalence proofs of processes (even over infinite alphabets) via fixpoint induction.

As an example for a pre-compiled translation, consider the \( \text{HOL-Z} \) embedding [11]. While most operators from Z’s “mathematical toolkit” can be compiled canonically, the expressions for the schema calculus must be pre-compiled. In particular, a schema expression \( A \) may be mapped to a function \( \alpha_1, \ldots, \alpha_n \Rightarrow \text{bool} \), where both the \( \alpha_i \) and the \( n \) depend on previous declarations.

We turn now to a classification of type systems resulting from an embedding function \( \text{embedding} : L \Rightarrow \text{HOL} \) mapping expressions of an object-language \( L \) to expressions in \( \text{HOL} \). We assume the existence of predicates \( \text{welltyped}_L \) (assuring that an expression of language \( L \) is welltyped with respect to type discipline) and \( \text{welltyped}_\text{hol} \) (assuring that a \( \text{HOL} \)-expression is welltyped with respect to the simple-type type-system including parametric polymorphism called \( \lambda^\alpha \)).

Then we can characterize a type translation underlying an embedding as

- **approximate**, if and only if for any \( e \), \( \text{welltyped}_\text{hol}(\text{embedding } e) \) is implied by \( \text{welltyped}_L(e) \),

- **tight**, if and only if the type translation is approximative and we have additionally for any \( e \), \( \text{welltyped}_L(e) \) is implied by \( \text{welltyped}_\text{hol}(\text{embedding } e) \).

Thus, if an embedding is tight, a possible implementation for a type-checker consists in applying the \( \text{embedding} \)-function and trying to type-check its result in \( \text{HOL} \); an error on the converted term indicates the existence of a type error on the original term. Note, however, that the practical usability of such a conceptual implementation is usually very limited, in particular for semi-canonical or pre-compiled languages, since the error messages gained by this process are hard to decipher for users not familiar with the semantic details of the embedding.
Chapter 3. Foundations

<table>
<thead>
<tr>
<th></th>
<th>Process-Oriented</th>
<th>Data-Oriented</th>
<th>Object-Oriented</th>
</tr>
</thead>
<tbody>
<tr>
<td>Example</td>
<td>HOL(\rightarrow)CSP</td>
<td>HOL-Z</td>
<td>HOL-OCL</td>
</tr>
<tr>
<td>Syntax</td>
<td>canonical Translation</td>
<td>pre-compiled</td>
<td>OCL semi-canonical, HOAS</td>
</tr>
<tr>
<td>Typing</td>
<td>tight</td>
<td>approximate</td>
<td>tight</td>
</tr>
<tr>
<td>Semantics</td>
<td>denotational semantics, complex side-calculi</td>
<td>fairly trivial</td>
<td>complex side-calculi, three-valued logics, theory morphisms</td>
</tr>
</tbody>
</table>

Table 3.1.: Comparing Logical Embeddings

On this basis, we can compare logical embeddings like \(\text{HOL}\rightarrow\text{CSP}\) [49], HOL-Z [11] and the present HOL-OCL in Tab. 3.1. The reason for the non-tightness of the Z embedding lies in a mere technicality: the Z type-system supports \(n\)-ary tuples or records. They are distinct to their (isomorphic) representation as nested pairs: \((a, b, c)\) has not the same type as \((a, (b, c))\). Since the latter is the representation of the former in HOL-Z, an expression like \((a, b, c) = (a, (b, c))\) can be typed after applying the embedding function although it cannot be typed with the Z type discipline.

Some explanation for the remarks in the line summarizing semantic features: in a shallow embedding, it is necessary to express syntactic constraints (e.g., for admissibility as in the case of fixpoint-induction, or continuity in the case of process-refinement, or state-passing in the case of rewriting in (OCL) semantically, i.e., by second or third-order predicates expressing conditions that were otherwise characterized by inductively defined subsets of the syntax. These semantic predicates result in side-conditions which can be established by derived rules having a similar form and purpose than the inductive rules of a sub-language definition (for example, the admissibility rules characterize formulae built over logical constants, \(\_\land\_\), \(\_\lor\_\) and universal quantification; thus, existential quantification or negation are ruled out except that they occur in the construction of constant terms). However, their semantic construction leads sometimes to unexpected generalizations and often improves the insight into their logical nature than their (traditional) syntactic counterparts.
Chapter 4.

Faithfully Representing \textbf{UML/OCL}

In the first two sections of this core chapter on our formalization of UML and OCL semantics, we refine our overall goal for a faithful formal semantics with respect to the standard into several sub-goals and then present an overall architecture of HOL-OCL meeting these goals. The subsequent sections are concerned with various aspects of this architecture.

4.1. A Note About Standard Compliance

We claim that we provide a semantic representation compliant with the OCL standard semantics definition. In this section, we make our claim more precise, in particular we have to discuss to which parts we claim to be compliant. First, the OCL standard is divided into normative parts and informative, i.e., not normative, parts. The semantics\footnote{A nice overview of the different usages of the word “semantics” is given in [22].} of the standard appears in the following chapters of [41]:

Chapter 7 “OCL Language Description”: This informative chapter motivates the use of OCL and introduces it in an informal way, mostly by showing examples. We used this chapter mainly for catching the intentions of the standard in cases where the other parts of the standard are unclear or contradictory.

Chapter 10 “Semantics Described using UML”: This normative chapter describes the “semantics” of OCL using the UML itself. Merely an underspecified “evaluation” environment is presented. Nevertheless, some of the information presented in this chapter is helpful for formalizing the standard.

Chapter 11 “The OCL Standard Library”: This normative chapter is, in our opinion, the best source of the normative part of the standard describing the intended semantics of OCL. It describes the semantics of the OCL expressions as requirements (in form of pairs of pre- and postconditions) they must fulfill. Overall, we prove these requirements for our embedding and thus show that our embedding satisfies these requirements.

Appendix A “Semantics”: This informative appendix defines the syntax and semantics of OCL formally in a textbook style paper-and-pencil notion. It is mostly based on [44].
Chapter 4. Faithfully Representing UML/OCL

Overall, we see the semantic foundations of the standard critical for several reasons:

1. The normative part of the standard does not contain a formal semantics of the language.

2. The consistency and completeness of the formal semantics given in “Appendix A” is not checked formally.

3. There is no proof, neither formal nor informal, that the formal semantics given in the informative “Appendix A” satisfies the requirements given in the normative chapter 10.

Nevertheless, we think the OCL standard [41] (“ptc/03-10-14”) is mature enough to serve as a basis for a machine-checked semantics and formal tools support.

When we claim to be compliant to the standard, we do not mean that we converted “literally” the “Semantics” chapter of the OCL standard [41, appendix A] into an Isabelle theory. The deviations from the standard can be grouped into the following six classes:

Making the standard more precise: The most important point here is the need for a typed set theory as meta-language for the description of the data-universe and the concepts “model” and “signature” left implicit in the standard: Formalizing this in HOL is neither possible nor desirable, since consistency can no longer be guaranteed. Another issue in this line is that collection types have “smashed semantics” in HOL-OCL (see subsection 4.7.1).

Presentational issues: This covers our decision to turn OCL into a shallow embedding, as well as our decision to use a combinator style presentation for the bulk of semantic definitions both for conceptual and technical reasons. In section 4.5, we show in detail why these formulations are equivalent to the ones used in the standard.

Generalizations: This covers for example our decision to use an infinite collection type Set,$\tau$, since logical connections between, e.g., $\rightarrow$ IsType and class invariants can therefore be satisfactorily treated inside an own typed set theory in OCL (see section 4.9)

Repairing glitches: The standard contains—as can be expected for a large semi-formal document—several errors in local definitions which were revealed during our formalization. Such cases are marked by with a danger sign on the margin throughout this document, and our definitions can be seen as our proposal for repair.

Suggesting Extensions: In some cases we see our proposals as an extension of the standard, these cases are marked with an exclamation sign on the margin.

Proofs for Compliance Requirements: The OCL standard contains a collection of formal requirements in its mandatory part with no established link to the informative “Appendix A” of the standard [41]. We provide formal proofs for the compliance of our OCL semantics with these requirements (see section 4.6).
4.2. Building-Blocks of the HOL-OCL Architecture

Providing alternative mathematical syntax: Being the first who did substantial proof work in OCL, we early noticed the need for a compact, mathematical notation for OCL specifications as alternative to the programming-language like notation used in the OCL 2.0 standard. However, we accept the latter one in our front-ends, and as an alternative input format in the proof engine, too. This proceeding can be seen as generalization and is also covered by the standard, see Appendix A.

4.2. Building-Blocks of the HOL-OCL Architecture

In section 3.3, we have already seen that HOL-OCL can be decomposed in a UML part concerned with a typed object store and underlying object universes on the one hand and the OCL part just assuming abstract contexts on the other. This decomposition gives a natural basis for a separation of concern.

Before describing architectural issues in more detail, one general remark on terminology is appropriate here: The UML standard makes a clear separation between operations and operation specifications on the one hand and methods on the other; the former are entirely a specification construct, the latter an implementation construct. The UML allows an operation to have several methods, even in different programming languages. Consequently, methods are out of the scope of our work so far. Further, we distinguish operations from operation specifications by viewing the former as mathematical functions, the latter as relations.

With respect to the OCL part, we have to provide:

- A technique for giving semantics to basic (value) types of OCL, i.e., Boolean, Integer, Set and collection types such as Set, or Sequence.
- A technique for giving semantics for built-in operations over contexts, capturing arithmetics, logic and collection type theories.
- A technique for giving semantics for user-defined operations.

With respect to the UML part (resulting in a semantics for path expressions), we have to provide:

- A mechanism to generate formal theories of typed object structures associated to classes and their relationships (e.g., inheritance).
- A technique for giving semantics for user-defined operations in the context of classes, leading also to a formal semantics of path expressions. In principle, this constitutes an embedding of a subset of the UML foundation package [40].
- And last but not least, we have to bring both embeddings together, i.e., generating semantics for invariants and operation specifications consisting of pre and post conditions.

Further, we aim for mechanisms providing modularization and extensibility:
Chapter 4. Faithfully Representing UML/OCL

- The object store should allow for modular proofs, i.e., one should be able to add new classes without the need of re-proving the properties of existing classes.

- The embedding of OCL should be easily useable with another kind of object store, e.g., one idea is to use OCL for a Java-like object store allowing, e.g., a distinction between $\perp$ and the "null" reference (pointer).

- The object-store should be useable without OCL, e.g., for a programming language description that allows for method definitions associated to an operation; thus, one could verify a method with respect to its operation specification in a Hoare-logic style of reasoning.

In this chapter we present techniques and concepts; the concrete collection of theories implementing them is contained in Appendix B.

Each of the mentioned techniques and encoding mechanisms can be organized into level, which were built in their core by formally defined theory morphisms called layer, in particular:

**Level 0:** This level defines the ground work for the embeddings. It consist out of two layers:

**Defining new Datatypes:** In this layer, we define HOL types, in particular auxiliary types for classes.

**Datatype Adaption:** In this layer, the HOL datatypes are adapted as needed, e.g., we glue basic datatypes together to objects or extend all datatypes by an special “undefined” element.

In summary, this level defines all datatypes and provides a extensible object store.

**Level 1:** This adds, respectively adapts, the functional behavior and finalizes the embeddings. it consists out of two layers:

**Functional Adaption:** This layer adapts and extend the functional behavior of our embeddings, e.g., it defines the strictness of operations and defines the semantics of operations invocations in the context of our object store.

**Embedding Adaption:** This layer adds infrastructure for handling contexts.

In summary, this level provides an embedding of core OCL (i.e., there are only OCL formulae without context declarations) and an embedding of an extensible object store with operation invocation.

**Level 2:** This level combines the two embeddings, i.e., it introduces the context of OCL formulae and defines the semantics of objects and method invocations with respect to the validity of the corresponding preconditions, postconditions and invariants.

An overview of this architecture is shown in Figure 4.1 and will describe both, levels and layers, in more detail in the following, in particular in section 4.7.
4.3. Formal Preliminaries for Representing Semantics in \textit{HOL}

First of all, we use the usual notation for Cartesian product $\alpha \times \beta$ and sum types $\alpha + \beta$. Recall that polymorphic type variables were represented by Greek letters. The theory of these type constructors is developed in the \textit{HOL} library and provides the constructor $(x, y)$ for pairs and the projections $\text{fst}$ and $\text{snd}$. The constructors for sum are $\text{Inl}$ and $\text{Inr}$; the projection is hidden in the “pattern match function”

$$\text{sumCase}(f, g, x) = \begin{cases} f(k) & \text{if } x = \text{Inl } k; \\ g(k) & \text{if } x = \text{Inr } k. \end{cases}$$

In \textit{OCL}, the notion of explicit undefinedness plays a fundamental role, both for the logical and non-logical expressions: some expressions will, when evaluated, have an undefined value. For instance, typecasting with $\text{oclAsType()}$ to a type that the object does not support or getting the $\text{-} \text{first()}$ element of an empty collection will result in undefined. \cite{OCL Specification, page 15}

Thus, concepts like \textit{definedness} and \textit{strictness} play a major role in the \textit{OCL}. We used Isabelle’s concept of a \textit{type class} to specify the class of all types but that contain the undefinedness element $\bot$. Additionally, we required from this class the postulate “all \textit{OCL} types must have one element different from the undefined value” to rule out certain pathological cases revealed during the proofs. For all types in this class, concepts such
as definedness
\[ \text{def}(x :: \alpha :: \text{bot}) \equiv (x \neq \bot) \]

or strictness of a function
\[ \text{isStrict}(f) \equiv (f \bot) = \bot \]

are introduced. We define a combinator strictify by
\[
\text{strictify}\ f(x) \equiv \begin{cases} 
  x \quad \text{if} \; x = \bot \\
  f(x) \quad \text{else}
\end{cases}
\]
with type \((\alpha :: \text{bot} \Rightarrow \beta :: \text{bot}) \Rightarrow \alpha \Rightarrow \beta\).

The operator strictify yields a strict version of an arbitrary function \(f\) defined over the type class bot.

Further, we use the type constructor \(\tau\up\) that assigns to each type \(\tau\) a type lifted by \(\bot\), for which we also write \(\tau\bot\). Since any type in \(\text{HOL}\) contains at least one element, each type \(\tau\bot\) is in fact in the type class bot. The function \(\up\) denotes the injection, the function \(\down\) its inverse for defined values. The case distinction function is defined by:
\[
\text{upCase}(f, c, x) = \begin{cases} 
  c \quad \text{if} \; x = \bot \\
  f(k) \quad \text{if} \; x = k 
\end{cases}
\]

We will complete our formalization section of the machinery specific to our shallow embedding by the type synonym \(V\tau(\alpha)\) already mentioned in Section 3.3 and some combinators that capture the semantical essence of context lifting. The type synonym \(V\tau(\alpha)\) is defined by:
\[
V\tau(\alpha) = \tau \Rightarrow \alpha
\]

On the expression level, context lifting combinators fining the distribution of contexts are defined as follows:
\[
\begin{align*}
\text{lift}_0\ f & \equiv \lambda \tau. \ f \\
\text{lift}_1\ f & \equiv \lambda X\ \tau. \ f(X\ \tau) \\
\text{lift}_2\ f & \equiv \lambda X\ Y\ \tau. \ f(X\ \tau)(Y\ \tau)
\end{align*}
\]
with type \((\alpha \Rightarrow \beta) \Rightarrow V\tau(\alpha) \Rightarrow V\tau(\beta)\), and
\[
\begin{align*}
\text{lift}_0\ f & \equiv \lambda \tau. \ f \\
\text{lift}_1\ f & \equiv \lambda X\ \tau. \ f(X\ \tau) \\
\text{lift}_2\ f & \equiv \lambda X\ Y\ \tau. \ f(X\ \tau)(Y\ \tau)
\end{align*}
\]
with type \((\alpha, \beta) \Rightarrow \gamma\) \Rightarrow [V\tau(\alpha), V\tau(\beta)] \Rightarrow V\tau(\gamma)\).

The types of these combinators reflect their purpose: they “lift” operations from \(\text{HOL}\) to semantic functions that are operations on contexts.

Operations constructed by context lifting enjoy the property, that the context is just passed, not changed. We call an operation context passing if it satisfies exactly this property. This is expressed formally as follows:
\[
\text{cp}(P) \equiv \exists f. \ \forall X\ \tau. \ P\ X\ St = f(X\ \tau) \ \tau \quad \text{with type} \ (V\tau(\alpha) \Rightarrow V\tau(\beta)) \Rightarrow \text{bool}.
\]

Context invariance of expressions will turn out to be a key concept allowing for converting an equivalence on \(\text{OCL}\) expressions into a congruence; thus \(\text{cp}\) will play a major role in side-calculi for \(\text{OCL}\).

The fundamental theory for undefinedness, elementary strictness combinators, lift-combinators and \(\text{cp}\) is developed in subsection B.3.1.
4.4. Encoding Standard OCL Operations

The standard contains “principles” for the semantics of the operations. Consider the example:

In general, an expression where one of the parts is undefined will itself be undefined.

\(\text{(OCL Specification [41], page 15)}\)

In other words, one could rephrase this semantic principle as “all operations are strict,” be it standard or user-defined operations. The OCL standard requires as default for all operations to be strict, both for the case of built-in like the \(\_ + \_ \text{ on Integer}_\tau\) or user defined operations declared in class diagrams. Other “principles” are hidden in the semantic definitions; for example the passing of the context. Since we have to define around hundred operators, it is tempting to cover these underlying principles in combinators once and forall, both for conceptual and technical reasons (as we will see in section 4.7 in more detail). For the combinators introduced in section 4.3, it is now straight forward to meet the semantic principles. For the new types \(\text{Boolean}_\tau\) (subsection B.4.2 “OCL Boolean”) and \(\text{Integer}_\tau\) (subsection B.4.6 “OCL Integer”), for example, a representation as types from the HOL library looks as follows:

\[
\begin{align*}
\text{Boolean} &= \text{bool}_\bot, \\
\text{Integer} &= \text{int}_\bot, \\
\text{Boolean}_\tau &= \text{V}_\tau(\text{Boolean}), \\
\text{Integer}_\tau &= \text{V}_\tau(\text{Integer}).
\end{align*}
\]

These definitions are contained in theories subsection B.4.14. These type definitions by type synonyms are typical for our semi-canonical type translation.

Basic constant definitions for \(T, F, \bot\) in \(\text{Boolean}_\tau\) or 0, 1, etc., in \(\text{Integer}_\tau\) using the lifting combinators of section 4.3 is straight-forward:

\[
\begin{align*}
\bot &\equiv \text{lift}_0(\bot), & \text{with type } \text{Boolean}_\tau, \\
T &\equiv \text{lift}_0(\text{true}), & \text{with type } \text{Boolean}_\tau, \\
F &\equiv \text{lift}_0(\text{false}), & \text{with type } \text{Boolean}_\tau, \\
\bot &\equiv \text{lift}_0(\bot), & \text{with type } \text{Integer}_\tau, \\
0 &\equiv \text{lift}_0(0), & \text{with type } \text{Integer}_\tau, \text{ and} \\
1 &\equiv \text{lift}_0(1), & \text{with type } \text{Integer}_\tau.
\end{align*}
\]

In fact, the definition for undefinedness is done for the polymorphic constant \(\bot\), expressing the fact that undefinedness is omnipresent in all types of the OCL language. Furthermore, we use within the theories the mathematical syntax; a translation table can be found in Appendix A.
An example for a strict boolean operation is the logical negation:

$$\neg \_ \equiv \text{lift}_1(\text{strictify}(\_ \circ (\neg \_)) \circ \text{lift}_1) \text{ with type } \text{Boolean} \rightarrow \text{Boolean}.$$  

From this definition, the usual logical laws for a strict negation are derived:

$$\neg \bot = \bot \quad \neg T = F \quad \neg F = T$$

A typical example for strict binary operation is the addition on integers; all other unary and binary operators of the theory of the follow this scheme of a constant definition:

$$\_ + \_ \equiv \text{lift}_2(\text{strictify}(\lambda x. \text{strictify}(\lambda y. [\_ x + [\_ y]))) .$$

From these definitions, computational rules on numbers can be derived, which perform computations like $3 + 4$ on the basis of binary representations; thus, computations of this kind can be handled fairly efficiently by the Isabelle rewriter. This representation technique for numbers is by no means new, it is merely a standard in Isabelle and adapted for these new types.

The encoding of the operations along the scheme described above is the default encoding; cases requiring special treatment are non-strict operators as well of higher-order constructs like quantifiers and the iterators of the OCL language similar to fold in functional programs:

> These iterator expressions always have a collection expression as their source, as is defined in the well-formedness rules . . .

(\text{OCL Specification [41], page 149})

Non-strict constructs such as $\partial \text{self}$ or $\partial \text{self}$ were defined via context lifting from the definedness predicate \textit{def} introduced in section 4.3, without using the strictness combinator.

### 4.5. Textbook vs. Combinator Style Semantics of Operations

As mentioned previously, we use a combinator-style presentation rather than a textbook-style presentation as used in the OCL standard, both for reasons of conciseness as well as better amenability techniques. It is formally shown in our Isabelle theories that from our definitions, the requirements [41, Chapter 11 (normative)] of the standard follow. In this section, we give an alternative reason for this: namely, we formally show that our semantics is equivalent to (a formalized version of) the semantics given standard [41, appendix A (non normative)].

The OCL 2.0 standard presents a definition scheme for all \textit{strict basic operations} just by one example. For the + operator on integers, it looks as follows: OCL [41] page
4.5. Textbook vs. Combinator Style Semantics of Operations

A-11] this definition is presented as:

\[ I(+)\langle i_1, i_2 \rangle = \begin{cases} i_1 + i_2 & \text{if } i_1 \neq \bot \text{ and } i_2 \neq \bot, \\ \bot & \text{otherwise.} \end{cases} \]

This semantic function for basic operations is integrated in the more general semantic interpretation function for OCL expressions in

Let Env be the set of environments \( \tau = (\sigma, \beta) \). The semantics of an expression \( e \in \text{Expr}_t \) is a function \( I[e] : \text{Env} \rightarrow I(t) \) that is defined as follows.

iv. \( I[w(e_1, \ldots, e_n)]\tau = I(w)(\tau)(I[e_1](\tau), \ldots, I[e_n](\tau)) \)

(OCL Specification [41], page A-26, definition A.30)

There are two more semantic interpretation functions; one concerned with path expressions (i.e., attribute and navigation expressions [41] Definitions A.21], and one concerning the interpretation of pre and postconditions \( \tau \models P \) which is used in two different variants.

To show the equivalence of the two formalization styles, we re-introduce a kind of “explicit semantic function” into our shallow embedding. Of course, which with respect to HOL semantics, this is just the identity. \( \text{Sem}E\gamma \tau \) can be thought of as the fusion of the two semantic functions \( I(\circ) \) and \( I[E] \):

\[ \text{Sem}[x] \equiv x \quad \text{with type } \alpha \rightarrow \alpha. \]

The definitions and exemplary proofs shown here are contained in theory subsection B.4.2.

Now we show for our first strict operation in OCL, the not operator, that it is in fact an instance of the standards definition scheme:

\[ \text{Sem}[\neg X] \gamma = \begin{cases} \neg \text{Sem}[X] \gamma \; \& \; \text{if } \text{Sem}[X] \gamma \neq \bot, \\ \bot & \text{otherwise.} \end{cases} \]

This is formally proven as lemma “not_faithfully_represented” in the theory subsection B.4.2. The proof is trivial and canonical: it consists of the unfolding of all combinator definitions (they are just abbreviations of re-occurring patterns in the textbook style definitions!) and the semantic function \( \text{Sem} \) which is merely a syntactic marker in our context.

For the binary example of the integer addition, one proceeds analogously and receives as result:

\[ \text{Sem}[X + Y] \gamma = \begin{cases} \text{Sem}[X] \gamma + \text{Sem}[Y] \gamma & \text{if } \text{Sem}[X] \gamma \neq \bot \text{ and } \text{Sem}[Y] \gamma \neq \bot, \\ \bot & \text{otherwise.} \end{cases} \]

(4.1)

This is formally proven in subsection B.4.2.

In the following, we summarize the differences between the OCL standards textbook definitions and our combinator-style approach:
Chapter 4. Faithfully Representing UML/OCL

1. The standard [41] chapter A assumes an “untyped set of values and objects” as semantic universe of discourse. Since we reuse the types from the HOL-library to give boolean, integers and reals a semantics, meta-expressions like \{true, false\} ∪ \{⊥\} used in the standard are simply illegal in our interpretation. This makes the injections \_\_ and projections \_\_ necessary.

2. The semantic functions in the standard are split into \(I(x), I[e]τ, I_{\text{Att}}[e]τ\) and \(τ ⊨ P\). Since we aim at a shallow embedding (which ultimately suppresses the semantic interpretation function), we prefer to fuse all these semantic functions into one.

3. The environment \(τ\) in the sense of the standard is a pair of a variable map and a state pair. The variable map is superfluous in a shallow embedding (binding is treated by HOL itself), our contexts \(τ\) just comprises the pair of pre and post state, thus an implementation of our notion of context.

4.6. Compliance to the Standards OCL Requirements

As already described, the semantics of OCL is spread over several chapters in the OCL standard. For example, with respect to undefinedness, it is stated that:

In general, an expression where one of the parts is undefined will itself be undefined. There are some important exceptions to this rule, however. First, there are the logical operators:

- True OR-ed with anything is True
- False AND-ed with anything is False
- False IMPLIES anything is True
- anything IMPLIES True is True

The rules for OR and AND are valid irrespective of the order of arguments and they are valid whether the value of the other sub-expression is known or not.

\[\text{(OCL Specification [41], page 15)}\]

which implies explicitly that OCL is based on a strong Kleene/uniLogic. Thus, most operators of the logical type like \_ and \_ (written \_∧\_) are explicitly stated exceptions from the “all operations are strict” principle.

In the normative part [41] chapter 11], requirements were formally stated on the standard operations of OCL; the question, if these requirements are met by the informative semantics description[appendix A] [41] has not been systematically investigated.

Although, for most of the logical connective this can be settled by rephrasing the requirement given in form of a truth table [41] page A-12, Table A.2], there is a contradiction with respect to the truth table for \text{implies} and the requirement given in the normative part of the standard [41] page 139].

It is a contribution of our work that we can in fact formally prove the requirements are met by our semantics. In the case of the logical connectives, compliance to the standard is proven by deriving lemmas representing the complete truth table as required in the standard. Further, we also prove the normative requirements, and
thus connect the informative formal semantics with the normative requirements of the standard. Moreover, lemmas were derived for special calculi allowing for automated reasoning in OCL (see chapter 5).

With respect to the requirements of the standard [41, chapter 11], e.g., for `isEmpty`:

```plaintext
isEmpty():Boolean
Is self the empty collection?
post: result = (self->size() = 0)  (OCL Specification [41], page 114)
```

we prove formally requirement satisfaction lemmas:

\[
\models \partial (self->size())
\]

\[
self->isEmpty() = (self :: (α Setτ))->size() = 0.
\]  (4.1)

Instead of using operation specifications, we prefer their reformulation as algebraic properties that are directly usable in proofs. The constraint \(\models \partial (self->size())\) (the size of the set must be defined) is a tribute to our extension of the standard to infinite sets; it has the effect to constrain this specification to finite sets, i.e., to the domain the requirement is intended to hold.

### 4.7. Organizing the Embedding into Layer

In the previous sections we developed a standard representation technique for OCL operations. In this section, we will revise and generalize this technique to meet a particular technical problem of our approach.

The conservative embedding approach for yielding semantics for a large language such as OCL must provide derivations for several thousand “folklore” theorems to be practically relevant. In this section, we present an approach for deriving the mass of these theorems mechanically from the existing HOL library. The approach assumes a structured theory morphism mapping library types and library functions to new types and new functions of the specification language (i.e., OCL) while uniformly modifying some semantic properties. It turns out that the technique also greatly facilitates the technical organization of the automated generation of theories from class diagrams (see section 4.9).

The key idea is to represent the structure of the theory morphism by the semantic combinators \(\_\_\_\_, V(\_), strictify(\_), lift0, lift1, lift2, lift3\) introduced in the previous sections. We say that a theory morphism is layered, if in each form of conservative extension the following decomposition into elementary theory morphisms (the layer) is possible:

1. For type synonyms \((α_1, \ldots, α_m)T\), there must be type constructors \(C_1\) to \(C_n\) such that

\[
(α_1, \ldots, α_m)T = C_n\left(\cdots (C_1(T))\right)\]
Chapter 4. Faithfully Representing UML/\textit{OCL}

2. for conservative type definitions \((\alpha_1, \ldots, \alpha_m)T\), there must be functions \(C_1\) to \(C_n\) such that

\[
(\alpha_1, \ldots, \alpha_m)T = \{ x :: C_n\left(\cdots\left(C_1\left(T'\right)\right)\right) \mid P(x) \}
\]

3. for constant definitions \(c\), there must be functions \(E_1\) to \(E_n\) such that:

\[
c = (E_n \circ \cdots \circ E_1)(c')
\]

where each \(C_i\), or \(E_i\) are (type constructor) expressions build from semantic combinators of layer \(S_i\) and \(T'\). Where \(c'\) is a construct from the meta logic. A layer \(S_i\) is represented by a specification defining the semantic combinators, i.e., constructs that perform the semantic transformation from meta-level definitions to object-level definitions. In Figure 4.2 we show a classification for such layers.

In the following sections, we will present a collection of layers and their combinators in more detail. We will associate the semantic combinators one by one to the specific layers and collect them in a distinguished variable \textit{SemCom}. Finally, we will put them together for our example \textit{OCL} and describe generic theorem proving techniques that exploit the layering of the theory morphism for \textit{OCL}.
4.7. Organizing the Embedding into Layer

4.7.1. Datatype Adaption

Datatype adaption establishes the link between meta-level types and object-level types and meta-level constants to object-level constants. While meta-level definitions in libraries of existing theorem prover systems are geared toward good tool support, object-level definitions tend to be geared to a particular computational model, such that the gap between these two has to be bridged. For example, in Isabelle/HOL the head-function applied to an empty list is defined to yield an arbitrary but fixed element; in a typical executable object-language such as SML, Haskell or OCL, however, this function should be defined to yield an exception element that is treated particularly. Thus, datatype adaption copes with such failure elements, the introduction of boundaries (as maximal and minimal numbers in machine arithmetics), congruences on raw data (such as smashing; see below) and the introduction of additional semantic structure on a type such as \( \alpha :: \text{bot} \), or similarly, complete partial orders [36].

Now, revising the definition (cf. section 4.4):

\[
\begin{align*}
\text{Boolean} &= \text{bool}_{\bot}, \\
\text{Boolean}_\tau &= \text{V}_\tau(\text{Boolean}),
\end{align*}
\]

we easily recognize the layer structure of these definitions.

We turn now to the semantical function combinators of this layer. We identify them in the injection \( \_\bot \) and the projection \( \_\bot \). We mark them as combinators by including them in the specific set: \( _\bot, V(_\bot), \text{strictify}_{\_\bot}, \_\bot, \text{lift}_0, \text{lift}_1, \text{lift}_2, \text{lift}_3 \in \text{SemCom} \).

As an other example for a congruence construction, we will show the smashing on sets, which occurs in the semantics of SML or OCL, for example. In a language with semantic domains providing \( \bot \)-elements, the question arises how they are treated in type constructors like product, sum, list or sets. Two extremes are known in the literature; for products, for example, we can have:

\[
(\bot, X) \neq \bot
\]

or:

\[
(\bot, X) = \bot
\]

The latter variant is called smashed product and smashed set. The semantics chapters make no clear decision here; since OCL tends to define its constructs towards executability and proximity of object-oriented programming languages such as Java, we opt for smashed collection semantics. The constant definition for the semantic combinator smash reads as follows:

\[
\text{smash } f \ X \equiv \text{if } f \bot X \text{ then } \bot \text{ else } X
\]

On this basis, the collection type \( \text{Set} \), for example, is built via the type definition:

\[
\alpha \text{ Set} = \left\{ X :: (\alpha :: \text{bot} \ \text{set}_\bot) \mid X \notin \text{Set} \right\}
\]

In fact, in the theories, the equivalent formulation as semantic combinators \( \text{smash}(\lambda x.X. x \in \text{Set}) = X \) is used instead of \( \bot \notin \text{Set} \).

Undefiniedness in Collections
Chapter 4. Faithfully Representing UML/OCL

and the type synonym:

\[ \alpha \text{Set}_{\tau} = V_{\tau}(\alpha \text{Set}) . \]

As a consequence of these definitions, sets in HOL-OCL may be infinite which allows representing the syntactic category of \textit{types} as sets in HOL-OCL.

This quotient construction identifies all sets containing \( \bot \) in one class which is defined to be the \( \bot \) of the type \( \alpha \text{ set} \). All other sets were injected into an own class. Thus, an embedding of smashed sets into the class bot can be done as via the (overloaded) constant definition:

\[ \bot \equiv \text{AbsSet} \bot \]

The injection AbsSet (together with the projection RepSet) is a consequence of the conservative type definition above (cf. section 2.1).

For nested \textit{OCL} types such as \( \text{Set} \langle \text{Set} \langle \text{Integer} \rangle \rangle \), the \textit{HOL} type is \( \text{Integer Set Set}_{\tau} \) and not \( \text{Integer}, \text{Set}, \text{Set}_{\tau} \) since context lifting is only necessary on the topmost level for each argument of an operation.

We “mark” the semantic combinators smash, \( \bot :: \alpha \text{ Set} \), AbsSet, RepSet by including them into \textit{SemCom}.

4.7.2. Functional Adaption

\textit{Functional adaption} is concerned with the semantic transformation of a meta-level function into an object-level operation. Functional adaption may involve, for example, the

- \textit{strictification} of an operation, i.e., its result is undefined if one of its arguments is undefined,

- \textit{late-binding invocation} semantics for operations. This semantic conversion process is necessary for converting a function into an operation using dynamic overloading.

Technically, this is achieved by the strictify combinator already introduced in section 4.3. Overloading and late-binding can be introduced by the combinators \textit{invoke} and \textit{invokeS} described in this section.

As an example for strictification, we present a definition of a built-in operation over a non-trivial datatype adaption, \textit{OCL}’s union on sets. An intermediate version can now be defined as strictified version of \textit{HOL}’s union over the smashed type \( \alpha \text{ Set} \):

\[ \text{union} \equiv \text{strictify}(\lambda X. \text{strictify}(\lambda Y. \text{AbsSet}('\text{RepSet} X \cup '\text{RepSet} Y'))) \]

with type \( [\alpha :: \text{bot Set}, \alpha \text{ Set}] \Rightarrow \alpha \text{ Set} \).

The, from object-oriented programming languages, well-known concept of method-overloading is not yet fully supported by \textit{OCL}. We believe, this is more or less due to some accidental circumstances:
1. The UML standard [40, chapter 4.4.1] requires that operation names are unique within the same namespace. Albeit, the UML standard allows one to (explicitly) overwrite methods, i.e., implementation of operations.

2. The OCL standard [41, chapter 7.3.41] restricts the use of the precondition and postcondition declarations to operations or other behavioral features. Sadly, all OCL tools we know of do not support the specification of preconditions and postconditions for methods.

3. Whereas the OCL standard speaks on several places from operation calls, it does not give an hints how operation overloading should be solved, neither does it explain in details concepts like operation (method) calls or operation (method) invocations.

Bringing these together, one has to conclude, that operation overloading, and thus late-binding, is underspecified, or even not supported in OCL. Nevertheless, we think that overwriting inherited operations or methods is a very important feature of object-orientation and thus should be supported by the OCL. Thus we already provide the theoretical foundations for supporting late-binding (and thus overloading of operations) within HOL-OCL, nevertheless a concrete syntax for specifying this has to be worked out.\(^3\)

As many object-oriented languages provide a particular call-scheme for functions, called method invocation which increases the reusability of code. Moreover, many object-oriented programming languages such as Java assume a particular overload-resolution strategy for invocations; this well-known construction in programming language theory is called late-binding. Since we believe that future versions will overcome the self-restriction to operation calls, and since the UML definition expresses at several places a clear preference for overloading operations, we will discuss in this document and partially implement in HOL-OCL a late-binding semantics of method invocation.

The treatment of late-binding requires a particular pre-compilation step concerning the declaration of overloaded-methods discussed in section 4.13 in more detail; in this section, we will concentrate on the callee-aspect of method invocations, i.e., how to represent sub-expressions occuring in post conditions representing an invocation of a user-defined operation specification:

<table>
<thead>
<tr>
<th>context A::m(a1:t1,...,an:tn):t</th>
</tr>
</thead>
<tbody>
<tr>
<td>pre: ...</td>
</tr>
<tr>
<td>post: ...</td>
</tr>
</tbody>
</table>

In an “invocation,” e.g., a sub-expression \(a \cdot m(a_1, \ldots, a_n)\), the semantic value of the “dynamic type” (see section 4.9 for more details) of \(self\) is detected, i.e., a set of “objects” whose structure is to be discussed later. This “type” helps to lookup the concrete operation specification in the table. This specification can be turned into a function (just by picking some function satisfying the specification) which is applied

\(^3\)As simple workarounds, one ignore for operations the well-formedness constraint of UML that requires operation names to be unique within one namespace, or one could introduce new context declarations allowing one to specify preconditions and postconditions for methods.
Chapter 4. Faithfully Representing UML/OCL

to a as first argument (together with the other arguments). This semantics is captured in the \( \nu \)-indexed family of analogous \texttt{invoke} and \texttt{invokeS} semantic combinators; the former define call-by-name semantics, the latter call-by-value semantics satisfying the general strictness principle of OCL. Since \texttt{invokeS} can be realized on top of \texttt{invoke} as usual, we concentrate on the former.

The \texttt{invoke}-combinator is defined for the case \( \nu = 1 \), for example, as follows:

\[
\text{invoke} C t a \texttt{result} \equiv \lambda \tau. \begin{cases} 
\text{arbitrary} & \text{if } t (\text{Least } X. X \in \text{dom } t \land C(a \tau) \in X) = \text{None} ; \\
\text{f a result } \tau & \text{if } t (\text{Least } X. X \in \text{dom } t \land C(a \tau) \in X) = \text{Some } f .
\end{cases}
\]

Here, \texttt{Least} is a HOL operator selecting the least set of a set of sets, that satisfies a certain property. In this case, this property is that self (suitably converted by a casting function \( C \), see 4.9 for details) is contained in one of the domains of the lookup table \( \text{tab}_m \) generated during the processing of the declaration discussed in section 4.13), i.e., some set (of objects) characterizing a type. For such an element of the domain of the lookup table, the specification of the operation is selected and returned.

The conversion \( C \) will be instantiated by a suitable “coercion” of a dynamic type to a class type to be discussed later (see section 4.9). Such a conversion is known to the pre-parser from the OCL types.

The process of selecting an arbitrary, but fixed function from a specification (i.e., a relation) is handled by the \texttt{Choose}-combinator omitted here. It is defined essentially as the context-lifting of the HOL Hilbert-Operator \( \varepsilon x. P x \) that just gives one result element satisfying \( P \); if this does not exist, the Hilbert-Operator picks an arbitrary element of this type.

Thus, the semantic code for the call-by-value invocation \( c.m(a_1, \ldots, a_n) \) is given by:

\[
\text{Choose}(\texttt{invokeS } C[A] \texttt{tab}_m c a_1 \ldots a_n)
\]

where \( c \) is assumed as an object of class \( C \) and to have a subtype of \( A \) and \( C[A] \) is a casting-function that converts \( C \) objects to \( A \) objects.

4.7.3. Embedding Adaption for Shallow Embedding

Semantic combinators for embedding adaptions are related to the embedding technique itself, namely the lifting/\( \overline{\text{over}} \)-contexts. Recalling section 3.1 any function \( o \) with type \( T_1, \ldots, T_n \rightarrow T_{n+1} \) of the object-language has to be transformed to a function:

\[
I[o] \quad \text{with type } [V_{\tau}(T_1), \ldots, V_{\tau}(T_n)] \Rightarrow V_{\tau}(T_{n+1}).
\]

As an example for a binary function like the built-in operation \( \cup \) (based on union defined in subsection 4.7.2), we present its constant definition:

\[
\cup \equiv \text{lift}_2 \texttt{union} \quad \text{with type } [(\alpha :: \texttt{bot}) \texttt{Set}_{\tau}, \alpha \texttt{Set}_{\tau}] \Rightarrow \alpha \texttt{Set}_{\tau}.
\]

\( ^4 \)It is a methodological issue to avoid this situation; i.e., a notion of a consistent OCL specification is needed which can be reduced to proof obligations. Such a proof methodology is discussed in a future version of this document.
4.8. Extensible Universes in Typed Meta-Language

Summing up the intermediate results of the local theory morphisms (i.e., the layers) in the previous subsections, the definition of our running example $\mathcal{U}$ is given directly by:

$$\mathcal{U} \equiv \text{lift}_2 \left( \text{strictify} \left( \lambda X. \text{strictify} \left( \lambda Y. \text{AbsSet} \downarrow \text{RepSet} X \cup \text{RepSet} Y \right) \right) \right)$$

One easily recognizes our standard definition scheme, having AbsSet and RepSet as additional semantic combinators. During mechanical lifting of HOL theorems to OCL theorems (such as $A \cup B = B \cup A$), these operators require proofs for the invariance of the underlying quotient constructions; i.e., in this example, it must be proved that the union on representations of OCL sets will again be representations of an OCL set, (i.e., HOL sets not containing $\bot$).

4.8. Extensible Universes in Typed Meta-Language

The standard [41, appendix A] uses naive set theory and an informal notion of “model” (in the sense of mathematical model theory) over a signature given by an informal class model. It implicitly assumes one big universe for values and objects (where undefinedness elements are just there . . . ) and algebras over it without any concern of existence and consistency.

The standard cannot be formalized in this form in Isabelle, neither in an untyped set theory like Isabelle/ZF or a typed set theory underlying Isabelle/HOL; major efforts to make the foundations precise will have to be invested in both options. But since OCL is a typed language at the end, and since we wanted to have type-issues handled by the Isabelle type-checker and not inside the logic representation, it seemed for us also most natural to use a typed meta-language and typed set theory for this. In particular, in a shallow embedding, there should be a typed representation for each OCL expression, where the OCL type corresponds one-to-one to an HOL type (although not necessarily vice versa). Nevertheless, even our object universe construction should preserve the intentions of the standard, in the sense that general laws reflecting inheritance and subtyping on sets of objects can be derived. If possible, there should be no change of the users perspective of the OCL language, although its foundations have been worked out more precisely.

Recall that the OCL language specifies states and relations over them. The states are object structures, abstract representations of pieces of memory that were linked via references to each other. Formally, an object structure can be represented by a state $\tau$ mapping references or object id’s oid’s to objects, i.e., tuples of elementary values like integers or strings or object identifiers to other objects in the object structure. The type of these tuples corresponds to the type of the class they are belonging to; the components of the tuple correspond to the attributes of a class. The type of such an object structure is therefore oid $\Rightarrow \%$.

Instead of constructing a “universe of all objects” (which is either untyped or “too large” for a (simply) typed set theory, where all type sums must be finite), one could think of generating an object universe for each given class diagram. Ignoring subtyping
and inheritance for a moment, this would result in a universe $U_0 = A + B + C$ for some class diagram with the classes $A$, $B$ and $C$. Unfortunately, such a construction is not extensible: If we add a new class to an existing class diagram, say $D$, then the "obvious" construction $U_1 = A + B + C + D$ results in a different type to $U_0$, turning the two object structure types and all values constructed over them into something incomparable. This has quite dramatic consequences: such a representation rules out a modular, incremental construction of larger object systems as used in the UML standard document itself, for example. Properties, that have been proven over $U_0$ will not hold over $U_1$; practically, this means that all proof scripts will have to be rerun over an extended universe. Since even library proofs would have to be rerun (in current HOL-OCL, this takes about half an hour . . .), such an approach is unfeasible.

Our solution to the problem is to use parametric polymorphisms for families for universes $U_i$ (see Fig. 4.3 for a first overview of this idea), where the families stand for the "possible class diagram extensions". Further, we extend the scheme sketched above by assigning to Classes not directly $\alpha \colon bot$, but merely object extensions. This "incremental" object view (also used in many implementations) allows for the representation of object inheritance and leads, as we will see, to a smooth integration of object subtyping into the world of parametric polymorphism.

4.9. Encoding Object Structures

4.9.1. The Basics

Recall that object universes are the core of our notion of state, which is the building block of our notion of context $\tau$, which is again the building block of the semantic domain of HOL-OCL expressions: $\tau \Rightarrow \alpha : bot$. In this section, we focus on families
of object universes $\mathcal{U}^i$, each of which corresponding to a class diagram. Each $\mathcal{U}^i$ comprises all value types (Real, Integer, String, Boolean, ...) and an extensible class type representation induced by a class hierarchy. To each class in a given class diagram, a class type is associated which represents the set of object instances or objects. The structure of a $\mathcal{U}^i$ is to provide a family of injections and projections to and from each class type. More precisely, if we assume a class $A$, this results in:

$$mk^{(0)}_A \text{ with type } \mathcal{U}^i \Rightarrow A, \text{ and}$$

$$get^{(0)}_A \text{ with type } A \Rightarrow \mathcal{U}^i$$

allowing to inject any semantic value of OCL into some $\mathcal{U}^i$. Note, as we need also lifted version of these definitions, we will mark the different versions by different upper indexes. This in turn makes a family of states (containing “object systems”) possible:

$$\text{state with type oid } \Rightarrow \mathcal{U}$$

from which concrete values may be accessed via an oid and then be projected via $get^{(0)}_A$. On this basis, the accessor functions composing OCL path expressions can be built.

The “families” of types and states are represented by parametric polymorphism in our approach. In the big picture, this encodes a subtype type system into the typed $\lambda$-calculus with parametric polymorphism underlying HOL in “shallow style” (c.f. [48]).

The extensibility of a universe type is reflected by “holes” (polymorphic variables), that can be filled when “adding” extensions to class objects which means adding subclasses to the class hierarchy. Our construction will ensure that $\mathcal{U}^{i+1}$ (corresponding to a particular class diagram) is just a type instance of $\mathcal{U}^i$ (where $\mathcal{U}^{(i+1)}$ is constructed by adding new classes to $\mathcal{U}^i$). Thus, properties proven over object systems “living” in $\mathcal{U}^i$ remain valid in $\mathcal{U}^{i+1}$.

The universe construction introduced so far follows the interpretation of object models given in the formal semantics of the standard:

Each object is uniquely determined by its identifier and vice versa. Therefore, the actual representation of an object is not important for our purposes.

(OCL Specification [44], page paragraph A.1.2.1)

The standard does not discuss constructors\(^5\) of objects. However, it is interesting to interpret this explicitly stated bijection between oids and object instances from the point of view of an operational semantics for them. There are essentially two choices:

1. an object constructor may have sharing semantics, i.e., the tuple of attribute values is searched in the global store; if existing, its reference into the store is returned, otherwise the tuple is entered into the state and associated to a fresh object identifier which is returned, or

\(^5\text{Although, OCL includes the operation } \text{self.isNew()} \text{ which, somehow, evaluates to true if } \text{self is a “fresh” object.}\)
Chapter 4. Faithfully Representing UML/OCL

2. An object constructor may have creation semantics, i.e., generate a fresh object identifier and store it together with the tuple of attribute values in the store.

The former seems to be the underlying understanding of the original authors of Appendix A. Even so, the author of the normative part always identifies objects by a reference to it, e.g.:

\[
\text{If } \textit{self} \text{ is a reference to an object, then } \textit{self.property} \text{ is the value of the property } \textit{property} \text{ on } \textit{self}. 
\]

\[\text{(OCL Specification [41], page 15)}\]

However, the latter view has the advantage to be closer to the usual programming language semantics of object constructors. Furthermore, in this setting, the reference to an object in the store can always be reconstructed which paves the way for reference types as in Java or C++, and simple semantic definitions for \texttt{@pre} or referential equality (as discussed in definition 4.13).

Therefore, we provide, among others, one HOL-OCL configuration supporting non-referential universes and one supporting referential universes. Overall, we suggest to resolve this ambiguity in favor of the referential setting, which we also see as the default HOL-OCL configuration.

In our framework, the distinction between sharing semantics and creation semantics is reflected in two alternative universe constructions, namely the non-referential universe and the referential universe. In more detail, we can choose one of the following two types for \texttt{OclAny} and as basis for our universe construction:

\[
\begin{align*}
\alpha \text{ OclAny} &= \text{OclAny}_\text{tag} \times \alpha_\bot \quad \text{for building a non-referential universe, or} \\
\alpha \text{ OclAny} &= (\text{OclAny}_\text{tag} \times \text{oid}) \times \alpha_\bot \quad \text{for building a referential universe.}
\end{align*}
\]

Where \texttt{OclAny}_\text{type} is an abstract datatype which makes the type for \texttt{OclAny} unique within our universe construction.

The “initial” universe type \(\mathbb{U}_0^\alpha\) is defined as a sum of the built-in value types and the type \texttt{OclAny} (including all its extensions representing subtypes of \texttt{OclAny}):

\[
\mathbb{U}_0^\alpha = \alpha \text{ OclAny} + \text{Values}.
\]

Since a class can be extended in several ways, the class hierarchy has a tree-like structure; where the leaves may be “holes.” Extending a universe type by instantiating the hole \(\alpha\) is done as follows:

\[
\alpha \mapsto T \times \alpha_\bot + \beta
\]

where \(T\) is the accumulative type of the attributes of the class extension and \(\alpha_\bot\) represents the possible (future) extensions of this class. We present this idea in more formal detail in the next section.

---

\(^6\)Technically, this different HOL-OCL configuration requires another theory for the basic type definition for \texttt{OclAny} (see subsection B.3.8 “Type Definition for \texttt{OclAny}” for details).
4.9.2. The Formal Details of Encoding Object Structures

We will present the framework of our object encoding together with a small example: assume a class Node with an attribute \( i \) of type integer and two attributes \( \text{left} \) and \( \text{right} \) of type Node, and a derived class Cnode (thus, Cnode is a subtype of Node) with an attribute \( \text{color} \) of type Boolean (see Figure 4.4 and Listing 4.1 for details). The \text{oclAsType()} expressions are not necessary since the OCL standard [41, p. 20] allows their syntactic omission. In OCL [41, p. 8], an invariant of class must evaluate to \text{true} for all instances (i.e., objects of this class) at any time. Moreover, OCL is based on a three-valued logic. These two choices lead to several consequences:

1. Invariants must be valid, i.e., evaluate to \text{true} and are therefore defined.

2. From the definedness of the invariant and the strictness of \( \_ < \_ \) follows the

\[ \text{invariant: } (\text{not } \text{self.oclAsType(Node).left.oclIsUndefined()} \implies \\
(\text{not } (\text{self.oclAsType(Node).left.}
\text{oclAsType(Cnode).color = self.color}))) \]

\[ \text{and } \\
(\text{not } \text{self.oclAsType(Node).right.oclIsUndefined()} \implies \\
(\text{not } (\text{self.oclAsType(Node).right.}
\text{oclAsType(Cnode).color = self.color}))) \]

\[ \text{endpackage} \]

Listing 4.1: Modelling Directed Graphs: 

| OCL Specification |

---

A more detailed discussion of a similar example can be found in section C.1.
definedness of \texttt{self.i}.

3. From the definedness of the invariant and the strictness of the equality follows that \texttt{self.color} must be defined, if the term \texttt{self.oclAsType(Node).left} or \texttt{self.oclAsType(Node).right} is defined.

4. From the definedness of the invariant and the strictness of the equality follows that \texttt{self.oclAsType(Node).left.oclAsType(Cnode).color} must be defined, if \texttt{self.oclAsType(Node).left} is defined. The case for \texttt{right} is symmetric.

5. From \texttt{self.oclAsType(Node).left.oclAsType(Cnode).color} being defined follows the definedness of \texttt{self.oclAsType(Node).left.oclAsType(Cnode)} and thus in this case \texttt{left} must be of type \texttt{Cnode}. Moreover, this means that \texttt{left} must fulfill the invariant for \texttt{Cnode}.

4.9.3. Type Constructions

In the following we define several type sets which all are subsets of the types of the HOL type-system. This set, although denoted in usual set-notation, is a meta-theoretic construct, i.e., it cannot be formalized in HOL. We start by defining all possible types for class attributes.

\textbf{Definition 4.1 (Attribute Types)} The set of attribute types \(\mathcal{A}\) is defined inductively as follows:

1. \(\{\text{Boolean, Integer, Real, String, oid}\} \subset \mathcal{A}\), and
2. \(\{a \text{ Set, a Sequence, a Bag, a OrderedSet}\} \subset \mathcal{A}\) for all \(a \in \mathcal{A}\).

Attributes with class types, i.e., the attribute \texttt{left} of class \texttt{Node}, are encoded using the type \texttt{oid}. These object identifiers (i.e., references) will be resolved by accessor functions for a given state; an access failure will be reported by \(\bot\).

Similar to the description in the OCL standard we represent the classes as a pairs of its attribute types which we extend by an an abstract datatype for each class which makes guarantees that each class type is unique. This gives a foundation for a strongly typed universe (with regard to the object-oriented type system).

\textbf{Definition 4.2 (Tag Types)} For each class \(C\) we assign a tag type \(t \in \mathcal{I}\) which is just an abstract type used to make class types unique. The set \(\mathcal{I}\) is called the set of tag types.

For class \texttt{Node} we assign an abstract datatype \texttt{Node}, with the only element \texttt{Node.key}, and we introduce the base type for classes:

\textbf{Definition 4.3 (Base Class Types)} The set of base class types \(\mathcal{B}\) is defined as follows:

1. classes without attributes are represented by \((t \times \text{unit}) \in \mathcal{B}\), where \(t \in \mathcal{I}\) and unit is a special HOL type denoting the empty product.
2. if \(t \in \mathcal{I}\) is a tag type and \(a_i \in \mathcal{A}\) for \(i \in \{0, \ldots, n\}\) then \((t \times a_0 \times \cdots \times a_n) \in \mathcal{B}\).
Thus, the base object type of class Node is \( \text{Node} \times \text{Integer} \times \text{oid} \times \text{oid} \) and of class Cnode is \( \text{Cnode} \times \text{Boolean} \).

Without loss of generality, we assume in our object model a common supertype of all objects. In the case of OCL, this is \( \text{OclAny} \), in the case of Java this is \( \text{Object} \). This assumption is no restriction because such a common supertype can always be added to a given class structure without changing the overall semantics of the original object model.

**Definition 4.4 (OclAny)** Let \( \text{OclAny} \) be the tag of the common supertype \( \text{OclAny} \) and \( \text{oid} \) the type of the object identifiers,

1. in the non-referential setting, we define \( \alpha \text{OclAny} := (\text{OclAny} \times \alpha \perp) \).
2. in the referential setting, we define \( \alpha \text{OclAny} := ((\text{OclAny} \times \text{oid}) \times \alpha \perp) \).

Now we have all the foundations for defining the type of our family of universes formally:

**Definition 4.5 (Universe Types)** The set of all universe types \( U_{\text{ref}} \) resp. \( U_{\text{nonref}} \) (abbreviated \( U_{x} \)) is inductively defined by:

1. \( U_{x}^{0} \in U_{x} \) is the initial universe type with one type variable (hole) \( \alpha \).
2. \( U_{(a_{0}, \ldots, a_{n}, b_{1}, \ldots, b_{m})} \in U_{x}, n, m \in \mathbb{N}, i \in \{0, \ldots, n\} \) and \( c \in B \) then \( U_{(a_{0}, \ldots, a_{n}, b_{1}, \ldots, b_{m})} \left[ \alpha_{i} := \left( (c \times (\alpha_{n+1})_{1}) + \beta_{m+1} \right) \right] \in U_{x} \)
3. \( U_{(a_{0}, \ldots, a_{n}, b_{1}, \ldots, b_{m})} \in U_{x}, n, m \in \mathbb{N}, i \in \{0, \ldots, m\}, \) and \( c \in B \) then \( U_{(a_{0}, \ldots, a_{n}, b_{1}, \ldots, b_{m})} \left[ \beta_{i} := \left( (c \times (\alpha_{n+1}))_{1} + \beta_{m+1} \right) \right] \in U_{x} \)

This definition covers the introduction of “direct object extensions” by instantiating \( \alpha \)-variables.

This definition covers the introduction of “alternative object extensions” by instantiating \( \beta \)-variables.

The initial universe \( U_{x}^{0} \) represents the common supertype (i.e., \( \text{OclAny} \)) of all classes, i.e., a simple definition would be

\[ U_{x}^{0} = \alpha \text{OclAny}. \]

Alternatively one can also encode values \( \text{Values} = \text{Real} + \text{Integer} + \text{Boolean} + \text{String} \) within the initial universe type, e.g.,

\[ U_{x}^{0} = \alpha \text{OclAny} + \text{Values}. \]

For HOL-OCL we choose to represent also values within the universe which makes extensions possible that need to store values within the store. Thus we define the universes as follows:

**Definition 4.6 (Referential Universe Types)** The referential universe \( U_{x}^{0} \) is constructed using the referential definition \( \alpha \text{OclAny} := ((\text{OclAny} \times \text{oid}) \times \alpha \perp) \) in the definition of the initial universe:

\[ U_{x}^{0} = \alpha \text{OclAny} + \text{Real} + \text{Integer} + \text{Boolean} + \text{String}. \]
Chapter 4. Faithfully Representing UML/OCL

Definition 4.7 (Non-Referential Universe Types) The non-referential universe \( U_{\text{nonref}} \) is constructed using the non-referential definition \( \alpha \text{OclAny} := (\text{OclAny}_\text{tag} \times \alpha_\bot) \) in the definition of the initial universe:

\[
\mathcal{U}_0^\alpha = \alpha \text{OclAny} + \text{Real} + \text{Integer} + \text{Boolean} + \text{String}.
\]

Extending the initial universe \( \mathcal{U}_0^\alpha \), in parallel, with the classes \( \text{Node} \) and \( \text{Cnode} \) leads to the following universe type:

\[
\mathcal{U}_1^{(\alpha_\text{C}, \beta_\text{C}, \beta_\text{N})} = \left( \left( \text{Node}_t \times \text{Integer} \times \text{oid} \times \text{oid} \right) \times \left( (\text{Cnode}_t \times \text{Boolean}) \times (\alpha_\text{C}_\bot + \beta_\text{C}_\bot + \beta_\text{N} \right) \right) \text{OclAny} + \text{Values}
\]

We pick up the idea of a universe representation without values for a class with all its extensions (subtypes). We construct for each class a type that describes a class and all its subtypes. They can be seen as “paths” in the tree-like structure of universe types, collecting all attributes in Cartesian products and pruning the type sums and \( \beta \)-alternatives.

Definition 4.8 (Class Type) The set of class types \( \mathcal{C} \) is defined as follows: Let \( \mathcal{U} \) be the universe covering, among others, class \( \text{C}_n \), and let \( \text{C}_0, \ldots, \text{C}_{n-1} \) be the supertypes of \( \text{C}_n \), i.e., \( \text{C}_1 \) is inherited from \( \text{C}_{i-1} \). The class type of \( \text{C}_n \) is defined as:

1. \( \alpha_i \in \mathbb{B}, i \in \{0, \ldots, n\} \) then

\[
\mathcal{U}_n^\alpha = \left( C_0 \times \left( C_1 \times \ldots \times (C_n \times \alpha_\bot)_\bot \right) \right)_\bot \in \mathcal{C}.
\]

2. \( \mathcal{U}_\mathcal{C} \supset \mathcal{C} \), where \( \mathcal{U}_\mathcal{C} \) is the set of universe types with \( \mathcal{U}_\alpha = \mathcal{U}_n^\alpha \).

Thus in our example we construct for the class type of class \( \text{Node} \) the type

\[
(\alpha_\text{C}, \beta_\text{C}) \text{Node} = \left( \left( \text{Node}_t \times \text{Integer} \times \text{oid} \times \text{oid} \right) \times \left( (\text{Cnode}_t \times \text{Boolean}) \times (\alpha_\text{C}_\bot + \beta_\text{C}_\bot) \right) \right) \text{OclAny},
\]

and for \( \text{Cnode} \) the class type

\[
\alpha_\text{C} \text{Cnode} = \left( \left( \text{Node}_t \times \text{Integer} \times \text{oid} \times \text{oid} \right) \times \left( (\text{Cnode}_t \times \text{Boolean}) \times (\alpha_\text{C}_\bot) \right) \right) \text{OclAny}.
\]

Here, \( \alpha_\text{C} \) allows for extension with new classes by inheriting from \( \text{Cnode} \) while \( \beta_\text{C} \) allows for direct inheritance from \( \text{Node} \). Alternatively, one could omit the lifting of the base types of the supertypes in the definition of class types. This would lead to:

\[
\mathcal{U}_n^\alpha = \left( C_0 \times \left( C_1 \times \ldots \times (C_n \times \alpha_\bot) \right) \right)_\bot
\]

We see our definition as the more general one, since it allows for “partial objects” potentially relevant for other object-oriented semantics for programming languages.
4.9. Encoding Object Structures

For example Java, for which partial class objects may occur during construction. This paves the way for establishing the definedness of an object “lazy.” Furthermore, since the injections and projections are only built to define attribute accessors, partial objects are hidden from the \texttt{OCL} level.

In both cases the outermost \(\bot\) reflects that class objects may also be undefined, in particular after projection from some elements in the universe or from failing type casts. This choice has the consequence that constructor arguments may be undefined.

4.9.4. Treatment of Instances

For each class we provide injections and projects for each class. In the case of \texttt{OclAny} these definitions are quite easy, e.g., using the \texttt{Inl} constructors for type sums we can easily insert an \texttt{OclAny} object into the initial universe via

\[
\text{mk}^{(0)}_{\texttt{OclAny}} o = \text{Inl} o \quad \text{with type} \quad \alpha \rightarrow \mathcal{U}_0
\]

and the inverse function for constructing an \texttt{OclAny} object out of an universe can be defined as follows:

\[
\text{get}^{(0)}_{\texttt{OclAny}} u = \text{sumCase}(\lambda x. x, \lambda x. \varepsilon x. \text{true}, u) \quad \text{with type} \quad \mathcal{U}_0 \rightarrow \alpha \rightarrow \texttt{OclAny}.
\]

In the general case, the definitions of the injections and projections is a little bit more complex, but follows the same schema: e.g., for the injections we have to find the “right” position in the type sum and insert the given object into that position. Using the following auxiliary functions

\[
\text{base} x = \begin{cases} b & \text{if } x = ((a, b)_\downarrow), \\
\text{down} & \text{otherwise}, \end{cases}
\]

\[
\text{sup} o = \text{fst} o, \quad \text{and}
\]

\[
\text{FromL} x = \begin{cases} \text{sumCase}(\lambda x. x, \text{down}, x) & \text{if } x = (x)_\downarrow, \\
\text{down} & \text{otherwise}, \end{cases}
\]

the injections for the classes \texttt{Node} and \texttt{Cnode} can be defined as follows:

\[
\text{get}^{(0)}_{\texttt{Node}} o = \text{get}^{(0)}_{\texttt{OclAny}}(\text{sup} o, (\text{Inl} o \circ \downarrow \circ \text{base}) o) \quad \text{and}
\]

\[
\text{mk}^{(0)}_{\texttt{Cnode}} o = \text{get}^{(0)}_{\texttt{Node}}(\text{sup} o, (\text{sup} o \circ \text{base}) o, (\text{Inl} o \circ \downarrow \circ \text{base} \circ \text{base}) o)_\downarrow.
\]

And analogous for the projectors:

\[
\text{get}^{(0)}_{\texttt{Node}} u = ((\text{sup} o \circ \text{get}^{(0)}_{\texttt{OclAny}}) u, (\text{FromL} o \circ \text{base} \circ \text{get}^{(0)}_{\texttt{OclAny}}) u)_\downarrow, \quad \text{and}
\]

\[
\text{get}^{(0)}_{\texttt{Cnode}} u = ((\text{sup} o \circ \text{get}^{(0)}_{\texttt{Node}}) u, (\text{FromL} o \circ \text{base} \circ \text{base} \circ \text{get}^{(0)}_{\texttt{Node}}) u)_\downarrow.
\]
Further, we define in a similar way projectors for all class attributes, e.g., for the attributes \( i \) and \( \text{self} \) of class \( \text{Node} \) we generate:

\[
\text{self} . i^{(0)} \equiv \text{fst} \circ \text{snd} \circ \text{fst} \circ \text{snd} \circ \text{fst} \circ \text{base} \text{self}
\]
with type \((\alpha_C, \beta_C) \text{Node} \to \text{int}\), and

\[
\text{self} . \text{left}^{(0)} \equiv \text{fst} \circ \text{snd} \circ \text{snd} \circ \text{fst} \circ \text{base} \text{self}
\]
with type \((\alpha_C, \beta_C) \text{Node} \to \text{oid}\).

In a next step, we define type test functions; for universe types we need functions for testing if a universe represents a specific type, i.e., we need to test that the corresponding extensions of the universe type are defined. For \( \text{OclAny} \) we define:

\[
\text{isUniv}^{(0)}_{\text{OclAny}} u = \text{sumCase}(\lambda x. \text{true}, \lambda x. \text{false}, u)
\]
with type \( \mathcal{U}_0^{\alpha} \to \text{bool} \).

For class types we define two type tests, an exact one that tests if an object is exactly of the given \emph{dynamic type} and a more liberal one that tests if an object is of the given type or a subtype thereof. Testing the latter one, which is called \emph{kind} in the \emph{OCL} standard, is quite easy. We only have to test, that the base type of the object is defined, e.g., not equal to \( \bot \):

\[
\text{isKind}^{(0)}_{\text{OclAny}} o = \text{def} o
\]
with type \( \alpha \text{OclAny} \to \text{bool} \).

An object is exactly of a specific type, if it is of the given kind and the extension is undefined, e.g.:

\[
\text{isType}^{(0)}_{\text{OclAny}} o = \text{isKind}^{(0)}_{\text{OclAny}} \wedge \neg \text{def} \circ \text{base} o
\]
with type \( \alpha \text{OclAny} \to \text{bool} \).

The type tests for user defined classes are defined in a similar way by testing the corresponding extensions for definedness.

Finally, we define coercions, i.e., ways to type-cast classes along their subtype hierarchy. Thus we define for each class a cast to its direct subtype and to its direct supertype. We need no conversion on the universe types where the subtype relations are handled by polymorphism. Therefore we can define the type casts as simple compositions of projections and injections, e.g.:

\[
\text{Node}^{(0)}_{\text{Object}} = \text{get}^{(0)}_{\text{Object}} \circ \text{mk}^{(0)}_{\text{Node}}
\]
with type \((\alpha_1, \beta) \text{Node} \to \alpha \text{Any OclAny}\),

\[
\text{Object}^{(0)}_{\text{Node}} = \text{get}^{(0)}_{\text{Node}} \circ \text{mk}^{(0)}_{\text{Object}}
\]
with type \(\alpha \text{Any OclAny} \to (\alpha_1, \beta_1) \text{Node}\),

where \(\alpha \text{Any} = (\alpha_1, \beta) \text{Node} + \beta \text{Any}\), i.e., the free type variable \(\alpha \text{Any}\) of \(\alpha \text{Any OclAny}\) in the Universe \(\mathcal{U}_{\text{Any}}\) is instantiated to match type of the \(\mathcal{U}_{\text{Any}}^{(1)}\).

These type-casts are changing the \emph{static type} of an object, while the \emph{dynamic type} remains unchanged, i.e., one can always re-cast an object to the type it was initially created.

Note, for a universe construction without values, e.g.,

\[
\mathcal{U}_0^{\alpha} = \alpha \text{OclAny},
\]
the universe type and the class type for the common supertype are the same. In that case there is a particular strong relation between class types and universe types on the one hand and on the other there is a strong relation between the conversion functions and the injections and projections function. In more detail, see also Figure 4.5, one can understand the projections as a cast from the universe type to the given class type and the injections are inverse.

As a consequence, a theorem over class invariants (based finally on these projections, injections, casts, characteristic sets, etc.), it will remain valid even if we extend the universe via $\alpha$ and $\beta$ instantiations. In particular, this holds if we “finalize” classes by instantiating the $\alpha$’s and $\beta$’s related to this class by instantiating them by the unit type. We consider this fact as a solution to the long-standing problem of structured extensionally for object-oriented languages, enabling to represent “open world” and “closed world” assumptions as polymorphism on data universes.

4.9.5. Adaption to Higher Layers

The previous presented definitions are on the lowest layer, i.e., the introduction of new datatypes. Just as the HOL definition are adopted, over several layers, to match the OCL definitions, we have to adopt the new definitions for the UML core. For example,
after the functional and embedding adoptions, the accessor for the attributes \( i \) is:

\[
self . i^{(1)} \equiv \text{strictify} \circ \text{lift}_0 (self . i^{(0)})
\]

with type \((\alpha_C, \beta_C) \text{Node}_\tau \rightarrow \text{Integer}_\tau\), and for \textbf{left}:

\[
(sself \tau \tau). \text{left}^{(1)} \equiv \text{upCase} (\text{get}^{(0)}_{\text{Node}_\tau} , \bot , \text{fst} (sself \tau \tau . \text{left}^{(0)}))
\]

with type \((\alpha_C, \beta_C) \text{Node}_\tau \rightarrow (\alpha_C, \beta_C) \text{Node}_\tau\). The accessor for the \textbf{left} is somewhat more complicated as we have to resolve reference, i.e., note the change of types during adaption.

As already shown in these examples, we will use an upper index to distinguish the different definitions on different levels.

### 4.10. Faithful Representing \textbf{UML} Object Structures

For backing our claim that the presented encoding of object structures models faithfully the concepts that are normally described as object-oriented (e.g., in the sense of programming language like Java or Smalltalk or the \textbf{UML} standard [40]), we prove several of the required properties. These theorems are proven for each class, e.g., during loading a specific \textbf{UML} model. This is similar to other datatype packages in interactive theorem provers like Isabelle/\textbf{HOL}. Further, these theorems are also a prerequisite for a successful reasoning over object structures.

First we prove some basic properties of our injections and projects, i.e., mainly that our conversion between universe representations and object representation is lossless. Therefore we show for very class \( C \) that

\[
\text{isKind}^{(0)}_C o \implies \text{get}^{(0)}_C (\text{mk}^{(0)}_C o) = o \quad \text{and} \quad \text{isUniv}^{(0)}_C u \implies \text{mk}^{(0)}_C (\text{get}^{(0)}_C u) = u .
\]

holds. Further, we show that the injection

\[
\text{isType}^{(0)}_C o \implies \text{isUniv}^{(0)}_C (\text{mk}^{(0)}_C o)
\]

results in a correct universe type and that the projection

\[
\text{isUniv}^{(0)}_C u \implies \text{isKind}^{(0)}_C (\text{get}^{(0)}_C u)
\]

results in the right class type. We also show the important subtyping relation, i.e., that injections a class \( C \) creates a universe type which is of the kind of the parent class \( P \):

\[
\text{isKind}^{(0)}_P o \implies \text{isUniv}^{(0)}_P (\text{mk}^{(0)}_C o) ,
\]

and that every universe of type \( C \) is also of the kind of the parent class \( P \):

\[
\text{isUniv}^{(0)}_C u \implies \text{isUniv}^{(0)}_P u .
\]
Moreover, we show that our definitions support lossless re-casting, i.e.,

\[ \text{isKind}^{(0)}_{C} \text{self} \implies P^{(0)}_{[C]}(C^{(0)}_{\text{self}}) = \text{self} \]

### 4.11. A Constrained Object Store: Encoding Invariants

In this section, we bring OCL and UML together, i.e., we enrich the pure UML data model with invariants which supports the encoding of recursive object structures with class invariants. This constructions follows the spirit of the already presented scheme: we begin with a co-recursive construction of type sets (which can be seen as a kind of first level invariant) which later on define a second level construct, providing a representation that looks similar to people involved in object-oriented modeling.

#### 4.11.1. A Co-Recursive Type and Kind Set Construction

In a setting with subtyping, we need two characteristic type sets, a sloppy one, the characteristic kind set, and a fussy one, the characteristic type set. We define these sets co-recursively. As basis for our co-recursive construction, we built for each invariant a “HOL representation,” i.e., in each formula where we replace recursively the logical connectives of OCL with them from HOL by requiring the validness of the subformula. This is done using the logical judgement $\tau \models P$ which means that the OCL formula $P$ is valid (i.e., evaluates to $T$) in context $\tau$. This is explained in more detail in chapter 5.

As we want to use these invariants for a co-recursive construction we parametrize them over the current state $\tau$, the object $\text{self}$ and the type set $C$ we are constructing.

Recall our previous example (see Figure 4.4 and Listing 4.1), where the class Node describes a potentially infinite recursive object structure. The invariant of class Node constrains the attribute $i$ to values greater than 5. For this constraint, we generate

\[ \text{hol} _ \text{inv} _ \text{range} _ \text{of} _ i \tau C \text{self} \equiv \tau \models \text{self} . \text{i}^{(1)} > 5 \]

Further, we generate the two invariants expressing the fact that left and right must be either undefined or of type Node:

\[
\begin{align*}
\text{hol} _ \text{inv} _ \text{type} _ \text{left} \tau C \text{self} & \equiv \tau \models \emptyset \text{self} . \text{left}^{(1)} \\
& \lor \tau \models \text{self} . \text{left}^{(1)} \in (\lambda f. f \tau) \setminus C \\
\text{hol} _ \text{inv} _ \text{type} _ \text{right} \tau C \text{self} & \equiv \tau \models \emptyset \text{self} . \text{right}^{(1)} \\
& \lor \tau \models \text{self} . \text{right}^{(1)} \in (\lambda f. f \tau) \setminus C
\end{align*}
\]

Now we define a function for construction the kind set for Node which approximates
Chapter 4. Faithfully Representing UML/OCL

the set of possible instances of the class \texttt{Node} and its subclasses:

\[
\text{NodeKindF} :: \nu_{(\alpha_C, \beta_C, \beta_N)} \rightarrow (\alpha_C, \beta_C) \text{ Node set} \\
\Rightarrow \nu_{(\alpha_C, \beta_C, \beta_N)} \rightarrow (\alpha_C, \beta_C) \text{ Node set}
\]

\[
\text{NodeKindF} \equiv \lambda \tau. \lambda X. \{ \text{self} \mid \text{hol\_inv\_type\_left} \tau X \text{self} \\
\land \text{hol\_inv\_type\_left} \tau X \text{self} \\
\land \text{hol\_inv\_type\_right} \tau X \text{self} \}
\]

By adding the conjunct \( \tau \models \text{isType}^{(1)} \text{Node self} \), we can construct another approximation function (which has obviously the same type as \text{NodeKindF}):

\[
\text{NodeTypeF} :: \nu_{(\alpha_C, \beta_C, \beta_N)} \rightarrow (\alpha_C, \beta_C) \text{ Node set} \\
\Rightarrow \nu_{(\alpha_C, \beta_C, \beta_N)} \rightarrow (\alpha_C, \beta_C) \text{ Node set}
\]

\[
\text{NodeTypeF} \equiv \lambda \tau. \lambda X. \{ \text{self} \mid (\text{self} \in (\text{NodeKindF} \tau X)) \\
\land \text{\tau isType}^{(1)}_{\text{Node} \text{self}} \}
\]

Thus, the characteristic kind set for the class \texttt{Node} can be defined as the greatest fixedpoint over the function \text{NodeKindF}:

\[
\text{NodeKindSet} :: \nu_{(\alpha_C, \beta_C, \beta_N)} \rightarrow (\alpha_C, \beta_C) \text{ Node set} \\
\Rightarrow \nu_{(\alpha_C, \beta_C, \beta_N)} \rightarrow (\alpha_C, \beta_C) \text{ Node set}
\]

\[
\text{NodeKindSet} \equiv \lambda \tau. (\text{gfp(} \text{NodeKindF} \tau)) .
\]

For the characteristic type set we proceed analogously. Further, we prove automatically, using the monotonicity of the approximation functions, the point-wise inclusion of the kind and type sets:

\[ \text{NodeTypeSet} \subset \text{NodeKindSet} \]

This property represents semantically the subtype relation. This kind of theorem remains valid if we add further classes in a class system.

Now we relate class invariants of subtypes to class invariants of supertypes. The core of the construction for characteristic sets taking the class invariants into account is a greatest fixedpoint construction (reflecting their co-algebraic properties). We proceed by defining a new approximation for an inherited class \texttt{Cnode} on the basis of the approximation function of the superclass:

\[
\text{CnodeF} \equiv \lambda \tau. \lambda X. \{ \text{self} \mid \text{self}^{(1)}_{\text{Node}} \in (\text{NodeKindF} \tau (\lambda o. \phi^{(1)}_{\text{Node}} X) \\
\land (\varphi X \text{self}) \}
\]

where \( \varphi \) stand for the constraints specific to the subclass. Note \( \varphi \) must appropriately include \( \tau \models \text{self} . \text{right}^{(1)} (\lambda f. f \tau) X \) to make the implicit recursion in the \texttt{Cnode} invariant explicit.
4.11. A Constrained Object Store

Similar to [42] we can handle mutual-recursive datatype definitions by encoding
them into a type sum. However, we already have a suitable type sum together with
the needed injections and projections, namely our universe type with the make and
get methods for each class. The only requirement is, that a set of mutual recursive
classes must be introduced “in parallel,” i.e., as one extension of an existing universe.

Our construction for type sets and kind sets provides for an object a tight connection
between “being of a type” and “fulfilling its invariant,” i.e., the invariant for Node can
be defined semantically as follows:

\[ \text{Node\_sem\_inv } \equiv \text{self } \in \text{NodeKindSet } \text{ with type } (\alpha_C, \beta_C) \text{ Node } \rightarrow \text{Boolean}. \]

4.11.2. Functional Adoption: Invariant Awareness

Using our invariant definition we can do the last step needed to combine our OCL
embedding with our embedding of the UML core, i.e., we have to test if an object
fulfills its invariant. This can easily done, e.g., for the accessor for attribute \( i \) of class
Node we define:

\[
\text{self}.i \equiv \text{if Node\_sem\_inv } \text{then } \text{self}.i^{(1)} \text{else } \bot \text{endif}.
\]

All other, previously introduced definitions, are also adopted by wrapping them in an
if-then-else-endif statement.

4.11.3. Defining Invariants

The semantic invariant definition previously introduced is not a representation an
object-oriented modeler would expect. Therefore we define for each class a “user repre-
sentation.” This representation is based on the accessor introduced in the last section,
e.g., for Node we define:

\[
\text{Node\_defined } \equiv \partial \text{self}, \quad \text{Node\_inv\_range\_of\_i } \equiv \text{self}.i > 5, \text{ and} \quad \text{Node\_inv } \equiv \text{Node\_defined self } \land \text{Node\_inv\_range\_of\_i self}.
\]

The constraint Node\_defined ensures the definedness of self, which allows us to prove
the equivalence of both invariant representations, e.g., for Node we prove:

\[
\text{Node\_inv self } = \text{Node\_sem\_inv self}
\]

Thus, we can provide the users of HOL-OCL an invariant representation that looks
familiar to the OCL specification:

```plaintext
context Node
inv range_of_i: self.i > 5
```

Moreover, this representations allows proving of many system properties without the
need of using co-recursive induction schemes.
4.12. An Object-Oriented Type-System

Within HOL-OCL we represent types via their characteristic set. Therefore, we lift the both, the characteristic type set and characteristic kind set of each type (i.e., all basic types like Integer and also for all user defined types) introduced in subsection 4.11.1 to the OCL level. For doing so, we need to extend the OCL standard in two ways:

- The characteristic set, i.e., the set of all instances, can be infinite (e.g., for the type Integer). Therefore we use an infinite set theory for HOL-OCL. In subsection 5.6.3 we discuss the advantages of this setup in more detail.

- For supporting characteristic sets in front-end tools, we introduce two new operations:
  
  ```
  typeSetOf() -- Returns all possible instances of self, this may be an infinite set. The Type T is equal to self.
  OclType::typeSetOf(): Set(T)
  ```

  ```
  kindSetOf() -- Returns all possible instances of self and its subtypes, this may be an infinite set. The Type T is equal to self.
  OclType::kindSetOf(): Set(T)
  ```

For example, they allow one to specify, that the addition on Integers is associative:

```plaintext
Integer.typeSetOf()->forall(x,y | (not x.oclIsUndefined() and not y.oclIsUndefined()) implies x + y = y + x)
```

Our encoder will map the expression `type.typeSetOf() and type.kindSetOf()` directly to the corresponding type (kind) set.

4.13. Operation Invocations

We distinguish built-in operations (i.e., all library operations such as the logical operation ¬X, the arithmetical operation X + Y or the collection operation X ∪ Y) and user-defined operations declared in class diagrams.

While built-in operations cannot be overloaded in HOL-OCL in order not to hamper algebraic reasoning on OCL expressions, user-defined operations can be overloaded which is considered as a main feature of object-oriented programming. However, as already discussed in subsection 4.7.2, the OCL 2.0 standard [41] assumes that operation calls are statically known and makes therefore no statement on which overload resolution strategy is used; in earlier documents of the OCL semantics definition, it is suggested to users to be compliant to an arbitrary resolution strategy.
4.13. Operation Invocations

It is clearly feasible to make any static resolution strategy explicit in OCL by suitable ->oclIsType() or ->oclIsKind() tests on the arguments. However, as an experimental feature, we suggest to extend the OCL 2.0 standard by a dynamic late-binding strategy. In this section, we show how the semantics of strict operation invocations is encoded using the semantic combinator for strict invoke defined in 4.7.2.

4.13.1. The Invocation Encoding Scheme

Initial Operation Definition

In the following, we show the semantic representation scheme of invocation for user-defined operations by an example. We assume the three classes A, B and C, where C inherits from B and B inherits from A, see Figure 4.5 on page 59 for an illustration of this scenario. Further, we assume that an operation m, specified in the topmost class A, with arguments a1 : T1, ..., an : Tn, return type T, the precondition P, and postcondition Q.

In the following, we will present declaration related infrastructure for invocations as introduced in subsection 4.7.2 more detail. We reconsider the scheme of an operation specification first: an operation m, with arguments a1 :: T1, ..., an :: Tn, return type T, is declared by a pair of precondition and postcondition, thus, for each method we introduce a precondition and a postcondition:

\[
\text{pre } \text{self } a_1 \ldots a_n \text{ of type } V_\tau(A), V_\tau(T_1), \ldots, V_\tau(T_n) \Rightarrow \text{Boolean}_\tau \text{ and }
\]
\[
\text{post } \text{self } a_1 \ldots a_n \text{ result of type } V_\tau(A), V_\tau(T_1), \ldots, V_\tau(T_n), V_\tau(T) \Rightarrow \text{Boolean}_\tau.
\]

The precondition pre is a predicate function depending on the input parameter including the implicit input parameter self. The postcondition post is a predicate function depending on the input parameter and the implicit result parameter. The implicit parameters self and result are defined in OCL; they represent a common concept in object-oriented languages.

The standard defines the semantics of an operation specification by

\[
\tau_{\text{pre}} \equiv \text{pre self } a_1 \ldots a_n \land (\tau_{\text{pre}}, \tau_{\text{post}}) \equiv \text{post self } a_1 \ldots a_n \text{ result},
\]
assuming technically two different validity predicates \(\equiv\). We represent the former by the latter (replacing all accessors occurring in pre by their @pre-variant) and slightly generalize to:

\[
S_m \text{ self } a_1 \ldots a_n \text{ result } \tau \equiv \tau \equiv ((\text{pre self } a_1 \ldots a_n) \land (\text{post self } a_1 \ldots a_n \text{ result}))
\]
allowing to distinguish undefined specifications from contradictory ones. We will abbreviate this by \(S_m\) in the sequel.
Furthermore, we define the totalized operation specification:

\[ S_{\text{m-tot}}(\text{self} \ a_1 \ldots a_n \ \text{result} \ \tau \equiv \tau \models \varnothing(\text{pre self} \ a_1 \ldots a_n) \land \text{pre self} \ a_1 \ldots a_n \ \text{result} \]

\[ \text{then post self} \ a_1 \ldots a_n \ \text{result} \]

\[ \text{else } \varnothing \ \text{result} \]

\[ \text{endif} \]

which can be used alternatively in HOL-OCL and which is preferable for methodological issues, namely for proofs of consistency of OCL specifications (not discussed in this document). We will abbreviate this by \( S_{\text{m-tot}} \) in the sequel.

Now, when processing the above part of the class declaration, the following HOL declaration of a lookup table \( \text{tab}_m \) is generated:

\[
\text{tab}_m :: (A \ 	ext{set} \rightarrow [V_\tau(A), V_\tau(T_1), \ldots, V_\tau(T_n), V_\tau(T)] \Rightarrow \text{Boolean}_\tau)
\]

Here, the arrow \( \rightarrow \) stands for the type of partial maps from the HOL library. The main difference of partial maps compared to total functions \( \rightarrow \) is that partial functions have a domain operator \( \text{dom} \) with type \( \alpha \rightarrow \beta \Rightarrow \alpha \ 	ext{set} \). Additionally, the axiom:

\[
\text{tab}_m \ A \equiv \text{Some}(S_m)
\]  

or

\[
\text{tab}_{\text{m-tot}} \ A \equiv \text{Some}(S_{\text{m-tot}})
\]

where \( A \) is the characteristic set of the class \( A \). In the concluding subsection, we will discuss the conservativity issue for this type of axioms which is similar, but technically unequal to a constant definition since the table is not defined once and for all, but pointwise for a finite set of arguments.

**Inheritance of Operation**

Now we consider the case, that the class \( B \) is declared, but operation \( m \) is not redeclared, i.e., inherited from class \( A \). This leads to the axioms:

\[
\text{tab}_m \ B \equiv \text{Some}(S_m)
\]

or

\[
\text{tab}_{\text{m-tot}} \ B \equiv \text{Some}(S_{\text{m-tot}})
\]

Recall that due to our object universe construction, the type of \( B \) is an instance of the type of \( A \) even if the class \( B \) has been inserted into the system in a later stage than the compilation of \( A \), i.e., \( A \) and \( B \) live in different universes. Moreover, in the later universe, the property \( B \subset A \) holds and has been proven automatically.
4.13. Operation Invocations

Operation Overriding

Now we consider the case of an operation overriding, i.e., a new declaration, we introduce a new specification for the operation \( m \) for class \( C \) (and its subclasses) with precondition \( P' \) and postcondition \( Q' \). Again, see Figure 4.5 for details. Analogously to the overloading case, the two axioms:

\[
\text{tab}_m \ B \equiv \text{Some}(S'_m)
\]

or

\[
\text{tab}_{m-tot} \ B \equiv \text{Some}(S'_{m-tot})
\]

were constructed where \( S'_m \) and \( S'_{m-tot} \) were the operation specifications or totalized operation specifications as defined in 4.13.1.

4.13.2. Considering Conservativity

The axioms generated in the previous sections are conservative; however, they do not fit into one of the standard schemes such as constant definition (the argument of \( \text{tab}_m \) and \( \text{tab}_{m-tot} \) are a changing constant not allowed in this scheme). Rather, it is a finite family of constant definitions, where the overall type is refined from universe to universe. In the following, we characterize the syntax of this axiom scheme and sketch a proof of conservativity for it.

Definition 4.9 (Finite Constant Definition Family) A finite constant definition family is a theory extension \((\Sigma, A)\) where \( \Sigma \) is a constant declaration \( c : \tau_1 \rightarrow \tau_2 \) and \( A \) is a finite sequence of axioms of the form \( c \ D_1 \equiv E_1 \ldots c \ D_n \equiv E_n \) where \( D_i \) and \( E_i \) are closed expressions. One further optional rule, the catch-all rule, has the form:

\[
X / \notin \{D_1, \ldots, D_n\} \Rightarrow c \ X \equiv E_{n+1}
\]

Furthermore, the following conditions must be satisfied:

1. \( c \) does not occur in \( E_1 \).
2. \( c \) does only occur in \( E_j \) (in fact: in no defining expression \( E \) in a definition except that the catch-all rule is known) in the form \( c \ D_i \) with \( i < j \).
3. The type of \( c \) in axiom \( j \) must be an instance of the type of \( c \) in axiom \( i \) with \( i < j \).
4. All type variables occurring in any type of a subterm of \( E_j \) must occur in the type of \( c \ D_j \)
5. \( D_j = D_k \Rightarrow j = k \), i.e., the \( D_j \) must be pairwise disjoint.

The catch-all rule is used when a class is finalized, i.e., the class cannot be used as starting point for further inheritance.

Theorem 4.1 (Conservativity) A finite constant definition family is conservative, i.e., provided the original theory \((\Sigma, A)\) is consistent ("has models"), the theory \((\Sigma \cup \Sigma', A \cup A')\) extended by the extension \((\Sigma', A')\) is also consistent.
In case that the catch-all rule is unknown, translate the constant definition family into a family of constant definitions with constants \( c_{D_j} \). In case that the catch-all rule is known, the constant definition family can be replaced equivalently by the constant definition \( c \ X \equiv \text{if } X = D_1 \text{ then } E_1 \text{ else if } \ldots \text{ else } E_{n+1} \).

The axioms above represent a constant definition family since partial maps \( \alpha \twoheadrightarrow \beta \) are just a synonym for \( \alpha \Rightarrow \beta \) option. The pairwise disjointness follows from the full inclusion of the characteristic sets assured by construction.

4.13.3. Limits to Recursive Invocations

Let us briefly reconsider why conservativity is fundamental for HOL-OCL: an UML/OCL model can be inconsistent in the sense that there is no state that satisfies all invariants or that a method produces a state contradicting the invariants. However, no axiom generated during the compilation of the UML/OCL model into Isabelle/HOL should introduce a logical inconsistency into the meta-logic HOL, i.e., reasoning in itself should never be corrupted. A proof over the consistency of an UML/OCL model in the above sense should be valid in any case, independent of any generated axiom. Therefore, we require that method definitions in class diagrams satisfy the constraints of a finite constant definition family.

This has consequences on the form of admissible recursive operation invocations in HOL-OCL: the conditions 1 and 2 in definition 4.9 rule out a general recursive invocation of the operation to be specified. Consider again the operation \( m \) specified in class \( C \), which overrides the operation \( m \) already defined in class \( A \). Assume that the postcondition of \( m \) defined in class \( C \) is given by

\[
\text{context } C::m(a_1:T_1, \ldots, a_n:T_n):\text{Integer} \\
\text{post: result} = 1 + \text{self.m}(a_1, \ldots, a_n) \\
\text{and self.m}(a_1, \ldots, a_n).\text{isDefined()} \\
\]

Following subsection 4.7.2, the operation invocation is represented by:

\[
\text{Choose(invokeC}_A\text{[A]} tab_m self a_1 \ldots a_n) \\
\]

which conflicts with \( tab_m \) directly to condition 2 of definition 4.9.

The example also shows why this kind of syntactic restriction is necessary: from \( \text{self.m}(a_1, \ldots, a_n) = 1 + \text{self.m}(a_1, \ldots, a_n) \) and the definedness of the result one can infer \( 1 = 0 \) in OCL and then false in HOL, and then simply everything from there.

The OCL standard vaguely requires that recursions should always be terminating to rule out such kind of problems:

\[
\text{Underspecification of Recursion} \\
\]

The right-hand-side of this definition may refer to operations being defined (i.e., the definition may be recursive) as long as the recursion is not infinite.

\[
\text{(OCL Specification [41], page paragraph 7.5.2, pp.16)} \\
\]

and also:

\[
\text{Underspecification of Recursion} \\
\]
For a well-defined semantics, we need to make sure that there is no infinite recursion resulting from an expansion of the operation call. A strict solution that can be statically checked is to forbid any occurrences [...] However, allowing recursive operation calls considerably adds to the expressiveness of OCL. We therefore allow recursive invocations as long as the recursion is finite. Unfortunately, this property is generally undecidable. (OCL Specification [41], page A-31)

Unfortunately, in a proof-environment we have to be sufficiently more specific than this. Furthermore, HOL-OCL is designed to live with the open-world assumption, i.e., with the potential extensibility of object universes, as a default; further restrictions such as finalizations of class diagrams or a self-restriction to Liskov’s Principle may be added on top, but the system in itself does not require them. This has the consequence that even in the following version:

```context C::m(a1:T1,...,an:Tn):Integer
  post: result = if a1.p()
    then 1 + self.m(a1.q(),...,an)
    else 0
  endif
```

the termination for the invocation `self.m(a1.q(),...,an)` is fundamentally unknown (even if `p` and `q` are known and terminating): a potential overriding may destroy the termination of this recursive scheme.

In form of a pre-translation process, operation specifications with a limited form of recursive invocations can be converted into the format that satisfies the constraints of a finite constant definition family. These limited forms are assumed to occur in the postcondition Q’ and can be listed as follows:

- calls to superclass operations, i.e., `(self.asType(A)).m(x1,...,xn)`, or
- direct recursive well-founded invocations, i.e., `(self.asType(C)).m(x1,...,xn).

The former invocation can be translated simply into `Choose(S_m self x1 ... xn)` or `Choose(S_{m,tot} self x1 ... xn)`. The latter is based on the theory of well-founded orders and the well-founded recursor `wfrec` in Isabelle/HOL and so to speak an application of the standard HOL methodology to OCL. We assume the following version:

```context C::m(a1:T1,...,an:Tn):Integer
  post: result = if a1.p()
    then 1 + (self.asType(C)).m(a1.q(),...,an)
    else 0
  endif
```
The idea is to abstract away the occurrence of the recursive call:

\[
\text{post}_{\text{rec}} f \ self \ a_1 \ldots \ a_n \ result \equiv \text{result} \triangleq \begin{cases} 
& \text{if } a_1.p() \\
& \text{then } 1 + (f\ (a_1.q()) \ldots \ a_n) \\
& \text{else } 0 \\
& \text{endif}
\end{cases}
\]

and to build the operation specification notions on top of it: Let

\[
S_{\text{m-rec}} f \ self \ a_1 \ldots \ a_n \ result \ \tau \equiv \tau \models (\text{pre self } a_1 \ldots \ a_n \\
\land \text{post}_{\text{rec}} f \ self \ a_1 \ldots \ a_n \ result)
\]

and

\[
S_{\text{m-tot-rec}} f \ self \ a_1 \ldots \ a_n \ result \ \tau \equiv \tau \models \begin{cases} 
& \text{if } \partial(\text{pre self } a_1 \ldots \ a_n) \\
& \land \text{pre self } a_1 \ldots \ a_n \\
& \text{then post}_{\text{rec}} f \ self \ a_1 \ldots \ a_n \ result \\
& \text{else } \not\partial \ \text{result} \\
& \text{endif}
\end{cases}
\]

then, for direct recursive calls, the operation specifications are defined as 

\[
S_m \equiv \text{Wfrec} \ M \ S_{\text{m-rec}} \ \text{or} \ S_{\text{m-tot}} \equiv \text{Wfrec} \ M \ S_{\text{m-tot-rec}}
\]

where \(\text{Wfrec}\) is the context-lifted version of \(\text{wfrec}\) and \(M\) is an ordering (such as \(x < y\) restricted on positive integers). If this ordering is well-founded, from this definition, the original user-specified post-condition follows from this definition. Since the critical call is now incorporated into the well-founded recursion construction, the definition is conservative; and provided the user gives a suitable ordering, it can be shown that the desired specification follows from the constructed definitions.

Alternatively, in case of a finalized class, an invocation

\[
\text{invoke}S \ C \ \text{tab}_m \ self \ a_1 \ldots \ a_n
\]

can be replaced by the case-switch:

\[
\text{if } \text{self} \rightarrow \text{IsType}(A) \text{ then } \text{S else if } \ldots \text{ else } \text{S'}
\]

Summing up, conservativity implies that only limited forms of recursive invocations are admissible in HOL-OCL. In an open world (no class finalization so far), only operation calls can be treated, in a closed world (the class hierarchy has been finalized), an invocation can be expanded to a case-switch considering the dynamic type of \(\text{self}\) over calls.

### 4.14. Comparing the Referential and Non-Referential Universe

Historically, object-oriented systems are equipped with a variety of different “equalities” [30]. Answering the question if two objects are equal is not so obvious: e.g., are
two objects equal only if their object identifier is equal (are they the same object?) or are two objects equal if their values are equal, or are they equal if they are observably equivalent with respect to the accessor functions? In the following we will use this problem as a linchpin to discuss the differences between a referential universe and a non-referential universe.

Whereas in traditional specification formalism the equality is defined over values, the most basic equality over objects is the reference quality or identity equality which is also the kind of equality that is usually provided as a default, i.e., “built-in,” equality in object-oriented languages. Thus, there is usually a fundamental difference between values and objects.

**Definition 4.10 (Value Types)** The set of *value types* $\mathcal{V}$ is defined inductively as follows:
1. $\{\text{Boolean, Integer, Real, String}\} \subset \mathcal{V}$, and
2. $\{v\text{ Set, } v\text{ Sequence, } v\text{ Bag, } v\text{ OrderedSet}\} \subset \mathcal{V}$ for all $v \in \mathcal{V}$.

**Definition 4.11 (Values)** An instance of a value type, e.g., $x :: v$ with $v \in \mathcal{V}$ is called *value*.

Normally one expects that an equality is an equivalence relation.

**Definition 4.12 (Equivalence Relation)** An *equivalence relation* $\sim$ is a binary relation over a set $S$ for which the following properties hold:
- Reflexivity: $a \sim a$, for all $a \in S$.
- Symmetry: $a \sim b$ if and only if $b \sim a$, for all $a, b \in S$.
- Transitivity: if $a \sim b$ and $b \sim c$ then $a \sim c$, for all $a, b, c \in S$.

Now we introduce in an abstract way the basic qualities of object-oriented systems, we ignore undefinedness in these definitions. In a second step, we will show, that the handling of undefinedness is orthogonal and can be combined with any of the following equalities.

Most object-oriented languages have the concepts of references or object identifiers where a reference uniquely identifies an object. Thus it seems a natural choice to use these references for defining an equality, namely the reference quality.

**Definition 4.13 (Reference Equality)** The *reference equality* or *identity equality* is defined as follows:
1. Two values are reference equal, if they are of the same type and represent the same value.
2. Two objects are reference equal, if their object identifiers (references) are equal.

Thus, the reference equality tests if two objects represent in fact the same object in a store. If we want to test, if two objects are identical (i.e., they represent the same value) we have two options: a shallow and a deep one:

**Definition 4.14 (Shallow Value Equality)** The *shallow value equality* or just *value equality* is defined as follows:
Chapter 4. Faithfully Representing UML/OCL

1. Two values are shallow value equal, if they are of the same type and represent the same value.
2. Two objects are shallow value equal, if they are of the same type and all attributes with value types are pairwise shallow value equal.

This definition is not recursive, hence the name shallow equality. The main idea behind the shallow equality is to compare two singular objects as values. In contrast to this, we can define the deep value equality for comparing the values of two object structures.

**Definition 4.15 (Deep Value Equality)** The *deep value equality* is defined as follows:

1. Two values are deep value equal, if they are of the same type and represent the same value.
2. Two objects are deep equal, if they are of the same type and
   
a) all attributes with value types are pairwise deep value equal.
b) all attributes with type *oid* (object type) are pairwise deep value equal if they objects represented by the object identifiers are deep value equal.

Summarizing, we have already three different equalities:

1. the reference equality which checks if two objects are in fact the same object,
2. the shallow value equality which compares the values of the attributes on the first level, and
3. the deep value equality which compares recursively the object structure comparing the equality of the corresponding parts.

Assuming a setting, where all values and oid are defined, i.e., the classical two-valued view, each of them is an equivalence relation on objects. It seems to be obvious that that in a universe without undefinedness (definition 4.15) refines (definition 4.14) and definition 4.14 refines definition 4.13. Thus, two objects, that are reference equal are also shallow equal and deep equal. But if we have undefined values and object it is not clear how these equalities relate to each other. First, in a world with undefinedness we can apply the concept of strictness to equivalence relations:

Taking undefinedness into account, e.g., values and references can be undefined, the setting gets more complicated. First we generalize the concept of equivalence relations by introducing equivalence operators.

**Definition 4.16 (Equivalence Operators)** An equality operator ∼ is a binary operator that satisfies the following properties for a state τ and a context-passing P:

- **Quasi-reflexivity:**

  \[
  \tau \models x \Rightarrow \tau \models x \sim x
  \]

- **Quasi-symmetry:**

  \[
  \tau \models x \sim y \quad \tau \models y \sim x
  \]

72

- Quasi-transitivity:
  \[
  \tau \models \partial x \quad \tau \models \partial y \quad \tau \models \partial z \quad \tau \models x \sim y \quad \tau \models y \sim z
  \]
  \[
  \tau \models x \sim z
  \]

- Quasi-substitutivity:
  \[
  \tau \models \partial x \quad \tau \models \partial y \quad \tau \models x \sim y \quad \tau \models P(x)
  \]
  \[
  \tau \models P(y)
  \]

This definition uses the logical judgement \( \tau \models P \) which means that the OCL formula \( P \) is valid (i.e., evaluates to \( T \)) in context \( \tau \); this judgement is defined and discussed at length in chapter 5.

In the following, we characterize certain classes of three-valued equality operators.

**Definition 4.17 (Strong Equality)** An equality operator is called a strong equality if it satisfies the property: \( (\bot \triangleq \bot) = T \).

This equality operation is quasi-reflexive, quasi-symmetric, quasi-transitive and quasi-substitutive even for undefined values which explains its outstanding importance in deduction.

Applying the concept of strictness to an equality operator results in the following definition:

**Definition 4.18 (Strict Equality)** An equality operator is called strict equality if it evaluates to undefined whenever one of its arguments is undefined, i.e., if the following properties hold:

\[
(o \triangleq \bot) = \bot, \quad (\bot \triangleq o) = \bot, \quad \text{and} \quad (\bot \triangleq \bot) = \bot.
\]

Strictly speaking, these last definitions are merely algebraic characterizations and not definitions. These operation symbols were characterized by some properties, but they are obviously not defined up to isomorphism. In our context, two interpretations of the equalities into the semantic domain of universes are of particular importance: when comparing objects, we can define the equality operation via HOL-equality in the object representation in the referential or the non-referential universe (when comparing values, we compare them via HOL-equality anyway; see subsection B.4.3 for technical details).

Thus, the concept of strictness is orthogonal to the semantics of equality if the arguments are defined. Thus we can combine this with all of our previous equality variants. In principle this results in six different equalities for the object logic (see also Table 4.1). Albeit, with respect to the interpretation of the equality operators (assuming only objects in the range of the state whose reference field just contains the reference to the object in the store), strong and strict equality operators both coincide with referential equality since we have a bijective mapping between the values of an object and the object identifier. As a consequence, the advantage of a non-referential universe is its potential for efficient implementations of strict and strong equality for an executable fragment of OCL.

73
Chapter 4. Faithfully Representing UML/OCL

<table>
<thead>
<tr>
<th>Referential Equality</th>
<th>Strict</th>
<th>Strong (Non-Strict)</th>
</tr>
</thead>
<tbody>
<tr>
<td>Shallow Value Equality</td>
<td>$o_1 \triangleq o_2$</td>
<td>$o_1 \trianglelefteq o_2$</td>
</tr>
<tr>
<td>Deep Value Equality</td>
<td>$o_1 \triangleleft o_2$</td>
<td>$o_1 \triangleleft \triangleleft o_2$</td>
</tr>
</tbody>
</table>

Table 4.1.: Equalities in an Object-oriented Setting

<table>
<thead>
<tr>
<th>Equality</th>
<th>Strict</th>
<th>Strong (Non-Strict)</th>
</tr>
</thead>
<tbody>
<tr>
<td>$\tau \models o_1 \triangleq o_2$</td>
<td>✓</td>
<td>✓</td>
</tr>
<tr>
<td>$\tau \models o_1 \trianglelefteq o_2$</td>
<td>✓</td>
<td>✓</td>
</tr>
<tr>
<td>$\tau \models o_1 \triangleleft o_2$</td>
<td>✓</td>
<td>✓</td>
</tr>
<tr>
<td>$\tau \models o_1 \triangleleft \triangleleft o_2$</td>
<td>✓</td>
<td>✓</td>
</tr>
</tbody>
</table>

Table 4.2.: Comparing Equalities (non-referential universe)

The OCL standard defines equality as the strict equality over values [21, Sec. A.2.2], and since objects are values, but object identifiers are not distinguished from object values [21, Definition A.10] we choose the strict reference quality _ as the default OCL equality. In addition, we provide both referential equalities and for each class both shallow value equalities. The deep value equalities can be defined by the user within HOL-OCL at the HOL-level, if needed.

Note, besides smashed sets and unsmashed sets, we could also define smashed objects (i.e., an undefined attribute would result in an undefined object) and unsmashed objects. This would also influence the relations between the different equalities, i.e., compare Table 4.2 with Table 4.3.

Finally, we report on some strange effect of the self.OclIsNew()-operator in the non-referential universe which can be seen as a non-standard behavior: if we assume a class A with an attribute i and if we assume the natural specification of the creation-method:

```plaintext
context A::create(val:Integer):Boolean
post: self.OclIsNew() and self.i = val
```

This operation specification is not satisfiable if the objects has already been created and exists already in the store—this makes the whole concept of self.OclIsNew()-operator quite useless in the non-referential universe interpretation of the OCL 2.0 standard.

74
4.15. Specifying Frame Properties

The OCL does not guarantee that an operation only modifies the path-expressions mentioned in the postcondition, i.e., it allows arbitrary relations from pre states to post states. For most applications this is too general: there must be a way to express that parts of the state do not change during a system transition, i.e., to specify the frame properties of system transition. Thus we suggest to extend OCL one to specify the frame properties explicitly:

\[
\text{->modifiedOnly}() \quad \text{where } S \text{ is a set of object identifiers (i.e., a set of } \text{OclAny objects). This also allows recursive operations to collect the set of objects that are potentially changed within a recursive data structure. Obviously, similar to } \text{@pre} \text{ the use of } ->\text{modifiedOnly}() \text{ is restricted to postconditions.}
\]

The semantics for \( ->\text{modifiedOnly}() \) can be defined in HOL-OCL quite easily. First we define a predicate oidOf for accessing the object identifier of an object:

\[
\text{oidOf} \quad \tau \quad X \equiv \{ x. (\tau \quad x) = \text{Some(mkOclAny} \quad X) \}\]

which allows a uniform definition of \( ->\text{modifiedOnly}() \) for the referential universe and the non-referential universe:

\[
X->\text{modifiedOnly}() \equiv \lambda (\tau, \tau'). \\\ \forall i \notin (\bigcup (\text{oidOf} \quad \tau) \setminus \text{RepSet}(X(\tau, \tau'))). \tau \quad i = \tau' \quad i_j.
\]

Thus requiring \( \text{Set}\{\}->\text{modifiedOnly}() \) in a postcondition of an operation allows for stating explicitly that an operation is a query in the sense of the OCL standard, i.e., the isQuery property is true. Further, constructs like

\[
\text{Set}\{x,y\}->\text{modifiedOnly}() \text{ and Set}\{z\}->\text{modifiedOnly}()
\]

\[
\begin{array}{cccccccc}
\tau \models o_1 \equiv o_2 & \checkmark & \checkmark & \checkmark & \checkmark & \checkmark & \checkmark & \checkmark \\
\tau \models o_1 \wedge o_2 & \checkmark & \checkmark & \checkmark & \checkmark & \checkmark & \checkmark & \checkmark \\
\tau \models o_1 \wedge o_2 & \checkmark & \checkmark & \checkmark & \checkmark & \checkmark & \checkmark & \checkmark \\
\tau \models o_1 \wedge o_2 & \checkmark & \checkmark & \checkmark & \checkmark & \checkmark & \checkmark & \checkmark \\
\tau \models o_1 \wedge o_2 & \checkmark & \checkmark & \checkmark & \checkmark & \checkmark & \checkmark & \checkmark \\
\tau \models o_1 \wedge o_2 & \checkmark & \checkmark & \checkmark & \checkmark & \checkmark & \checkmark & \checkmark \\
\tau \models o_1 \wedge o_2 & \checkmark & \checkmark & \checkmark & \checkmark & \checkmark & \checkmark & \checkmark \\
\tau \models o_1 \wedge o_2 & \checkmark & \checkmark & \checkmark & \checkmark & \checkmark & \checkmark & \checkmark \\
\end{array}
\]

4.15. Specifying Frame Properties

Table 4.3.: Comparing Equalities (referential universe)
is semantically the intersection of the argument lists, and thus equivalent to the term \( \text{Set\{}\rightarrow \text{modifiedOnly}\(\), i.e., “nothing is changed.”}
Chapter 5.

Calculi

One contribution of HOL-OCL is the development of proof calculi for OCL. Having a conservative embedding of OCL in HOL, one might ask why this is necessary. Of course, one could always unfold the definitions, thus converting an OCL formula into a HOL expression, and try to prove the latter. However, this is not an effective proof technique, analogously to the fact that the Turing machine is not necessarily an effective means to simulate an up-to-date microprocessor. Abstract, human-readable statements on OCL formulae are both important for the human user performing interactive proof as well as automated decision- or normalization procedures for certain fragments of OCL.

To this means, appropriate logical statements have to be conceived that were linked via derived inference rules (i.e., formally proven in Isabelle/HOL) to each other, such that their use in a programmed tactic process finally yields the necessary automated support for effective interactive proof.

As the OCL standard does not provide any calculi we had to develop them based on previous works [28, 20] for three valued logics and own considerations and experiences with previous interactive proof environments [11]. We extended these works to support UML-like data-models and we also developed a tool-supported methodology. One central point of the OCL semantics presented in the in the OCL standard [41, appendix A] is the notion of local validity judgements:

\[(\tau_{\text{pre}}, \tau_{\text{post}}) \models Q \iff I[Q](\tau_{\text{pre}}, \tau_{\text{post}}) = \text{true}\]

(OCL Specification [41], page A.33)

We use this as a basis of our calculi and generalize this form of judgements to:

\[(\tau \models P) \equiv (I[P](\tau) = \text{true}),\]  
(5.1a)

\[(\tau \models_F P) \equiv (I[P](\tau) = \text{false}), \text{ and}\]  
(5.1b)

\[(\tau \models_{\bot} P) \equiv (I[P](\tau) = \bot).\]  
(5.1c)

As a shorthand for all three variants, we will write \(\tau \models_x P\) for \(x \in \{\bot, F, T\}\). When omitting the index, we assume T as default (which will turn out to be complete). Moreover, we generalize local validity judgements to a notion of global or universal judgments:

\[(\forall_x P) \equiv (\forall \tau. \tau \models_x P)\]  
(5.2)
In principle, reasoning over OCL formulae can either be based on a decomposition strategy of judgements or on exploiting equivalences between formulae or judgements over them; in the latter case, the transport of knowledge of contexts is a major technical issue in reasoning over OCL\textsuperscript{1} which turns out to be even more important (i.e., more fundamental) than reasoning over definedness of subterms.

In the following sections, we will explore the potential of deduction via equivalences, in section 5.9 we will discuss a tableaux calculus that can process proof states built from judgements on a completely different way.

5.1. A Theory of Basic Judgement

Following the previous definitions, we can easily check the following link between judgements and equalities:

\[(\tau \models A) = (A \tau = \text{true}) = (A \tau = T\tau)\]

(5.3)

The following analogous equations reveal that only one kind of judgements is needed, as canonical form we take the validity judgement:

\[(X = \bot) = (\models \bot X)\]

(5.4a)

\[(X = F) = (\models \neg X)\]

(5.4b)

\[(X = T) = (\models X)\]

(5.4c)

\[(X \tau = \bot \tau) = (\tau \models \bot X)\]

(5.4d)

\[(X \tau = F \tau) = (\tau \models \neg X)\]

(5.4e)

\[(X \tau = T \tau) = (\tau \models X)\]

(5.4f)

Applied from right to left, these theorems reveal also the character of judgements as rewrite-rules that can be used by automatic rewriting procedures. From these equalities, the base cases for judgements follow directly:

\[\neg(\models \bot)\]

(5.5a)

\[\neg(\models F)\]

(5.5b)

\[\models T\]

(5.5c)

\[\neg(\tau \models \bot)\]

(5.5d)

\[\neg(\tau \models F)\]

(5.5e)

\[\tau \models T\]

(5.5f)

A last fundamental fact of judgements is related to the three-valuedness of OCL, i.e., non quatrium datur:

\[(\tau \models A) \lor (\tau \models \neg A) \lor (\tau \models \emptyset A)\]

(5.6)

With this fact, a defined formula can be converted into formulae which are true or which are false; this gives rise for six corresponding case-split lemmas.

\textsuperscript{1}there is a notable similarity to labelled deduction systems [16, 51].
5.2. Basic Equivalences and Congruences

In principle, we distinguish four equivalences over OCL formulae; one of them is already a congruence.

**UC**: Universal (Formula) Congruence. This congruence requires that two formulae $A$ and $B$, both of type $\text{Boolean}_\tau$, agree in all contexts $\tau$ and in all three truth values of type $\text{Boolean}_\tau$. They have the form

$$A = B \quad \text{or} \quad \frac{A_1 = B_1 \quad \cdots \quad A_n = B_n}{A_{n+1} = B_{n+1}}.$$

**LE**: Local (Formula) Equivalence. This equivalence requires that two formulae agree on all three truth values of $\text{Boolean}_\tau$ in a specific context $\tau$: They have the form

$$A_\tau = B_\tau \quad \text{or} \quad \frac{H_1 \quad \cdots \quad H_n}{A_{n+1}_\tau = B_{n+1}_\tau}.$$

Premises can have the form $A_\tau = B_\tau$ or instances of this scheme such as $\tau \vDash_x A$.

**UJE**: Universal Judgement Equivalence. This equivalence requires that for all contexts $\tau$ agree on one value $X$ from $\text{Boolean}$. They have the form $\vDash_x A = \vDash_x A$ or horn-clauses over them.

**LJE**: Local Judgement Equivalence. This equivalence requires that two formulae agree on a specific truth value $X$ of type $\text{Boolean}_\tau$ in a specific context $\tau$: $\tau \vDash_x A = \tau \vDash_x B$ or horn-clauses over them.

Since all three possible kinds of judgements $\vDash_x A$ (universal) respectively $\tau \vDash_x A$ (local) called validity, invalidity and undefinedness, can be converted into each other, we can choose just one of them, validity, as representative. We will abbreviate $\vDash_A$ or $\tau \vDash_A$ by $\vDash A$ or $\tau \vDash A$ resp.

The **UJE**-format is only of notational interest: it is not possible to build a complete calculus using only **UJE**-rules. While the rule:

$$\vDash \partial A \quad \vDash \partial B \quad \frac{\vDash (A \land B)}{(\vDash A \land (\vDash B))} \quad (5.7)$$

is in fact valid (due to distribution of universal quantification over $\land$), but even an analogue version for the OCL disjunction, however, does not hold.

The **LE**-format, however, is flexible enough to build both practical relevant conversion calculi as well as (relative) complete calculi. Consider:

$$\tau \vDash \partial A \quad \tau \vDash \partial B \quad \frac{(\tau \vDash A \land B)}{((\tau \vDash A) \land (\tau \vDash B))} \quad (5.7)$$
as a propositional equivalence or

\[(\tau \vDash \forall x \in S. A \land B) = (\forall x. (\tau \vDash x \in S) \longrightarrow (\tau \vDash A)) \land (\forall x. (\tau \vDash x \in S) \longrightarrow (\tau \vDash B))\]  

(5.8)

as a predicative equivalence.

Since judgements are propositional and LJE’s can be decomposed into implications from left-to-right and from right-to-left, there is another line to automated reasoning over OCL-formulae: they can be turned into a tableau calculus (the local judgement tableaux calculus, LTC, see section 5.9).

Unfortunately, there is a trade-off between completeness of the various calculi based on these equivalences and deductive efficiency. UC is the only congruence that can be directly processed by Isabelle’s simplifier; normalizations in UC can be computed relatively efficiently. While UC comprises several thousands of rules (among them, the strictness and computational rules of operators) it does not form a complete calculus for principle reasons: Some properties in OCL are inherently context dependent, in particular when referring to paths. Others are difficult to formalize in the form of the straight-jacket of a universal congruence. On the other end of the spectrum, since local judgements are simply propositions, they are extremely flexible. When extending LJE’s to equivalences over propositional (predicative) formulae, it is not difficult at all to convert them into a fairly abstract but still provably complete calculus with analytic rules (rules whose application yield to an equivalent transformation of the proofstate. Their use is “safe” in the sense that no logical content is lost.)

With respect to LTC, it is well-known that the complexity for tableaux-based reasoning in Kleene-Logic is unfavorably high [19]. However, the logic is only one little fragment of the overall problem of building decision procedures in OCL: Most operations are strict, and from the data-invariants, definedness of many literals can be inferred, such that large fragments of the language are in fact two-valued. Furthermore, we do not only have the logic but also a rich datatype theory with collection types which give the overall language a flavour in its own. As consequence, a good combination of all these types of calculus is a prerequisite for automated reasoning for practically relevant reasoning work in OCL.

5.3. Reasoning on Context-Passing Predicate cp

In the following, we discuss the first side-calculus, the reasoning over context-passingness or cp. Revising the definition

\[\text{cp} :: (V_\tau'(\alpha) \Rightarrow V_\tau'(\beta)) \Rightarrow \text{bool}\]  

(5.9)

\[\text{cp}(P) \equiv (\exists f. \forall X_\tau. P X_\tau = f(X_\tau)\tau)\]  

(5.10)

presented in section 4.3, one might wonder why this quite arcane definition is so pivotal for reasoning in OCL. An answer can be drawn from the following lemma:

\[\frac{A \tau = B \tau \quad \text{cp} P}{P \ A \tau = P \ B \tau}\]  

(5.10)
5.4. Reasoning on Undefinedness and Definedness

Definedness and undefinedness are indeed opposite concepts in [OCL], i.e., they satisfy the “classical” rule for all \( X \) in all types:

\[
\forall \tau \in \mathfrak{OCL} \quad \tau \vdash \emptyset X \lor \tau \vdash \partial X \tag{5.16}
\]

Table 5.1.: The Context Passing Side-Calculus (Excerpt)

<table>
<thead>
<tr>
<th>Rule</th>
<th>Equation</th>
</tr>
</thead>
<tbody>
<tr>
<td>( \text{cp}(\lambda X. c) )</td>
<td>(5.11)</td>
</tr>
<tr>
<td>( \text{cp}(\lambda X. X) )</td>
<td>(5.12)</td>
</tr>
<tr>
<td>( \text{cp}(\text{lift}_0 c) )</td>
<td>(5.13)</td>
</tr>
<tr>
<td>( \text{cp}P )</td>
<td>(5.14)</td>
</tr>
<tr>
<td>( \text{cp}(\lambda X. \text{lift}_1 f(\text{PX})) )</td>
<td>(5.14)</td>
</tr>
<tr>
<td>( \text{cp}P \quad \text{cp}P' )</td>
<td>(5.15)</td>
</tr>
<tr>
<td>( \text{cp}(\lambda X. \text{lift}_2 f(\text{PX})(\text{P}' X)) )</td>
<td>(5.15)</td>
</tr>
</tbody>
</table>

In other words, any local equivalence (LE) \( A \tau = B \tau \) is in fact a congruence for all terms \( P X \) that are \( \text{cp} \). As a consequence, \( \text{cp} \) is a pre-requisite for replacing a term through another in some (context-passing) term \( P X \); \( P \) can also be interpreted as the “surrounding term” marked by the “position” \( X \). Since global equivalence is semantically closely connected to strong equality, this means that all sorts of term-rewriting in [OCL] will be constrained to “adequate surrounding terms,” i.e., those that are \( \text{cp} \). Context-passingness is a tribute to the fact that [OCL] is a context dependent typed logic; it can be seen as an invariant of semantic functions representing the [OCL] operations.

The inference rules for establishing \( \text{cp} \) are contained in [Table 5.1] and follow an inductive scheme over the structure of [OCL] expressions: The base-cases are straightforward, i.e., constant expressions or identities are context passing. The latter three lemmas contain the step-cases and work for all operators that had been defined via the context lifting combinators.

Since we presented all [OCL] operations “operator style” with these combinators, these generic step-cases pave the way for the automatic generation of one \( \text{cp} \)-rule with a uniquely defined pattern for each [OCL] operator. Thus, for all expressions built entirely from [OCL] operators—as those formulae generated from the “encoder” when loading class-diagram and an associated [OCL] specification, the derivation of \( \text{cp} P \) formulae are done fully automatic by backward chaining (using both by the Isabelle simplifier as well as the Isabelle classical reasoner).
Chapter 5. Calculi

This gives, of course, rise for case-split techniques that can be applied automatically in \texttt{UJE} or \texttt{LJE} calculi. However, since \texttt{OCL} is biased towards strict operations, the use of undefinedness in deduction is easier than its counterpart.

Undefinedness can be propagated throughout a proof state via forward reasoning and exploited via rewriting. The forward reasoning part is covered by rules like:

\[
\frac{\tau \models \emptyset X \quad \text{cp} \ P}{P \ X \ \tau = P \bot \ \tau}
\]  

and several variants used for technical purposes. Having replaced some term $X$ by $\bot$, strictness rules like $f \bot Y$ or $f Y \bot$ can reduce the size of subgoals drastically.

We now focus on the far more involved treatment of definedness. The core of reasoning over definedness is in fact representable in an \texttt{UC} calculus. It is summarized in \textbf{Equation 5.2(a)}. This rule set contains also a class of rules for “strict standard operations $f$.” With this set of operations, we refer to operations that had been defined by constant definitions of the form:

\[
f \equiv \text{lift}_2(\text{strictify}(\lambda x. \text{strictify}(\lambda y. \text{``}x\text{''}y)))
\]  

As in the case of the generation of \texttt{cp}-inference rules, we exploit the combinator-style definitions of the standard operators here and generate this type of rules in pre-computation steps once and for all.

The power and the main drawback of this type of \texttt{UC}-based calculus stems from the rules listed in \textbf{Equation 5.2(b)}. They result in the generation of numerous case splits, which may be inappropriate in most deductions: if we know, that all variables in a subgoal are defined (and this is an important special case that we can achieve by initial case-splits done once and for all), simple conditional rules leading to direct backward-chaining are sufficient.

By the way, this strong definedness calculus can be extended to quantifiers reads as follows\footnote{The analogous rule for existential quantification is omitted here.}:

\[
\text{cp} \ P \\
\partial(\forall x \in S \cdot Px) = \partial S \land ((\exists x \in S \cdot (\neg Px)) \lor (\forall x \in S \cdot (\partial Px)))
\]  

To overcome this drawback, we derived the following alternative rule-set listed in \textbf{Equation 5.2(c)}. It reduces the burden of applicability to the question, if the definedness of a term can be derived. We will discuss in section 5.10 how this can be assured by prior normal form computations. By the way, the $\tau \models \emptyset x$-part in the last rule is strictly speaking redundant (as we will see when we discuss the Set-theory in more detail), but facilitates the establishment of $\tau \models \partial Px$ since this additional assumption will be used if $x$ occurs in (the instance of) $Px$.\footnote{\texttt{The analogou}}
5.4. Reasoning on Undefinedness and Definedness

(a) Core Definedness Rules

\[ \partial F = T \quad \partial T = T \] (5.17)
\[ \partial \bot = F \quad \partial \partial X = T \] (5.18)
\[ \partial(\neg X) = \partial X \quad \partial(\neg X \land \partial X) = T \] (5.19)
\[ \partial(X \land \partial X) = T \quad \partial(\partial X \land \neg X) = T \] (5.20)
\[ \partial(X \land \neg X) = T \] (5.21)
\[ \partial X = Y \land \partial X \land \partial Y \] (5.22)
\[ \partial f X = \partial X \quad \partial f X Y = \partial X \land \partial Y \] (5.23)

for all strict standard operations \( f \)

(b) Strong Definedness Rules

\[ \partial(\text{if } X \text{ then } Y \text{ else } Z \text{ endif}) = \partial X \land (X \land \partial Y \lor \neg X \land \partial Z) \] (5.24)
\[ \partial(X \land Y) = \partial X \land \partial Y \lor \partial X \land \neg X \lor \partial Y \land \neg Y \] (5.25)
\[ \partial(X \rightarrow Y) = \partial X \land \partial Y \lor \partial X \land \neg X \lor \partial Y \land \neg Y \land \neg Y \] (5.26)
\[ \partial(X \lor Y) = \partial X \land \partial Y \lor \partial X \land X \lor \partial Y \land Y \] (5.27)

(c) Weak Definedness Rules

\[ \tau \models \partial X \quad \tau \models \partial Y \quad \tau \models \partial Z \]
\[ \tau \models \partial(\text{if } X \text{ then } Y \text{ else } Z \text{ endif}) \] (5.28)
\[ \tau \models \partial X \quad \tau \models \partial Y \]
\[ \tau \models \partial(X \rightarrow Y) \] (5.29)
\[ \tau \models \partial X \quad \tau \models \partial Y \]
\[ \tau \models \partial(X \lor Y) \] (5.30)
\[ \tau \models \partial X \quad \tau \models \partial Y \]
\[ \tau \models \partial(X \land Y) \] (5.31)
\[ \text{cp } P \]
\[ (\tau \models \partial(\forall x \in S \cdot Px)) = (\tau \models \partial S \land ((\exists x. \tau \models x \in S \land \neg \tau \models Px) \lor (\forall x. \tau \models \partial x \land \tau \models x \in S \rightarrow \tau \models \partial Px))) \] (5.32)

Table 5.2.: The Definedness Calculi
5.5. Logical Bridges Between \([\text{UC, LE, LJE, and LTC}]\)

The following characterizations between statements in these logical formats hold:

\[
(A = B) = (\forall \tau. A \tau = B \tau)
\]  
(5.36)

\[
(A = B) = \left( (\forall \tau. (\tau \Vdash A) = (\tau \Vdash B)) \land (\forall \tau. (\tau \Vdash \neg A) = (\tau \Vdash \neg B)) \right)
\]  
(5.37)

\[
(A = B) = \left( (\forall \tau. (\tau \Vdash A) = (\tau \Vdash B)) \land (\forall \tau. (\tau \Vdash \neg A) = (\tau \Vdash \neg B)) \right)
\]  
(5.38)

\[
(A = B) = \left( (\forall \tau. (\tau \Vdash A) = (\tau \Vdash B)) \land (\forall \tau. (\tau \Vdash \neg A) = (\tau \Vdash \neg B)) \right)
\]  
(5.39)

\[
(A = B) = \left( (\forall \tau. (\tau \Vdash A) = (\tau \Vdash B)) \land (\forall \tau. (\tau \Vdash \neg A) = (\tau \Vdash \neg B)) \right)
\]  
(5.40)

\[
(\tau \Vdash (a \triangleq b)) = (a \tau = b \tau)
\]  
(5.41)

The two variants Equation 5.38 and Equation 5.40 implicitly exploit Equation 5.6: If two formulae agree in two truth-values, they have also to agree on the third. There is a third variant that is omitted here.

The characterization Equation 5.41 justifies an own tableaux calculus on the basis of local judgements. This rule is the “entry point” into section 5.9.

The rule Equation 5.42 also shows the connection of strong equality to local formula equivalence \([\text{LE}]\).

5.6. The Logics

5.6.1. Reasoning over Strong and Strict Equality

The strong equality satisfies the usual properties \textit{except} the Leipnitz rule (substitutivity); instead, the slightly weaker form Equation 5.42 of substitutivity (for context passing contexts \(P\)) holds. This side-constraint is not surprising, since since Equation 5.42 shown in section 5.5 we know that strong equality and global equivalence are semantically equivalent.

\[
\tau \Vdash a \triangleq a
\]  
(5.42)

\[
\tau \Vdash a \triangleq b
\]

\[
\tau \Vdash b \triangleq a
\]  
(5.42)

\[
\tau \Vdash a \triangleq b \quad \tau \Vdash b \triangleq c
\]

\[
\tau \Vdash a \triangleq c
\]  
(5.42)

\[
\tau \Vdash a \triangleq b \quad \tau \Vdash P a \quad \tau \Vdash P b
\]

\[
\tau \Vdash P b
\]  
(5.42)
The following lemmas show that strict equality is indeed convertible into strong equality and is a stronger equality:

$$\tau \models \varnothing a \quad \tau \models \varnothing b$$

$$(a \equiv b)\tau = (a \equiv b)\tau$$

$$\tau \models a \equiv b$$

$$\tau \not\models a \equiv b$$

(5.42)

5.6.2. Core-Logic (Boolean)

With \textit{core-logic} of OCL we refer to the sub-language consisting of the logical connectives $\neg,$ $\land,$ $\lor,$ etc., which we also call the \textit{propositional fragment} of OCL (although the underlying semantics is strictly speaking not propositional but Strong Kleene Logic SKL). The core-logic is contained in subsection B.4.2. Besides the computational rules like $\perp \land F = F,$ the core-logic enjoys a lattice-like structure with the rules shown in Equation 5.3(a).

In Equation 5.3(b), the rules are shown that deal with “logical reasoning” related to implication. Note, however, that the UC-rules do not form a complete calculus. The problem is (obviously) hidden in the only conditional rule, which has to be rephrased as LE-rule to achieve completeness. Unfortunately, this conditional rule corresponds to “assumption” and is therefore particularly vital in deduction. The Boolean case-split rule in Equation 5.3(c) is interesting for automated reasoning. Consequent case-splits over all Boolean variables yields a proof procedures of sufficient power, as can be seen for many facts in the basic library.

5.6.3. Set Theory and Logics

Set theory is the theory of membership $x \in S$ on the one hand and set constructions like comprehensions (corresponding to the $\rightarrow select()$-construct in OCL) on the other.

In OCL we have a typed form of a set theory which rules out, for example, Russels Paradox. With respect to typedness, OCL’s set theory is more related to HOL’s set theory, but more distant to ZF. Undefinedness, on the other hand, is a distinguishing feature of OCL’s set theory: Inclusion of elements in a set may result in an undefined set or in a set, that just contains undefined elements. As mentioned in subsection 4.7.1, we opt for a smashed sets semantics; the OCL standard is not clear with this respect.

On the deduction level, smashed semantics boils down to the following rule:

$$\tau \models x \in S$$

$$(\tau \models \varnothing x) \land (\tau \models \varnothing S)$$

(5.61)

This has the consequence, that whenever we eliminate a universal or existential OCL quantification, we know that the variable over which a quantifier ranges is defined. In itself, this is also useful to deduce that the body of a quantifier is defined—which it will usually not.
Chapter 5. Calculi

(a) Lattice

\[
\begin{align*}
F \land X &= F \quad (5.43) \\
T \land X &= X \quad (5.44) \\
T \lor X &= T \quad (5.45) \\
F \lor X &= X \quad (5.46) \\
X \quad &\text{op} \quad X = X \quad (5.47) \\
X \quad &\text{op} \quad Y = Y \quad \text{op} \quad X \quad (5.48) \\
X \quad &\text{op} \quad (Y \quad \text{op} \quad Z) = (X \quad \text{op} \quad Y) \quad \text{op} \quad Z \\
\end{align*}
\]

where \( \text{op} \in \{\land, \lor\} \quad (5.50) \)

\[
\begin{align*}
\neg (\neg X) &= X \quad (5.51) \\
(X \lor Y) \land Z &= (X \land Z) \lor (Y \land Z) \quad (5.52) \\
Z \land (X \lor Y) &= (Z \land X) \lor (Z \land Y) \quad (5.53) \\
(X \land Y) \lor Z &= (X \lor Z) \land (Y \lor Z) \quad (5.54) \\
Z \lor (X \land Y) &= (Z \lor X) \land (Z \lor Y) \quad (5.55) \\
\neg (X \land Y) &= \neg (\neg (X) \lor \neg (Y)) \quad (5.56) \\
\neg (X \lor Y) &= \neg (\neg (X) \land \neg (Y)) \quad (5.57) \\
\neg (X \lor Y) &= \neg (\neg (X) \land \neg (Y)) \quad (5.58) \\
\end{align*}
\]

(b) Logic

\[
\begin{align*}
X \rightarrow F &= \neg X \quad (5.59a) \\
X \rightarrow T &= T \quad (5.59b) \\
F \rightarrow X &= T \quad (5.59c) \\
T \rightarrow X &= X \\
\partial (X) &= T \\
(X \rightarrow X) &= T \\
X \rightarrow (Y \land Z) &= (X \rightarrow Y) \land (X \rightarrow Z) \quad (5.59d) \\
X \rightarrow (Y \lor Z) &= (X \rightarrow Y) \lor (X \rightarrow Z) \quad (5.59e) \\
(X \land Y) \rightarrow Z &= X \rightarrow (Y \rightarrow Z) \quad (5.59f) \\
(X \lor Y) \rightarrow Z &= (X \rightarrow Z) \land (Y \rightarrow Z) \quad (5.59g) \\
X \rightarrow (Y \rightarrow Z) &= Y \rightarrow (X \rightarrow Z) \quad (5.59h) \\
\end{align*}
\]

(c) Boolean Case-Split

\[
\begin{align*}
P \perp &\Rightarrow P' \perp \\
P \uparrow &\Rightarrow P' \uparrow \\
P \land F &= P' \land F \\
\text{cp}(P) \text{ cp}(P') \\
\hline
P \ X &= P' \ X \\
\end{align*}
\]

Table 5.3.: The \text{UC} Core-Calculus (“Propositional Calculus”)
Further, we allow sets to be infinite. This generalization has the advantage, that general sets can be used to represent the syntactic category of “types” in the standard as “characteristic sets” in HOL-OCL and enables the possibility to reason over them. From the pragmatics point of view, this also allows for quantifications of sets of “values” in the sense of OCL 2.0. For example, it is possible in HOL-OCL (see section 4.12 for details) to express the commutativity law on Integers inside OCL as follows:

\[
\text{Integer.typeSetOf()} \to \forall (x, y | \text{not } x. \text{oclIsUndefined()} \\
\quad \text{and} \\
\quad \text{not } y. \text{oclIsUndefined()} \\
\quad \text{implies} \\
\quad x + y = y + x)
\]

In an infinite set theory, each OCL type can be assigned to a characteristic set. Based on the characteristic sets for the base types Integer, Real, Boolean, String, we have defined set constructors \( \text{Set} : \alpha :: \bot \to \alpha \text{Set}_{\tau} \) that mimic the semantic effect of type constructors. The construction assures that characteristic sets are always defined objects. Equations for \( \partial \text{Integer} = T \), \( \partial X = T \Rightarrow \partial \text{Set}(X) = T \) etc.

Infiniteness of a set in the OCL can be naturally expressed by testing if the cardinality of the set is defined:

\[
\partial ||\text{Boolean}|| = T \\
\partial ||\text{Integer}|| = F
\] (5.62)

In contrary, finiteness of sets paves the way for an induction-scheme in HOL-OCL.

In the following, we discuss the core of the collection theories at the example of the set theory. We omit the type constraints from this presentation which enforce that the overloaded symbols are in fact interpreted as operations on sets.

Quantifiers and set constructors in OCL have a highly operational character with respect to undefinedness; in the standard, quantifiers were defined via iterators, hence fold-like constructs which also reflect the behaviour in case of undefinedness. This represents a particular challenge for deduction; on the other hand, similar languages like Spec# or VDM have the same characterizations, too, and the problem needs to be resolved for the forthcoming generation of “light-weight”-specification languages that are semantically “operation-like” and therefore closer to programming languages.

The combinator-based definition:

\[
\forall x \in S. \ P x \equiv \lambda x. \ \text{if def}(S \tau) \\
\quad \text{then if } \forall x \in \text{set} ' \text{RepSequence}(S \tau)' . \ P(\text{lift}_0 x) \tau = (\text{true}_\tau) \\
\quad \text{then true}_\tau \\
\quad \text{else if } \exists x \in \text{set} ' \text{RepSequence}(S \tau)' . \ P(\text{lift}_0 x) \tau = (\text{false}_\tau) \\
\quad \text{then false}_\tau \\
\quad \text{else } \bot
\]

turns out to be equivalent (for the finite case) to the operational formulation of the
Chapter 5. Calculi

standard:

\[ \tau \models \partial \| S \| \]
\[ (\forall x \in S. P x) \equiv (S \rightarrow \text{iterate}(x; y = T| (P x) \land y)) \tau \]

(5.63)

Of course, we are interested for the general case of infinite sets. As general \( UC \)-rules, we have:

\[ \forall x \in \bot. P x = \bot \quad (5.64) \]
\[ \exists x \in \bot. P x = \bot \quad (5.65) \]
\[ \forall x \in \emptyset. P x = T \quad (5.66) \]
\[ \exists x \in \emptyset. P x = F \quad (5.67) \]

\[ \tau \models \partial X \quad \tau \models \partial a \quad \text{cp P} \]
\[ (\forall x \in X \rightarrow \text{including}(a). P x) \tau = ((P a) \land (\forall x \in X. P x)) \tau \]

(5.68)

\[ \tau \models \partial X \quad \tau \models \partial Y \]
\[ (\forall x \in (X \cup Y). P x) \tau = ((\forall x \in X. P x) \land (\forall x \in Y. P x)) \tau \]

(5.69)

The latter two rules allow for the elimination of quantifications over known finite sets via computation.

Besides, there is a tableaux calculus for quantifier elimination can be directly derived from the rules in Equation 5.4(b).

The core of OCL set theory is the relation between the element-hood and the set comprehension (\( \rightarrow \text{select}(\_ | \_\_\)) and the relation to equality, i.e., a form of set
5.7. Arithmetic Computational Rules

An important source of deduction is computation. Computation is needed when \( \tau \models 3 + 5 = 4 \) is refuted. So far, we have only used declarative concepts to introduce numbers; the question arises how this can be used for computation, and even: how can this be used fairly efficiently in deduction.

This problem is by no means new and deeply intertwined with the existing solution in Isabelle/HOL. In the HOL library, a type bin for binary two’s complement representation has been introduced—by classical, conservative means. Now, the Isabelle parser is configured to parse a literal like \( 3 \) to bitstring representation representing \((101)_2\). Further, an axiomatic class num is defined providing a function declaration \( \text{numberOf} :: \text{bin} \Rightarrow \alpha :: \text{num} \) that can be overloaded for each type declared to be an instance of class num. Thus, for new datatypes, just a new function is defined that converts a bitstring representation to this new type. In the library, such a conversion has been provided, for example, for int. Based on this definitions, suitable rules have been derived that perform the integer operations like addition on the two’s complement representation directly; these rules can be directly processed by the simplifier.

With respect to the types \( \text{Integer}_\tau \), \( \text{Real}_\tau \) and \( \text{String}_\tau \), we can proceed analogously. For example, after declaring \( \text{Integer}_\tau \) to be an instance of num, we provide the following definition \( \text{numberOf} \) for the representation conversion: 

\[
(\text{numberOf} :: \text{bin} \Rightarrow \text{Integer}_\tau) b \equiv \text{lift}_0((\text{numberOf} :: \text{bin} \Rightarrow \text{int}) b_0)
\]  

(5.74)

This means, that the “new” \( \text{numberOf} \) with type \( \text{bin} \Rightarrow \text{Integer}_\tau \) is the context-lifted, \( \bot \)-lifted version of the “old” \( \text{numberOf} \) on integers. From this definition, among others,
Chapter 5. Calculi

the following facts follow:

\[ \partial(\text{numberOf } a) = T \tag{5.75} \]
\[ (\text{numberOf } a) + (\text{numberOf } b) = \text{numberOf}(a + b) \tag{5.76} \]
\[ (\text{numberOf } a) \cdot (\text{numberOf } b) = \text{numberOf}(a \cdot b) \tag{5.77} \]
\[ \neg \text{iszero}(\text{numberOf } (a - 2 b)) \]
\[ (\text{numberOf } a) \triangleq (\text{numberOf } b) = F \tag{5.78} \]

Thus, besides definedness-related computations (“all values are defined”), computations in OCL were mapped directly to computations in the underlying meta-logic HOL. This setup enables the standard simplifier of Isabelle to refute judgements like

\[ \tau \vdash 3 + 2 \equiv 7 \tag{5.79} \]

fully automatic, i.e., without user interaction.

5.8. Converting OCL to HOL

For a fragment of OCL that is built for expressions that are always defined, the following “conversion” of OCL formula into standard HOL formulae over local judgements equivalences LJE are possible. The propositional part of the translation is described in Equation 5.4(a), the predicative part in Equation 5.4(b). The rules for the other collection types are accordingly.

5.9. The Judgement Tableaux Calculus LTC

The conversion technique discussed in section 5.8 requires reasoning on side-conditions such as \( cp \ P \) or definedness \( \partial X \). The question arises, if this can be avoided when performing a logical decomposition of the OCL formulae directly.

The tableaux methodology is one of the most popular approaches to design and implement proof-procedures. While originally geared towards first-order theorem proving, in particular for non-clausal formulae accommodating equality, renewed research activity is being devoted to investigating tableaux systems for intuitionistic, modal, temporal and many-valued logics, as well as for new families of logics, such as non-monotonic and sub-structural logics. Many of these recent approaches are based on a special labeling technique on the level of judgments, called labeled deduction \([16, 51]\). Of course, labeling can also be embedded into a higher-order, classical meta-logic. Being a special case of a many-valued logic, tableaux calculi for SKL based on labeled deduction have been extensively studied \([29, 20, 19]\). In this section, we present an experimental tableaux calculus for the predicative fragment of OCL, i.e., for SKL roughly following \([29]\). It is designed to be processed by Isabelle’s generic proof procedures, which are geared towards natural deduction.
5.10. Towards Automated Deduction in HOL-OCL

(a) Propositional Conversion

\[ \tau \models \partial A \]
\[ (\tau \models \neg A) = (\neg \tau \models A) \quad (5.80) \]
\[ \tau \models \partial A \quad \tau \models \partial B \]
\[ (\tau \models A \land B) = (\tau \models A \land \tau \models B) \quad (5.81) \]
\[ \tau \models \partial A \quad \tau \models \partial B \]
\[ (\tau \models A \lor B) = (\tau \models A \lor \tau \models B) \quad (5.82) \]
\[ \tau \models \partial A \quad \tau \models \partial B \]
\[ (\tau \models A \implies B) = (\tau \models A \implies \tau \models B) \quad (5.83) \]

(b) Predicative Conversion

\[ \tau \models \partial(S :: (\beta :: \text{bot} \, \text{Set}) \, \text{cp} \, P) \]
\[ (\tau \models \forall x \in S, P(x :: \tau \Rightarrow \beta)) = (\forall x. \tau \models x \in S \implies \tau \models P \, x) \quad (5.84) \]
\[ \tau \models \partial(S :: (\beta :: \text{bot} \, \text{Set}) \, \text{cp} \, P) \]
\[ (\tau \models (\exists x \in S, P(x :: \tau \Rightarrow \beta))) = (\exists x. \tau \models x \in S \land \tau \models (P \, x)) \quad (5.85) \]

Table 5.4.: The OCL to HOL Conversion Rules

Tableau proofs may be viewed as trees where the nodes are lists of formulae. Tableau rules extend the leaves of a tree by a new subtree, i.e., by adding leaves below, where the latter case is called “branching” and is used for case splits. Classical tableau rules are purely analytic: each rule captures the full logical content of the expanded connective. Backtracking from a rule application is never necessary. The goal of the process is to construct trees in a deterministic manner, where the leaves can eventually all be detected as “closed,” i.e., a logical contradiction is detected. This last step, however, may be combined with the non-deterministic search for a substitution making this contradiction possible.

The LTC appears in Table 5.5(a).

5.10. Towards Automated Deduction in HOL-OCL

In the current HOL-OCL distribution, only two major OCL-related proof procedures have been implemented:

- OCL_hypsubst
- OCL_subst

They mimick the Isabelle [3] standard tactics hypsubst and subst and serve as an abstract interface to to the various rules that express local congruences, strict and strong
Chapter 5. Calculi

(a) Definedness Introduction and Elimination

\[ \begin{array}{c|c|c|c}
\tau \models A & \tau \models \neg A & \neg(\tau \models \neg A) \\
\hline
\tau \models \partial(A) & \bar{R} & \bar{R} & \tau \notmodels A \\
\tau \models \neg \partial(A) & \neg(\tau \models \neg A) & \tau \models A \\
\end{array} \]  
Equation 5.86

(b) Negation

\[ \begin{array}{c|c|c|c|c|c|c|c}
\tau \models \neg(\neg A) & \tau \models A & \tau \models \neg(A) & \tau \models \neg A & \tau \models \neg A & \tau \models \neg A & \tau \models \neg A \\
\hline
\tau \models \neg(A) & \tau \models \neg(A) & \tau \models \neg \partial(A) & \tau \models \neg \partial(A) & \tau \models \neg \partial(A) & \tau \models \neg \partial(A) & \tau \models \neg \partial(A) \\
\end{array} \]  
Equation 5.88

(c) Conjunction Introduction and Elimination

\[ \begin{array}{c|c|c|c|c|c|c|c}
\tau \models A, \tau \models B & \tau \models \neg(A \land B) & \tau \models \neg B & \tau \models \neg A & \tau \models \neg A & \tau \models \neg A & \tau \models \neg A \\
\hline
\tau \models \neg(\neg(A \land B)) & \tau \models \neg(A \land B) & \tau \models \neg B & \tau \models \neg A & \tau \models \neg A & \tau \models \neg A & \tau \models \neg A \\
\end{array} \]  
Equation 5.89

\[ \begin{array}{c|c|c|c|c|c|c|c}
\tau \models \neg(A \land B) & \tau \models \neg A & \tau \models \neg B & \tau \models \neg(A \land B) \\
\hline
\tau \models A & \tau \models \neg B & \tau \models \neg(A \land B) & \tau \models \neg(A \land B) & \tau \models \neg(A \land B) & \tau \models \neg(A \land B) & \tau \models \neg(A \land B) \\
\end{array} \]  
Equation 5.90

\[ \begin{array}{c|c|c|c|c|c|c|c}
\tau \models \neg(A \land B) & \tau \models \neg A & \tau \models \neg B & \tau \models \neg(A \land B) \\
\hline
\tau \models \neg(A \land B) & \tau \models \neg A & \tau \models \neg B & \tau \models \neg(A \land B) & \tau \models \neg A & \tau \models \neg B \\
\end{array} \]  
Equation 5.91

(d) Contradictions

\[ \begin{array}{c|c|c|c|c|c|c|c}
\tau \models A & \tau \models \neg A & \tau \models A & \tau \models \neg A & \tau \models \neg A & \tau \models \neg A & \tau \models \neg A & \tau \models \neg A \\
\hline
\tau \models \neg(\neg(A \land B)) & \tau \models \neg(A \land B) & \tau \models \neg A & \tau \models \neg A & \tau \models \neg A & \tau \models \neg A & \tau \models \neg A & \tau \models \neg A \\
\end{array} \]  
Equation 5.92

Table 5.5: The Core of LTC
5.10. Towards Automated Deduction in HOL-OCL

equalities and congruences hidden in local judgements as pointed out in section 5.1. For the moment, this tactic setup allows for a step-by-step reasoning using the rules of the logic and UC-rules (including computational rules) in the Isabelle simplifier. Provided that sufficient information on the definedness of free variables is available in a proof state, this enables a conversion to HOL formulae with the rules discussed in section 5.8 possible. A converted formula can be treated by the standard Isabelle proof procedures like safe_tac, blast_tac and auto_tac possibly intertwined with OCL_hypsubst. This covers to a certain extent logical reason automatically.

However, the situation is clearly not satisfactory for larger, application oriented proof projects in OCL so far. Here is a list of the most painful shortcomings when comparing it with proofs in “pure” Isabelle/HOL:

- The library on datatypes and on datatype-oriented rules is far from being sufficiently developed.
- An arithmetic decision procedure such as arith_tac is missing.
- All rules of the LE-format are not usable by the simplifier. Since the complete core calculus and many datatype-oriented rules are in this format, the proof engineer is limited to elementary proof techniques excluding the simplifier whenever these rules are involved in a proof.
- Rules of the LE-format are also excluded from the classical reasoner, i.e., the fast_tac, blast_tac and auto_tac procedures.
- A combined automatic tactic integrating all these local procedures like the auto_tac-procedure.

Partly, the situation is comparable to HOL 10 years ago—and a fair comparison to similar logical languages has to take into account that the development of HOL libraries and proof procedures needed this time. For OCL, the development of proof procedures and, more critical, the technical support of formal methods based on OCL is still at the beginning. The latter will have to cope with path-expressions, modifies-clauses, and refinement-like situations.

In the following, we will summarize our ideas about potential future tactics to reason over OCL automatically.

With respect to arithmetic, besides a step-by-step reasoning, only the following paths to use automated procedures seem to be viable: defined arithmetic terms have to be converted (by unfolding semantic combinators and blowing away the cascades of definedness-conditions) into pure HOL arithmetic formulae and reuse the existing procedure (the adaption approach). Alternatively, arith_tac must be rewritten to cope with definedness issues (the re-engineering approach).

With respect to rewriting, we see (besides the not very attractive re-engineering approach) the following techniques to adapt to existing Isabelle technology:

**Proof-object transformation.** Since one can instantiate the simplifier with new equalities obeying the Leipnitz rule, one can run it in an unsafe-mode without
Chapter 5. Calculi

checking the cp-side conditions. Proof-objects generated in an unsafe mode could be extended to “full” proof objects where the missing parts are reconstructed. It remains to be explored how costly this approach would be (in development time as well as runtime; previous experience (blast_tac) suggests that at least the runtime costs are insignificant).

Making context-passing explicit. One can transform proof-states and rules in a format where context-passingness is encoded directly at all positions in a term. As a consequence, the simplifier can process the transformed rules directly. Additionally to the conversion tactics that perform this term-transformation in forward and backward proof, the major changes for this technique boil down to the management of transformed and un-transformed rule-sets used by the simplifier and auto_tac.

It is worth to present the letter option a bit more in detail:

\[
\text{cp } P \\
\frac{P \; X \; \tau}{P \; (\text{lift}_o (X \; \tau)) \; \tau} \quad (5.93)
\]

One can also annotate any redex \( X \) explicitly with the context it is referring to. This leads to a linear blowup on the size of terms. On such redexes, LE-rules match directly and can be processed by the simplifier. To enable the simplification of assumptions of a rewriting rule, this technique requires a pre-computation on all rules which should be hidden from the user.

Depending on the decision for the simplification, it is either possible to adopt tableaux calculi for the classical reasoning (however, first experiments showed that the built-in strategies are not compatible with three-valued logic) or to convert a proof-goal to a HOL-formula to be processed by standard classical reasoning.

At present, we consider the latter strategy for an automatic setup as more promising. Thus, an auto_tac could proceed as follows:

1. general logical simplification and normalization with the UC calculi,
2. performing a definedness case-split on all free variables in a subgoal, which are not already known to be defined or undefined. Moreover, for all potentially undefined subterms (like \( 1 \div 0 \)), a suitable case-split is generated,
3. using OCL_hypsubst, all \( \bot \) were propagated and simplified,
4. a conversion to HOL-formulae is performed (following the rules in 5.8), extended by rules converting equalities on collection types, and simplifying path expressions along their structure,
5. making cp explicit (see above),
6. and apply standard auto_tac.
5.10. Towards Automated Deduction in HOL-OCL

We call the first three phases the *splintering* of an OCL formula, the resulting subgoals are the *splinters*. In principle, there may be exponential many splinters depending on the number of free variables in the formula. Variables bound by OCL quantifier are implicitly defined due to smashed collection semantics. However, due to the fact that expressions are usually built from strict operators, this growth is unlikely in practice. The subsequent phases will deeply interact with the datatype specific rules of the library.
Chapter 5. Calculi
Chapter 6.
The HOL-OCL System Architecture

6.1. An Architectural Overview

HOL-OCL is integrated into a framework [10], which among others provides a model repository, su4sml, written in SML. For a quick overview over main system components of HOL-OCL see Figure 6.1. HOL-OCL is based on the SML interface of Isabelle/HOL and the UML/OCL model repository. As front-end, HOL-OCL provides a special instance of ProofGeneral [5] and a LaTeX based documentation generation.

In this chapter, we will briefly present the main components of the HOL-OCL core system, namely:

- our datatype package, or encoder, which encodes UML/OCL models into HOL-OCL, i.e., from a user’s perspective it provides the XMI import facilities.
- the HOL-OCL library which provides the core theorems needed for verification and also provides a formal semantics for OCL.
- the theory morpher which derives many of the core OCL theorems by “lifting” them based on the corresponding theorems already proven for HOL.

As further background, we will also give a short introduction of the model repository su4sml.

6.2. The Model Repository: su4sml

The Model Repository su4sml [10] provides an interface to models expressed in the UML core (mainly class diagrams and state machines) and OCL. OCL expressions naturally translate into an abstract datatype in SML, as shown in Listing 6.1 and Listing 6.2. This abstract datatype is modeled closely following the standard OCL 2.0 metamodel. In addition to these datatype definitions, the repository structure defines a couple of normalization functions, for example for converting association ends into attributes with corresponding type, together with an invariant expressing the cardinality constraint.

For class models (see Listing 6.3), su4sml resembles the tree structure given by the “containment hierarchy.” For example, a class contains attributes, operations, or
Chapter 6. The HOL-OCL System Architecture

Figure 6.1.: Overview of the HOL-OCL architecture

```sml
signature REP_OCL_TYPE = sig
  type Path = string list
  datatype OclType = Integer | Real | String | Boolean (* Primitive Types *)
                    | OclAny | OclVoid
                    | Set of OclType | Sequence of OclType
                    | OrderedSet of OclType | Bag of OclType
                    | Collection of OclType
                    | Classifier of Path (* user-defined classifiers *)
                    | DummyT (* dummy type for untyped expressions *)
end
```

Listing 6.1: su4sml: Representing OCL types
6.2. The Model Repository: su4sml

signature REP_OCL_TERM = sig
include REP_OCL_TYPE

datatype OclTerm =
  Literal of string * OclType (* literal with type *)
  | CollectionLiteral of CollectionPart list * OclType (* content with type *)
  | If of OclTerm * OclType (* condition *)
     * OclTerm * OclType (* then *)
     * OclTerm * OclType (* else *)
     * OclType (* result type *)
  | AssociationEndCall of OclTerm * OclType (* source *)
     * Path (* assoc -enc *)
     * OclTerm (* result type *)
     * OclType (* result type *)
  | AttributeCall of OclTerm * OclType (* source *)
     * Path (* attribute *)
     * OclType (* result type *)
  | OperationCall of OclTerm * OclType (* source *)
     * Path (* operation *)
     * (OclTerm * OclType) list (* parameters *)
     * OclType (* result type *)
  | OperationWithType of OclTerm * OclType (* source *)
     * string * OclType (* type parameter *)
     * OclType (* result type *)
  | Variable of string * OclType (* name with type *)
  | Let of string * OclType (* variable *)
     * OclTerm * OclType (* rhs *)
     * OclTerm * OclType (* in *)
  | Iterate of (string * OclType) list (* iterator variables *)
     * string * OclType (* result variable *)
     * OclTerm * OclType (* source *)
     * OclType (* result type *)
  | Iterator of CollectionPart = CollectionItem of OclTerm * OclType (* element with type *)
     * CollectionRange of OclTerm (* first *)
     * OclTerm (* last *)
     * OclType (* type of range *)

end

Listing 6.2: su4sml: Representing OCL expressions
**Chapter 6. The HOL-OCL System Architecture**

```
signature REP_CORE = sig

  type Scope
  type Visibility
  type operation = { name : string,
                    precondition : (string option * OclTerm) list,
                    postcondition : (string option * OclTerm) list,
                    arguments : (string * OclType) list,
                    result : OclType,
                    isQuery : bool,
                    scope : Scope,
                    visibility : Visibility }

  type associationend = { name : string,
                          aend_type : OclType,
                          multiplicity : (int * int) list,
                          ordered : bool,
                          visibility : Visibility,
                          init : OclTerm option }

  type attribute = { name : string,
                     attr_type : OclType,
                     visibility : Visibility,
                     scope : Scope,
                     stereotypes : string list,
                     init : OclTerm option }

  datatype Classifier = Class of { name : Path,
                                  parent : Path option,
                                  attributes : attribute list,
                                  operations : operation list,
                                  associationends : associationend list,
                                  invariant : (string option * OclTerm) list,
                                  stereotypes : string list,
                                  interfaces : Path list,
                                  activity_graphs : ActivityGraph list}
       | Interface of { ... } (* similar to Class *)
       | Enumeration of { ... }
       | Primitive of { ... }
end
```

**Listing 6.3:** swisml: Representing the UML core
6.3. The Encoder: An Object-oriented Datatype Package

Encoding object-oriented data structures in HOL, as needed for HOL-OCL, is a tedious and error-prone activity, which should be automated. In this section, we give an overview of the su4sml-based datatype package we implemented to automate this process. In the theorem prover community, a datatype package [33] is a module that allows one to introduce new datatypes and automatically derive certain properties over them. A (conservative) datatype package has two main tasks:

1. generate all required (conservative) constant definitions, and
2. prove as much (interesting) properties over the generated definitions as possible automatically behind the scenes.

For our datatype package, we use the possibility to build SML programs performing symbolic computations over formulae in a logically safe way.

In the following, we give a brief overview what our package does. The datatype package is implemented on top of the su4sml interface on one hand and on top of the Isabelle core on the other (see Listing 6.4 for details). During the encoding, our datatype packages extends the given theory by a HOL-OCL-representation of the given UML/OCL model. This is done in an extensible way, i.e., classes can be added later on to an existing theory preserving all proven properties. The obvious tasks of the datatype package are:

1. declare HOL types for the classifiers of the model,
2. encode the core data model into HOL, and
3. encode the OCL specification and combine it with the core data model.

Listing 6.4: The Top-level Interface of the Repository Encoder

```
signature REP_ENCODER =

sig
  type mdr = { theory : theory,
               universe : typ,
               classifiers : Classifier list }

  val add_classifiers : Classifier list -> mdr -> mdr
end
```

statemachines. We also decided to ignore associations as such. We only represent their association ends, again as part of the participating classifiers.

Overall, the top-level data structures (see Listing 6.1-6.3) of su4sml are inspired by the metamodels of OCL [41, Chapter 8] and UML [40] and readers familiar with these metamodels should recognize the similarities.
Chapter 6. The HOL-OCL System Architecture

In fact, the most important task is probably not that obvious: The package has to generate formal proofs that the generated encoding of object-structures is a faithful representation of object-orientation (e.g., in the sense of the UML standard \cite{40}, or Java). These theorems have to be proven for each model during its encoding phase. Among many other properties, our package proves for each pair of classes $A$ and $B$ where $B$ is a generalization\footnote{inherited from} of $A$ the following facts:

\[ \text{self.oclIsType}(B) \Rightarrow \text{self.oclIsKind}(A). \]  

(6.1)

as well as the more complicated property:

\[ \text{self.oclIsDefined}() \land \text{self.oclIsType}(B) \Rightarrow \text{self.oclAsType}(A).\text{oclAsType}(B).\text{oclIsDefined}() \land \text{self.oclAsType}(A).\text{oclAsType}(B).\text{oclIsType}(B). \]  

(6.2)

Listing 6.5 presents a simplified version of the SML function \textit{cast_class_id} that proves the property (6.2). The expression starting in line 5 generates a type-checked instance of the current theorem to prove with respect to the current class (and its parent). Readers familiar with LCF-style theorem provers will recognize the “proof script” in lines 10 to 23. Finally, the function registers the proven theorem in Isabelle’s theorem database. Logical rules like (6.1) or (6.2) or co-induction schemes given by class invariants constitute the object-oriented datatype theory of a given class diagram and represent the basic weapon for proofs over them, in particular verifications of UML/OCL specifications. Stating these rules could be achieved by adding axioms (i.e., unproven facts) during the encoding process, which is definitively easier to implement. Instead, our datatype package generates entirely conservative definitions and derives these rules from them; this also includes the definition of recursive class invariants, which are in itself not conservative (see section 4.11 for details).

This strategy, i.e., stating entirely conservative definitions and formally proving the datatype properties for them, ensures two very important properties:

1. our encoding fulfills the required properties, otherwise the proofs would fail, and
2. doing all definitions conservatively together with proving all properties ensures the consistency of our model (provided that HOL is consistent and Isabelle/HOL is a correct implementation).

To get a feeling for the amount of work needed, the import of the “Company” model (including the OCL specification) presented in the OCL standard \cite{41} Chapter 7 generates 1147 conservative definitions and proven theorems, the larger “Royals and Loyal” model \cite{52} model generates 2472 conservative definitions and proven theorems.
6.4. The HOL-OCL-Library

An important part of HOL-OCL is a collection of Isabelle theory files describing the formalization in detail. These theories also contain new proof procedures (tactics) written in SML. These theory files are extensively documented in Appendix B in the appendix, the developed tactics are also described in section 9.3.

6.5. The Theory-Morpher

The theory morpher provides automatic support lifting theorems from the HOL level to the HOL-OCL level. This is based on our organization of the library function definitions for our typed shallow embedding in a layered theory morphism. The theory morpher, or lifter is in principle a tactic-based program that lifts meta-level theorems to their object-level counterparts and meta-level prover configurations to object-level ones.

Our approach can be seen as an attempt to liberate the shallow embedding technique from the “point-wise-definition-style” in favor of more global semantic transformations from one language level to another. We abstracted the underlying conceptual notions into a generic framework that shows that the overall technique is applicable in a wide range of embeddings in type systems; embedding-specific dependencies arise only from the specifications of semantic combinator (the layers), and technology specific dependencies from the used tactic language.
Chapter 6. The HOL-OCL System Architecture

In the present version, the tactic proof engine cannot handle data adaption invariants which limits its potential to 10%. However, this limitation will be overcome soon.
Part III.

System Description
Chapter 7.
Installing HOL-OCL

7.1. Prerequisites
HOL-OCL is built on top of Isabelle/HOL-Complex, version 2005, thus you need an
working installation of Isabelle 2005. At the moment, HOL-OCL requires an Isabelle
based on SML/NJ [6] to be full functional. We strongly recommend also to install the
generic proof assistant front-end Proof General [5]. The overall system architecture
is depicted in Figure 7.1. For doing larger applications, you should install a complete
tool-chain comprising a CASE-Tool, e.g., ArgoUML, and an OCL type-checker (from the
Dresden OCL2 toolkit) for generating type-correct UML/OCL models in a comfortable
way (see Figure 9.1 for an overview of the proposed HOL-OCL work flow).

7.2. Installation
In the following we will give a brief guide how to install the prerequisites for an HOL-OCL
base system (i.e., HOL-OCL and Isabelle), please refer to the respective documentation
for details. We will not describe how to install the CASE-tool and the OCL type checker,
but we recommend to install them in any case. We have tested our XMI import using
the following setup:

1 The XMI import uses internally the Word32 structure which is not supported by Poly/ML [4].
2 Currently HOL-OCL was only tested on GNU/Linux systems on the i386 architecture, please report
your experiences when you try to install HOL-OCL on a different operating system or architecture.

![Figure 7.1: Overview of the High-level System Architecture of HOL-OCL](image)
Chapter 7. Installing HOL-OCL

- A recent version (0.20 or higher) of the CASE tool “ArgoUML” (http://argouml.tigris.org/).
- The type checker “Dresden-OCL 1.1” for OCL 2.0 (http://dresden-ocl.sf.net). Our toolchain only relies on the “Parser GUI” which can be started using the command

\[
\text{java -jar ocl20parsertools.jar}
\]

within the top-level directory of the Dresden OCL 2 toolkit. Alternatively, a command-line version of the type-checker is available:

\[
\text{java -cp ocl20parsertools.jar tudresden.ocl20.core.parser.OCL20CLI}
\]

7.2.1. Installation from Source

SML and Isabelle

1. Download a recent version of SML/NJ [6] install it following its documentation. We recommend version 110.56 which can be downloaded from http://www.smlnj.org/dist/working/110.56/index.html.

2. Download the Isabelle 2005 source code [3] from the from the Isabelle web-site (http://isabelle.in.tum.de/dist/packages.html), after unpacking the Isabelle source, replace the file src/Pure/defs.ML with contrib/defs.ML which is part of the HOL-OCL distribution [4]. After that you can follow the normal instructions for building Isabelle. Note that you need to build the HOL-Complex image, i.e., you have to call

\[
\text{$ISABELLE_HOME/build -m HOL-Complex HOL}
\]

3. Finally install a version of Proof General that supports Isabelle 2005, e.g. we strongly recommend a 3.6 pre-release. E.g., download http://proofgeneral.inf.ed.ac.uk/releases/ProofGeneral-3.6pre051004.tar.gz and follow the supplied instructions.

Installing HOL-OCL

In the following we assume that you have a running Isabelle 2005 environment including the Proof General based front-end. The installation of HOL-OCL requires the following steps:

1. Unpack the HOL-OCL distribution, e.g.:

\[
\text{tar zxfv holocl-0.9.0.tar.gz}
\]

[3] Note that you cannot use the pre-compiled heap images as they are based on Poly/ML.
[4] This fixes a performance bug in Isabelle 2005 which renders our XMI import unusable.
This will create a directory holocl-0.9.0 containing the HOL-OCL distribution.

2. Check the settings in the configuration file holocl-0.9.0/make.config. If you can use the isatool tool from Isabelle on the command line, the default settings should work.

3. Change into the src directory

   cd holocl-0.9.0/src

and build the HOL-OCL heap image for Isabelle by calling

   isatool make

4. HOL-OCL queries the environment variable $HOLOCL_HOME for finding runtime configurations. Thus you have to set this variable pointing to the top-level directory of your HOL-OCL installation.

5. Finally, you need to extend your ~/.emacs file with the following lines:

   (add-to-list 'load-path (concat (getenv "HOLOCL_HOME") "~/etc/"))
   (load "x-symbol-holocl-startup") ;; Proof General extension
   ;; for HOL-OCL

   This will make Proof General aware of the new syntax and commands provided by HOL-OCL. Further, if you are using GNU Emacs, you can also add the following lines to your ~/.emacs file:

   (require 'ocl) ;; simple mode for editing
   ;; OCL files

   This will provide syntax highlighting for plain OCL specifications.

### 7.2.2. Using Debian Packages

Installing HOL-OCL on an Debian GNU/Linux on a i386 architecture should be straightforward. Just add the IsaMorph apt-repository to the sources of your package manager, e.g., by adding the following lines

# IsaMorph repository
deb-src http://kisogawa.inf.ethz.ch/isamorph/debian stable main
deb http://kisogawa.inf.ethz.ch/isamorph/debian stable main

to /etc/apt/sources.list file. Please replace stable by the distribution you are using (we provide packages for all three flavours, i.e., stable, testing or unstable). After that, update your package list, i.e., by executing

   aptitude update
Chapter 7. Installing HOL-OCL

Now install a complete (eventually you have to install other logics, if you need them) Isabelle setup by executing

```
aptitude install x-symbol proofgeneral-misc isabelle \
isabelle-thy-hol-complex
```

and HOL-OCL by executing

```
aptitude install hol-ocl
```

This should give you a running installation of sm/NJ Isabelle, Proof General and last but not least, HOL-OCL in its default configuration (based on smashed sets and a referential universe). If you need a different setup, please install HOL-OCL from source and change the configuration in the make.config file.

7.3. Starting

Regardless of the installation method you are using, you should now be able to start HOL-OCL using the command:\(^5\)

```
Isabelle -L HOL-OCL
```

As HOL-OCL provides new top-level commands, the -L HOL-OCL is mandatory. After a few seconds you should see a Emacs window similar to the one shown in Figure 7.2.

\(^5\)If you installed the Debian packages, you can also start HOL-OCL using the command hol-ocl or start HOL-OCL using the menu entry of your favourite desktop environment.
7.3. Starting

Figure 7.2.: A HOL-OCL session Using the Isar Interface of Isabelle
Chapter 7. Installing HOL-OCL
Chapter 8.
Limitations of HOL-OCL

In this chapter we give a brief overview of the supported \textit{OCL}/\textit{UML} subset and current limitations of HOL-OCL.

8.1. The Supported \textbf{UML} Subset

HOL-OCL aims for supporting a relevant subset of the “\textit{UML Core}” module, i.e., more sloppy: the subset needed to express \textit{UML} class diagrams.\footnote{Support for dynamic diagram types, i.e., activity charts or state charts is planned for a future release.} The most restricting limitations are:

\begin{itemize}
  \item \textit{Enumeration} are not supported, but planned for a future release.
  \item \textit{Associations} are represented by their associations ends (with additional constraints). Direct association support (i.e., as relations) is planned.
  \item \textit{Association classes} are not supported, support is planned together with the direct support for associations.
  \item Qualifiers for \textit{association ends} are not supported.
  \item \textit{Ordered associations ends} have the type \textbf{Sequence} by the Dresden \textit{OCL} Toolkit which is thus also the type used by HOL-OCL. In contrast, the \textit{UML} standard \cite[p. 3-71]{40} requires the representation as \textbf{OrderedSet} (i.e., the elements of the set have an ordering, but duplicates are still prohibited).
\end{itemize}

Further note that the \textit{UML} standard \cite[p. 5-10]{40} defines the following primitive datatypes: \texttt{Integer}, \texttt{UnlimitedInteger}, \texttt{String}, and \texttt{Enumerations}. We consider these datatypes different from the datatypes defined in the \textit{OCL} standard \cite{41}, e.g. HOL-OCL supports the \textit{OCL} types \texttt{Integer}, \texttt{Real}, \texttt{String}, and \texttt{Boolean} (the datatype \texttt{enumeration} is not supported at the moment). During modeling, we advise you to place the \textit{OCL} datatypes in a package called \texttt{UML\_OCL} (as proposed by the \textit{UML} standard version 1.4).
Chapter 8. Limitations of HOL-OCL

8.2. The Supported OCL Subset

8.2.1. OCL Syntactical Variants

In general, the OCL standard introduces a lot of syntactical variants to reduce the amount of text one has to write, i.e., bound variables do not need to be annotated with a concrete type, or can even be omitted. For example, the following two invariants are only syntactical different:

context A:
inv: self.b->select(i=5)
inv: self.b->select(a | a.i =5 )

At the moment, our tool chain does not support all these variants, in particular, one has to make bound variable explicit and has to annotate them with type information:

context A:
inv: self.b->select(b:B | b.i = 5)

Furthermore, introducing alternative names for self, i.e.:

context foo:A:
inv: foo.i = 5

is currently not supported.

8.2.2. OCL context declarations

The OCL standard [41, pp. 157] introduces the different context declarations, which are handled by HOL-OCL as follows:

inv: fully supported.
pre: fully supported.
post: fully supported.
init: are supported by converting them into an invariant, e.g.

context A:
init: self.x = 5

will be converted into

context A:
inv: self.oclIsNew() implies self.x = 5

Note, this formulae can be considered as non-standard with regards to the OCL standard but is valid in HOL-OCL.

114
8.2. The Supported OCL Subset

def: not supported due to Dresden OCL (will be supported after Dresden OCL is fixed), i.e., for HOL-OCL there is no difference between statements defined (graphical) within the case tool or as OCL formulae.

derive: not supported due Dresden OCL (will be supported after Dresden OCL is fixed), i.e., for HOL-OCL there is no difference between statements defined (graphical) within the case tool or as OCL formulae.

body: supported by converting them into a post condition, e.g.

```
context A::f(): Integer
body: 5
```

will be converted into

```
context A::f(): Integer
post: result = 5
```

guard: not supported, support will be developed once we support dynamic diagram types.

8.2.3. OCL Types

OCL introduces a variety of pre-defined types (and meta-types), for some of them certain restrictions apply within HOL-OCL. In more detail:

- The type OclVoid is modeled implicit, for details see the lifting construction and the handling of the type class bot.

- The type OclMessage is not support, support for it will be eventually developed together with the support for dynamic diagram types (e.g. activity charts).

- Enumeration are not supported, but planned for a future release.

- The types OclModelElementType and OclType are modeled implicit, respectively replaced by the type of the characteristic set (of a type).

- The enumeration OclModelElement is not supported, respectively modeled implicit.

- The type Tuple (respectively TupleType) is not yet supported. Support for tuples is planned for a future release.

- The type OrderedSet is not yet supported by the Dresden OCL Toolkit but is supported by HOL-OCL. If you want to use OrderedSet you have to enter the corresponding constraints directly into HOL-OCL.
Chapter 8. Limitations of HOL-OCL

8.2.4. Predefined Properties

HOL-OCL introduces a variety of pre-defined properties and operations, for some of them certain restrictions apply within HOL-OCL. In more detail:

- **OclInState** is not supported at the moment; it will be eventually developed together with the support for dynamic diagram types (e.g. state charts and activity charts).

- The generic “->iterate()” expression is at the moment not supported by the XMI import. Nevertheless, it can be entered directly within HOL-OCL. The pre-defined iterator expressions, e.g. ->one or select, are fully supported.

- Collection literals, e.g., Set{1,3,4,9} or Sequence{1,...,42} are at the moment not supported by the XMI import. Nevertheless, they can be entered directly within HOL-OCL.

8.3. Reporting Bugs

If you find a bug in HOL-OCL, please send electronic mail to hol-ocl@brucker.ch. Include the version number, which you can find by running the info() in the interactive HOL-OCL-environment. Also include in your message the output that the program produced and the output you expected.

If you have other questions, comments or suggestions about HOL-OCL, contact the author via electronic mail to hol-ocl@brucker.ch. The authors will try to help you out, although they may not have time to fix your problems.
Chapter 9.

A HOL-OCL Reference Manual

9.1. The Basic Workflow

HOL-OCL allows one to reason formally over UML/OCL specifications. The overall HOL-OCL work-flow (see Figure 9.1 for an overview) is divided into two phases:

Modelling Phase: During the modeling and design phase one formalizes the (often informal given) requirements. The result of this phase should be a formal model of the software system being built (in our case, formalized using UML/OCL). Technically, the modelling phase is twofold:

1. design your (data) model using a CASE tool, e.g., ArgoUML and export it in XMI format.
2. refine your data model by adding OCL constraints using the Dresden OCL Toolkit. Export your refined model (including the OCL formulae in XMI format).

Verification Phase: In the analyzing and verification phase, the UML/OCL model is formally explored using a Isar-based 

In the following, we will give a brief overview of the modelling phase and describe the verification phase in more detail.

9.2. The Modelling Phase

In this section, we give a brief overview of the modelling phase, i.e., we concentrate ourselves on describing the most important tasks and pitfalls of this phase. Please refer to the manuals of your CASE tool and Dresden OCL for more information about the technical details of the modelling phase.

Overall, the modeling phase is split into two parts:

Figure 9.1.: The HOL-OCL Workflow

Figure 9.2.: Using ArgoUML for Data Modelling
9.2. The Modelling Phase

```ocl
package RoyalLoyal

context LoyaltyAccount::isEmpty(): Boolean
  pre: true
  post: result = (points = 0)

context LoyaltyAccount::points : Integer
  init: 0

context LoyaltyAccount
  inv Transactions: transactions.points
      ->exists(p : Integer | p = 500 )
endpackage
```

**Listing 9.1:** A OCL Specication (Excerpt of the Royals and Loyals Model)

1. Modelling the data model (based on the diagrammatic UML notions) using a CASE tool, e.g., ArgoUML (see Figure 9.2). We assume that the reader is familiar with the CASE tool he is using. The result of this process should be an XMI file that can be imported into the Dresden OCL 2 toolkit. Our tool-chain requires the following organization of your model:

   - The complete model should be placed into a top-level UML package. This package is ignored by the Dresden OCL 2 Toolkit.
   - Within the top-level package you can specify arbitrary packages for your models.
   - The top-level package should contain a UML package called UML_OCL that contains all OCL datatypes (i.e., the OCL library). All basic datatypes, e.g., Integer, should be taken from this package.

Please have a look at the examples provided by the HOL-OCL distribution for details.

2. Modelling and type-checking the OCL specification. The Dresden OCL 2 Toolkit uses a simple text format for the OCL specification, e.g., see Item 9.1 for an example. The type-checker (included in the “Parser Tool”) of the Dresden OCL 2 toolkit can now be used to type-check such a OCL specification against a given data model. This can be either done using a graphical interface (see Figure 9.3) in the following way:

   a) Click on the “Model” tab and load your UML model by clicking on “Load XMI” (either stored as XMI or ArgoUML’s native format (.zargo).
   b) Click on the “Constraint” tab and load your OCL specification.

Figure 9.3.: Using the Dresden OCL Parser and Type-checker

c) Click on the “CST” tab and click on “Parse(CS)”. This will parse the concrete syntax of your OCL specification, i.e., up to now, no type-checking is involved.

d) If the last step succeeds, i.e., you see a parse tree of the concrete syntax, click on the “ASTGen log” tab and click on “Generate”. This will generate type-check your OCL specification.

e) If the type-checking succeeds without errors, you can export (by clicking on “Export XMI”) a XMI file containing both, the UML and OCL part of your model. This XMI file can be loaded into HOL-OCL.

Alternatively, you can type-check your models on the command-line using the following syntax:

```
java -cp ocl20parsertools.jar tudresden.ocl20.core.parser.OCL20CLI \
   --outfile output.xmi input.xmi input.ocl
```

where input.xmi is your data model (either in XMI format or ArgoUML’s native format), input.ocl is your OCL specification, and output.xmi the file where (after successful type-checking) the output is written to.

9.3. Using HOL-OCL: The Verification Phase

In this section we give a brief overview of HOL-OCL related extension of the Isar proof language. We also use a presentation similar to the one in the Isar Reference
9.3. Using HOL-OCL: The Verification Phase

theory royals_and_loyals
imports
   OCL
begin
   load_xmi "royals_and_loyals_ocl.xmi"
end

Listing 9.2: A simple HOL-OCL Theory File

Manual [53], e.g. “missing” non-terminals (e.g., (goalspec)) of our syntax diagrams are defined in [53].

9.3.1. Getting Started

For using HOL-OCL you have to build your Isabelle theories (i.e. test specifications) on top of the theory UML_OCL (plain UML/OCL) or OCL (UML/OCL with extensions) instead of Main. A sample theory is shown in Listing 9.2.

9.3.2. Loading XMI files

A well-typed UML/OCL models can be imported using the load_xmi command, e.g.:

load_xmi "royals_and_loyals_ocl.xmi"

9.3.3. Canonizing Hypotheses

Similar to hypsubst_tac on the HOL level, HOL-OCL provides ocl_hypsubst_tac for the elimination of variables $A$ within OCL equality assumptions. The format of these equalities is quite general here, a number of equalities stated implicitly is also handled:

1. $A = t$ and $t = A$ (as in standard hyp_subst_tac),
2. $A \tau = t \tau$, $t \tau = A \tau$ (local congruences),
3. $\tau \vdash \emptyset A$, $\vdash \emptyset A$ (local and global undefinedness),
4. $\tau \vdash \neg A$, $\vdash \neg A$ (local and global falsities),
5. $\tau \vdash A \triangleq t$, $\tau \vdash t \triangleq A$, $\vdash A \triangleq t$, $\vdash t \triangleq A$ (local and global strong equalities), and
6. $\tau \vdash A \triangleq t$, $\tau \vdash A \equiv t \equiv A \equiv t$ (local and global strict equalities).

The syntax for the command reads as follows:

ocl_hypsubst_tac $\setminus (goalspec) \vdash f$.

The optional goal specification allows for
9.3.4. One-Step-Rewriting

The major limitation of canonization as described in the previous section is that it is restricted to variables - on the other hand, the canonization can apply an equality from left to right and from right to left. In contrast, rewriting (actually in this context: narrowing) proceeds only from left to right, but admitting simplification of non-variable terms.

The possible formats of implicit equalities look similarly as the list in ocl_hypsubst:

1. \( \tau \models \partial \text{lhs}, \models \partial \text{lhs} \) (local and global undefinedness; replaces lhs by \( \bot \)),
2. \( \tau \models \partial \text{lhs}, \models \partial \text{lhs} \) (local and global definedness; replaces \( \partial \text{lhs} \) by \( T \)),
3. \( \tau \models \neg \text{lhs}, \models \neg \text{lhs} \) (local and global falsities; replaces \( \partial \text{lhs} \) by \( F \)),
4. \( \tau \models \text{lhs} \models \text{lhs} \) (local and global validities; replaces \( \text{lhs} \) by \( T \)),
5. \( \text{lhs} \tau = t \tau \) (local congruences),
6. \( \models \text{lhs} \models t \) (local and global strong equalities), and
7. \( \tau \models \text{lhs} \models t \) (local and global strict equalities).

The syntax for the command reads as follows:

\[
\text{ocl_subst} \quad \ldots
\]

With the attribute specification, the range of the possible rewrite-step can be limited. Specifying \texttt{no_asm_use} excludes assumptions in a goal from rewriting. \texttt{no_asm_simp} excludes them from being rewritten. By \texttt{no_asm_concl}, the rewriting of the conclusion can be prevented. The method accepts a direct specification of a goal to be rewritten, and a list of theorems that can be used for rewriting (possibly empty).

The method fails if no redex can be found; thus, by using the repetition operator, a simple exhaustive rewriting process can be formulated.

9.3.5. Automated Proof-Procedures

As an equivalent to auto ...

\[
\text{ocl_auto} \quad \ldots
\]
9.4. Extending HOL-OCL

In this section we explain commands that are useful for extending HOL-OCL, i.e., they are more thought for developers than for users.

Exploiting theory morphisms: The theory lifting mechanism is used on a “per operator” basis, e.g.:

\[ \text{ocl_setup_op [OclIncludes , OclExcludes]} \]

The procedure performs at present no sophisticated ocl-specific machinery.
Chapter 10.
Case Studies

10.1. Encoding a Stack in UML/OCL

theory stack
imports OCL
begin

We use the class diagram presented in Fig. 10.1 together with the following OCL formulae shown in Listing 10.1.

load_xmi stack_ocl.xmi

An interesting system property would be

\[ \forall s0. (\text{push } s0 \text{ e0 } s1) \implies ((\text{top } s1 \text{ e1}) \land (e1 \equiv e0)) \]

or

\[ \forall s0. \text{size } s0 \geq \text{size } \text{pop } s0 \]

Note that such system properties specified in HOL-OCL allow the use of operations that are not side-effect free (e.g. push)

end

Figure 10.1.: Modelling a Stack: Data Model
package Stack

context Stack::pop(): Stack
    pre notEmpty: isEmpty() = false
    post topElementRemoved: top() <> self.top@pre()
    post: elements = elements@pre->subSequence(2, self@pre->size())
    post: elements->size() = elements@pre->size() -1

class

context Stack::top(): ElemType
    pre notEmpty: isEmpty() = false
    post: result = elements->first()
    post: self = self@pre

class

context Stack::push(e: ElemType): Stack
    post pushedElemTypeIsOnTop: top() = e
    post: elements = elements@pre->prepend(e)
    post: result = self

context Stack::isEmpty(): Boolean
    post: result = (elements->size() <> 0)

endpackage

Listing 10.1: Modelling a Stack: OCL specification
Part IV.

Appendix
Appendix A.

The Syntax of OCL

OCL, being advertised with the slogan “Mathematical Foundation, But No Mathematical Symbols” [52], is normally written using a concrete syntax that is inspired by object-oriented programming languages. To give a first impression of this syntax to readers unfamiliar with it, we present a core fragment of OCL using Extended Backus-Naur Form (EBNF) notation (see Tab. A.1). This fragment in particular omits many syntactic variants resulting from naming expressions. This simplified concrete syntax used to denote our examples contains some redundancies: the variant `expr->simpleName` is semantically equivalent with dereferencing `expr.simpleName`, it is a tribute to the OCL convention to distinguish the application of operations on collections such as `X->union(Y)`. In principle, this also holds for the prefix and infix operators. However, the semantics of this applications may be call-by-name or call-by-need; this is handled for each operator individually.

Whereas this textual notation pleases the people coming from object-oriented programming languages, it looks awkward for people coming from the mathematics and formal methods field. Especially for proof work, there seems a need for a compact, mathematical notation. Thus we developed a mathematics-oriented OCL syntax, as an alternative to the programming-language like notation used in the OCL 2.0 standard.

For example, compare the textual presentation of the proof rule:

\[
\begin{align*}
\tau \models S->\text{includes}(x) & \quad \tau \models \text{not}(P \; x) \quad \text{cp} \; P \\
\tau \models (S->\text{forall}(x \mid P(x)))->IsDefined() 
\end{align*}
\] (A.1)

to its presentation in mathematical notation:

\[
\begin{align*}
\tau \models x \in S & \quad \tau \models \neg(P \; x) \quad \text{cp} \; P \\
\tau \models \partial(\forall \; x \in S. P(x)) 
\end{align*}
\] (A.2)

Clearly, both syntax’s have their advantages and disadvantages and therefore we support both of them in HOL-OCL.

In Table A.2 we provide a brief comparison between the different concrete OCL syntax’s, namely the syntax as proposed in the OCL standard, our textual notation that tries to follows the standard syntax as close as possible, and finally our new mathematical syntax. The table follows the OCL library presentation from the standard [41, Chapter 11], constructs that are not supported by HOL-OCL are written in a gray typeface, e.g. `ooclIsInState(s)`.
Appendix A. The Syntax of OCL

\[
\text{invSpec ::= context pathName inv : expr}
\]

\[
\text{opSpec ::= context operation pre : expr post : expr}
\]

\[
\text{operation ::= [pathName ::] NAME ( [varDecl {}, varDecl] ] [: type]}
\]

\[
\text{varDecl ::= NAME [: type] [= expr]}
\]

\[
\text{type ::= pathName | collKind ( type )}
\]

\[
\text{expr ::= literal | − expr | not expr | expr infixOp expr}
\]

\[
\text{| pathName [@pre] | expr.NAME [@pre] | expr − > NAME}
\]

\[
\text{| expr([expr,]expr]) | expr(varDecl:expr) |}
\]

\[
\text{| expr − > bindOp(varDecl:[varDecl]) expr |}
\]

\[
\text{| if expr then expr else expr endif |}
\]

\[
\text{| let varDecl {}, varDecl in expr |}
\]

\[
\text{infixOp ::= * | / | div | mod | + | − | < | > | <= | >= | = | <> | and | or | xor | implies |}
\]

\[
\text{bindOp ::= iterate | forall | exists |}
\]

\[
\text{literal ::= Integer | Real | String | true | false | OclUndefined |}
\]

\[
\text{| collKind{[collLitPart,}collLitPart} |}
\]

\[
\text{collKind ::= Set | Bag | Sequence | Collection | OrderedSet |}
\]

\[
\text{collLitPart ::= expr | expr..expr |}
\]

\[
\text{pathName ::= [pathName::]NAME}
\]

**Table A.1.:** Formal Grammar of OCL (fragment)
### Table A.2.: Comparison of different concrete syntax variants for OCL

<table>
<thead>
<tr>
<th>OCL (standard)</th>
<th>mathematical HOL-OCL</th>
<th>textual HOL-OCL</th>
</tr>
</thead>
<tbody>
<tr>
<td>( x = y )</td>
<td>( x \equiv y )</td>
<td>( x == y )</td>
</tr>
<tr>
<td>( x &lt;&gt; y )</td>
<td>( x \neq y )</td>
<td>( x \neq y )</td>
</tr>
<tr>
<td>( x \triangle y )</td>
<td>( x \triangleq y )</td>
<td>( x \triangleq y )</td>
</tr>
<tr>
<td>( ooclIsNew() )</td>
<td>( \varnothing o )</td>
<td>( ooclIsNew() )</td>
</tr>
<tr>
<td>( ooclIsUndefined() )</td>
<td>( \bot )</td>
<td>( ooclIsUndefined() )</td>
</tr>
<tr>
<td>( ooclAsType(t) )</td>
<td>( o\rightarrow\text{AsType}(t) )</td>
<td>( o\rightarrow\text{AsType}(t) )</td>
</tr>
<tr>
<td>( ooclIsType(t) )</td>
<td>( o\rightarrow\text{IsType}(t) )</td>
<td>( o\rightarrow\text{IsType}(t) )</td>
</tr>
<tr>
<td>( ooclIsKindOf(t) )</td>
<td>( o\rightarrow\text{IsKindOf}(t) )</td>
<td>( o\rightarrow\text{IsKindOf}(t) )</td>
</tr>
<tr>
<td>( ooclIsKindOf(s) )</td>
<td>( o\rightarrow\text{AllInstances()} )</td>
<td>( o\rightarrow\text{AllInstances()} )</td>
</tr>
<tr>
<td>( o\rightarrow\text{HasReturned()} )</td>
<td>( o\rightarrow\text{Result()} )</td>
<td>( o\rightarrow\text{IsDefined()} )</td>
</tr>
<tr>
<td>( o\rightarrow\text{IsSignalSent()} )</td>
<td>( o\rightarrow\text{IsOperationCall()} )</td>
<td>( o\rightarrow\text{IsDefined()} )</td>
</tr>
<tr>
<td>( x + y )</td>
<td>( x + y )</td>
<td>( x + y )</td>
</tr>
<tr>
<td>( x - y )</td>
<td>( x - y )</td>
<td>( x - y )</td>
</tr>
<tr>
<td>( x \ast y )</td>
<td>( x \ast y )</td>
<td>( x \ast y )</td>
</tr>
<tr>
<td>(-x)</td>
<td>(-x)</td>
<td>(-x)</td>
</tr>
<tr>
<td>( x / y )</td>
<td>( x/y )</td>
<td>( x/y )</td>
</tr>
<tr>
<td>( x.abs() )</td>
<td>(</td>
<td>x</td>
</tr>
<tr>
<td>( x.floor() )</td>
<td>( \lfloor x \rfloor )</td>
<td>( \lfloor x \rfloor )</td>
</tr>
<tr>
<td>( x.max(y) )</td>
<td>( \max(x, y) )</td>
<td>( \max(x, y) )</td>
</tr>
<tr>
<td>( x.min(y) )</td>
<td>( \min(x, y) )</td>
<td>( \min(x, y) )</td>
</tr>
<tr>
<td>( x &lt; y )</td>
<td>( x &lt; y )</td>
<td>( x &lt; y )</td>
</tr>
<tr>
<td>( x &gt; y )</td>
<td>( x &gt; y )</td>
<td>( x &gt; y )</td>
</tr>
<tr>
<td>( x &lt;= y )</td>
<td>( x \leq y )</td>
<td>( x \leq y )</td>
</tr>
<tr>
<td>( x &gt;= y )</td>
<td>( x \geq y )</td>
<td>( x \geq y )</td>
</tr>
</tbody>
</table>

Continued on next page
## Appendix A. The Syntax of OCL

<table>
<thead>
<tr>
<th>OCL</th>
<th>mathematical HOL-OCL</th>
<th>textual HOL-OCL</th>
</tr>
</thead>
<tbody>
<tr>
<td>x.div(y)</td>
<td>(x \div y)</td>
<td>(x \rightarrow \text{div}(y))</td>
</tr>
<tr>
<td>x.mod(y)</td>
<td>(x \mod y)</td>
<td>(x \rightarrow \text{mod}(y))</td>
</tr>
<tr>
<td>x.max(y)</td>
<td>(\max(x, y))</td>
<td>(x \rightarrow \text{max}(y))</td>
</tr>
<tr>
<td>x.min(y)</td>
<td>(\min(x, y))</td>
<td>(x \rightarrow \text{min}(y))</td>
</tr>
<tr>
<td>s.size()</td>
<td>(</td>
<td>s</td>
</tr>
<tr>
<td>s.concat(z)</td>
<td>(s \concat z)</td>
<td>(s \rightarrow \text{concat}(z))</td>
</tr>
<tr>
<td>s.substring(i,j)</td>
<td>(s \text{substring}(i,j))</td>
<td>(s \rightarrow \text{substring}(i,j))</td>
</tr>
<tr>
<td>s.toInteger()</td>
<td>(s \toInteger())</td>
<td>(s \rightarrow \text{toInteger}())</td>
</tr>
<tr>
<td>s.toReal()</td>
<td>(s \toReal())</td>
<td>(s \rightarrow \text{toReal}())</td>
</tr>
<tr>
<td>s.toUpper()</td>
<td>(s \text{toUpper}())</td>
<td>(s \rightarrow \text{toUpper}())</td>
</tr>
<tr>
<td>s.toLower()</td>
<td>(s \text{toLower}())</td>
<td>(s \rightarrow \text{toLower}())</td>
</tr>
<tr>
<td>(\equiv p)</td>
<td>(p \equiv \top)</td>
<td>(\top)</td>
</tr>
<tr>
<td>(\tau \equiv p)</td>
<td>(p \equiv \bot)</td>
<td>(\bot)</td>
</tr>
<tr>
<td>true</td>
<td>(\top)</td>
<td>(\top)</td>
</tr>
<tr>
<td>false</td>
<td>(\bot)</td>
<td>(\bot)</td>
</tr>
<tr>
<td>x or y</td>
<td>(x \lor y)</td>
<td>(x \lor y)</td>
</tr>
<tr>
<td>x xor y</td>
<td>(x \oplus y)</td>
<td>(x \oplus y)</td>
</tr>
<tr>
<td>x and y</td>
<td>(x \land y)</td>
<td>(x \land y)</td>
</tr>
<tr>
<td>not x</td>
<td>(\neg x)</td>
<td>(\neg x)</td>
</tr>
<tr>
<td>x implies y</td>
<td>(x \rightarrow y)</td>
<td>(x \implies y)</td>
</tr>
<tr>
<td>x not implies y</td>
<td>(x \not\rightarrow y)</td>
<td>(x \not\implies y)</td>
</tr>
</tbody>
</table>

### Boolean

- \(\equiv p\) → Valid \(p\)
- \(\tau \equiv p\) → Valid \(p\)
- \(\top\) → \(\top\)
- \(\bot\) → \(\bot\)
- \(\lor\) → \(\lor\)
- \(\oplus\) → \(\oplus\)
- \(\land\) → \(\land\)
- \(\neg\) → \(\neg\)
- \(\rightarrow\) → \(\implies\)
- \(\not\rightarrow\) → \(\not\implies\)

### Collection

- \(\exists e \in X.P(e)\) → \(\exists e : T | P(e)\)
- \(\forall e \in X.P(e)\) → \(\forall e : T | P(e)\)
- \(\text{isUnique}(e : T | P(e))\) → \(\text{isUnique}(e : T | P(e))\)
- \(\text{any}(e : T | P(e))\) → \(\text{any}(e : T | P(e))\)
- \(\text{one}(e : T | P(e))\) → \(\text{one}(e : T | P(e))\)
- \(\text{collect}(e : T | P(e))\) → \(\{e \in X | P(e)\}\)

Continued on next page
<table>
<thead>
<tr>
<th>OCL</th>
<th>mathematical HOL-OCL</th>
<th>textual HOL-OCL</th>
</tr>
</thead>
<tbody>
<tr>
<td>Set{}</td>
<td>{}</td>
<td>( )</td>
</tr>
<tr>
<td>( X \rightarrow \text{union}(Y) )</td>
<td>( X \cup Y )</td>
<td>( X \rightarrow \text{union}(Y) )</td>
</tr>
<tr>
<td>( X \equiv Y )</td>
<td>( X \equiv Y )</td>
<td>( X \equiv Y )</td>
</tr>
<tr>
<td>( X \rightarrow \text{intersection}(Y) )</td>
<td>( X \cap Y )</td>
<td>( X \rightarrow \text{intersection} )</td>
</tr>
<tr>
<td>( X \rightarrow \text{complement}(Y) )</td>
<td>( X^{-1} )</td>
<td>( X \rightarrow \text{complement}() )</td>
</tr>
<tr>
<td>( X \rightarrow \text{including}(y) )</td>
<td>( y \in X )</td>
<td>( X \rightarrow \text{includes}(y) )</td>
</tr>
<tr>
<td>( X \rightarrow \text{excluding}(y) )</td>
<td>( y \notin X )</td>
<td>( X \rightarrow \text{excludes}(y) )</td>
</tr>
<tr>
<td>( X \rightarrow \text{symmetricDifference}(Y) )</td>
<td>( X \ominus Y )</td>
<td>( X \rightarrow \text{symmetricDifference}(Y) )</td>
</tr>
<tr>
<td>( X \rightarrow \text{count}(y) )</td>
<td>( #X )</td>
<td>( X \rightarrow \text{count}() )</td>
</tr>
<tr>
<td>( X \rightarrow \text{flatten}() )</td>
<td>![X]!</td>
<td>( X \rightarrow \text{flatten}() )</td>
</tr>
<tr>
<td>OrderedSet{}</td>
<td>( )</td>
<td>OrderedSet{}</td>
</tr>
<tr>
<td>( X \rightarrow \text{append}(y) )</td>
<td>( X : y )</td>
<td>( X \rightarrow \text{append}(y) )</td>
</tr>
<tr>
<td>( X \rightarrow \text{prepend}(y) )</td>
<td>( y : X )</td>
<td>( X \rightarrow \text{prepend}(y) )</td>
</tr>
<tr>
<td>( X \rightarrow \text{insertAt}(i, y) )</td>
<td>( X \rightarrow \text{insertAt}(i, y) )</td>
<td>( X \rightarrow \text{insertAt}(i, y) )</td>
</tr>
<tr>
<td>( X \rightarrow \text{subOrderedSet}(i, j) )</td>
<td>( X \rightarrow \text{subOrderedSet}(i, j) )</td>
<td>( X \rightarrow \text{subOrderedSet}(i, j) )</td>
</tr>
<tr>
<td>( X \rightarrow \text{at}(i) )</td>
<td>( ^i X )</td>
<td>( X \rightarrow \text{at}(i) )</td>
</tr>
<tr>
<td>( X \rightarrow \text{indexOf}(y) )</td>
<td>( \textbf{?} X )</td>
<td>( X \rightarrow \text{indexOf}(y) )</td>
</tr>
<tr>
<td>( X \rightarrow \text{first}() )</td>
<td>( ^1 X )</td>
<td>( X \rightarrow \text{first}() )</td>
</tr>
<tr>
<td>( X \rightarrow \text{last}() )</td>
<td>( ^$ X )</td>
<td>( X \rightarrow \text{last}() )</td>
</tr>
<tr>
<td>Bag{}</td>
<td>( )</td>
<td>Bag{}</td>
</tr>
<tr>
<td>( X \equiv Y )</td>
<td>( X \equiv Y )</td>
<td>( X \rightarrow \text{union}(Y) )</td>
</tr>
<tr>
<td>( X \rightarrow \text{complement}(Y) )</td>
<td>( X^{-1} )</td>
<td>( X \rightarrow \text{complement}() )</td>
</tr>
<tr>
<td>( X \rightarrow \text{including}(y) )</td>
<td>( y \in X )</td>
<td>( X \rightarrow \text{includes}(y) )</td>
</tr>
<tr>
<td>( X \rightarrow \text{excluding}(y) )</td>
<td>( y \notin X )</td>
<td>( X \rightarrow \text{excludes}(y) )</td>
</tr>
<tr>
<td>( X \rightarrow \text{count}(y) )</td>
<td>( #X )</td>
<td>( X \rightarrow \text{count}() )</td>
</tr>
<tr>
<td>( X \rightarrow \text{flatten}() )</td>
<td>![X]!</td>
<td>( X \rightarrow \text{flatten}() )</td>
</tr>
<tr>
<td>Bag{}</td>
<td>( )</td>
<td>Bag{}</td>
</tr>
<tr>
<td>( X \rightarrow \text{append}(y) )</td>
<td>( X : y )</td>
<td>( X \rightarrow \text{append}(y) )</td>
</tr>
<tr>
<td>( X \rightarrow \text{prepend}(y) )</td>
<td>( y : X )</td>
<td>( X \rightarrow \text{prepend}(y) )</td>
</tr>
<tr>
<td>( X \rightarrow \text{insertAt}(i, y) )</td>
<td>( X \rightarrow \text{insertAt}(i, y) )</td>
<td>( X \rightarrow \text{insertAt}(i, y) )</td>
</tr>
<tr>
<td>( X \rightarrow \text{subOrderedSet}(i, j) )</td>
<td>( X \rightarrow \text{subOrderedSet}(i, j) )</td>
<td>( X \rightarrow \text{subOrderedSet}(i, j) )</td>
</tr>
<tr>
<td>( X \rightarrow \text{at}(i) )</td>
<td>( ^i X )</td>
<td>( X \rightarrow \text{at}(i) )</td>
</tr>
<tr>
<td>( X \rightarrow \text{indexOf}(y) )</td>
<td>( \textbf{?} X )</td>
<td>( X \rightarrow \text{indexOf}(y) )</td>
</tr>
<tr>
<td>( X \rightarrow \text{first}() )</td>
<td>( ^1 X )</td>
<td>( X \rightarrow \text{first}() )</td>
</tr>
<tr>
<td>( X \rightarrow \text{last}() )</td>
<td>( ^$ X )</td>
<td>( X \rightarrow \text{last}() )</td>
</tr>
</tbody>
</table>

Continued on next page
### Appendix A. The Syntax of OCL

<table>
<thead>
<tr>
<th>OCL</th>
<th>mathematical HOL-OCL</th>
<th>textual HOL-OCL</th>
</tr>
</thead>
<tbody>
<tr>
<td>Sequence()</td>
<td>[]</td>
<td>[]</td>
</tr>
<tr>
<td>X-&gt;count()</td>
<td>X-&gt;count(y)</td>
<td>X-&gt;count(y)</td>
</tr>
<tr>
<td>X = Y</td>
<td>X = Y</td>
<td>X = Y</td>
</tr>
<tr>
<td>X-&gt;union(Y)</td>
<td>X ∪ Y</td>
<td>X-&gt;union(Y)</td>
</tr>
<tr>
<td>X-&gt;flatten()</td>
<td>[]</td>
<td>X-&gt;flatten()</td>
</tr>
<tr>
<td>X-&gt;append(y)</td>
<td>X :: y</td>
<td>X-&gt;append y</td>
</tr>
<tr>
<td>X-&gt;prepend(y)</td>
<td>y : X</td>
<td>X-&gt;prepend y</td>
</tr>
<tr>
<td>X-&gt;insertAt(i,y)</td>
<td>X-&gt;insertAt(i,y)</td>
<td>X-&gt;insertAt(i,y)</td>
</tr>
<tr>
<td>X-&gt;subSequence(i,j)</td>
<td>X-&gt;subSequence(i,j)</td>
<td>X-&gt;subSequence(i,j)</td>
</tr>
<tr>
<td>X-&gt;at(i)</td>
<td>i X</td>
<td>X-&gt;at(i)</td>
</tr>
<tr>
<td>X-&gt;indexOf(y)</td>
<td>X (? y)</td>
<td>X-&gt;indexOf(y)</td>
</tr>
<tr>
<td>X-&gt;first()</td>
<td>1 X</td>
<td>X-&gt;first()</td>
</tr>
<tr>
<td>X-&gt;last()</td>
<td>$ X</td>
<td>X-&gt;last()</td>
</tr>
<tr>
<td>X-&gt;including(y)</td>
<td>y ∈ X</td>
<td>X-&gt;includes(y)</td>
</tr>
<tr>
<td>X-&gt;excluding(y)</td>
<td>y ∉ X</td>
<td>X-&gt;excludes(y)</td>
</tr>
<tr>
<td>X-&gt;asBag()</td>
<td>X-&gt;asBag()</td>
<td>X-&gt;asBag()</td>
</tr>
<tr>
<td>X-&gt;asSequence()</td>
<td>X-&gt;asSequence()</td>
<td>X-&gt;asSequence()</td>
</tr>
<tr>
<td>X-&gt;asSet()</td>
<td>X-&gt;asSet()</td>
<td>X-&gt;asSet()</td>
</tr>
<tr>
<td>X-&gt;asOrderedSet()</td>
<td>X-&gt;asOrderedSet()</td>
<td>X-&gt;asOrderedSet()</td>
</tr>
<tr>
<td>X-&gt;select(e:T</td>
<td>P(e))</td>
<td>{ e ∈ X</td>
</tr>
<tr>
<td>X-&gt;reject(e:T</td>
<td>P(e))</td>
<td>{ e ∈ X</td>
</tr>
<tr>
<td>X-&gt;collectNested(e:T</td>
<td>P(e))</td>
<td>{ { e ∈ X</td>
</tr>
<tr>
<td>X-&gt;sortedBy(e:T</td>
<td>P(e))</td>
<td>X-&gt;sortedBy(e : T</td>
</tr>
</tbody>
</table>

let e=x in P(s) end

\[
\begin{bmatrix}
  x_1 \\
  x_2
\end{bmatrix}
\]
Appendix B.

Isabelle Theories

B.1. Introduction

In this chapter we present the Isabelle theories of HOL-OCL. The main dependencies are shown in Fig. B.1 on the next page. HOL-OCL is build on top of Isabelle[HOL]. This chapter was automatically generated by Isabelle, i.e., it is a formal document: all definitions and theorems presented here were checked and formally proven by Isabelle. Nevertheless, we took the privilege of omitting theory some parts, in particular part’s concerning the setup of the syntax engine, the interface to Isar the implementation of several tactics, the lifter, and the datatype package. Thus, in this chapter we present the semantic foundation of HOL-OCL. If you are also interested in the technical details of the implementation, please read the source code which is included in the HOL-OCL distribution.

B.1.1. HOL-OCL Configurations

Note, HOL-OCL can be built in different configurations, e.g., one can choose between non-referential and referential universes and between smashed and unsmashed collection types:

Universe: One issue to be raised here is the semantics of equality; are two objects equal only if their object identifier is equal or are two objects equal if their values are equal? The OCL semantics is not specific here since equality is defined as equality over values [41, Sec. A.2.2], and since objects are values, but object identifiers are not distinguished from object values [41, Definition A.10]. However, since many object-oriented programming languages are centered around referential equality (which gave the motivation to opt for the latter in [7]), HOL-OCL can be configured such that the above definition leads to referential equality.

- **non-referential**: here objects are just identified with their values, i.e., two objects are equal if and only if all their attributes are equal.

- **referential universe**: here a referential equality is present, which is is the strong equality for “boxed types” (that is Real, Boolean, String) and an equality on the reference to a value.

---

1 More precise, HOL-OCL is build on top of Isabelle/HOL/Complex but only to support real numbers. If you don’t need them, HOL-OCL should be build just fine on top of a plain Isabelle/HOL.
Figure B.1: Session Graph
B.2. Overview

This manual describes HOL-OCL with referential universes. We recommend the use of referential universes.

This option has effects on the theory OCL_OclAny_type.

Collections: [OCL [41] Sec. A.2.5.2] allows for collections to include ⊥, i.e. the constructor of sets and the membership tests are non-strict. This has several undesired consequences for executability and proof support. Thus we provide two collection libraries:

smashed collections: smashing data-structures is a key-concept in denotational semantics [35] [54]. For example, pairs are smashed if (a, ⊥) is identified with ⊥ as in e.g. Java or SML, or, with respect to sets, \{a, ⊥\} = ⊥.

unsmashed collections: In this configuration all collection types can contain the element ⊥.

This manual describes HOL-OCL with smashed collections. We recommend the use of smashed collections.

This option has effects on the following theories: OCL_Bag.thy, OCL_OrderedSet.thy, OCL_Set.thy, OCL_Bag_type.thy, OCL_OrderedSet_type.thy, OCL_Set_type.thy, OCL_CharacteristicSet.thy, OCL_Sequence.thy, OCL_Collection_requirements.thy, and OCL_Sequence_type.thy.

The “recommended configuration” is more developed and contains more deductive support.

B.1.2. Notational Remarks

For presentational reasons, we used in the previous chapters for a few constructs a “Formal Syntax” that looks somewhat more “mathematical” than the syntax of Isabelle. To help readers that are not familiar with the syntax of Isabelle/HOL we compare the (already introduced) “Formal Syntax” with the Isabelle syntax used in this chapter in Table B.1.

B.2. Overview

The remainder of this chapter are the Isabelle theories using the \LaTeX-based presentation facilities of Isabelle. We structured the theory presentation in this chapter into the following subsections:

Foundations: In this section, we introduce the foundations of HOL-OCL, namely the concept of lifting and the basic datatype definitions.

Library: In this section, we introduce the OCL datatypes presented in [41] Chapter 11 and prove basic properties over them. Especially, we prove that our definitions fulfill the requirements presented in the OCL standard [41] Chapter 11.
Appendix B. Isabelle Theories

<table>
<thead>
<tr>
<th>Formal Syntax</th>
<th>Isabelle Syntax</th>
</tr>
</thead>
<tbody>
<tr>
<td>[ A_1 \ldots A_n \Rightarrow C ]</td>
<td>([A_1; \ldots; A_n] \Rightarrow C)</td>
</tr>
</tbody>
</table>
| \(\text{sumCase}(f, g, x) = \begin{cases} 
  f(k) & \text{if } x = \text{Inl } k \\
  g(k) & \text{if } x = \text{Inr } k 
\end{cases}\) | \(\text{case } x \text{ of } \text{Inl } k \Rightarrow f(k) | \text{Inr } k \Rightarrow g(k)\) |
| \(\text{upCase}(f, c, x) = \begin{cases} 
  c & \text{if } x = \bot \\
  f(k) & \text{if } x = k
\end{cases}\) | \(\text{case } x \text{ of } k \Rightarrow f(k) | \bot \Rightarrow c\) |
| \(V_\tau(\alpha)\) | \((\tau, \alpha) \text{ VAL semfun } \) |
| \(\text{Sem}[(_)]\) | \(|\) |

Table B.1.: Comparison of the formal syntax and the Isabelle syntax

**State:** In this section, the concept of state and constructive concepts like proof calculi are presented.

**Requirements:** In this section, the proof (or disproof) of the requirements of [41, Chapter 11].

**OCL:** This section just contains the final OCL theory that should be used as basis for case studies, i.e., it it the HOL-OCL main theory.

### B.3. Foundations

#### B.3.1. The Theory of Lifting and its Combinators

```isabelle
theory Lifting
imports Main
begin

Besides the infrastructure for handling undefinedness, this theory also provides the basis for data type embedding smashing and context lifting including the generic core of theorems.

The main purpose of this theory is to provide a generic theory of undefinedness. The Isabelle standard mechanism for such a generic data type is the class mechanism. The standard method is to declare a class to which all related operations are associated.

A Generic Theory of Undefinedness

Since the very first OCL publications [39] explicit undefinedness is part of the language, both for the logic and the basic values, e.g., [39, 7.4.9 Undefined Values] states (this is similarly stated in [41, page 15]):

138
Whenever an OCL expression is being evaluated, there is a possibility that one or more of the queries in the expression are undefined. If this is the case, then the complete expression will be undefined.

There are two exceptions to this for the Boolean operators

- True OR-ed with anything is True
- False AND-ed with anything is False

The above two rules are valid irrespective of the order of the arguments and the above rules are valid whether or not the value of the other sub-expression is known.

(OCL Specification [39], page 7-11)

This requirement postulates the strictness of all operations (except for the logic) and rules out a modeling of undefinedness via Hilbert-Operators and underspecification (e.g., as done for Z [26] in Isabelle/HOL-Z [11]).

All bottom types possess a constant \( \bot \)

- const :: \( \alpha :: \text{bot} \)
- syntax :: \( \alpha :: \text{bot} \) (\( \bot \))

- axclass :: \( \text{bot} \subseteq \text{bot0} \)
- nonEmpty :: \( \exists x. x \neq \bot \)

- constdefs :: \( \text{DEF} : \alpha :: \text{bot} \Rightarrow \text{bool} \)
- constdefs :: \( \text{DEF} x \equiv (x \neq \bot) \)
- constdefs :: \( \text{isStrict} : (\alpha :: \text{bot} \Rightarrow \beta :: \text{bot}) \Rightarrow \text{bool} \)
- constdefs :: \( \text{isStrict} f \equiv (f \bot = \bot) \)
- constdefs :: \( \text{strictify} : (\alpha :: \text{bot} \Rightarrow \beta :: \text{bot}) \Rightarrow \alpha \Rightarrow \beta \)
- constdefs :: \( \text{strictify} f x \equiv \text{if } x = \bot \text{ then } \bot \text{ else } f x \)
- constdefs :: \( \text{smash} : ([\beta :: \text{bot}, \alpha :: \text{bot}] \Rightarrow \text{bool}, \alpha] \Rightarrow \alpha \)
- constdefs :: \( \text{smash} f X \equiv \text{if } f \bot X \text{ then } \bot \text{ else } X \)

**Semantic Constructions: Liftings, Functions, Products**

We introduce now the lifting construction (see Winskel [54, p.131]) by a type constructor defined as free data type:

- datatype :: \( \uparrow \text{up} = \text{lift} \uparrow \text{bot} | \text{down} \)

- constdefs :: \( \text{drop} : \uparrow \text{up} \Rightarrow \uparrow \alpha \)
Appendix B. Isabelle Theories

\[ \text{drop } x \equiv \text{case } x \text{ of lift } v \Rightarrow v \mid \text{down} \Rightarrow \epsilon \ x \ . \ True \]

**syntax**
\[
\@lift \quad :: \:'\alpha \Rightarrow \:'\alpha \uparrow (\{\_\},\}) \\
Lifting\,\text{drop} \quad :: \:'\alpha \uparrow \Rightarrow \:'\alpha \ (\{\_\},\})
\]

**syntax (xsymbols)**
\[
\@lift \quad :: \:'\alpha \Rightarrow \:'\alpha \uparrow (\_\) \\
Lifting\,\text{drop} \quad :: \:'\alpha \uparrow \Rightarrow \:'\alpha \ (\_\)
\]

**translations**
\[
|\cdot \cdot | \equiv \text{lift } a \\
\|a\| \equiv \text{drop } a
\]

**instance**
\[
\text{up} :: (\text{zero}) \text{ zero} \\
\quad \text{by intro\_classes} \\
\text{instance} \quad \text{up} :: (\text{one}) \text{ one} \\
\quad \text{by intro\_classes} \\
\text{instance} \quad \text{up} :: (\text{plus}) \text{ plus} \\
\quad \text{by intro\_classes} \\
\text{instance} \quad \text{up} :: (\text{minus}) \text{ minus} \\
\quad \text{by intro\_classes} \\
\text{instance} \quad \text{up} :: (\text{times}) \text{ times} \\
\quad \text{by intro\_classes} \\
\text{instance} \quad \text{up} :: (\text{inverse}) \text{ inverse} \\
\quad \text{by intro\_classes}
\]

**Semantic Constructions: Liftings of Type Constructors**
The class is then propagated across lifting, function space and cartesian products.

**instance**
\[
\text{up} :: (\text{type}) \text{ ord} \\
\quad \text{by intro\_classes} \\
\text{arities} \quad \text{up} :: (\text{type}) \text{ bot} \\
\text{instance} \quad \text{fun} :: (\text{type,ord}) \text{ ord} \\
\quad \text{by intro\_classes} \\
\text{arities} \quad \text{fun} :: (\text{type,bot}) \text{ bot} \\
\text{instance} \quad * :: (\text{ord,ord}) \text{ ord} \\
\quad \text{by intro\_classes} \\
\text{instance} \quad + :: (\text{ord,ord}) \text{ ord} \\
\quad \text{by intro\_classes} \\
\text{arities} \quad * :: (\text{bot,bot}) \text{ bot} \\
\text{arities} \quad + :: (\text{bot,bot}) \text{ bot}
\]

**defs**
\[
UU\_\text{up\_def[simp]}: \bot \equiv \text{down}
\]

140
B.3. Foundations

\[ UU\_fun\_def[simp]: \bot \equiv (\lambda x. \bot) \]
\[ UU\_pair\_def[simp]: \bot \equiv (\bot, \bot) \]
\[ UU\_sum\_def[simp]: \bot \equiv (Inl \bot) \]

\textbf{Semantic Constructions Dealing with Context Lifting}

\textbf{types} \quad (\tau, b) \text{ VAL} = \tau \Rightarrow b

\textbf{instance} \quad \text{fun} :: (type, zero) \text{ bot}
\text{by intro_classes}

\textbf{instance} \quad \text{fun} :: (type, one) \text{ bot}
\text{by intro_classes}

\textbf{instance} \quad \text{fun} :: (type, plus) \text{ bot}
\text{by intro_classes}

\textbf{constdefs}

\textbf{lift0} :: \alpha \Rightarrow (\tau, \alpha) \text{ VAL}
\text{lift0} \equiv \lambda c. \lambda s. c

\textbf{lift1} :: (\alpha \Rightarrow \beta) \Rightarrow (\tau, \alpha) \text{ VAL} \Rightarrow (\tau, \beta) \text{ VAL}
\text{lift1} \equiv (\lambda f X \tau. f (X \tau))

\textbf{lift2} :: ((\alpha, \beta) \Rightarrow \gamma) \Rightarrow ((\tau, \alpha) \text{ VAL}, (\tau, \beta) \text{ VAL}) \Rightarrow (\tau, \gamma) \text{ VAL}
\text{lift2} \equiv (\lambda f X Y \tau. f (X \tau)(Y \tau))

\textbf{lift3} :: ((\alpha, \beta, \gamma) \Rightarrow \delta) \Rightarrow ((\tau, \alpha) \text{ VAL}, (\tau, \beta) \text{ VAL}, (\tau, \gamma) \text{ VAL}) \Rightarrow (\tau, \delta) \text{ VAL}
\text{lift3} \equiv (\lambda f X Y Z \tau. f (X \tau)(Y \tau)(Z \tau))

\textbf{cp} :: (\langle \tau, \alpha \rangle \text{ VAL}, \tau) \Rightarrow (\tau) \Rightarrow \text{ bool}
\text{cp}(P) \equiv (\exists f. \forall X \tau. P X \tau = f (X \tau) \tau)

\textbf{syntax}
Appendix B. Isabelle Theories

\[
\begin{align*}
lift0 &:: \alpha \Rightarrow (\tau, \alpha) \, \text{VAL} \\
(lift0) &\ \\
lift1 &:: (\alpha \Rightarrow \beta) \Rightarrow (s, \alpha) \, \text{VAL} \Rightarrow (s, \beta) \, \text{VAL} \\
(lift1) &\ \\
lift2 &:: (\alpha, \beta) \Rightarrow (\tau, \alpha) \, \text{VAL} \Rightarrow (\tau, \beta) \, \text{VAL} \\
(lift2) &\ \\
lift3 &:: (\alpha, \beta, \gamma) \Rightarrow (\tau, \alpha) \, \text{VAL} \Rightarrow (\tau, \beta) \, \text{VAL} \Rightarrow (\tau, \gamma) \, \text{VAL} \\
(lift3) &\ \\
\end{align*}
\]

A Generic Theory of Undefinedness, Strictness, and Smashing

\textbf{lemma} not_DEF_UU [simp]: \(\neg \text{DEF}(\bot)\)
by (simp (no_asm) add: DEF_def)

\textbf{lemma} not_DEF_X [simp]: \((\neg \text{DEF } X) = (X = \bot)\)
by (simp (no_asm) add: DEF_def)

\textbf{lemma} exists_DEF [simp]: \(\exists x. \text{DEF}(x)\)
by (simp add: DEF_def nonEmpty)

\textbf{lemma} isStrict_strictify [simp]: isStrict(strictify f)
by (simp add: isStrict_def strictify_def)

\textbf{lemma} strict2a_UU [simp]: strictify f \(\bot = \bot\)
by (simp (no_asm) add: strictify_def)

\textbf{lemma} strict2b_UU [simp]: strictify f \(\bot X = \bot\)
by (simp (no_asm) add: strictify_def UU_fun_def)

\textbf{Strictness versus Definedness}

\textbf{lemma} DEF_strictify_DEF_args2 :
\begin{align*}
\text{DEF}(\text{strictify}(\lambda x. \text{strictify}(f x)) X Y) &\Rightarrow \text{DEF}(X \land \text{DEF } Y) \\
\text{apply} & (\text{simp add: strictify_def DEF_def}) \\
\text{apply} & (\text{case_tac } X=\bot) \\
\text{apply} & \text{auto} \\
\text{done}
\end{align*}

\textbf{lemma} DEF_strictify_DEF_fun :
\begin{align*}
[ \forall X. \text{DEF}(f X) \land \text{DEF } X ] &\Rightarrow \text{DEF}(\text{strictify } f X) \\
\text{by} & (\text{simp add: strictify_def DEF_def})
\end{align*}

\textbf{lemma} DEF_strictify_DEF_args :
\begin{align*}
\text{DEF}(\text{strictify } f X) &\Rightarrow \text{DEF } f \land \text{DEF } X \\
\text{by} & (\text{simp add: strictify_def DEF_def, auto})
\end{align*}
B.3. Foundations

lemma isStrict_compose :
[ isStrict f, isStrict g ] \Rightarrow isStrict (f o g)
by (simp add: isStrict_def o_def)

lemma smash_strict [simp]: smash f ⊥ = ⊥
by (simp (no_asm) add: smash_def)

lemma smashed_sets_nonempty [simp]: ⊥ : {X. smash f X = X}
by (simp (no_asm))

This lemmas are useful for proofs of non-emptyness of type-definition based on smashed collection types.

A Theory of Undefinedness and Strictness in Lifted Types

lemma not_down_exists_lift : x ≠ down = (∃ y. x = ⌞ y⌟)
by (induct_tac x, auto)

lemma not_down_exists_lift2 : x ≠ ⊥ = (∃ y. x = ⌞ y⌟)
by (induct_tac x, auto)

lemma drop_lift [simp]: ⌞ f⌟ = f
by (simp add: drop_def UU_up_def)

lemma drop_down [simp]: ⌞ ⊥⌟ = (ε x. True)
by (simp add: drop_def)

lemma drop_down2 [simp]: ⌞ down⌟ = (ε x. True)
by (simp add: drop_def)

lemma not_DEF_down [simp]: ¬DEF(down)
by (simp (no_asm) add: DEF_def)

lemma DEF_lift [simp]: DEF(⌜x⌟)
by (simp (no_asm) add: DEF_def)

lemma DEF_X_up : DEF(X::′a up) = (∃ x. X = ⌞ x⌟)
by (simp add: not_down_exists_lift DEF_def)

lemma not_DEF_X_up : (¬DEF(X::′a up)) = (X = ⊥)
by (simp (no_asm) add: DEF_def)

lemma DEF_fun_lift [simp]: DEF(λx. f ⌞ x⌟)

143
Appendix B. Isabelle Theories

apply (simp (no_asm) add: DEF_def)
apply (rule notI)
apply (erule fun_cong)
apply simp
done

lemma DEF_fun_fun_lift [simp]:
  \( \deflam{x}{y}. f x y \) = \( \deflam{x}{y}. f x y \)
apply (simp (no_asm) add: DEF_def)
apply (rule notI)
apply (erule fun_cong)
apply (erule fun_cong)
apply simp
done

lemma lift_defined:
  \( \bot \notin X \rightarrow \bot \notin X \)
by (auto)

A Generic Theory of Strictness

lemma strict2c_UU [simp]:
  \( \strictify f \downarrow = \downarrow \)
by (simp (no_asm) add: strictify_def)

lemma strict2d_UU [simp]:
  \( \strictify f \downarrow X = \downarrow X \)
by (simp (no_asm) add: strictify_def)

lemma strict2e_UU [simp]:
  \( \lambda x. \strictify (f x) \) \( \bot \) = \( \bot \)
by (simp add: strictify_def)

lemma strict2f_UU [simp]:
  \( \lambda x. \strictify (f x) \) \( \bot \) \( X \) = \( \bot \)
by (simp add: strictify_def)

lemma strict2_DEF [simp]:
  \( \DEF X \implies \strictify f X = f X \)
by (simp add: strictify_def, auto)

lemma strict3_DEF [simp]:
  \[ \DEF X; \DEF Y \] \implies \( \strictify (\lambda X. \strictify (f X)) \) \( X \) \( Y \) = \( f X \) \( Y \)
by (simp add: strictify_def, auto)

Generic Undefinedness-Reduction Rules for 2-Lifted Strict Operations

lemma lift1b_undef [simp]:
  \( \lift 1 (\strictify f) \downarrow = \downarrow \)
by (simp add: lift1_def)

lemma lift2_undef1a [simp]:
  \( \lift 2 (\strictify (\lambda x. \strictify (f x))) \) \( X \) \( (\bot \Rightarrow (b::bot)) \) = \( \bot \)
by (simp add: lift1_def lift2_def)

lemma lift2_undef2a [simp]:
  \( \lift 2 (\strictify (\lambda x. \strictify (f x))) \) \( X \) \( (\tau \Rightarrow (b::bot)) \) = \( \bot \)

144
B.3. Foundations

by (simp add: lift1_def lift2_def)

lemma lift1_undef_fw: \( g \equiv \text{lift1}(\text{strictify } f) \Rightarrow g \perp = \perp \)
  by (simp del: UU_fun_def)

lemma lift2_undef1_fw: \( g \equiv \text{lift2}(\text{strictify}(\lambda x. \text{strictify}(f \ x))) \Rightarrow g \perp X = \perp \)
  by (simp del: UU_fun_def)

lemma lift2_undef2_fw: \( g \equiv \text{lift2}(\text{strictify}(\lambda x. \text{strictify}(f \ x))) \Rightarrow g X \perp = \perp \)
  by (simp del: UU_fun_def)

Generic Theorems on Context Lifted Combinators

lemma cp_charn: \( \llbracket A \tau = B \tau; \ cp \ P \rrbracket \Rightarrow \ cp \ A \tau = P B \tau \)
  by (auto simp: cp_def)

lemma cp_by_cpify: \( \cp P = (\forall X \tau. \cp (\lambda x. \cp (f \ x))) \)
  by (auto simp: cp_def lift0_def, rule exI, assumption)

lemma cp_lift0 [simp, intro!]: \( \cp (\lambda X \tau. c) \)
  by (simp add: cp_def lift0_def, fast)

lemma cp_eq [simp, intro!]: \( \llbracket \cp P; \ cp P' \rrbracket \Rightarrow \cp (\lambda u ua. \cp u ua = P' u ua) \)
  by (auto simp: cp_def)

lemma cp_const [simp, intro!]: \( \cp (\lambda X \tau. c) \)
  by (simp add: cp_def, fast)

lemma cp_id [simp, intro!]: \( \cp (\lambda X \tau. X) \)
  by (simp add: cp_def, fast)

lemma cp_compose [simp, intro!]: \( \llbracket \cp P; \ cp P' \rrbracket \Rightarrow \cp (P \circ P') \)
  apply (simp add: cp_def o_def)
  apply (erule exE)
  apply (erule exE)
  apply (simp (no_asm_simp))
  apply fast
  done

lemma cp_compose2: \( \llbracket \cp P; \ cp P' \rrbracket \Rightarrow \cp (\lambda x. P' x) \)
  by (simp add: cp_def o_def, auto)

lemma cp_lift1 [simp]: \( \cp P \Rightarrow \cp (\lambda X. \text{lift1}(f \ (P \ x))) \)
  apply (simp add: cp_def lift1_def)

145
Appendix B. Isabelle Theories

apply (erule exE)
apply (simp (no_asm_simp))
apply fast
done

lemma cp_lift2 [simp]:
  [cp P; cp P'] \Rightarrow cp (\lambda X. lift2 f (P X) (P' X))
apply (simp add: cp_def lift2_def)
apply (erule exE)
apply (erule exE)
apply (simp (no_asm_simp))
apply fast
done

lemma cp_lift3 [simp]:
  [cp P; cp P'; cp P''] \Rightarrow cp (\lambda X. lift3 f (P X) (P' X) (P'' X))
apply (simp add: cp_def lift3_def)
apply (erule exE)+
apply (simp (no_asm_simp))
apply fast
done

lemma cp_lift0_fw : f \equiv lift0 g \Rightarrow cp f
  by (simp add: lift0_def cp_def)

lemma cp_lift1_fw [simp]:
  [f \equiv lift1 g; cp P] \Rightarrow cp (\lambda X. f (P X))
by (simp add: cp_def lift1_def, auto)

lemma cp_lift2_fw : 
  [f \equiv lift2 g; cp P; cp P']
  \Rightarrow cp (\lambda X. f (P X) (P' X))
by (simp add: cp_def lift2_def, auto)

lemma cp_lift3_fu :
  [f \equiv lift3 g; cp P; cp P'; cp P'']
  \Rightarrow cp (\lambda X. f (P X) (P' X) (P'' X))
by (simp add: lift3_def cp_def,auto)

lemma ocl.Undef_split : X \tau = \bot \lor (\exists a. X \tau = (\langle a \rangle))
  by (simp add: UU_up_def,simp add: not_down_exists_lift [symmetric])

lemma ocl_cp.Undef_split:
  [cp P; cp P'; P \bot = P' \bot; X \neq \bot \Rightarrow P X = P' X]
  \Rightarrow P X = P' X
b.3. Foundations

by (simp add: UU_fun_def auto)

lemma ocl_cpsubst :
[X τ = X' τ; P X τ = C; cp P'] \implies P X' τ = C
by (simp add: cp_def, auto)

lemma ocl_cpsubst2 :
[X τ = X' τ; P X' τ = P' X' τ; cp P; cp P'] \implies P X τ = P' X τ
by (simp add: cp_def, auto)

lemma ocl_cpsubst3 :
[X τ = X' τ; P X' τ = P' X τ; cp P; cp P'] \implies P' X τ = P'' X τ
by (simp add: cp_def, auto)

lemma drop_lift_idem [simp]:(⌜⌜ x ⌝⌟) = x
by (simp_all split add: up.split)

lemma lift_drop_idem [simp]:[[DEF](x)] \implies (⌜⌜ x ⌝⌟) = x
  apply (simp_all add: drop_def DEF_def not_down_exists_lift)
  apply (erule exE, auto)
  done

lemma DEF_set_charn : (∀x∈X. DEF x) = (⊥/∈ X)
by (simp add: DEF_def, auto)

lemma cp_option_case : [cp P; cp P'; cp (λ(X τ . (λx. P'' x X τ)))]
  \implies cp (λX τ . case (P X τ ) of None ⇒ P' X τ | Some x ⇒ P'' x X τ)
apply (simp add: cp_def)
apply (auto)
apply (rule_tac x=λ C τ . case (f C τ ) of None ⇒ (fa C τ ) | Some x ⇒ (fb C τ ) x in excl)
apply (auto)
apply (rule_tac t=P' X τ in subst)
apply (rotate_tac 1)
apply (erule allE)
apply (rotate_tac -1)
apply (erule allE)
apply (rule sym)
apply (assumption)
apply (rule_tac t=λ x. P'' x X τ in subst)
apply (rotate_tac 2)
apply (erule allE)

147
Appendix B. Isabelle Theories

apply\(\text{rotate Tac} - 1\)
apply\(\text{erule allE}\)
apply\(\text{rule sym}\)
apply\(\text{assumption}\)
apply\(\text{rule refl}\)
done

Higher-order Context-passingness

This theory is intended to develop higher-order rewriting for LJE-rewriting, i.e., rewriting in the body of iterators or quantifiers with respect to strong equality. This part of the theory is still under development.

constdefs

cp0 :: (\((\tau',\alpha)\ \text{VAL}, \tau\) \Rightarrow bool)
cp0 P \equiv \forall Q \tau. P Q \tau = P (\text{lift}0 (Q \tau)) \tau

cp1 :: (\((\tau',\alpha)\ \text{VAL} \Rightarrow (\tau',\beta)\ \text{VAL} \Rightarrow (\tau,\gamma)\ \text{VAL} \Rightarrow bool)
cp1 P \equiv \forall Q \tau. P Q \tau = P (\lambda x. (\text{lift}0 (Q x \tau))) \tau

cp2 :: (\((\tau',\alpha)\ \text{VAL} \Rightarrow (\tau',\beta)\ \text{VAL} \Rightarrow (\tau,\gamma)\ \text{VAL} \Rightarrow bool)
cp2 P \equiv \forall Q \tau. P Q \tau = P (\lambda x y. (\text{lift}0 (Q x y \tau))) \tau

Building a ruleset to unfold all cp-ness definitions
lemmas ss_cp_defns = cp_def cp0_def cp1_def cp2_def

The functional behaviour of \text{lift}0. This rule is sometimes useful in a proof.

lemma lift0_apply: \text{lift}0 X \tau = X
by\(\text{simp add: lift0_def}\)

The rules to “cp-unfold” a term by hand

lemma cp0_unfold:
[ cp0 P; Q \tau = R \tau ] \Longrightarrow P Q \tau = P (\text{lift}0 (R \tau)) \tau
apply\(\text{unfold cp0_def, erule_tac x=Q in allE, erule_tac x=\tau in allE}\)
by\(\text{simp add: lift0_apply}\)

lemma cp1_unfold:
[ cp1 P; \forall x. Q x \tau = R x \tau ] \Longrightarrow P Q \tau = P (\lambda x. \text{lift}0 (R x \tau)) \tau
apply\(\text{unfold cp1_def, erule_tac x=Q in allE, erule_tac x=\tau in allE}\)
by\(\text{simp add: lift0_apply}\)

lemma cp2_unfold:
[ cp2 P; \forall x y. Q x y \tau = R x y \tau ] \Longrightarrow P Q \tau = P (\lambda x y. \text{lift}0 (R x y \tau)) \tau
apply\(\text{unfold cp2_def, erule_tac x=Q in allE, erule_tac x=\tau in allE}\)
by\(\text{simp add: lift0_apply}\)
The rules to "cp-fold" a term by hand

**lemma cp0_fold:**
\[
\llbracket \text{cp0 } P; \ Q \tau = R \tau \rrbracket \implies P (\text{lift}_0 (Q \tau)) \tau = P R \tau
\]
by(erule cp0_unfold[symmetric], simp)

**lemma cp1_fold:**
\[
\llbracket \text{cp1 } P; \ \forall x. Q x \tau = R x \tau \rrbracket \implies P (\lambda x. \text{lift}_0 (Q x \tau)) \tau = P R \tau
\]
by(erule cp1_unfold[symmetric], simp)

**lemma cp2_fold:**
\[
\llbracket \text{cp2 } P; \ \forall x y. Q x y \tau = R x y \tau \rrbracket \implies P (\lambda x y. \text{lift}_0 (Q x y \tau)) \tau = P R \tau
\]
by(erule cp2_unfold[symmetric], simp)

### Rules to Establish cp

cp0 rules

**lemma cp0_const [simp]:** cp0 (λ x τ. P)
by(simp add: cp0_def)

**lemma cp0_id [simp]:** cp0 (λ x. x)
by(simp add: cp0_def lift0_apply)

**lemma cp0_app:**
\[\text{assumes } \text{cp0}_1_P : \ \forall y. \text{cp0} (\lambda x. (P x y)) \]
\[\text{and } \text{cp0}_2_P : \ \forall x. \text{cp0} (P x) \]
\[\text{shows } \text{cp0} (\lambda x. P x x) \]
apply(unfold cp0_def, clarify)
apply(rule trans, rule_tac P=λ x. P x ?Y and τ=τ in cp0_unfold)
apply(rule cp0_1_P, refl)
apply(rule sym, rule trans, rule_tac P=λ x. P ?X x and τ=τ in cp0_unfold)
apply(rule cp0_2_P, refl)
done

**lemma cp0_of_cp0:**
\[\text{assumes } \text{cp0}_P : \ \text{cp0 } P \]
\[\text{and } \text{cp0}_Q : \ \text{cp0 } Q \]
\[\text{shows } \text{cp0} (\lambda x. P (Q x)) \]
apply(unfold cp0_def, clarify)
apply(rule trans, rule_tac P=λ x. P x and τ=τ in cp0_unfold)
apply(rule cp0_P, refl)
apply(rule sym, rule trans, rule_tac P=λ x. P x and τ=τ in cp0_unfold)
apply(rule cp0_P)
apply(rule_tac P=λ x. Q x and τ=τ in cp0_fold)
apply(rule cp0_Q, refl)
done
Appendix B. Isabelle Theories

lemma cp0_of_cp1:
  assumes cp1_P: cp1 P
  and cp0_1_Q: \( \forall a. \text{cp}0 (\lambda x. Q x a) \)
  shows \( \text{cp}0 (\lambda x. P (Q x)) \)
  apply (unfold cp0_def, clarify)
  apply (rule trans, rule_tac P=\( (\lambda x. P x) \) and \( \tau=\tau \) in cp1_unfold)
  apply (rule cp1_P, rule refl)
  done

lemma cp0_of_cp2:
  assumes cp2_P: cp2 P
  and cp0_1_Q: \( \forall a b. \text{cp}0 (\lambda x. Q x a b) \)
  shows \( \text{cp}0 (\lambda x. P (Q x)) \)
  apply (unfold cp0_def, clarify)
  apply (rule trans, rule_tac P=\( (\lambda x. P x) \) and \( \tau=\tau \) in cp1_unfold)
  apply (rule cp2_P, rule refl)
  done

In the following, we prove the cp1 rules.

lemma cp1_const [simp]: cp1 (\( \lambda x. P \))
  by (simp add: cp1_def)

This rule corresponds to \( \text{cp}0\_id \). It has the same structure except for the constant
evaluation which is of course not needed for a nullary argument.

lemma cp1_const_eval [simp]: cp1 (\( \lambda x. x X \))
  by (simp add: cp1_def lift0_apply)

lemma cp1_app:
  assumes cp1_1_P: \( \forall y. \text{cp}1 (\lambda x. (P x y)) \)
  and cp1_2_P: \( \forall x. \text{cp}1 (P x) \)
  shows \( \text{cp}1 (\lambda x. P x x) \)
  apply (unfold cp1_def, clarify)
  apply (rule trans, rule_tac P=\( (\lambda x. P x ?Y) \) and \( \tau=\tau \) in cp1_unfold)
  apply (rule cp1_1_P, rule refl)
  done
lemma cp1_of_cp0:
assumes cp0_P: cp0 P
and cp1_Q: cp1 Q
shows cp1 (λ x. P (Q x))
apply(unfold cp1_def, clarify)
apply(rule trans, rule_tac P=(λ x. P x) and τ=τ in cp0_unfold)
apply(rule sym, rule trans, rule_tac P=(λ x. P x) and τ=τ in cp0_unfold)
apply(rule cp0_P, rule refl)
apply(rule_tac P=(λ x. P x) and τ=τ in cp1_fold)
apply(rule cp1_Q, rule refl)
apply(rule refl)
done

lemma cp1_of_cp1:
assumes cp1_P: cp1 P
cp1_1_Q: ∀ a. cp1 (λ x. Q x a)
shows cp1 (λ x. P (Q x))
apply(unfold cp1_def, clarify)
apply(rule trans, rule_tac P=(λ x. P x) and τ=τ in cp1_unfold)
apply(rule sym, rule trans, rule_tac P=(λ x. P x) and τ=τ in cp1_unfold)
apply(rule cp1_P, rule refl)
apply(rule_tac P=(λ x. P x) and τ=τ in cp1_fold)
apply(rule cp1_1_Q, rule refl)
apply(rule refl)
done

lemma cp1_of_cp2:
assumes cp2_P: cp2 P
cp1_1_Q: ∀ a b. cp1 (λ x. Q x a b)
shows cp1 (λ x. P (Q x))
apply(unfold cp1_def, clarify)
apply(rule trans, rule_tac P=(λ x. P x) and τ=τ in cp2_unfold)
apply(rule sym, rule trans, rule_tac P=(λ x. P x) and τ=τ in cp2_unfold)
apply(rule cp2_P, rule refl)
apply(rule_tac P=(λ x. P x) and τ=τ in cp1_fold)
apply(rule cp1_1_Q, rule refl)
apply(rule refl)
done

cp2 rules

lemma cp2_const [simp]: cp2 (λ x. τ. P)
by(simp add: cp2_def)

lemma cp2_const_eval [simp]: cp2 (λ x. x X Y)
by(simp add: cp2_def lift0_apply)
Appendix B. Isabelle Theories

lemma cp2_app:
  assumes cp2_1_P: \( \forall y. \, cp2 (\lambda x. (P \circ y)) \)
  and cp2_2_P: \( \forall x. \, cp2 (P \circ x) \)
  shows \( cp2 (\lambda x. \, P \circ x) \)
  apply (unfold cp2_def, clarify)
  apply (rule trans, rule_tac P=\( (\lambda x. \, P \circ x) \) \& \& \( \tau = \tau \) \in cp2_unfold)
  apply (rule cp2_1_P, rule refl)
  apply (rule refl)
  done

lemma cp2_of_cp0:
  assumes cp0_P: cp0 P
  and cp2_Q: \( \forall x. \, cp2 (Q \circ x) \)
  shows \( cp2 (\lambda x. \, P \circ (Q \circ x)) \)
  apply (unfold cp2_def, clarify)
  apply (rule trans, rule_tac P=\( (\lambda x. \, P \circ x) \) \& \& \( \tau = \tau \) \in cp0_unfold)
  apply (rule cp0_P, rule refl)
  apply (rule sym, rule trans, rule_tac P=\( (\lambda x. \, P \circ x) \) \& \& \( \tau = \tau \) \in cp0_unfold)
  apply (rule cp0_P)
  apply (rule_tac P=\( (\lambda x. \, Q \circ x) \) \& \& \( \tau = \tau \) \in cp2_fold)
  apply (rule cp2_Q, rule refl)
  apply (rule refl)
  done

lemma cp2_of_cp1:
  assumes cp1_P: \( \forall x. \, cp1 P \)
  and cp2_1_Q: \( \forall a. \, cp2 (\lambda x. (Q \circ a)) \)
  shows \( cp2 (\lambda x. \, P \circ (Q \circ x)) \)
  apply (unfold cp2_def, clarify)
  apply (rule trans, rule_tac P=\( (\lambda x. \, P \circ x) \) \& \& \( \tau = \tau \) \in cp1_unfold)
  apply (rule cp1_P, rule refl)
  apply (rule sym, rule trans, rule_tac P=\( (\lambda x. \, P \circ x) \) \& \& \( \tau = \tau \) \in cp1_unfold)
  apply (rule cp1_P)
  apply (rule_tac P=\( (\lambda x. \, Q \circ x) \) \& \& \( \tau = \tau \) \in cp2_fold)
  apply (rule cp2_Q, rule refl)
  apply (rule refl)
  done

lemma cp2_of_cp2:
  assumes cp2_P: \( \forall x. \, cp2 P \)
  and cp2_1_Q: \( \forall a b. \, cp2 (\lambda x. (Q \circ a \circ b)) \)
  shows \( cp2 (\lambda x. \, P \circ (Q \circ x)) \)
  apply (unfold cp2_def, clarify)
  apply (rule trans, rule_tac P=\( (\lambda x. \, P \circ x) \) \& \& \( \tau = \tau \) \in cp2_unfold)
  apply (rule cp2_P, rule refl)
  apply (rule sym, rule trans, rule_tac P=\( (\lambda x. \, P \circ x) \) \& \& \( \tau = \tau \) \in cp2_unfold)
  apply (rule cp2_P)
  done

152
B.3. Foundations

apply\(\text{rule_tac P}=\lambda x. Q x \ ?X \ ?Y\) and \(\tau=\tau\) in cp2\_fold
apply\(\text{rule cp2\_I Q, rule refl}\)
apply\(\text{rule refl}\)
done

Deciding cp using cp\_unfold

By converting a term into explicit cp-representation one can also decide all context-passingness questions. Since the tactic uses a stepwise unfolding it can deal with cases where simp combined with chain rules for cp-ness does not work. These are cases where currently the rule cp\_compose2 is applied by hand. The decision using cp\_unfold works as follows:

1. apply \text{cp\_unfold}
2. if one of the rules \text{cp0\_cp\_unfolded}, \text{cp1\_cp\_unfolded}, or \text{cp2\_cp\_unfolded} matches were done otherwise the cp-ness proposition is false (assuming that the simpset of cp\_unfold is complete with respect to the set of OCL operations)

\begin{verbatim}
lemma cp0\_cp\_unfolded:
  cp0 (λ x τ. P (lift0 (x τ)) τ)
by(simp add: cp0\_def lift0\_apply)

lemma cp1\_cp\_unfolded:
  cp1 (λ Q τ. P (λ x. lift0 (Q x τ)) τ)
by(simp add: cp1\_def lift0\_apply)

lemma cp2\_cp\_unfolded:
  cp2 (λ Q τ. P (λ x y. lift0 (Q x y τ)) τ)
by(simp add: cp2\_def lift0\_apply)
\end{verbatim}

Higher-Order lifting construction

\begin{verbatim}
constdefs
  lift1' :: \([\gamma \Rightarrow \eta \Rightarrow \alpha, \alpha \Rightarrow \beta] \Rightarrow [\gamma, \eta] \Rightarrow \beta\)
lift1' a1 ≡ (λ f x τ. f (a1 x τ))

  lift2' :: \([\alpha \Rightarrow \gamma \Rightarrow \eta, \beta \Rightarrow \gamma \Rightarrow \delta] \Rightarrow [\alpha, \beta, \gamma] \Rightarrow \delta\)
lift2' a1 a2 ≡ (λ f x y τ. f (a1 x τ) (a2 y τ))

  lift3' :: \([\eta \Rightarrow h \Rightarrow \alpha, \gamma \Rightarrow g \Rightarrow \beta, \alpha \Rightarrow \gamma \Rightarrow \delta] \Rightarrow [\eta, \gamma, \delta]\)
lift3' a1 a2 a3 ≡ (λ f x y z τ. f (a1 x τ) (a2 y τ) (a3 z τ))

  lift\_arg0 :: ('τ \Rightarrow \beta) \Rightarrow ('τ \Rightarrow \beta)
lift\_arg0 ≡ id

  lift\_arg1 :: (('τ \Rightarrow \gamma) \Rightarrow ('τ \Rightarrow \delta)) \Rightarrow ('γ \Rightarrow \delta)
lift\_arg1 ≡ (λ f τ x. f (lift0 x τ))
\end{verbatim}
Appendix B. Isabelle Theories

\[
lift\_arg2 :: ([\alpha \Rightarrow \beta, \gamma \Rightarrow \delta] \Rightarrow (\eta \Rightarrow \tau)) \Rightarrow ([\eta, \beta, \delta] \Rightarrow \tau)
\]

\[
lift\_arg2 \equiv \lambda f \tau x y. f (lift\_0 x) (lift\_0 y) \tau
\]

Rule sets to unfold the whole lifting constructions

lemmas ss_lifting = lift1_def lift2_def lift3_def
lift1\'_def lift2\'_def lift3\'_def
lift_arg0_def lift_arg1_def lift_arg2_def
DEF_def strictify_def isStrict_def
smash_def

lemmas ss_lifting\' = lift0_def ss_lifting

The Link to the old Combinator Set.

Note, lift1, lift2, lift3 are special cases of these general combinators.

lemma lift1\_by\_lift1\': lift1 = (lift1\'_lift_arg0)
  by(simp add: lift1_def lift1\'_def lift_arg0_def)
lemma lift2\_by\_lift2\': lift2 = (lift2\'_lift_arg0 lift_arg0)
  by(simp add: lift2_def lift2\'_def lift_arg0_def)
lemma lift3\_by\_lift3\': lift3 = (lift3\'_lift_arg0 lift_arg0 lift_arg0)
  by(simp add: lift3_def lift3\'_def lift_arg0_def)

Forward rules for the currently used constructions:

As we have in the section about cp_unfold, transforming a term into explicit cp-representation is very powerful. Its foundation is cp_unfold which relies on a simplifier set containing all context-passingess predicates for every argument of every OCL operation. These set can be created by putting all forward rules from below whose first assumption is unifyable with the definition of the OCL function into this set.

The fw,OCL cp0\_chain rules are the old OCL_cp rules who can be put into the standard simplifier set.

Context-passeness by constrution

ML

val cpR_prover_simpset = simpset() addsimps ((thms ss_lifting') @ (thms ss_cp_defs))

in

fun cpR_prover mk_name (goal,arity,position,lifting) =
  let val (rname, name_prefix) = mk_name arity position lifting
  val thm = prove_goalw (the_context()) [] goal
  in fn prems => [ALLGOALS(simp_tac cpR_prover_simpset) thm]
  end

B.3. Foundations

```ml
in

```
Appendix B. Isabelle Theories

(f \equiv (\text{lift}' \ \text{lift}_0 \ \text{arg}0) \ P \implies f \ (\text{lift}_0 \ (X \ \tau)) \ \tau = f \ X \ \tau, \ 0, \ 1, \ \text{lift}'

(f \equiv \text{lift}2 \ P \implies f \ (\text{lift}_0 \ (X \ \tau)) \ Y \ \tau = f \ X \ Y \ \tau, \ 0, \ 1, \ \text{lift}2)

(f \equiv \text{lift}2 \ P \implies f \ X \ (\text{lift}_0 \ (Y \ \tau)) \ \tau = f \ X \ Y \ \tau, \ 0, \ 2, \ \text{lift}2)

(f \equiv (\text{lift}2' \ \text{lift}_0 \ \text{arg}0 \ aY) \ P \implies f \ (\text{lift}_0 \ (X \ \tau)) \ Y \ \tau = f \ X \ Y \ \tau, \ 0, \ 1, \ \text{lift}2'

(f \equiv (\text{lift}2' \ aX \ \text{lift}_0 \ \text{arg}0) \ P \implies f \ X \ (\text{lift}_0 \ (Y \ \tau)) \ \tau = f \ X \ Y \ \tau, \ 0, \ 2, \ \text{lift}2'

(f \equiv (\text{lift}2' \ aX \ \text{lift}_0 \ \text{arg}1) \ P \implies f \ X \ (\lambda \ x. \ \text{lift}_0 \ (Y \ x \ \tau)) \ \tau = f \ X \ Y \ \tau, \ 1, \ 2, \ \text{lift}2'

(f \equiv (\text{lift}2' \ aX \ \text{lift}_0 \ \text{arg}2) \ P \implies f \ X \ (\lambda \ x. \ y. \ \text{lift}_0 \ (Y \ x \ y \ \tau)) \ \tau = f \ X \ Y \ \tau, \ 2, \ 2, \ \text{lift}2'\n
end

Unary [OCL] operations: old lifting construction.

lemma \text{fw}_\text{OCL} \text{cp0} \text{_chain0} \_\text{lift1}:
assumes \text{f_def}: f \equiv \text{lift1} \ P
and \text{cp0}_\text{X}: \text{cp0} \ X
shows \ (\text{cp0} \ (\lambda \ x. \ f \ (X \ x)))
by(insert \text{cp0}_\text{X}, \text{simp add: f_def cp0_def lift1_def lift0_apply})

lemma \text{fw}_\text{OCL} \text{cp0} \text{_chain0} \_\text{lift1'}:
assumes \text{f_def}: f \equiv (\text{lift1}' \ \text{lift}_0 \ \text{arg}0) \ P
and \text{cp0}_\text{X}: \text{cp0} \ X
shows \ (\text{cp0} \ (\lambda \ x. \ f \ (X \ x)))
by(insert \text{cp0}_\text{X}, \text{simp add: f_def cp0_def lift1'_def lift0_apply})

Binary [OCL] operations: old lifting construction.

lemma \text{fw}_\text{OCL} \text{cp0} \text{_chain00} \_\text{lift2}:
assumes \text{f_def}: f \equiv \text{lift2} \ P
and \text{cp0}_\text{X}: \text{cp0} \ X
and \text{cp0}_\text{Y}: \text{cp0} \ Y
shows \ (\text{cp0} \ (\lambda \ x. \ f \ (X \ x) \ (Y \ x)))
by(insert \text{cp0}_\text{X} \text{cp0}_\text{Y}, \text{simp add: f_def cp0_def lift2_def lift0_apply})
B.3. Foundations

lemma fw_OCL_cp0_chain00_lift2':
assumes f_def: f ≡ (lift2' lift_arg0 lift_arg0) P
and cp0_X: cp0 X
and cp0_Y: cp0 Y
shows cp0 (λ x. f (X x) (Y x))
by(insert cp0_X cp0_Y,
  simp add f_def cp0_def lift_arg0_def lift2'_def lift0_apply)

lemma fw_OCL_cp0_chain01_lift2':
assumes f_def: f ≡ (lift2' lift_arg0 lift_arg1) P
and cp0_X: cp0 X
and cp0_Y: ∀a. cp0 (λ x. Y x a)
shows cp0 (λ x. f (X x) (Y x))
by(insert cp0_X cp0_Y,
  simp add f_def cp0_def lift_arg0_def lift_arg1_def lift2'_def lift0_apply)

lemma fw_OCL_cp0_chain02_lift2':
assumes f_def: f ≡ (lift2' lift_arg2 lift_arg0) P
and cp0_X: cp0 X
and cp0_Y: ∀a b. cp0 (λ x. Y x a b)
shows cp0 (λ x. f (X x) (Y x))
by(insert cp0_X cp0_Y,
  simp add f_def cp0_def lift_arg0_def lift_arg2_def lift2'_def lift0_apply)

Tertiary OCL operations: ->iterate and if.

lemma fw_OCL_cp0_chain000_lift3:
assumes f_def: f ≡ lift3 P
and cp0_X: cp0 X
and cp0_Y: cp0 Y
and cp0_Z: cp0 Z
shows cp0 (λ x. f (X x) (Y x) (Z x))
by(insert cp0_X cp0_Y cp0_Z,
  simp add f_def cp0_def lift3_def lift0_apply)

lemma fw_OCL_cp0_chain000_lift3':
assumes f_def: f ≡ (lift3' lift_arg0 lift_arg0 lift_arg0) P
and cp0_X: cp0 X
and cp0_Y: cp0 Y
and cp0_Z: cp0 Z
shows cp0 (λ x. f (X x) (Y x) (Z x))
by(insert cp0_X cp0_Y cp0_Z,
  simp add f_def cp0_def lift_arg0_def lift3'_def lift0_apply)

lemma fw_OCL_cp0_chain020_lift3':
assumes f_def: f ≡ (lift3' lift_arg0 lift_arg0 lift_arg2) P
and cp0_X: cp0 X
and cp0_Y: ∀a b. cp0 (λ x. Y x a b)
and cp0_Z: cp0 Z
shows cp0 (λ x. f (X x) (Y x) (Z x))
Appendix B. Isabelle Theories

by (insert cp0_X cp0_Y cp0_Z,
simp add: f_def cp0_def lift_arg0_def lift_arg2_def lift3'_def)

Chain rules for cp usable by simp or fast.

Unary OCL operations: old lifting construction.

lemma \textit{fw\_OCL\_cp\_chain0\_lift1'}:
assumes f_def: \( f \equiv (\text{lift1'} \text{lift_arg0}) \) \( P \)
and cp_X: \( \text{cp X} \)
shows \( \text{cp (\( \lambda \) x. \( f (X x) \))} \)
by (insert cp_X, simp add: f_def cp_by_cpify lift_arg0_def lift1'_def lift0_apply)

Binary OCL operations: old lifting construction.

lemma \textit{fw\_OCL\_cp\_chain00\_lift2'}:
assumes f_def: \( f \equiv (\text{lift2'} \text{lift_arg0} \text{lift_arg0}) \) \( P \)
and cp_X: \( \text{cp X} \)
and cp_Y: \( \text{cp Y} \)
shows \( \text{cp (\( \lambda \) x. \( f (X x) (Y x) \))} \)
by (insert cp_X cp_Y,
  simp add: f_def cp_by_cpify lift_arg0_def lift2'_def lift0_apply)

lemma \textit{fw\_OCL\_cp\_chain01\_lift2'}:
assumes f_def: \( f \equiv (\text{lift2'} \text{lift_arg0} \text{lift_arg1}) \) \( P \)
and cp_X: \( \text{cp X} \)
and cp_Y: \( \text{cp Y} \)
shows \( \text{cp (\( \lambda \) x. \( f (X x) (Y x) \))} \)
by (insert cp_X cp_Y,
  simp add: f_def cp_by_cpify lift_arg0_def lift_arg1_def lift2'_def lift0_apply)

lemma \textit{fw\_OCL\_cp\_chain02\_lift2'}:
assumes f_def: \( f \equiv (\text{lift2'} \text{lift_arg0} \text{lift_arg2}) \) \( P \)
and cp_X: \( \text{cp X} \)
and cp_Y: \( \text{cp Y} \)
and cp_Z: \( \text{cp Z} \)
shows \( \text{cp (\( \lambda \) x. \( f (X x) (Y x) (Z x) \))} \)
by (insert cp_X cp_Y cp_Z,
  simp add: f_def cp_by_cpify lift_arg0_def lift_arg2_def lift2'_def lift0_apply)

Tertiary OCL operations: \textit{iterate} and if.

lemma \textit{fw\_OCL\_cp\_chain000\_lift3'}:
assumes f_def: \( f \equiv (\text{lift3'} \text{lift_arg0} \text{lift_arg0} \text{lift_arg0}) \) \( P \)
and cp_X: \( \text{cp X} \)
and cp_Y: \( \text{cp Y} \)
and cp_Z: \( \text{cp Z} \)
shows \( \text{cp (\( \lambda \) x. \( f (X x) (Y x) (Z x) \))} \)
by (insert cp_X cp_Y cp_Z,
  simp add: f_def cp_by_cpify lift_arg0_def lift3'_def lift0_apply)

lemma \textit{fw\_OCL\_cp\_chain020\_lift3'}:
B.3. Foundations

assumes \( f \equiv (\text{lift3}' \text{ lift_arg0 lift_arg2 lift_arg0}) \) \( P \)
and \( \text{cp}_X : \text{cp} \ X \)
and \( \text{cp}_Y : \forall a b. \text{cp} (\lambda x. Y x a b) \)
and \( \text{cp}_Z : \text{cp} \ Z \)
shows \( \text{cp} (\lambda x. f (X x) (Y x) (Z x)) \)
by\((\text{insert cp}_X \text{ cp}_Y \text{ cp}_Z, \text{simp add: f_def cp_by_cpify lift_arg0_def lift_arg2_def lift3'_def})\)

Note, all of these rules are registered for the use with ocl_setup_op in the theory OCLL_Kernel.

cp0 is equivalent to what we already have with cp. Thus it is currently added to the simps set to allow for a decision of cp0-questions based on the machinery already built for deciding cp-questions. This will change when the theories are cleaned up.

lemma \( \text{cp0}_\text{eq}_\text{cp} \):
\( \text{cp0} \ P = \text{cp} \ P \)
by\((\text{simp add: cp0_def cp_by_cpify})\)

Accessing Type Sums

constdefs
\( \text{FromL} \:: (\alpha + \beta) \uparrow \rightarrow \alpha \uparrow \)
\( \text{FromL} \ x \equiv \text{case} \ x \ of \ \text{lift} \ v \Rightarrow (\text{sum_case} (\lambda x. x) (\lambda x. \downarrow)) \ v \)
\| \ down \Rightarrow \ down \)

constdefs
\( \text{FromR} \:: (\alpha + \beta) \uparrow \rightarrow \beta \uparrow \)
\( \text{FromR} \ x \equiv \text{case} \ x \ of \ \text{lift} \ v \Rightarrow (\text{sum_case} (\lambda x. \downarrow) (\lambda x. x)) \ v \)
\| \ down \Rightarrow \ down \)

lemma \( \text{FromL}_\text{UU} \) [simp]: \( \text{FromL} \ \bot = \bot \)
by \((\text{simp add: FromL_def})\)

lemma \( \text{FromR}_\text{UU} \) [simp]: \( \text{FromR} \ \bot = \bot \)
by \((\text{simp add: FromR_def})\)

lemma \( \text{FromL}_\text{down} \) [simp]: \( \text{FromL} \ \text{down} = \text{down} \)
by \((\text{simp add: FromL_def})\)

lemma \( \text{FromR}_\text{down} \) [simp]: \( \text{FromR} \ \text{down} = \text{down} \)
by \((\text{simp add: FromR_def})\)

lemma \( \text{FromL}_\text{lift_Inl_id} \) [simp]: \( \text{FromL} \ _\text{Inl} \ x = x \)
by \((\text{simp add: FromL_def})\)

lemma \( \text{FromR}_\text{lift_Inr_id} \) [simp]: \( \text{FromR} \ _\text{Inr} \ x = \text{lift} \ x \)
by \((\text{simp add: FromR_def})\)

lemma \( \text{FromR}_\text{Inl}_\text{UU} \) [simp]: \( \text{FromR} \ _\text{Inl} \ x = \bot \)

159
Appendix B. Isabelle Theories

by (simp add: FromR_def)

lemma FromL_Inr_UU [simp]: FromL Inr x = ⊥
  by (simp add: FromL_def)

lemma h1: DEF (FromL(x)) ⇒ Inl (FromL x) = x
  apply (simp add: FromL_def lift_drop_idem)
  apply (case_tac x, simp add: lift_drop_idem drop_lift_idem, auto)
  apply (case_tac α, simp add: lift_drop_idem drop_lift_idem, auto)
  done

lemma h2: DEF (FromL x) ⇒ Inl (FromL x) = x
  apply (simp add: FromL_def)
  apply (case_tac x, simp add: lift_drop_idem drop_lift_idem, auto)
  apply (case_tac α, simp add: lift_drop_idem drop_lift_idem, auto)
  done

lemma DEF_FromL_exists_Inl: DEF (FromL x) = (∃ y. x = Inl y)
  apply (simp add: FromL_def)
  apply (case_tac x, simp, auto)
  apply (case_tac α, simp, auto)
  done

lemma DEF_FromR_exists_drop_Inr: DEF (FromR x) ⇒ Inr (FromR x) = x
  apply (simp add: FromR_def lift_drop_idem)
  apply (case_tac x, simp add: lift_drop_idem drop_lift_idem, auto)
  apply (case_tac α, simp add: lift_drop_idem drop_lift_idem, auto)
  done

lemma FromL_Inl_inv [simp]: (FromL (xb) = X) = (xb = Inl X)
  apply (auto)
  apply (cases xb)
  apply (auto)
  done

lemma FromR_Inr_inv [simp]: (FromR (xb) = X) = (xb = Inr X)
  apply (auto)
  apply (cases xb)
  apply (auto)
  done

lemma Inv_FromR_drop: DEF (FromR(x)) ⇒ Inv (FromR x) = x
  apply (simp add: FromR_def lift_drop_idem)
  apply (case_tac x, simp add: lift_drop_idem drop_lift_idem, auto)
  apply (case_tac α, simp add: lift_drop_idem drop_lift_idem, auto)
  done
lemma \textit{DEF\_FromR\_exists\_Inr}: \textit{DEF} (\text{FromR}\ x) = (∃ y. x = \text{Inr} y)
apply (simp add: \text{FromR\_def})
apply (case_tac x, simp, auto)
done

lemma \textit{DEF\_FromLD}: \textit{DEF}(\text{FromL}\ x) \implies \textit{DEF}(x)
apply (simp add: \text{DEF\_def FromL\_def not\_down\_exists\_lift})
apply (case_tac x, simp, simp)
done

lemma \textit{DEF\_FromRD}: \textit{DEF}(\text{FromR}\ x) \implies \textit{DEF}(x)
apply (simp add: \text{DEF\_def FromR\_def not\_down\_exists\_lift})
apply (case_tac x, simp, simp)
done

end

B.3.2. The OCL Kernel

theory OCL\_Kernel
imports "$\text{HOLOCL\_HOME}/src/Lifting$
begin

Global Syntax Infrastructure

In the theory files, this section contains the SML code for setting up the infra-structure for switching between mathematical and other notations for OCL.

ocl\_setup\_op

In the theory files, this section contains the SML code that implements the rulebase for the context-passingness rules that are registered to the simplifier to decide cp-ness in proofs without the help of ocl\_cp\_unfold or ocl\_simp.

Definition of the Isar interface to ocl\_setup\_op

In the theory files, this section contains the SML code that sets up the isar syntax for command setup\_op.

Setting up the Rulebase for cp-reasoning.
Appendix B. Isabelle Theories

In the theory files, this section contains the SML code that sets up the rulebase for the context-passingness rules that are registered to the simplifier to decide cp-ness questions in interactive proofs without the help of ocl_cp_unfold or ocl_simp.

Registering the cp-rules used to decide cp-ness questions in cp_unfold and the cp-fold rules used to convert a term in explicit cp-representation to a term in implicit cp-representation.

end

B.3.3. Type Definition for OclUndefined

theory OCL_Undefined
imports $HOLOCL_HOME/src/OCL_Kernel
begin

As previously mentioned, all types, let it be basic types or class types, my contain the element ⊥ denoting undefinedness. In classifier types, this can be interpreted as equivalent in unboxed types to a null-reference. Therefore, there is a universal constant in OCL called ⊥., that is constructed as follows:

constdefs
  OclUndefined :: ('a,'b:bot) VAL
  OclUndefined ≡ lift0 ⊥

syntax
  _OclUndefined_std :: ('a,'b:bot) VAL (OclUndefined)

syntax
  _OclUndefined_ascii :: ('a,'b:bot) VAL (undef)

syntax
  _OclUndefined_math :: ('a,'b:bot) VAL (⊥)

Rules describing the effect of strictify on the OCL level

lemma fw_OCL_undef_1_lift1_0:
  assumes f_def: f ≡ (lift1' lift_arg0) (strictify P)
  shows f OclUndefined = OclUndefined
  by(simp add: f_def OclUndefined_def ss_lifting')
  binary rules

lemma fw_OCL_undef_1_lift2_00:
  assumes f_def: f ≡ (lift2' lift_arg0 lift_arg0) (strictify P)
  shows f OclUndefined Y = OclUndefined

162
B.3. Foundations

B.3.4. Type Definition for Boolean

theory OCL_Boolean_type
imports
  $HOLOCL_HOME/src/OCL_Kernel
begin

  This theory is the prototype for a basic type definition in OCL. The value types are defined by extending the basic HOL types with the bottom element \( \bot \).

  types
    Boolean_0 = bool up
    'a Boolean = ('a, Boolean_0) VAL

end
Appendix B. Isabelle Theories

B.3.5. Type Definition for Integer

theory OCL_Integer_type
imports $HOLOCL_HOME/src/OCL_Kernel
begin

The value types are defined by extending the basic HOL types with the bottom element ⊥.

types
  Integer_0 = int up
  'a Integer = ('a, Integer_0) VAL

translations
  a Integer ← (a, Integer_0) VAL

end

B.3.6. Type Definition for Real

theory OCL_Real_type
imports $HOLOCL_HOME/src/OCL_Kernel Complex_Main
begin

The value types are defined by extending the basic HOL types with the bottom element ⊥.

types
  Real_0 = real up
  'a Real = ('a, Real_0) VAL

translations
  a Real ← (a, Real_0) VAL

end

B.3.7. Type Definition for String

theory OCL_String_type
imports $HOLOCL_HOME/src/OCL_Kernel
begin

The value types are defined by extending the basic HOL types with the bottom element ⊥.

end
B.3. Foundations

```
types
  String_0 = string up
  'a String = ('a, String_0) VAL

translations
  a String ← (a, String_0) VAL
end

B.3.8. Type Definition for OclAny

theory OCL_OclAny_type
imports $HOLOCL_HOME/src/Lifting
begin
typedecl oid
arities
  oid :: bot
datatype OclAny_key = OclAny
types OclAny_type = OclAny_key × oid
  This definition of OclAny_type results in a “referential universe”, i.e., a universe
  where a referential equality is present. A referential equality is the strong equality
  for “boxed types” (that is Real, Boolean, String) and an equality on the reference to
  a value. Since our universe construction is designed to mirror the subtype discipline
  of OCL, i.e., objects are records of a certain type, the reference to an object must be
  stored in the object itself in our model, namely at the first position of the record.

  If OCL expressions refer to states, where all objects contain the reference to itself
  in the state, the logical equality co-incides with a referential equality.

types
  'a OclAny_ext = (OclAny_type × 'a up)
  'a OclAny_0 = ('a OclAny_ext up
  Alternative we could define which would allow an additional ‘exception’ element with
  a non-strict behavior. This would model something like a null reference (or pointer)
  known from object-oriented programming languages.

translations
  a OclAny_0 <= (type ((OclAny_key × oid) × a up)up

types
  ('st,'a)OclAny = ('st, 'a OclAny_0) VAL

lemma [simp]: DEF (obj::(('a)OclAny_0)) ⇒
  (fst(fst(drop(obj)))) = OclAny_key.OclAny
apply (auto simp: DEF_def not_down_exists_lift)
apply (case_tac a)
```

165
Appendix B. Isabelle Theories

apply (assumption)
done
end

B.3.9. Type Definition for Collections

theory OCL_Collection_type
imports $HOLOCL_HOME/src/OCL_Kernel
begin
axclass collection < bot
end

B.3.10. Type Definition for Set

theory OCL_Set_type
imports $HOLOCL_HOME/src/library/collection/OCL_Collection_type
begin
typedef 'a Set_0 = {X::('a::bot) set. up.
(smash (\x. DEF X \land x \in drop X) X) = X}
apply (rule_tac x=\bot in exI)
by (simp add: smash_def)

instance Set_0 :: (type) ord
  by intro_classes

arities Set_0 :: (type) collection

defs UU_Set_def: \bot \equiv Abs_Set_0 down
end

B.3.11. Type Definition for Sequence

theory OCL_Sequence_type
imports $HOLOCL_HOME/src/library/collection/OCL_Collection_type
List
begin

In contrast to other collections, the collection Sequence is indeed implemented by finite lists. Consequently, size is defined for all defined sequences.

typedef 'a Sequence_0 = {X::('a::bot) list. up.
(smash (\x. DEF X \land x \in set "X") X) = X}
  by (rule_tac x=\bot in exI, simp add: smash_def)
B.3. Foundations

arities Sequence_0 :: (type) collection

defs UU_Sequence_def. ⊥ ≡ Abs_Sequence_0 down

The HOL type constructor for representing OCL sequence types is defined as follows:
types ('st, 'b) Sequence = ('st, 'b Sequence_0) VAL

end

B.3.12. Type Definition for Bag

theory OCL_Bag_type
imports $HOLOCL_HOME/src/library/collection/OCL_Collection_type
Multiset
begin
typedef 'a Bag_0 = {X::('a::bot) multiset up.
(smash (λ x X. DEF X & x:# drop X) X) = X}
apply (rule_tac x=⊥ in exI)
by (simp add: smash_def)

instance Bag_0 :: (type) ord
by intro_classes

arities Bag_0 :: (type) bot
arities Bag_0 :: (type) collection

defs UU_Bag_def. ⊥ ≡ Abs_Bag_0 down

types ('st, 'b) Bag = ('st, 'b Bag_0) VAL

end

B.3.13. Type Definition for OrderedSet

theory OCL_OrderedSet_type
imports $HOLOCL_HOME/src/library/collection/OCL_Collection_type
List
begin
typedef 'a OrderedSet_0 = {X::('a::bot) list up.
(smash (λ x X. DEF X & (distinct 'X' → x ∈ set 'X')) X) = X}
by (rule_tac x=⊥ in exI, simp add: smash_def)

instance OrderedSet_0 :: (type) ord

Appendix B. Isabelle Theories

by intro_classes

arities OrderedSet_0 :: (type) bot
arities OrderedSet_0 :: (type) collection

defs UU_OrderedSet_def ⊥ ≡ Abs_OrderedSet_0 down

types ('a,'b) OrderedSet = ('a, 'b OrderedSet_0) VAL

end

B.3.14. The OCL Universe

theory OCL_Universe
imports
  Datatype
  $HOME/src/library/basic/OCL_Boolean_type
  $HOME/src/library/basic/OCL_String_type
  $HOME/src/library/basic/OCL_Real_type
  $HOME/src/library/basic/OCL_Integer_type
  $HOME/src/library/basic/OCL_Undefined
  $HOME/src/library/basic/OCL_OclAny/OCL_OclAny_type
  $HOME/src/library/collection/$COLLECTION/OCL_Set_type
  $HOME/src/library/collection/$COLLECTION/OCL_Bag_type
  $HOME/src/library/collection/$COLLECTION/OCL_OrderedSet_type
  $HOME/src/library/collection/$COLLECTION/OCL_Sequence_type
begin hide const class

Note, the definitions of OclAny differs, depending if HOL-OCL is was built with a referential universe or not.$2

Foundations of the Universe Construction

The universe $U$ is a family of universe types which comprise all value types (Real, Integer,String, Boolean, ...) and an extendible class type representation induced by a class hierarchy. The concepts of the universe construction has been presented in Chapt. 4.

The “initial” universe type $U_0$ is built as sum of the built-in value types and the type OclAny (including all its extensions representing subtypes of OclAny.

types values = Real_0 + Integer_0 + Boolean_0 + String_0
'a U = ('a OclAny_0) + values

types

$2$This manual describes HOL-OCL with referential universes.

168
The following setup of Isabelle's syntax engine helps when presenting types corresponding to OCL types:

**translations**

\[
\begin{align*}
\text{a U} & \leq \text{(type) (OclAny_0 a)} + (\text{Real_0 + Integer_0 + Boolean_0 + String_0)} \\
\text{Real_0} & \leq \text{(type) real up} \\
\text{Integer_0} & \leq \text{(type) int up} \\
\text{String_0} & \leq \text{(type) char list up} \\
\text{Boolean_0} & \leq \text{(type) bool up}
\end{align*}
\]

**Constants in and Operations on the Universe**

The test operations for “being a Real” in a universe type were provided as follows:

**constdefs**

\[
\begin{align*}
\text{sup} & :: ((\text{a } \times \text{ b up}) \Rightarrow \text{a up}) \\
\text{sup obj} & \equiv \text{fst(drop(obj)))} \\
\text{base} & :: (\text{a } \times \text{ b up}) \Rightarrow \text{b up} \\
\text{base x} & \equiv \text{case x of lift (a,b) } \Rightarrow \text{b up} \\
| \text{down} & \Rightarrow \text{down}
\end{align*}
\]

**constdefs**

\[
\begin{align*}
\text{refbase} & :: (\text{a } \times \text{ b } \times \text{ c up}) \Rightarrow \text{c up} \\
\text{refbase x} & \equiv \text{case x of lift (a,(b,c)) } \Rightarrow \text{c | down } \Rightarrow \text{down}
\end{align*}
\]

**constdefs**

\[
\begin{align*}
\text{ref_of} & :: (\text{a } \times \text{ b } \times \text{ c up}) \Rightarrow \text{b} \\
\text{ref_of x} & \equiv \text{fst(snd(drop(x)))}
\end{align*}
\]

**lemma** DEF_baseD \[\text{simp} \]: DEF\text{(base\text{(obj))} \Rightarrow \text{DEF\text{(obj)}}

\[
\begin{align*}
\text{apply (simp add: DEF_def base_def not_down_exists_lift)} \\
\text{apply (cases obj, simp)} \\
\text{apply (auto)} \\
\text{done}
\end{align*}
\]

**lemma** DEF_baseD2 \[\text{simp} \]: DEF\text{(base\text{(obj))} \Rightarrow \text{DEF\text{(snd\text{(drop\text{(obj))}})

\[
\begin{align*}
\text{apply (simp add: DEF_def base_def not_down_exists_lift)} \\
\text{apply (cases obj, simp)} \\
\text{apply (auto)} \\
\text{done}
\end{align*}
\]

**lemma** base_snd_drop: DEF\text{(base\text{(obj))} \Rightarrow base\text{(obj)} = \text{snd\text{(drop\text{(obj))}}

\[
\begin{align*}
\text{apply (simp add: DEF_def base_def not_down_exists_lift)} \\
\text{apply (cases obj, simp)} \\
\text{apply (auto)} \\
\text{done}
\end{align*}
\]
Appendix B. Isabelle Theories

lemma DEF_refbaseD [simp]: DEF(refbase(obj)) ⇒ DEF(obj)
apply (simp add: DEF_def refbase_def not_down_exists_lift)
apply (cases obj, simp)
done

lemma DEF_refbaseD2 [simp]: DEF(refbase(obj)) ⇒ DEF(snd(drop(obj)))
apply (simp add: DEF_def refbase_def not_down_exists_lift)
apply (cases obj, simp)
done

lemma refbase snd drop: DEF(refbase(obj)) ⇒ refbase(obj) = snd(sndl drop(obj)))
apply (simp add: DEF_def refbase_def not_down_exists_lift)
apply (cases obj, simp)
done

lemma base UU [simp]: base ⊥ = ⊥
by (simp add: base_def )

lemma base down [simp]: base down = down
by (simp add: base_def )

lemma base snd lift [simp]: base (lift(a,X)) = X
by (simp add: base_def )

lemma refbase UU [simp]: refbase ⊥ = ⊥
by (simp add: refbase_def )

lemma refbase down [simp]: refbase down = down
by (simp add: refbase_def )

lemma refbase snd lift [simp]: refbase (lift(a,(b,X))) = X
by (simp add: refbase_def )

types 'a OCLtype = 'a U set

constdefs noext :: 'a ⇒ ('a × 'b up)
noext x ≡ (x,⊥)

lemma fst noext id [simp]: fst(noext x)=x
B.4. Library

apply (simp (no_asm) add: noext_def)
done

lemma snd_noext_UU [simp]: snd(noext x) = ⊥
apply (simp (no_asm) add: noext_def)
done

end

B.4. Library

B.4.1. The OCL let expression

theory OCL_Let
imports $HOLOCL_HOME/src/Lifting
begin

The OCL let expression is a purely syntactical abbreviation (see [41, p. A-26]) therefore it can be directly mapped to the HOL let.

constdefs
OclLet :: "[(′a,′b) VAL, (′a,′b) VAL ⇒ (′a,′b:bot) VAL] ⇒ (′a,′b) VAL"
OclLet ≡ Let

syntax _OclLet_std :: "[(′a,′b) VAL, (′a,′b) VAL ⇒ (′a,′b:bot) VAL] ⇒ (′a,′b) VAL"
(_OclLet_std) :: 

"([(′a,′b) VAL, (′a,′b) VAL ⇒ (′a,′b:bot) VAL] ⇒ (′a,′b) VAL"

syntax _OclLet_ascii :: "[(′a,′b) VAL, (′a,′b) VAL ⇒ (′a,′b:bot) VAL] ⇒ (′a,′b) VAL"
(_OclLet_ascii) :: 

syntax (xsymbols) _OclLet_math :: "[(′a,′b:bot) VAL] ⇒ (′a,′b) VAL"
(_OclLet_math) :: 

end

B.4.2. OCL Boolean

theory OCL_Boolean
imports $HOLOCL_HOME/src/OCL_University $HOLOCL_HOME/src/OCL_Let
begin
Appendix B. Isabelle Theories

The Characteristic Set of the Type Boolean

Case distinction over a boolean

lemma boolean_cases_sem:
  assumes undefC: x = ⊥ ⇒ R
  and  trueC: x = True ⇒ R
  and  falseC: x = False ⇒ R
  shows  R
  apply (rule disjE)
  prefer 2 apply (erule undefC)
  prefer 2 apply (rule disjE)
  prefer 2 apply (rotate_tac -1, erule trueC)
  prefer 2 apply (rotate_tac -1, erule falseC)
  apply (auto simp: not_down_exists_lift)
  done

constdefs
  is_Boolean_0 :: 'a U ⇒ bool
  is_Boolean_0 ≡ sum_case (λx. False)
                     (sum_case (λx. False)
                     (sum_case (λx. True) (λx. False)))

get_Boolean_0 :: 'a U ⇒ Boolean_0
get_Boolean_0 ≡ sum_case (λx. True)
                  (sum_case (λx. False)
                  (sum_case (λx. True) (λx. False)))

mk_Boolean_0 :: Boolean_0 ⇒ 'a U
mk_Boolean_0 ≡ Inr ◦ Inr ◦ Inr ◦ Inl

lemma get_mk_Boolean_id_0: get_Boolean_0 (mk_Boolean_0 x) = x
  apply (simp add: get_Boolean_0_def mk_Boolean_0_def)
  done

lemma is_mk_Boolean_0: is_Boolean_0 (mk_Boolean_0 x) = True
  apply (simp add: is_Boolean_0_def mk_Boolean_0_def)
  done

lemma mk_get_Boolean_id_0: is_Boolean_0 x ⇒ mk_Boolean_0 (get_Boolean_0 x) = x
  apply (simp add: is_Boolean_0_def mk_Boolean_0_def)
  apply (case_tac x, simp)
  apply (case_tac b, simp)
  apply (case_tac ba, simp)
  apply (case_tac bb, simp)
  done

constdefs
  Boolean_0 :: (′τ, Boolean_0 Set_0) VAL
  Boolean_0 ≡ lift0 (Abs_Set_0 ◦ lift ◦ UNIV)

172
Semantics of Boolean expressions, i.e., the logical operators of the language. They build a Kleene-Logic, i.e., the usual algebraic laws hold. Moreover, all operators are context passing. This results in proof support based on extensive ternary case distinctions \((P(X) \text{ holds iff } P(\bot) \text{ and } P(\top) \text{ and } P(\lnot))\).

**OCL Boolean expressions**

**The Core of the Language**

These operators are defined via lifting principles from \([\text{HOL}](see \[13\])). A more textbook-like “definition” of \([\text{OCL}](following the standard) is derived in the theorem section.

\[
\begin{align*}
\text{constdefs} &\quad \text{OclIsDefined} :: (\tau, \text{bot}) \Rightarrow \tau \\
\text{OclTrue} &\quad \equiv \text{lift}_1(\text{DEF}) \\
\text{OclFalse} &\quad \equiv \text{lift}_0(\text{False}) \\
\text{OclNot} &\quad \equiv \text{lift}_0(\text{Not}) \\
\text{OclAnd} &\quad \equiv \text{lift}_2(\lambda x y. \text{if DEF } x \text{ then if DEF } y \text{ then } (\lnot x) \land (\lnot y) \text{ else if DEF } y \text{ then } \bot \text{ else } \text{False} \text{ else } \bot) \\
\text{OclIf} &\quad \equiv \text{lift}_3(\lambda x y z. \text{strictify}(\lambda a. \text{if } \lnot a \text{ then } y \text{ else } z) x) \\
\text{OclSand} &\quad \equiv \text{lift}_2(\text{strictify}(\lambda x. \text{strictify}(\lambda y. (\lnot x) \land (\lnot y))))
\end{align*}
\]

The following statements represent the technical setup of Isabelle’s syntax engine for this core-language. All these technical setups can be ignored in first reading.
Appendix B. Isabelle Theories

syntax

_OclIsDefined_std :: (′τ,′α::bot) VAL ⇒ 'τ Boolean
  (→IsDefined() (_) [60]60)

_OclIsUndefined_std :: (′τ,′α::bot) VAL ⇒ 'τ Boolean
  (→oclIsUndefined() (_) [60]60)

_OclTrue_std :: 'τ Boolean
  (true)

_OclFalse_std :: 'τ Boolean
  (false)

_OclNot_std :: 'τ Boolean ⇒ 'τ Boolean
  (not ( ) [59]59)

_OclAnd_std :: [′τ Boolean,′τ Boolean] ⇒ 'τ Boolean
  ((_ and _) [57]57)

_OclIf_std :: [′τ Boolean,(′τ,b:bot)VAL,(′τ,b::bot)VAL] ⇒ (′τ,b)VAL
  ((if (_) / then (_) /else (_) /endif) 15)

_OclSand_std :: [′τ Boolean,′τ Boolean] ⇒ 'τ Boolean
  ((_ sand _) [57]57)

syntax

_OclIsDefined_ascii :: (′τ,′α::bot) VAL ⇒ 'τ Boolean
  (isdef ( ) [60]60)

_OclIsUndefined_ascii :: (′τ,′α::bot) VAL ⇒ 'τ Boolean
  (isundef ( ) [60]60)

_OclTrue_ascii :: 'τ Boolean
  (TRUE)

_OclFalse_ascii :: 'τ Boolean
  (FALSE)

_OclNot_ascii :: 'τ Boolean ⇒ 'τ Boolean
  (not ( ) [59]59)

_OclAnd_ascii :: [′τ Boolean,′τ Boolean] ⇒ 'τ Boolean
  ((_ and _) [57]57)

_OclIf_ascii :: [′τ Boolean,(′τ,b:bot)VAL,(′τ,b::bot)VAL] ⇒ (′τ,b)VAL
  ((if (_) / then (_) /else (_) /endif) 15)

_OclSand_ascii :: [′τ Boolean,′τ Boolean] ⇒ 'τ Boolean
  ((_ sand _) [57]57)

syntax (xsymbols)

_OclIsDefined_math :: (′τ,′α::bot) VAL ⇒ 'τ Boolean
  (∂ ( ) [60]60)

_OclIsUndefined_math :: (′τ,′α::bot) VAL ⇒ 'τ Boolean
  (∅ ( ) [60]60)

_OclTrue_math :: 'τ Boolean
  (T)

_OclFalse_math :: 'τ Boolean
  (F)

_OclNot_math :: 'τ Boolean ⇒ 'τ Boolean
  (¬ ( ) [39]39)
Is OCL Boolean Faithful to the OCL Standard?

In this section, we present background material and the proof work for the section 4.5.

constdefs
  semfun :: \('\alpha \Rightarrow '\alpha \) 
  \(\text{Sem}[[\_]\]](x) \equiv x\)

We show for our first strict operation in OCL, the \texttt{not} operator, that it is in fact an instance of the standards definition scheme:

\begin{verbatim}
lemma not_faithfully_represented:
  \(\text{Sem}[[\neg X]] \gamma = (if \text{Sem}[[X]] \gamma \neq \bot
  \text{then} \neg \text{Sem}[[X]] \gamma \text{ else } \bot)\)

by(simp add:OclNot_def
  DEF_def o_def
  lift0_def lift1_def lift2_def
  semfun_def)
\end{verbatim}

The proof is trivial and canonic: it consists of the unfolding of all combinator definitions (they are just abbreviations of re-occurring patterns in the textbook style definitions used to make these patterns explicit!) and the semantic function which is a mere syntactic marker in our context.

The definition of basic operations for the special case of boolean operations is done by a truth table [41, table A 2, page A-12]. We represent this table for the elementary "and" case as following:

\begin{verbatim}
lemma and_faithfully_represented:
  \(\text{Sem}[[X \land Y]] \gamma =
  (if \text{Sem}[[X]] \gamma = \bot \land \text{Sem}[[Y]] \gamma = \bot
  \text{then } \bot,\text{ else if } \text{Sem}[[X]] \gamma = \bot \land \text{Sem}[[Y]] \gamma = \bot
  \text{then } \bot,\text{ else if } \text{Sem}[[X]] \gamma = \bot \land \text{Sem}[[Y]] \gamma = \bot
  \text{then } \bot,\text{ else if } \text{Sem}[[X]] \gamma = \bot \land \text{Sem}[[Y]] \gamma = \bot
  \text{then } \bot,\text{ else } \bot)\)
\end{verbatim}
Appendix B. Isabelle Theories

apply (simp add: OclAnd_def DEF_def a_def strictify_def
lift0_def lift1_def lift2_def
semfun_def )

apply (cases X γ = down)
apply (cases Y γ = down)
apply (simp_all add: not_down_exists_lift)
by auto

For the operators or, xor, and implies, we refrain from an explicit proof of con-
formance with \[41, table A 2, page A-12\]. Instead, we preferred to define these
operators as usual in terms of and and not. However, the subsequent lemmas OclUn-
defined__or__True represent another way to directly check that our definitions meet
the standard.

The Extended Core of the Language
constdefs
OclOr :: [′τ Boolean, ′τ Boolean] ⇒ ′τ Boolean
OclOr S T ≡ ¬ (¬ S ∧ ¬ T)

OclSor :: [′τ Boolean, ′τ Boolean] ⇒ ′τ Boolean
OclSor S T ≡ ¬ (¬ S ⊔ ¬ T)

Alternative definitions for the use with ocl_setup_op:

lemma OclOr_alt_def: OclOr ≡ lift2 (λ x y.
(if (DEF x)
then (if (DEF y) then (⌜x⌝ ⊔ ⌜y⌝).
else (if (DEF y) then (True, else ⊥)))
else (if (DEF y) then (if (⌜y⌝ then (True, else ⊥) else ⊥)))))
apply (rule eq_reflection, (rule ext)+)
by (simp add: OclOr_def OclAnd_def OclNot_def ss_lifting)

lemma OclSor_alt_def: OclSor ≡ lift2 (strictify (λ x. strictify (λ y.⌜x⌝ ⊔ ⌜y⌝)))
apply (rule eq_reflection, (rule ext)+)
by (simp add: OclSor_def OclNot_def OclSand_def ss_lifting')

syntax
_ OclOr_std :: [′τ Boolean, ′τ Boolean] ⇒ ′τ Boolean
(_ or/ _) [53,54][53]

_ OclSor_std :: [′τ Boolean, ′τ Boolean] ⇒ ′τ Boolean
(_ sor/ _) [53,54][53]

syntax
_ OclOr_ascii :: [′τ Boolean, ′τ Boolean] ⇒ ′τ Boolean
(_ or/ _) [53,54][53]
B.4. Library

\_OclSor\_ascii :: ['τ Boolean, 'τ Boolean] \Rightarrow 'τ Boolean
\((\_\_\_\_\_\_\text{or}\_\_\text{or}) [53,54],[53]\)

\textbf{syntax}

\_OclOr\_math :: ['τ Boolean, 'τ Boolean] \Rightarrow 'τ Boolean
\((\_\_\_\_\_\_\_\_\_\text{or}\_\_\_\text{or}) [53,54],[53]\)

\textbf{constdefs}

\textbf{OclXor} :: ['τ Boolean, 'τ Boolean] \Rightarrow 'τ Boolean
\textbf{OclXor} S T ≡ \((\_\_\_\_\_\text{or}\_\_\text{or}) [53,54],[53]\)

\textbf{OclImplies} :: ['τ Boolean, 'τ Boolean] \Rightarrow 'τ Boolean
\textbf{OclImplies} S T ≡ \((\_\_\_\_\_\_\_\_\_\text{implies}\_\_\_\text{implies}) [53,54],[53]\)

Alternative definitions for the use with \texttt{ocl\_setup\_op}

\textbf{lemma} \textbf{OclXor\_alt\_def}: \textbf{OclXor} ≡ lift2 \((\_\_\_\_\_\_\_\_\_\text{or}\_\_\_\text{or}) [53,54],[53]\)
\begin{axistabular}{l}
\begin{align*}
\text{if} (\text{DEF} x) \\
\text{then} \text{if} (\text{DEF} y) \text{then} (\text{if} (\_\_\_\_\_\text{or}\_\_\text{or}) [53,54],[53]\) \text{else} \bot) \\
\text{else} \bot
\end{align*}
\end{axistabular}
\text{apply}\((\text{rule eq\_reflection}, (\text{rule ext}))+\)
\text{by}(\text{simp add: OclXor\_def OclOr\_alt\_def OclAnd\_def OclNot\_def ss\_lifting})

\textbf{lemma} \textbf{OclImplies\_alt\_def}: \textbf{OclImplies} ≡ lift2 \((\_\_\_\_\_\_\_\_\text{implies}\_\_\_\text{implies}) [53,54],[53]\)
\begin{axistabular}{l}
\begin{align*}
\text{if} (\text{DEF} x) \\
\text{then} \text{if} (\text{DEF} y) \text{then} (\text{if} (\_\_\_\_\_\text{or}\_\_\text{or}) [53,54],[53]\) \text{else} (\text{if} True_\_\_\_\_\text{or}\_\_\text{or} \text{else} \bot)) \\
\text{else} \text{if} (\text{DEF} y) \text{then} (\text{if} True_\_\_\_\_\text{or}\_\_\text{or} \text{else} \bot) \text{else} \bot)
\end{align*}
\end{axistabular}
\text{apply}\((\text{rule eq\_reflection}, (\text{rule ext}))+\)
\text{by}(\text{simp add: OclImplies\_def OclAnd\_def OclNot\_def OclOr\_alt\_def OclIsDefined\_def ss\_lifting})

There is an inconsistency between the normative part \[41\] Chapter 11] and the informative \[41\] Appendix A\] the standard. For reasons discussed below, we chose the \textbf{implies} version defined by \[41\] Table A-2, page A-12\] and define the other version later.

\textbf{constdefs}

\textbf{OclSxor} :: ['τ Boolean, 'τ Boolean] \Rightarrow 'τ Boolean
\textbf{OclSxor} S T ≡ \((\_\_\_\_\_\_\_\_\_\text{or}\_\_\_\text{or}) [53,54],[53]\)

\textbf{OclSimplies} :: ['τ Boolean, 'τ Boolean] \Rightarrow 'τ Boolean
\textbf{OclSimplies} S T ≡ \((\_\_\_\_\_\_\_\_\_\text{implies}\_\_\_\text{implies}) [53,54],[53]\)

Alternative definitions for the use with \texttt{ocl\_setup\_op}

\textbf{lemma} \textbf{OclSxor\_alt\_def}:
\textbf{OclSxor} ≡ lift2 \((\_\_\_\_\_\_\_\_\_\text{or}\_\_\_\text{or}) [53,54],[53]\)
\begin{axistabular}{l}
\begin{align*}
\text{if} (\text{DEF} x) \\
\text{then} \text{if} (\text{DEF} y) \text{then} (\text{if} (\_\_\_\_\_\text{or}\_\_\text{or}) [53,54],[53]\) \text{else} \bot) \\
\text{else} \bot
\end{align*}
\end{axistabular}
\text{apply}\((\text{rule eq\_reflection}, (\text{rule ext}))+\)
\text{by}(\text{simp add: OclSxor\_def OclSxor\_alt\_def OclAnd\_def OclNot\_def ss\_lifting})

The truth for \textbf{implies} does not fulfill the normative requirement.
Appendix B. Isabelle Theories

apply (rule eq_reflection, (rule ext)+)
by(auto simp: OclSxor_def OclSor_def OclNot_def OclSand_def ss_lifting)

lemma OclSimplies_alt_def:
OclSimplies ≡ lift2 (strictify (λ x. strictify (λ y.⌜x⌝ →⌜y⌝)))
apply (rule eq_reflection, (rule ext)+)
by(simp add: OclSimplies_def OclSxor_def OclNot_def OclSand_def ss_lifting)

syntax
_ OclXor_std :: [′τ Boolean, ′τ Boolean] ⇒ ′τ Boolean
(_ xor/ _) [55,56,55]
_ OclImplies_std :: [′τ Boolean, ′τ Boolean] ⇒ ′τ Boolean
(_ implies/ _) [53,52,52]
_ OclSxor_std :: [′τ Boolean, ′τ Boolean] ⇒ ′τ Boolean
(_ xor/ _) [55,56,55]
_ OclSimplies_std :: [′τ Boolean, ′τ Boolean] ⇒ ′τ Boolean
(_ simples/ _) [53,52,52]

syntax
_ OclXor_ascii :: [′τ Boolean, ′τ Boolean] ⇒ ′τ Boolean
(_ xor/ _) [55,56,55]
_ OclImplies_ascii :: [′τ Boolean, ′τ Boolean] ⇒ ′τ Boolean
(_ implies/ _) [53,52,52]
_ OclSxor_ascii :: [′τ Boolean, ′τ Boolean] ⇒ ′τ Boolean
(_ xor/ _) [55,56,55]
_ OclSimplies_ascii :: [′τ Boolean, ′τ Boolean] ⇒ ′τ Boolean
(_ simples/ _) [53,52,52]

syntax
_ OclXor_math :: [′τ Boolean, ′τ Boolean] ⇒ ′τ Boolean
(_ ⊕/ _) [55,56,55]
_ OclImplies_math :: [′τ Boolean, ′τ Boolean] ⇒ ′τ Boolean
(_ −→/ _) [53,52,52]
_ OclSxor_math :: [′τ Boolean, ′τ Boolean] ⇒ ′τ Boolean
(_ ⊕/ _) [55,56,55]
_ OclSimplies_math :: [′τ Boolean, ′τ Boolean] ⇒ ′τ Boolean
(_ −→/ _) [53,52,52]

An implies variant for three-valued logics introduced by Hähnle [18]:

constdefs
OclImplies1 :: [′τ Boolean, ′τ Boolean] ⇒ ′τ Boolean
OclImplies1 S T ≡ S ∨ S ∨ T

An implies variant for three-valued logics introduced by Hennicker et al [23]. This
B.4. Library

variant is also the version actually used in the normative part of the standard.

constdefs
  OclImplies2 :: int
  OclImplies2 S T ≡ (¬ S) ∨ (S ∧ T)

Alternative definitions for the use with ocl_setup_op

lemma OclImplies1_alt_def: OclImplies1 ≡ lift2 (λ x y. 
  if (DEF x) then (if⌜x⌝ then y else ⌞True⌟) 
  else ⌞True⌟) 
apply(rule eq_reflection, (rule ext)+) 
by(simp add: OclImplies1_def OclAnd_def OclNot_def 
  OclOr_alt_def OclIsDefined_def ss_lifting)

lemma OclImplies2_alt_def: OclImplies2 ≡ lift2 (λ x y. 
  if (DEF x) then (if⌜x⌝ then y else ⌞True⌟) 
  else ⊥) 
apply(rule eq_reflection, (rule ext)+) 
by(simp add: OclImplies2_def OclAnd_def OclNot_def 
  OclOr_alt_def OclIsDefined_def ss_lifting)

syntax
  _OclImplies1_std :: int
  _OclImplies2_std :: int
  _OclImplies1_ascii :: int
  _OclImplies2_ascii :: int
  _OclImplies1_math :: int
  _OclImplies2_math :: int
  _OclIf_std :: int
Appendix B. Isabelle Theories

\[(\text{if} \, (_,), \text{then} \, (,_),/\text{else} \, (,_)/\text{endif}) \, 15\]  

Setup of the Basics

lemma OclIsDefined_vs_DEF: \[\partial \, x = T \Rightarrow DEF \, (x \, St)\]
apply (simp add: OclIsDefined_def lift1_def o_def OclTrue_def lift0_def)
apply (drule fun_cong [of _ _ St ] )
apply (auto)
done

lemma DEF_vs_OclIsDefined:  
assumes \(1: (\forall \, St. \, DEF \, (x \, St))\)
s shows \(T = \partial \, x\)
apply (rule ext)
apply (simp add: OclIsDefined_def lift1_def o_def OclTrue_def lift0_def)
apply (rule 1)
done

ML setup ⟨
  val OclUndefined_def = get_def (theory OCL_Undefined) OclUndefined
⟩

ML ⟨
  val lift3_def = get_def (theory Lifting) lift3
  val lift2_def = get_def (theory Lifting) lift2
  val lift1_def = get_def (theory Lifting) lift1
  val lift0_def = get_def (theory Lifting) lift0
  val cp_def = get_def (theory Lifting) cp
⟩

ML ⟨
  val strictify_def = get_def (theory Lifting) strictify
  val isStrict_def = get_def (theory Lifting) isStrict

  val DEF_def = get_def (theory Lifting) DEF
  val not_down_exists_lift = thm not_down_exists_lift
  val up_split = thm up.split
  val cp_lift0-fw = thm cp_lift0-fw
  val cp_lift1-fw = thm cp_lift1-fw
  val cp_lift2-fw = thm cp_lift2-fw
  val cp_lift3-fw = thm cp_lift3-fw
  val DEF_strictify_DEF_args = thm DEF_strictify_DEF_args
  val DEF_strictify_DEF_args2 = thm DEF_strictify_DEF_args2
  val lift1b_undef = thm lift1b_undef
  val lift2_undef1a = thm lift2_undef1a
  val lift2_undef2a = thm lift2_undef2a
  val cp_compose2 = thm cp_compose2
  val UU_fun_def = get_def (theory Lifting) UU_fun

180
B.4. Library

ML ⟨ ⟨
val OclIsDefined_def = thm OclIsDefined_def
val OclTrue_def = thm OclTrue_def
val OclFalse_def = thm OclFalse_def
val OclNot_def = thm OclNot_def
val OclAnd_def = thm OclAnd_def
val OclIf_def = thm OclIf_def
val OclOr_def = thm OCL_Boolean.OclOr_def
val OclXor_def = thm OCL_Boolean.OclXor_def
val OclImplies_def = thm OCL_Boolean.OclImplies_def
val OclImplies1_def = thm OCL_Boolean.OclImplies1_def
val OclImplies2_def = thm OCL_Boolean.OclImplies2_def
⟩ ⟩

The Basic Laws for the Boolean Operators

ML_setup ⟨ ⟨
fun prover1 (goal,prems,name) =
  let val thm = prove_goalw (the_context()) (DEF_def::strictify_def::o_def::
lift0_def::lift1_def::lift2_def::lift3_def::prems) goal
    (fn _ =>
      rtac ext 1,
      ALLGOALS(asm_full_simp_tac
        (simpset())
        addsplits [split_if,up_split]),
      ALLGOALS(asm_full_simp_tac
        (simpset())
        addsimp [not_down_exists_lift]),
      auto_tac(claset(),simpset()))
  in bind_thm(name,thm);
  Adssimps[thm]
  end;
⟩ ⟩

ML_setup ⟨ ⟨
map prover1
[(\\bottom = \\false, [OclIsDefined__false,OclTrue__false,OclFalse__false,OclUndefined__false,
   OclIsDefined__undef],
   \\false T = T, [OclIsDefined__false,OclTrue__false,OclFalse__false,OclUndefined__false,
   OclIsDefined__true],
   \\false F = T, [OclIsDefined__false,OclTrue__false,OclFalse__false,OclUndefined__false,
   OclIsDefined__false],
   (\bottom \neg = \false, [OclNot__false,OclTrue__false,OclFalse__false,OclUndefined__false,
   not__false],
   (\true \neg = \false, [OclNot__true,OclTrue__true,OclFalse__false,OclUndefined__false,
   not__true],)}]
Appendix B. Isabelle Theories

\(- F = T, [OclNot_def, OclTrue_def, OclFalse_def, OclUndefined_def], not__False);\]

⟩ ⟩

This shows the following facts: \( \partial \perp = F \)
\( \partial T = T \)
\( \partial F = T \)
\( \neg \perp = \perp \)
\( \neg T = F \)
\( \neg F = T \)

...which also prove the compliance with Table A.2 in the standard 2.0.

ML ⟨ ⟨
map prover1
[⟨ ⟨\perp \land \perp = \perp, [OclAnd_def, OclTrue_def, OclFalse_def, OclUndefined_def], undef__and__undef⟩,
(\perp \land T = \perp, [OclAnd_def, OclTrue_def, OclFalse_def, OclUndefined_def], undef__and__True⟩,
(\perp \land F = F, [OclAnd_def, OclTrue_def, OclFalse_def, OclUndefined_def], undef__and__False⟩,
(T \land \perp = \perp, [OclAnd_def, OclTrue_def, OclFalse_def, OclUndefined_def], True__and__undef⟩,
(T \land T = T, [OclAnd_def, OclTrue_def, OclFalse_def, OclUndefined_def], True__and__True⟩,
(F \land T = F, [OclAnd_def, OclTrue_def, OclFalse_def, OclUndefined_def], False__and__True⟩,
(F \land F = F, [OclAnd_def, OclTrue_def, OclFalse_def, OclUndefined_def], False__and__False⟩);\⟩⟩

ML ⟨ ⟨
map prover1
[⟨ ⟨\perp \lor \perp = \perp, [OclOr_def, OclUndefined__or__undef⟩,
(\perp \lor T = T, [OclOr_def, OclUndefined__or__True⟩,
(\perp \lor F = F, [OclOr_def, OclUndefined__or__False⟩,
(T \lor \perp = T, [OclOr_def, True__or__undef⟩,
(T \lor T = T, [OclOr_def, True__or__True⟩,
(T \lor F = T, [OclOr_def, True__or__False⟩,
(F \lor \perp = \perp, [OclOr_def, False__or__False⟩,\⟩⟩

182
\((F \lor T = T, \text{OclOr_def, False\_or\_True})\),
\((F \lor F = F, \text{OclOr_def, False\_or\_False})\);
Appendix B. Isabelle Theories

\[ \vdash \bot  \rightarrow \bot = \bot, \quad \text{OclImplies2_def}, \quad \text{OclUndefined__implies2__undef} \]
\[ \vdash \bot  \rightarrow \top = \bot, \quad \text{OclImplies2_def}, \quad \text{OclUndefined__implies2__True} \]
\[ \vdash \bot  \rightarrow \bot = \bot, \quad \text{OclImplies2_def}, \quad \text{OclUndefined__implies2__FALSE} \]
\[ \vdash \top  \rightarrow \bot = \bot, \quad \text{OclImplies2_def}, \quad \text{True__implies2__undef} \]
\[ \vdash \top  \rightarrow \top = \top, \quad \text{OclImplies2_def}, \quad \text{True__implies2__True} \]
\[ \vdash \top  \rightarrow \bot = \bot, \quad \text{OclImplies2_def}, \quad \text{True__implies2__False} \]
\[ \vdash \bot  \rightarrow \top = \top, \quad \text{OclImplies2_def}, \quad \text{FALSE__implies2__True} \]
\[ \vdash \bot  \rightarrow \bot = \bot, \quad \text{OclImplies2_def}, \quad \text{False__implies2__False} \]

ML \( \langle \langle \text{map prover1} \rangle \rangle \)

\[ \langle \langle \text{if } \bot \text{ then } Y \text{ else } Z \text{ endif} = \bot, \quad \text{if_undef} \rangle \]
\[ \langle \langle \text{if } \top \text{ then } Y \text{ else } Z \text{ endif} = \bot, \quad \text{if_TRUE} \rangle \]
\[ \langle \langle \text{if } \bot \text{ then } Y \text{ else } Z \text{ endif} = \bot, \quad \text{if_FALSE} \rangle \]

All operators are contextpassing

lemmas \( \text{cp_undef} = \text{OclUndefined_def}[\text{THEN } \text{cp_lift0_fw}, \text{standard}] \)

lemmas \( \text{cp_OclIsDefined} = \text{OclIsDefined_def}[\text{THEN } \text{cp_lift1_fw}, \text{standard}] \)

lemmas \( \text{cp_not} = \text{OclNot_def}[\text{THEN } \text{cp_lift1_fw}, \text{standard}] \)

lemmas \( \text{cp_and} = \text{OclAnd_def}[\text{THEN } \text{cp_lift2_fw}, \text{standard}] \)

lemmas \( \text{cp_if} = \text{OclIf_def}[\text{THEN } \text{cp_lift3_fw}, \text{standard}] \)

declare \( \text{cp_undef} \) [simp, intro!]
\( \text{cp_OclIsDefined} \) [simp, intro!]
\( \text{cp_not} \) [simp, intro!]
\( \text{cp_and} \) [simp, intro!]
\( \text{cp_if} \) [simp, intro!]

ML \( \langle \langle \text{fun prover2} \rangle \rangle \)

fun prover2 (goal,prems,name) =
  let val thm = prove_goalw (the_context()) (prems) goal
  in bind thm(name,thm);
  AddSimpS [thm];
  AddSIs [thm];
end;
This shows the following facts:

\[ \lnot P \land \lnot P' \implies \lnot (P \lor P') \]
\[ \lnot P \land \lnot P' \implies \lnot (P \land P') \]
\[ \lnot P \land \lnot P' \implies \lnot (P \lor P') \]

Proof Technique: High-level Trichometry on context passing contexts

lemma base_distinct:

\[ X \tau = \bot \lor X \tau = \top \lor X \tau = \bot \]

apply (simp add: OclTrue_def OclFalse_def OclUndefined_def)

apply (auto)

apply (erule exE)

done

ML_setup "val base_distinct = thm base_distinct"

ML_setup "val prems = goalw (the_context()) []

(X \tau = \bot \lor X \tau = \top \lor X \tau = \bot)

by(cut_facts_tac [base_distinct] 1);

by(Simp_tac 1);

ba 1;

by(ALLGOALS(Assm_tac (simp_tac (simpset()) addsimps

[OclFalse_def, OclTrue_def, lift0_def, OclUndefined_def])));

qedbase_distinct_1;

val prems = goalw (the_context()) []

(X \tau = \bot \lor X \tau = \top \lor X \tau = \bot)

by(cut_facts_tac [base_distinct] 1);

by(Step_tac 1);

ba 1;

by(ALLGOALS(Assm_tac (simp_tac (simpset()) addsimps

[OclFalse_def, OclTrue_def, lift0_def, OclUndefined_def])));

qedbase_distinct_2;
Appendix B. Isabelle Theories

val prems = goalw (the_context()) []
  (X τ ~ T) = (X τ = T τ | X τ = F τ);
by (cut_facts_tac [base_distinct] 1);
by (Step_tac 1);
by (ALLGOALS (asm_full_simp_tac ([OclFalse_def,OclTrue_def,lift0_def,OclUndefined_def])));
by (Fast_tac 1);
qed distinct_3;

lemma cp_distinct_core:
  [ cp P; cp P’; 
    P ⊥ τ = P’ ⊥ τ;
    P T τ = P’ T τ;
    P F τ = P’ F τ 
  ] \implies X τ = P X τ 
apply (unfold cp_def)
apply (erule exE)
apply (erule exE)
apply (rotate_tac -2)
apply simp
apply (cut_tac X = X and τ = τ in base_distinct)
apply (erule disjE)
apply (erule_tac [2] disjE)
apply (auto)
done
ML ("val cp_distinct_core = thm cp_distinct_core")

lemma cp_distinct_core_P:
  [ cp P; cp P’; 
    X τ = ⊥ τ \implies P ⊥ τ = P’ ⊥ τ;
    X τ = T τ \implies P T τ = P’ T τ;
    X τ = F τ \implies P F τ = P’ F τ 
  ] \implies X τ = P’ X τ 
apply (unfold cp_def)
apply (erule exE)
apply (erule exE)
apply (rotate_tac -2)
apply simp
apply (cut_tac X = X and τ = τ in base_distinct)
apply (erule disjE)
apply (erule_tac [2] disjE)
apply (auto)
done
ML ("val cp_distinct_core_P = thm cp_distinct_core_P")
lemma local_validity_propagation1:
\[
\begin{align*}
X \tau &= \bot \tau; \\
P \bot \tau &= P' \bot \tau; \\
cp P; cp P' \\
\end{align*}
\]
\[\implies (P; (\tau) \text{Boolean} \Rightarrow (\tau', \alpha) \text{VAL}) X \tau = P' X \tau\]
apply (rule_tac P = P and P' = P' in cp_distinct_core_P)
apply auto
apply (simp_all add: OclTrue_def OclFalse_def OclUndefined_def lift0_def)
done
ML ⟨ ⟨ val local_validity_propagation1 = thm local_validity_propagation1 ⟩ ⟩

lemma local_validity_propagation2:
\[
\begin{align*}
X \tau &= T \tau; \\
P T \tau &= P' T \tau; \\
cp P; cp P' \\
\end{align*}
\]
\[\implies (P; (\tau) \text{Boolean} \Rightarrow (\tau', \alpha) \text{VAL}) X \tau = P' X \tau\]
apply (rule_tac P = P and P' = P' in cp_distinct_core_P)
apply auto
apply (simp_all add: OclTrue_def OclFalse_def OclUndefined_def lift0_def)
done
ML ⟨ ⟨ val local_validity_propagation2 = thm local_validity_propagation2 ⟩ ⟩

lemma local_validity_propagation3:
\[
\begin{align*}
X \tau &= F \tau; \\
P F \tau &= P' F \tau; \\
cp P; cp P' \\
\end{align*}
\]
\[\implies (P; (\tau) \text{Boolean} \Rightarrow (\tau', \alpha) \text{VAL}) X \tau = P' X \tau\]
apply (rule_tac P = P and P' = P' in cp_distinct_core_P)
apply auto
apply (simp_all add: OclTrue_def OclFalse_def OclUndefined_def lift0_def)
done
ML ⟨ ⟨ val local_validity_propagation3 = thm local_validity_propagation3 ⟩ ⟩

lemma cp_distinct:
\[
\begin{align*}
cp P; cp P'; \\
P \bot = P' \bot; \\
P T = P' T; \\
P F = P' F \\
\end{align*}
\]
\[\implies P X = P' X\]
apply (rule ext)
apply (drule_tac f = P \bot and x = x in fun_cong)
apply (drule_tac f = P T and x = x in fun_cong)
Appendix B. Isabelle Theories

apply (drule_tac f = P F and x = x in fun_cong)
apply (rule cp_distinct_core)
back
apply (auto)
done

ML ⟨ ⟨ val cp_distinct = thm cp_distinct ⟩ ⟩

lemma cp_distinct1:
[ \partial X = T; cp P; cp P';
  P T = P' T;
  P F = P' F
] \Longrightarrow P X = P' X
apply (unfold cp_def OclIsDefined_def o_def lift1_def)
apply (rule ext)
apply (drule_tac f = P T and x = x in fun_cong)
apply (drule_tac f = P F and x = x in fun_cong)
apply (drule_tac f = \lambda c. DEF (X c) \ and x = x in fun_cong)
apply (erule exE)
apply (erule exE)
apply (rotate_tac -2)
apply simp
apply (cut_tac X = X and \tau = x in base_distinct)
apply (erule disjE)
apply (erule_tac [2] disjE)
apply (simp)
apply (simp add: DEF_def lift0_def OclUndefined_def OclTrue_def)
apply (auto)
done

ML ⟨ ⟨ val cp_distinct1 = thm cp_distinct1 ⟩ ⟩

lemma cp_distinct2:
[ cp P; cp P';
  \land X. X = \bot \Longrightarrow P X = P' X;
  \land X. X = T \Longrightarrow P X = P' X;
  \land X. X = F \Longrightarrow P X = P' X
] \Longrightarrow P X = P' X
apply (rule cp_distinct)
apply auto
done

ML ⟨ ⟨ val cp_distinct2 = thm cp_distinct2 ⟩ ⟩

The lemmas for st_transissibility allow for a new, high-level proof style based on case-distinctions avoiding unfolding the definition of and, …

ML ⟨ ⟨
fun Ocl_case_tac str n =
    EVERY [(eres_inst_tac [(X,str)] cp_distinctI n) ORELSE
       (res_inst_tac [(X,str)] cp_distinct n),
       TRY(Fast_tac n), TRY(Fast_tac n),
       Auto_tac]
Appendix B. Isabelle Theories

false, or_and_distrib1),
(Z ∨ (X ∧ Y) = (Z ∨ X) ∧ (Z ∨ Y), [Z,Y,X,X],
false, or_and_distrib2);

ML ⟨
map prover3
(((X ∧ Y) = ¬(¬(X) ∨ ¬(Y)), [X,Y],[false, de_morgan],
(¬(X ∧ Y) = (¬(X) ∨ ¬(Y)), [X,Y],[false, de_morgan1],
¬(¬(X) ∧ ¬(Y)), [X,Y],[false, de_morgan2]);
⟩

ML ⟨
map prover3
((X ⊕ true, [X], true, xor_true_inverse_1),
(T ⊕ X = ¬ X, [X], true, xor_true_inverse_2),
(X ⊕ F = X, [X], true, xor_false_neutral_1),
(F ⊕ X = X, [X], true, xor_false_neutral_2),
(X ⊕ Y = Y ⊕ X, [X,Y],[false, xor_commute]),
((X ⊕ Y) ⊕ Z = X ⊕ (Y ⊕ Z),
[X,Y,Z,Z,Y,Z],[false, xor_assoc]);
⟩

We will not include the or-generating rewrite rules into the rewriter since they do not "simplify" the situation. As rules, they are important, of course.

ML ⟨
map prover3
((X → F = ¬ X,[X], true, implies_false),
((X → T) = T, [X], true, implies_true1),
((X → ⊥) = ¬ X ∨ ⊥, [X], false, implies_undef1),
((F → X) = T, [X], true, implies_true2),
((T → X) = X, [X], true, implies_idem),
((⊥ → X) = X ∨ ⊥, [X], false, implies_undef2),
((X → X) = ¬ X ∨ X, [X], false, implies_idem1),
(X → (Y ∧ Z) = ((X → Y) ∧ (X → Z)),
[X,Y,Z,Z],[false, implies_and1],
(X → (Y ∨ Z) = ((X → Y) ∨ (X → Z)),
[X,Y,Z,Z],[false, implies_or1],
(((X ∧ Y) → Z) = X → (Y → Z),
[Z,Y,X,Y,X],[false, implies_or1E],
(((X ∨ Y) → Z) = (X → Z) ∧ (Y → Z),
[Z,Y,X,Y,X,X,X,X],[false, implies_orE],
(X → (Y → Z) = Y → (X → Z),
[Z,Y,X,Y,X],[false, implies_commute]);
⟩

ML ⟨

190
map prover3

\[
\begin{align*}
& (\bot \rightarrow X = \bot, [X], true, \text{implies2}_\text{undef}), \\
& (T \rightarrow X = X, [X], true, \text{implies2}_\text{idem}), \\
& (F \rightarrow X = T, [X], true, \text{implies2}_\text{true2}), \\
& ((X \rightarrow X) = \neg X \vee X, [X], false, \text{implies2}_\text{idem1}), \\
& (X \rightarrow \bot = \neg X \vee \bot, [X], false, \text{implies2}_\text{undef2}), \\
& (X \rightarrow F = \neg X, [X], true, \text{implies2}_\text{false}), \\
& (X \rightarrow T = \neg X \lor X, [X], false, \text{implies2}_\text{true1}) \\
\end{align*}
\]

The properties in comments, crucial for deduction, could not be proven. In fact, they do not hold due to the counter-examples:

Lemma \[ X=\bot; Y=T; Z=T \] \implies (X \rightarrow (Y \land Z) \neq (X \rightarrow Y) \land (X \rightarrow Z))

apply(simp, auto simp: lift0_def OclUndefined_def OclTrue_def)
apply(drule fun_cong, simp)
done

Lemma \[ X=\bot; Y=F \] \implies (X \rightarrow (Y \land Z) \neq (X \rightarrow Y) \land (X \rightarrow Z))

apply(simp, auto simp: lift0_def OclUndefined_def OclTrue_def)
apply(drule fun_cong, simp)
done

Lemma \[ X=\bot; Y=F \] \implies (X \rightarrow (Y \land Z) \neq Y \rightarrow (X \rightarrow Z))

apply(simp, auto simp: lift0_def OclUndefined_def OclTrue_def)
apply(drule fun_cong, simp)
done

191
Appendix B. Isabelle Theories

All these higher logical arrangement rules do not hold for the standards definition \( \forall S \rightarrow \forall T \equiv \forall S \lor \forall T \) of the implication, in contrast to the "classical definition" \( \forall S \rightarrow \forall T \equiv \forall S \lor \forall T \). Although the standards choice is feasible, in the light of these dramatic algebraic deficiencies, we qualify it as glitch from the deduction point of view and suggest to apply the definition used in the appendix.

For the completeness of our presentation, we show the properties of \texttt{implies} introduced by Hähnle [18]:

\textbf{ML \$\langle$} map prover3
\[
\begin{align*}
& (\forall X. \varnothing(X) = \top, \text{true, implies1_idem}), \\
& (\forall X. \varnothing(X) = \top, \text{true, implies1_undef}), \\
& (\forall X. \varnothing(X) = \top \implies (\forall Y. \varnothing(Y) = \bot \rightarrow X, \\
& \text{false, implies1_false}), \\
& ((\forall X. \varnothing(X) = \top) \equiv (\forall Y. \varnothing(Y) = \bot \rightarrow X, \\
& \text{false, implies1_true1}), \\
& ((\forall X. \varnothing(X) = \top) \equiv (\forall Y. \varnothing(Y) = \bot \rightarrow X, \\
& \text{false, implies1_true2}), \\
& ((\forall X. \varnothing(X) = \top) \equiv (\forall Y. \varnothing(Y) = \bot \rightarrow X, \\
& \text{false, implies1_idem2}), \\
& (((\forall X. \varnothing(X) = \top) \equiv (\forall Y. \varnothing(Y) = \bot \rightarrow X, \\
& \text{false, implies1_undef}), \\
& (X \rightarrow Y \rightarrow Z) = Y \rightarrow (X \rightarrow Z), \\
& \text{false, implies1_orE}), \\
& (X \rightarrow Y \rightarrow Z) = Y \rightarrow (X \rightarrow Z), \\
& \text{false, implies1_commute}), \\
& (X \rightarrow Y \rightarrow Z) = (X \rightarrow Y) \land (X \rightarrow Z), \\
& \text{false, implies1_andI}), \\
& (X \rightarrow Y \rightarrow Z) = (X \rightarrow Y) \lor (X \rightarrow Z), \\
& \text{false, implies1_orI})
\end{align*}
\]

\textbf{ML \$\langle$} map prover3
\[
\begin{align*}
& (\neg(\forall X. \varnothing(X) \lor \forall X \land \neg X = \top), \\
& \text{false, kleenical0}), \\
& (\forall X. \varnothing(X) = \top \implies (\forall X \lor \neg X) = \top, \\
& \text{false, kleenical1}), \\
& (\forall X. \varnothing(X) = \top \implies (\neg X \lor X) = \top, \\
& \text{false, kleenical2}), \\
& ((\varnothing \land X \land \neg X) = \top, \\
& \text{false, absurd0}), \\
& (\forall X. \varnothing(X) = \top \implies (\forall X \land \neg X) = \top, \\
& \text{false, absurd1}), \\
& (\forall X. \varnothing(X) = \top \implies (\neg X \land X) = \top, \\
& \text{false, absurd2}), \\
& (\neg(\forall X. \varnothing(X) \lor (\forall X \rightarrow X) = \top), \\
& \text{false, implies_assume0}), \\
& (\forall X. \varnothing(X) = \top \implies (\forall X \rightarrow X) = \top, \\
& \text{false, implies_assume1})
\end{align*}
\]

\textbf{ML \$\langle$} map prover3
\[
\begin{align*}
& (\forall X. \varnothing X = \top \implies \varnothing(\neg X) = \top, \\
& \text{true, isDefined_not})
\end{align*}
\]

192
B.4. Library

(∧Z. [ θ X: T; θ Y: T ] ⇒⇒ ∂(X ∧ Y) = T, [X], true, isDefined_and),
(∧Z. [ θ X: T; θ Y: T ] ⇒⇒ ∂(X ∨ Y) = T, [X], true, isDefined_or),
(∧Z. [ θ X: T; θ Y: T ] ⇒⇒ ∂(X ⊕ Y) = T, [X], true, isDefined_xor),
(∧Z. [ θ X: T; θ Y: T ] ⇒⇒ ∂(X → Y) = T, [X], true, isDefined_implies),
(∧A. [ θ X: T; θ Y: T; θ Z: T ] ⇒⇒ ∂(if X then Y else Z endif) = T, [X], true, isDefined_if)

```
ML ⟨
map_prover3
|((if X then (Y; τ:=>τ;bot) else Z endif) = \ 

\(if \neg X then Y else Z endif)\), [X], false, if_swap),
(∧Z. θ X: T ⇒⇒ (if X then Y else Z endif) = ((X → Y) ∧ (∨ X → Z)), [X], false, if_to_implies
\)
```

The Laws for the is_def Operator

Some basic laws.

lemma isDefined_idem(simp) : ∂(θ X) = T

by(simp add: OclIsDefined_def lift0_def

lift1_def OclUndefined_def o_def OclTrue_def)

ML ⟨
map_prover3
[(∂(θ X ∧ X) = T, [X], true, isDefined_cong1),
(∂(θ X ∧ ¬ X) = T, [X], true, isDefined_cong2),
(∂(X ∧ θ X) = T, [X], true, isDefined_cong3),
(∂(¬ X ∧ θ X) = T, [X], true, isDefined_cong4)];
```

Some generic laws for the is_def operator

This section is essentially a rephrasing of the theory Lifting in the newly established
termology of this theory.

lemma is_isdef_lift0: ∂(lift0(θ, ε)) = T

apply (unfold OclIsDefined_def lift0_def lift1_def OclTrue_def o_def)
apply (rule ext)

193
Appendix B. Isabelle Theories

apply (auto)
done

lemma lift1_strictify_is_isdef:
\[ \frac{}{\partial (\text{lift}_1 (\text{strictify}(\lambda x. f x)))} X = \partial X \]
apply (unfold OclIsDefined_def lift1_def OclUndefined_def o_def)
apply (rule ext)
apply auto
apply (drule DEF_strictify_DEF_args)
apply auto
done

lemma lift2_strictify_is_isdef:
\[ \frac{}{\partial (\text{lift}_2 (\text{strictify}(\lambda x. \text{strictify}(\lambda y. f x y))))} X Y = (\partial X \land \partial Y) \]
apply (rule ext)
apply auto
apply (drule DEF_strictify_DEF_args2)
apply (drule_tac [2] DEF_strictify_DEF_args2)
apply auto
done

declare lift1_strictify_is_isdef [simp] lift2_strictify_is_isdef [simp]

lemma lift1b_undef-fw:
\[ f \equiv \text{lift}_1 (\text{strictify}(g)) \Rightarrow f \bot = \bot \]
apply (unfold OclUndefined_def lift0_def)
apply (tactic \langle \langle fold_tac [UU_fun_def] \rangle \rangle)
apply (rule lift1b_undef)
done

lemma lift2_undef1a-fw:
\[ f \equiv \text{lift}_2 (\text{strictify}(\lambda x. \text{strictify}(g x))) \Rightarrow f \bot (X : 'a : bot) = \bot \]
apply (unfold OclUndefined_def lift0_def)
apply (tactic \langle \langle fold_tac [UU_fun_def] \rangle \rangle)
apply (rule lift2_undef1a)
done

lemma lift2_undef2a-fw:
\[ f \equiv \text{lift}_2 (\text{strictify}(\lambda x. \text{strictify}(g x))) \Rightarrow f(X : 'a : bot) \bot = \bot \]
apply (unfold OclUndefined_def lift0_def)
apply (tactic \langle \langle fold_tac [UU_fun_def] \rangle \rangle)
apply (rule lift2_undef2a)
done

194
B.4. Library

ML ⟨ ⟨ val lift2_undef2a-fw = thm lift2_undef2a-fw ⟩ ⟩

lemma lift1_strict_is_isdef-fw:
  {lift1 (strictify (λx. g⌜x⌝))} ⇒ ∂(f X) = ∂ X
by(auto)

lemma lift1_strictify_is_isdef-fw:
  {lift1 (strictify (λx. g⌜x⌟))} ⇒ ∂(f X) = ∂ X
by(auto)

lemma lift2_strict_is_isdef-fw:
  {lift2 (strictify (λx. strictify (λy. g⌜x⌜y⌝⌟))))} =⇒ ∂(f X Y) = (∂ X ∧ ∂ Y)
by(auto)

lemma lift2_strictify_is_isdef-fw:
  {lift2 (strictify (λx. strictify (λy. g⌜x y⌝))))} =⇒ ∂(f X Y) = (∂ X ∧ ∂ Y)
by(auto)

Definedness reasoning for the new lifting construction

lemma fw_OCL_is_isdef_lift1′_0:
  assumes f_def: f ≡ (lift1′ lif宰_arg0) (strictify (λ x. g⌜x⌟))
  shows ∂(f X) = ∂ X
by(rule ext, simp add: f_def OclIsDefined_def ss_lifting′)

lemma fw_OCL_is_isdef_lift2′_00:
  assumes f_def: f ≡ (lift2′ lift_arg0 lift_arg0)
  shows ∂(f X Y) = (∂ X ∧ ∂ Y)
by(rule ext, simp add: f_def OclAnd_def ss_lifting′)

register rules for ocl_setup_op

ML ⟨ ⟨ val _ = let
  fun is_isdef_rules rname =
    let val rname' = fw_OCL_is_isdef ∧ rname;
    in RULE
      {rule = (rname', thm rname'),
       simp_tac = fn _ => fn x => x,
       is_atomic = fn _ => true,
       register = default_register_simponly,
       name_gen = default_namegen OCL_is_isdef_}
    end
  in add_setup_rules (_ lift1′_0, _ lift2′_00)
  ⟩ ⟩
Appendix B. Isabelle Theories

Some basic laws for the \texttt{not} operator

\begin{verbatim}
lemma not_injective: [\neg X = \neg Y ] \implies X = Y
  by (drule_tac f = \lambda x. \neg x in arg_cong, simp)

lemma not_bijective[simp]: (\neg X = \neg Y) = (X = Y)
  by (auto intro!: not_injective)
\end{verbatim}

Characterizations of logical operator definedness

\begin{verbatim}
lemma isDefined_notD: \partial (\neg X) = T \implies \partial X = T
  by (simp add: OclNot_def o_def lift1_strictify_is_isdef)
\end{verbatim}

The set of rules above is of fundamental importance for the forward-reasoning of
definedness expressions—it allows for computing for any core logic expression a
\texttt{DNF} composed of boolean literals and text \texttt{is_def(X)} terms (where
\texttt{X} is either a Boolean \texttt{OCL} atom (variable) or a non Boolean expression
(\texttt{if}). If the latter are composed by strict \texttt{OCL} operators, it can be further atomized to \texttt{is_def(X)} atoms where
\texttt{X} occurs in the expression.

Some generic laws for the \texttt{OCL if} operator

\begin{verbatim}
lemma if_distrib_strict:
  [ cp P; isStrict P ] \implies
  P(if X then Y:(\tau::\alpha::\bot)VAL) else Z endif) =
  (if X then ((P Y)::(\tau::\alpha::\bot)VAL) else P Z endif)
apply (rule_tac X = X in cp_distinct)
apply (rule_tac P = P in cp-compose2)
apply auto
\end{verbatim}

The set of rules above is of fundamental importance for the forward-reasoning of
definedness expressions—it allows for computing for any core logic expression a
\texttt{DNF} composed of boolean literals and text \texttt{is_def(X)} terms (where
\texttt{X} is either a Boolean \texttt{OCL} atom (variable) or a non Boolean expression
(\texttt{if}). If the latter are composed by strict \texttt{OCL} operators, it can be further atomized to \texttt{is_def(X)} atoms where
\texttt{X} occurs in the expression.
B.4. Library

apply(simp add: UU_fun_def OclUndefined_def
isStrict_def lift0_def
)
done

lemma if_distrib2 :

apply simp add: cp_def OclIf_def lift3_def,
erule exE, rule ext)
apply simp add: OclIsDefined_def DEF_def OclTrue_def
lift0_def lift1_def o_def
apply (drule_tac x =τ in fun_cong, simp)
apply (simp only: not_down_exists_lift, erule exE)
apply simp

done

Final Setup

lemma and_left_idem : P ∧ (P ∧ Q) = P ∧ Q

apply (subst and_assoc)
apply (auto)
done
ML ⟨ ⟨ val and_left_idem = thm and_left_idem ⟩ ⟩

lemma or_left_idem : P ∨ (P ∨ Q) = P ∨ Q

apply (subst or_assoc [symmetric])
apply (auto)
done
ML ⟨ ⟨ val or_left_idem = thm or_left_idem ⟩ ⟩

lemma and_left_assoc : X ∧ (Y ∧ Z) = Y ∧ (X ∧ Z)

apply (simp add: and_assoc)
apply (auto)
done

lemma or_left_assoc : X ∨ (Y ∨ Z) = Y ∨ (X ∨ Z)

apply (simp add: or_assoc [symmetric])
apply (auto)
done

Now we prepare a setup for Associativity, commutativity, and idempotency (ACI)
Appendix B. Isabelle Theories

rewriting of the boolean operators: These are:

\[ a \circ b = b \circ a \quad (B.1) \]

\[ (a \circ b) \circ c = a \circ (b \circ c) \quad (B.2) \]

\[ a \circ (b \circ c) = b \circ (a \circ c) \quad (B.3) \]

\[ a \circ a = a \quad (B.4) \]

\[ a \circ (a \circ b) = a \circ b \quad (B.5) \]

lemmas OCL_logic_ACI =

and_commute and_assoc[symmetric]

or_commute or_assoc

or_left_assoc or_idem or_left_idem

Boolean Relations

OCL Equalities

In the non-referential universe the following defines the two default equalities. In the referential universe, these definitions define the referential equalities.

constdefs OclStrongEq :: \([('τ, 'α::bot) VAL, ('τ, 'α) VAL] \Rightarrow \tau Boolean\)

\[ \text{OclStrongEq} \equiv \text{lift}_2 (\lambda x y. x = y) \]

OclStrictEq :: \([('τ, 'α::bot) VAL, ('τ, 'α) VAL] \Rightarrow \tau Boolean\)

\[ \text{OclStrictEq} \equiv \text{lift}_2 (\text{strictify}(\lambda x.\text{strictify}(\lambda y. x = y))) \]

consts

OclStrongValueEq :: \([('τ, 'α::bot) VAL, ('τ, 'α) VAL] \Rightarrow \tau Boolean\)

OclStrictValueEq :: \([('τ, 'α::bot) VAL, ('τ, 'α) VAL] \Rightarrow \tau Boolean\)

OclStrongDeepValueEq :: \([('τ, 'α::bot) VAL, ('τ, 'α) VAL] \Rightarrow \tau Boolean\)

OclStrictDeepValueEq :: \([('τ, 'α::bot) VAL, ('τ, 'α) VAL] \Rightarrow \tau Boolean\)

syntax

\text{OclStrongEq}\_std :: \([('τ, 'α::bot) VAL, ('τ, 'α) VAL] \Rightarrow \tau Boolean\)

\text{OclStrictEq}\_std :: \([('τ, 'α::bot) VAL, ('τ, 'α) VAL] \Rightarrow \tau Boolean\)

\text{OclStrongValueEq}\_std :: \([('τ, 'α::bot) VAL, ('τ, 'α) VAL] \Rightarrow \tau Boolean\)

\text{OclStrictValueEq}\_std :: \([('τ, 'α::bot) VAL, ('τ, 'α) VAL] \Rightarrow \tau Boolean\)

\text{OclStrongDeepValueEq}\_std :: \([('τ, 'α::bot) VAL, ('τ, 'α) VAL] \Rightarrow \tau Boolean\)

\text{OclStrictDeepValueEq}\_std :: \([('τ, 'α::bot) VAL, ('τ, 'α) VAL] \Rightarrow \tau Boolean\)

198
B.4. Library

syntax

translations

Is HOL-OCL Faithful to the OCL standard (Part II)?

lemma strictEq_factually_represented:
  Sem[X \equiv Y] \gamma = (if Sem[X] \gamma \neq \bot \land Sem[Y] \gamma \neq \bot
  \land \lnot Sem[X] \gamma \lor \lnot Sem[Y] \gamma
  then \bot
  else Sem[X] \gamma \lor Sem[Y] \gamma)
apply(simp add:OclStrictEq_def)
  DEF_def o_def
Appendix B. Isabelle Theories

apply (simp_all add: not_down_exists_lift, auto)
done

This operator—the first binary operator of a strict operation—represents directly the definition scheme as imposed by [41, Definition A.16]. The operators for Set, Bag, Integer and Real will follow this scheme.

B.4.3. OCL Logic Core

theory OCL_Logic_core
  imports $HOLOCL_HOME/src/library/basic/OCL_Boolean
  uses $HOLOCL_HOME/src/kernel_ext/isabelle_kernel_patch.ML
begin

Introduction

The purpose of this theory is to introduce the concept of OCL judgement (written $\tau \models \phi$) over a context $\tau$ and an OCL formula $\phi$ and to derive a number of mechanizable calculi for it. Here, a context is usually a system transition in OCL, i.e. a pair of two states $\tau = (\tau_{\text{pre}}, \tau_{\text{post}})$; however, since our theory on OCL does make perfect sense for generalizations such as "sequences of states" of Kripke structures, we prefer the more general notion of context to simple state pairs. A formula $\phi$ may contain paths such as "$\text{self.a.b}$" which were interpreted in $\tau$. The concept of a judgement occurs already in the standard in definition A.33 (Semantics of Operation Specifications). We turn it to the heart of deductive calculi for OCL: It will be the basis for equational calculi (both strict and strong equalities), congruences on formulas and local and global equivalences (terminology explained below) over judgements, and tableaux calculi over judgements. All of them are integrated into tactic procedures that allow for automated proof state transformations even automated proofs over OCL formula. Finally, these SML-programmed tactics were bound to a specific Isar syntax.

Terminology

An OCL formula is globally valid (or just valid if its evaluation yields for all contexts (e.g., state transitions $(s, s')$, Kripke-structures, ...) true; an OCL formula is locally valid if this is the case for a specific context.

We have three different equational calculi for OCL forming the bare bones of automated deduction in OCL.

1. an universal congruence (on OCL formulae) (UC) is a (conditional) formula with an equality $E = E'$ as conclusion where $E$ and $E'$ has type $V_{\alpha}(\beta)$. These equalities hold for all contexts $\tau$ and can be processed directly by Isabelle's rewriter.

200
2. A **local equivalence** (LE) is a (conditional) formula with an equality $E \tau = E' \tau$ as conclusion where $E$ and $E'$ has type $V_\alpha(\beta)$. These equalities hold only in some context $\tau$ of type $\tau$ and can only be processed by the tactic OCL_rewriter in subterms occurring in judgements and in the $E'$s in $E \tau = E' \tau$.

3. A **local judgement equivalence** (LJE) which states that two judgments over OCL formulae must be equivalent in a particular context $\tau$, so $(\tau | \models E) = (\tau | \models E')$.

Recall that judgements require that the formula holds, i.e., evaluates to true. In contrast to LE, the formulae must not agree on all three truth values.

### The Theory of OCL Judgements

In the following, the notation for validity judgements is introduced:

```plaintext
constdefs
  OclValid :: 'a Boolean ⇒ bool
  OclValid P ≡ (P = T)
  OclLocalValid :: ['a, 'a Boolean] ⇒ bool
  OclLocalValid τ P ≡ (P τ = T τ)

syntax
  _OclValid_std :: 'a Boolean ⇒ bool
  _OclLocalValid_std: ['a, 'a Boolean] ⇒ bool
    ((Valid (_,)) 50)
  _OclLocalValid_ascii: ['a, 'a Boolean] ⇒ bool
    ((1(())|=(_)) 50)
  _OclLocalValid_math: ['a, 'a Boolean] ⇒ bool
    ((1(())|=(_)) 50)

lemma cp OclLocalValid[simp, intro!]:
  cp P ⇒ cp(λ X τ. τ |= P X)
  by(simp add: OclLocalValid_def)
```

The rules needed by cp_unfold.

```plaintext
lemma OCL cp0 OclLocalValid: cp0 (λ x τ. τ |= x)
  by(simp add: OclLocalValid_def OclTrue_def ss_cp_defs ss_lifting)
```
Appendix B. Isabelle Theories

**Lemma**  \(OCL \_ cp0 \_ fold \_ OclLocalValid\): \(\tau \vDash (lift0 \ (x \ \tau)) = (\tau \vDash x)\)

**ML**
\[
\begin{aligned}
&cpR \_ simps := ![cpR \_ simps] \ addsimps \ [thm OCL \_ cp0 \_ OclLocalValid]; \\
&cpR \_ fold \_ simps := ![cpR \_ fold \_ simps] \ addsimps \ [thm OCL \_ cp0 \_ fold \_ OclLocalValid]; \\
\end{aligned}
\]

**Generic Definedness Rules on Local Validity**

**Lemma**  \(lift1 \_ strictify \_ implies \_ LocalValid \_ defined\):
\[
f \equiv lift1 \ (strictify \ (\lambda x \ . \ g \ x))
\implies (\tau \vDash 0 (f \ X)) = (\tau \vDash \partial X)
\]

**ML**
\[
\begin{aligned}
&apply \ (drule \ lift1 \_ strictify \_ is \_ isdef \_ fw) \\
&apply \ (rule \ _ \ tac \ t = \partial (f \ X) \ in \ ssubst) \\
&apply \ simp \ _ all \\
&done
\end{aligned}
\]

**Lemma**  \(lift2 \_ strictify \_ implies \_ LocalValid \_ defined\):
\[
f \equiv lift2 \ (strictify \ (\lambda x \ . \ strictify \ (\lambda y \ . \ g \ x \ y)))
\implies (\tau \vDash 0 (f \ X \ Y)) = (\partial X \land (\tau \vDash \partial Y))
\]

**ML**
\[
\begin{aligned}
&apply \ (drule \ lift2 \_ strictify \_ is \_ isdef \_ fw) \\
&apply \ (rule \ _ \ tac \ t = \partial (f \ X \ Y) \ in \ ssubst, \ assumption) \\
&apply \ (simp \ add: \ OclAnd \_ def \ OclTrue \_ def \ o \_ def \ OclIsDefined \_ def \ lift0 \_ def \ lift1 \_ def \ lift2 \_ def \ OclLocalValid \_ def) \\
&done
\end{aligned}
\]

**Semantic Representations of Validity**

**Lemma**  \(localValid2sem\):
\[
(\tau \vDash P) = (P \tau = \_\_True_\_) \\
by \ (simp \ add: \ OclLocalValid \_ def \ OclTrue \_ def \ lift0 \_ def)
\]

**Lemma**  \(localValidNot2sem\):
\[
(\tau \vDash \neg P) = (P \tau = \_\_False_\_) \\
by \ (simp \ add: \ OclLocalValid \_ def \ OclNot \_ def \ OclTrue \_ def \ strictify \_ def \ lift0 \_ def \ lift1 \_ def \ lift2 \_ def \ OclLocalValid \_ def) \\
\]

**Lemma**  \(localValidDefined2sem\):
\[
(\tau \vDash \partial S) = DEF(S \tau) \\
by \ (simp \ add: \ OclLocalValid \_ def \ OclIsDefined \_ def \ strictify \_ def \ o \_ def \ not \_ down \_ exists \_ lift, \ auto) \\
\]

**Lemma**  \(localValidUnDefined2sem\):
\[
(\tau \vDash \_\_ S) = (S \tau = \perp) \\
\]

202
lemma localValidToNotFalse2sem:
\( \tau \models P (x :: b :: type) \lor \neg (P x) = (P x \not\in \text{False}) \)
by (simp add: OclLocalValid_def OclIsDefined_def strictify_def
OclNot_def OclOr_def OclAnd_def
o_def lift0_def lift1_def lift2_def
OclTrue_def DEF_def lift not_down_exists_lift, auto)

lemma localValidToNotTrue2sem:
\( \tau \models \neg P (x :: b :: type) \lor \neg (P x) = (P x \not\in \text{True}) \)
by (simp add: OclLocalValid_def OclIsDefined_def strictify_def
OclNot_def OclOr_def OclAnd_def
o_def lift0_def lift1_def lift2_def
OclTrue_def DEF_def lift not_down_exists_lift, auto)

lemma isUndefined_charn:
\( X \equiv \bot \) = (\( \tau \not\models \bot \))
apply (auto simp: OclValid_def)
apply (simp add: OclUndefined_def OclTrue_def
lift0_def lift1_def o_def DEF_def OclNot_def OclIsDefined_def)
apply (rule ext)
apply (drule_tac x = s in fun_cong)
apply auto
done

lemma is_FALSE_charn:
\( X \equiv \text{False} \) = (\( \tau \not\models X \))
apply (auto simp: OclValid_def)
apply (simp add: OclUndefined_def OclFalse_def OclTrue_def
lift0_def lift1_def o_def DEF_def OclNot_def OclIsDefined_def)
apply (rule ext)
apply (drule_tac x = s in fun_cong)
apply (case_tac X s = down)
apply auto
apply (simp add: not_down_exists_lift)
apply auto
done

lemma is_TRUE_charn:
\( X \equiv \text{True} \) = (\( \tau \models X \))
by (auto simp: OclValid_def)

Local and global validity statements are in fact complete in the sense that they can inherently express all three basic values of booleans. This foundational fact gives rise
Appendix B. Isabelle Theories

to the idea to use these judgements both for tableaux and congruence-rewrite calculi.

Instead of representations in terms of semantic primitives, the local judgements can also be expressed on a more abstract level, i.e., in terms of global equivalences.

The following property reveals that strong equality is just mirrored by the \textit{HOL} equality. Note that the strong equality is implicitly use also in the construction of sets—its choice is therefore highly critical. Depending on the layout of objects, i.e., the concrete object universe, this results in different properties. For the case of the referential universe and states only containing objects with reference to themselves in the state, logical equality coincides with referential equality.

\textbf{lemma} isUndefined_charn_local:
\[
(X \tau = \bot \tau) = (\tau \not\models \emptyset X)
\]
by (simp add: OclLocalValid_def OclUndefined_def OclTrue_def lift0_def lift1_def o_def OclIsDefined_def DEF_def OclNot_def)

\textbf{lemma} is_FALSE_charn_local:
\[
(X \tau = \textit{F} \tau) = (\tau \not\models \neg X)
\]
apply (simp add: OclLocalValid_def OclUndefined_def OclTrue_def OclFalse_def lift0_def lift1_def o_def OclIsDefined_def DEF_def strictify_def OclNot_def)
apply auto
apply (case_tac X St = down)
apply (simp_all add: not_down_exists_lift)
apply auto
done

\textbf{lemma} is_TRUE_charn_local:
\[
(X \tau = \textit{T} \tau) = (\tau \models X)
\]
by (simp add: OclLocalValid_def)

\textbf{lemma} strongEq_charn:
\[
(\tau \models x \triangleleft y) = (x \tau = y \tau)
\]
apply (simp add: OclStrongEq_def OclLocalValid_def lift2_def)
done

\textbf{The Standardization Rules of (Global) Validity Statements.}

\textbf{lemma} OclTrue_is_Valid[simp]: \[
\vdash \textit{T}
\]
by (simp add: OclValid_def)

\textbf{lemma} OclFalse_is_Invalid[simp]: \[
\neg(\vdash \textit{F})
\]
apply (simp add: OclValid_def OclTrue_def OclFalse_def lift0_def)
by (unfold not_def, rule impl, drule fun_cong, simp)

\textbf{lemma} OclUndefined_is_Invalid[simp]: \[
\neg(\vdash \bot)
\]

204
apply (simp add: OclValid_def OclTrue_def OclUndefined_def lift0_def)
by (unfold not_def, rule impI, drule fun_cong, simp)

The Standardization Rules of Local Validity Statements.

lemma OclTrue_is_LocalValid[simp]: \( \tau \models T \)
by (simp add: OclLocalValid_def)

lemma OclFalse_is_LocalInvalid[simp]: \( \neg (\tau \models F) \)
by (simp add: OclLocalValid_def OclTrue_def OclFalse_def lift0_def)

lemma OclUndefined_is_LocalInvalid[simp]: \( \neg (\tau \models \bot) \)
by (simp add: OclLocalValid_def OclTrue_def OclUndefined_def lift0_def)

On the basis of the standardization rules, the simplifier can already rule out a number
of absurd validity statements:

lemma isdef_congr[simp]: \( (\tau \models \partial (\neg A)) = (\tau \models \partial (A)) \)
by simp

lemma \( (\tau \models \partial (\neg A)) = (\tau \models \partial (A)) \)
by simp

lemma \( \tau \models \partial (\neg A) \)
by simp

lemma \( \tau \models \partial (\neg A) \)
by simp

lemma \( \tau \models \partial (\bot) \)
by simp

lemma \( \neg (\tau \models \partial (\bot)) \)
by simp

lemma \( \neg (\tau \models \partial (\bot)) \)
by simp

lemma \( \tau \models \partial (\bot) \)
by simp

lemma \( \tau \models \partial (\bot) \)
by simp

Another Foundational Fact: non_quatrium_datur

lemma non_quatrium_datur: \( (\tau \models A) \lor (\tau \models \neg A) \lor (\tau \models \partial A) \)
apply (simp add: isUndefined_charn_local [symmetric])
apply (simp add: is_FALSE_charn_local [symmetric])
apply (cut_tac base_distinct)
apply (auto simp: OclLocalValid_def)
done

ML_setup ⟨ ⟨
fun prover4 (goal,name) =
let val thm = prove_goalw (the_context())
[OclFalse_def,OclTrue_def,OclUndefined_def,lift0_def]
goal (fn _ => > [auto_tac(claset()),simpset()]),
TRY(dr_tac fun_cong I),

205
Appendix B. Isabelle Theories

\[
\text{auto_tac(claset(),simpset())}}
\]

in bind_thm(name,thm);
Add_simp[thm]
end;
\]

ML_setup ⟨ ⟨
map prover4
[F \neq \bot, not__FALSE__undef],
(\bot \neq F, not__undef__FALSE),
(T \neq \bot, not__TRUE__undef),
(\bot \neq T, not__undef__TRUE),
(F \neq T, not__FALSE__TRUE),
(T \neq F, not__TRUE__FALSE),
(F \tau \neq \bot, not__FALSE__undef_st),
(\bot \neq F \tau, not__undef__FALSE_st),
(T \tau \neq \bot, not__TRUE__undef_st),
(\bot \neq T \tau, not__undef__TRUE_st),
(F \tau \neq T, not__FALSE__TRUE_st),
(T \tau \neq F, not__TRUE__FALSE_st)];
⟩ ⟩

lemma absurd1: \[ \tau \vdash A; \tau \vdash \neg A \] \implies R
by (auto simp add: is_FALSE_charn_local [symmetric],
simp add: OclLocalValid_def)

lemma absurd1S: \[ \tau \vdash A \] \implies \neg(\tau \vdash \neg A)
apply (rule_tac Pa = False in swap,simp_all)
by (erule absurd1,simp)

lemma absurd2: \[ \tau \vdash A; \tau \vdash \not\partial A \] \implies R
apply(simp add: isUndefined_charn_local[symmetric])
apply(simp add: is_TRUE_charn_local[symmetric])
done

lemma absurd2S: \[ \tau \vdash A \] \implies \not(\tau \vdash \not \partial A)
apply (rule_tac Pa = False in swap,simp_all)
by (erule absurd2,simp)

lemma absurd3: \[ \tau \vdash \neg A; \tau \vdash \not\partial A \] \implies R
apply(simp add: isUndefined_charn_local[symmetric])
apply(simp add: is_FALSE_charn_local[symmetric])
done

206
lemma absurd3S: \( \tau \models \neg A \implies \neg (\tau \models \emptyset A) \)
apply (rule_tac Pa = False in swap, simp_all)
by (erule absurd3, simp)

lemma absurd4: \( [\tau \models \emptyset (\partial A)] \implies R \)
by (simp add: OclLocalValid_def)

lemma absurd4S [simp]: \( \neg (\tau \models \emptyset (\partial A)) \)
by (simp add: OclLocalValid_def)

lemma absurd5: \( [\tau \models \bot] \implies R \)
by (simp add: OclLocalValid_def)

lemma absurd5S [simp]: \( \neg (\tau \models \bot) \)
by (simp add: OclLocalValid_def)

lemma absurd6: \( [\tau \models F] \implies R \)
by (simp add: OclLocalValid_def)

lemma absurd6S [simp]: \( \neg (\tau \models F) \)
by (simp add: OclLocalValid_def)

lemmas absurdE = absurd1 absurd2 absurd3 absurd4 absurd5 absurd6
lemmas absurdS = absurd1S absurd2S absurd3S absurd4S absurd5S absurd6S absurd6S

declare absurdE [elim]

ML_setup "
val OCL_absurd_tac = best_tac (HOL_cs addEs
[(thm absurd1),(thm absurd2),(thm absurd3)])"
"

lemma not_valid: \( \neg (\tau \models A) \) = ((\tau \models \neg A) \lor (\tau \models \emptyset A))
apply (cut_tac \( \tau = \tau \) \and \( A = A \) \in non_quatrium_datur)
apply auto
done

lemma not_invalid: \( \neg (\tau \models \neg A) \) = ((\tau \models A) \lor (\tau \models \emptyset A))
by (cut_tac \( \tau = \tau \) \and \( A = A \) \in non_quatrium_datur, auto)

lemma not_isUndefined: \( \neg (\tau \models \emptyset A) \) = ( (\tau \models A) \lor (\tau \models \neg A))
by (cut_tac \( \tau = \tau \) \and \( A = A \) \in non_quatrium_datur, auto)
Appendix B. Isabelle Theories

lemma isUndefined_eq_not_isDefined[auto]
\[ (~ (\tau \models A)) = (\tau \models \neg A) \]
apply auto
apply (drule not_valid [THEN iffD1], auto simp: OclLocalValid_def)
done

lemma not_isUndefined_eq_isDefined[auto]
\[ (~ (\tau \models \neg A)) = (\tau \models A) \]
apply (subst isUndefined_eq_not_isDefined [symmetric])
apply (rule HOL.not_not)
done

lemma not_isUndefined2: \[ (\tau \models A) = ( (\tau \models A) \lor (\tau \models \neg A)) \]
by (simp add: not_isUndefined [symmetric])

lemmas norm_validS =
not_valid
not_invalid
not_isUndefined
isUndefined_eq_not_isDefined
not_isUndefined_eq_isDefined

Global (Universal) vs. Local Validity

lemma global_vs_local_validity: \( (\forall \tau. (\tau \models A)) = (\vdash A) \)
by (auto simp:OclLocalValid_def OclValid_def,
rule ext,erule_tac x = x in allE, assumption)

lemma valid_intro: \( [\forall \tau. \tau \models A] \implies \vdash A \)
by (simp only: OclLocalValid_def OclValid_def,rule ext,auto)

lemma valid_elim: \( \vdash A \implies \tau \models A \)
by (simp only: OclLocalValid_def OclValid_def)

Universal Congruence vs. Judgement Equivalence

lemma equiv_I:
assumes a1: \( \forall \tau. ((\tau \models A) = (\tau \models B)) \)
and a2: \( \forall \tau. ((\tau \models \neg A) = (\tau \models \neg B)) \)
shows \( A = B \)
apply (rule ext)
apply (cut_tac \tau = x and A = A in non_quatrum_datur)
apply (erule disjE)
apply (erule_tac [2] disjE)
apply (cut_tac \tau l = x in a1 [THEN iffD1])
apply (cut_tac [4] \tau l = x in a2 [THEN iffD1])
B.4. Library

apply (auto)
pref 2
apply (simp_all add: isUndefined_charn_local [symmetric])
apply (cut_tac \tau = x and A = B in non_quatrium_datur)
apply (erule disjE)
apply (erule_tac [2] disjE)
pref 2
apply (simp only: is_FALSE_charn_local [symmetric])
apply (drule a1 [THEN iffD2])
apply (drule_tac [2] a2 [THEN iffD2])
apply (auto)
apply (simp_all add: OclLocalValid_def)
done

Absolute equivalence of OCL formula can be expressed in terms of absolute validity as follows:

lemma equiv_D:
\((A = B) = ((\forall \tau. (\tau \models A) = (\tau \models B))) \land (\forall \tau. ((\tau \models \emptyset A) = (\tau \models \emptyset B)))\)
by (auto intro!:equiv_I)

Local equivalence of OCL formula can be expressed in terms of local validity as follows:

lemma equiv_D_local:
\((A \tau = B \tau) = (((\tau \models A) = (\tau \models B)) \land (((\tau \models \emptyset A) = (\tau \models \emptyset B))))\)
apply (auto)
apply (simp_all add: isUndefined_charn_local[symmetric])
apply (simp_all only: not_valid)
apply auto
apply (simp_all add: OclLocalValid_def)
done

lemma equiv_I':
assumes A: \(\forall \tau. ((\tau \models A) = (\tau \models B))\)
and B: \(\forall \tau. ((\tau \models \neg A) = (\tau \models \neg B))\)
shows A = B
apply (rule ext)
apply (cut_tac \tau = x and A = A in non_quatrium_datur, safe)
apply (cut_tac \tau1 = x in A [THEN iffD1])
apply (cut_tac [3] \tau1 = x in B [THEN iffD1])
apply auto
apply (simp only: OclLocalValid_def)
apply (simp only: is_FALSE_charn_local [symmetric])
apply (cut_tac \tau = x and A = B in non_quatrium_datur)
apply (safe)
apply (drule A[THEN iffD2])

209
Appendix B. Isabelle Theories

apply (drule_tac [2] B THEN iffD2)
apply auto
apply (simp only: isUndefined_charn_local [symmetric])
done

lemma equiv_D':
(A = B) = ((∀ τ. (τ ⊨ A) = (τ ⊨ B)) ∧ (∀ τ. ((τ ⊨ ¬ A) = (τ ⊨ ¬ B)))
by (auto intro: equiv_I')

lemma equiv_D'_local:
(A τ = B τ) = (((τ ⊨ A) = (τ ⊨ B)) ∧ (((τ ⊨ ¬ A) = (τ ⊨ ¬ B)))
apply (auto)
apply (simp_all add: is_FALSE_charn_local [symmetric])
apply auto
apply (simp_all only: not_valid not_invalid is_FALSE_charn_local)
apply auto
apply (simp_all only: OclLocalValid_def)
done

The Theory of Strong and Strict Equality

The Setup: cp, strictness, definedness.
lemmas cp_strongEq = OclStrongEq_def [THEN cp_lift2_fw, standard]
lemmas cp_strictEq = OclStrictEq_def [THEN cp_lift2_fw, standard]
declare cp_strongEq [intro!] cp_strictEq [intro!]

lemma isDefined_strongEq [iff, simp]: ⊩ ∅ X ⇔ Y
by(auto simp: OclIsDefined_def OclStrongEq_def
   lift2_def lift1_def lift0_def
   OclUndefined_def o_def OclTrue_def
   DEF_def OclValid_def)

lemmas isDefined_strictEq = OclStrictEq_def [THEN lift2_strictify_is_isdef_fw, standard]
lemmas OclUndefined__strictEq = OclStrictEq_def [THEN lift2_undef1a_fw, standard]
lemmas strictEq__undef = OclStrictEq_def [THEN lift2_undef2a_fw, standard]
declare isDefined_strictEq [simp]
OclUndefined__strictEq [simp]
strictEq__undef [simp]

Properties of Strong Equality
lemma StrongEq_notvalid [simp]:
B.4. Library

\[ \lbrack \sigma \models \partial x; \tau \models \partial y \rbrack \implies (x \triangle y) \tau = \text{F} \tau \]

by (simp add: localValidDefined2sem localValidUnDefined2sem DEF_def OclStrongEq_def OclFalse_def lift0_def lift2_def)

lemma StrongEq_notvalid2(simp):
\[ \lbrack \sigma \models \partial x; \tau \models \partial y \rbrack \implies (x \triangle y) \tau = \text{F} \tau \]

by (auto simp: localValidDefined2sem localValidUnDefined2sem DEF_def OclStrongEq_def OclFalse_def lift0_def lift2_def)

lemma StrongEq_valid_undef:
\[ \lbrack \tau \models \partial x; \tau \models \partial y \rbrack \implies (x \triangle y) \tau = \text{T} \tau \]

will be proven in [OCL_Logic] by: by (ocl_hypsubst,simp)

oops

lemma strictEq_is_strongEq_LE:
\[ \lbrack \sigma \models \partial a; \tau \models \partial b \rbrack \implies (a \triangle b)\tau = (a \triangle b)\tau \]

by (simp add: OclStrictEq_def OclTrue_def OclIsDefined_def lift0_def lift1_def lift2_def OclLocalValid_def)

In the theory files, this section contains the SML code that implements OCL_normal_tac.

lemma strictEq_is_strongEq.simps:
\[ \sigma \models \partial a; \tau \models \partial b \implies a \triangle b = a \triangle b \]

by (tactic OCL_normal_tac (get_local_clasimpset ctxt) (thmstrictEq_is_strongEq_LE) 1)

lemma isDefined_if_valid:
\[ \tau \models A \implies \tau \models \partial A \]

by (rule classical, auto simp: OclIsDefined_def lift0_def lift1_def lift2_def o_def OclLocalValid_def)

lemma localvalid_strictEq_implies_localvalid_strongEq:
\[ \lbrack \tau \models a \triangle b \rbrack \implies \tau \models a \triangle b \]

apply (frule isDefined_if_valid, simp only: OclStrictEq_def)

apply (auto simp: OclTrue_def OclIsDefined_def OclStrongEq_def lift0_def lift1_def lift2_def o_def OclLocalValid_def)

done

211
Appendix B. Isabelle Theories

lemma **strongEq_subst**:
\[
\llbracket a \triangleq b = T; P a = T; cp P \rrbracket \Longrightarrow P b = T
\]
apply (unfold OclStrongEq_def OclTrue_def lift0_def lift1_def lift2_def)
apply (rule ext)
apply (drule_tac x = s in fun_cong)
apply (drule_tac x = s in fun_cong)
apply (simp add: cp_def)
apply (erule exE)
apply (rotate_tac -1)
simp
done

lemma **strongEq_refl_UC** [simp] :
\[
(\llbracket a \triangleq a \rrbracket) = T
\]
apply (rule ext)
by (simp add: OclStrongEq_def OclTrue_def lift0_def lift2_def)

lemma **strictEq_refl** :
\[
\models a \triangleq a
\]
by simp

lemma **strongEq_sym**:
\[
\models a \triangleq b \Longrightarrow \models b \triangleq a
\]
apply (unfold OclStrongEq_def OclTrue_def lift0_def lift2_def OclValid_def)
apply (rule ext)
apply (drule_tac x = \tau in fun_cong)
apply (drule_tac x = \tau in fun_cong)
simp
done

lemma **strongEq_trans**:
\[
\llbracket \models a \triangleq b; \models b \triangleq c \rrbracket \Longrightarrow \models a \triangleq c
\]
apply (unfold OclStrongEq_def OclTrue_def lift0_def lift2_def OclValid_def)
apply (rule ext)
apply (drule_tac x = \tau in fun_cong)
apply (drule_tac x = \tau in fun_cong)
simp
done

lemma **strongEq_refl_local** :
\[
\tau \models a \triangleq a
\]
by (simp add: OclStrongEq_def OclLocalValid_def OclTrue_def lift0_def lift2_def)

lemma **strongEq_sym_local** :
\[
\models \tau \models a \triangleq b \Longrightarrow \models \tau \models b \triangleq a
\]

212
by (simp add: OclLocalValid_def OclStrongEq_def OclTrue_def lift0_def lift2_def)

lemma strongEq_trans_local:
\[ \lbrack \tau \models a \triangleq b; \tau \models b \triangleq c \rbrack \implies \tau \models a \triangleq c \]
by (simp add: OclLocalValid_def OclStrongEq_def OclTrue_def lift0_def lift2_def)

Extensionality must swap parameters and therefore looks quite inelegant ...

lemma strongEq_ext_local:
\[(\forall x. \tau \models f x \triangleq g x) \implies \tau \models (\lambda X \tau. f \tau X) \triangleq (\lambda X \tau. g \tau X)\]
by (simp add: OclLocalValid_def OclStrongEq_def OclTrue_def lift0_def lift2_def)

thm strongEq_ext_local

constdefs
SW :: (′b ⇒ ′a ⇒ ′c) ⇒ ′a ⇒ ′b ⇒ ′c
SW f ≡ (λ X τ. f τ X)

lemma SW_idem[simp]: SW(SW f) = f
by(simp add: SW_def)

lemma strongEq_ext_local2:
\[(\forall x. \tau \models f x \triangleq g x) \implies \tau \models (SW f) \triangleq (SW g)\]
by (simp add: OclLocalValid_def OclStrongEq_def OclTrue_def lift0_def lift2_def SW_def)

lemma strongEq_subst_local:
\[ \lbrack \tau \models a \triangleq b; \tau \models P a; cp P \rbrack \implies \tau \models P b \]
apply (simp add: OclStrongEq_def OclLocalValid_def cp_def OclTrue_def lift0_def lift1_def lift2_def)
apply (erule exE)
apply (rotate_tac -1)
apply simp
done

lemmas strongEq_subst_local_sym = strongEq_sym_local [THEN strongEq_subst_local]

Technical Substitution Lemmas (for tactics)
lemma strongEq1:
\[ \lbrack \tau \models X \triangleq Y; \tau \models P Y; cp P \rbrack \implies \tau \models P X \]
by(drule strongEq_sym_local)
Appendix B. Isabelle Theories

erule strongEq_subst_local, simp)

lemmas strongEq1_rev = strongEq1[OF strongEq_sym_local]

lemma strongEq2:
\[
\begin{align*}
\tau \models X \triangleq Y; \ (\tau \models P Y) = (\tau \models P' Y); \ cp \ P; \ cp \ P' & \implies (\tau \models P X) = (\tau \models P' X) \\
apply & (rule \ iffI) \\
apply & (rule_tac X= X in strongEq1, simp_all, drule sym, simp) \\
apply & (erule strongEq1_rev, simp_all) \\
apply & (simp, erule strongEq1_rev, simp_all) \\
done
\end{align*}
\]

lemmas strongEq2_rev = strongEq2[OF strongEq_sym_local]

lemma strongEq3:
\[
\begin{align*}
\tau \models X \triangleq Y; \ P Y \tau = P' Y \tau; \ cp \ P; \ cp \ P' & \implies P X \tau = P' X \tau \\
apply & (rule_tac P= P and P'= P' in ocl_cp_subst3, simp_all) \\
apply & (simp add: OclLocalValid_def OclStrongEq_def strictify_def o_def lift0_def lift1_def lift2_def OclNot_def OclTrue_def OclFalse_def OclUndefined_def DEF_def) \\
done
\end{align*}
\]

lemmas strongEq3_rev = strongEq3[OF strongEq_sym_local]

lemma strongEq1_fw:
\[
\begin{align*}
\tau \models & P X; \tau \models Y \triangleq X; \ cp \ P & \implies \tau \models P Y \\
by & (erule strongEq1, simp)
\end{align*}
\]

lemma strongEq2_fw:
\[
\begin{align*}
[(\tau \models P X); \tau \models Y \triangleq X; \ cp \ P; \ cp \ P'] & \implies (\tau \models P X) = (\tau \models P' X) \\
by & (erule strongEq2, simp_all)
\end{align*}
\]

lemma strongEq3_fw:
\[
\begin{align*}
P X \tau = P' X \tau; \tau \models Y \triangleq X; \ cp \ P; \ cp \ P' & \implies P Y \tau = P' Y \tau \\
by & (erule strongEq3, simp_all)
\end{align*}
\]

lemma strongEq1-fw2:
\[
\begin{align*}
\tau \models & P X; \tau \models X \triangleq Y; \ cp \ P & \implies \tau \models P Y \\
by & (erule strongEq1_rev, simp)
\end{align*}
\]

lemma strongEq2-fw2:
\[
\begin{align*}
[(\tau \models P X); \tau \models X \triangleq Y; \ cp \ P; \ cp \ P'] & \implies (\tau \models P X) = (\tau \models P' X) \\
by & (erule strongEq2_rev, simp_all)
\end{align*}
\]

214
B.4. Library

lemma strongEq3_fw2:
\[ P \tau = P' \tau; \tau \models X \equiv Y; \text{cp P; cp P'} \implies P Y \tau = P' Y \tau \]
by (erule strongEq3_rev, simp_all)

Automated Canonization and Rewriting in OCL

OCL offers a number of equivalences: From strict and strong equality, we have three types of congruences/equivalences, such that reasoning about them is tedious and must be tactically supported. The ultimate goal of this section is to work out two tactic procedures for this purpose that pursue this goal. As basis for these tactic procedures, a number of rules and rule variants must be derived, whose existence is merely justified by deduction technical reasons.

validity propagation rules
lemma strictEq1 = strongEq1[OF localvalid_strictEq_implies_localvalid_strongEq]
lemma strictEq2 = strongEq2[OF localvalid_strictEq_implies_localvalid_strongEq]
lemma strictEq3 = strongEq3[OF localvalid_strictEq_implies_localvalid_strongEq]

lemma strictEq1_rev = strongEq1_rev
[OF localvalid_strictEq_implies_localvalid_strongEq]
lemma strictEq2_rev = strongEq2_rev
[OF localvalid_strictEq_implies_localvalid_strongEq]
lemma strictEq3_rev = strongEq3_rev
[OF localvalid_strictEq_implies_localvalid_strongEq]

lemma subst_LJ_TRUE:
[\tau \models X; \tau \models P \top; \text{cp P}] \implies \tau \models P X

apply (simp add: OclLocalValid_def, drule sym)
apply (erule_tac P = P in ocl_cp_subst, auto)
done

lemma subst_LJ_TRUE_fw:
[\tau \models X; \tau \models P X; \text{cp P}] \implies \tau \models P \top

apply (simp add: OclLocalValid_def)
apply (erule_tac P = P in ocl_cp_subst)
apply auto
done

lemma subst_LJ_TRUE_fw_rev:
[\tau \models P X; \tau \models X; \text{cp P}] \implies \tau \models P \top

apply (simp add: OclLocalValid_def)
apply (erule_tac P = P in ocl_cp_subst) back back
Appendix B. Isabelle Theories

apply auto
done

lemma subst_LE_TRUE:
[τ ⊨ X; P T τ = P' T τ; cp P; cp P'] ⇒ P X τ = P' X τ
apply(rule local_validity_propagation2 [simplified is_TRUE_charn_local]) back
by auto

lemma subst_LE_TRUE_fw:
[τ ⊨ X; P X τ = P' X τ; cp P; cp P'] ⇒ P T τ = P' T τ
apply(simp add: OclLocalValid_def)
apply(rule trans, rule sym)
apply(erule_tac τ = τ and P = P in cp_charn, simp)
apply(erule_tac τ = τ and P = P' in cp_charn, simp)
done

lemma subst_LE_TRUE-fw_rev:
[P (X::'a⇒Boolean_0) τ = P' X τ; τ ⊨ X; cp P; cp P'] ⇒ P T τ = P' T τ
by(erule subst_LE_TRUE_fw, auto)

lemma subst_LJE_TRUE:
[τ ⊨ X; (τ ⊨ P T) = (τ ⊨ P' T); cp P; cp P'] ⇒ (τ ⊨ P X)=(τ ⊨ P' X)
by(erule subst_LJ_TRUE, auto)

lemma subst_LJE_TRUE-fw:
[τ ⊨ X; (τ ⊨ P X]=(τ ⊨ P' X); cp P; cp P'] ⇒ (τ ⊨ P T) = (τ ⊨ P' T)
by(erule subst_LE_TRUE-fw, auto)

lemma subst_LJE_TRUE-fw_rev:
[(τ ⊨ P X]=(τ ⊨ P' X); τ ⊨ X; cp P; cp P'] ⇒ (τ ⊨ P T) = (τ ⊨ P' T)
by(erule subst_LJE_TRUE-fw, auto)

lemma subst_TRUE_LJE:
[τ ⊨ X; cp P ] ⇒ (τ ⊨ P X)=(τ ⊨ P T)
apply(auto)
apply(erule subst_LJ_TRUE-fw, auto)
apply(erule subst_LJ_TRUE, auto)
done
lemma subst_LJ_FALSE:
\[
\begin{array}{l}
\begin{align*}
\tau \vdash \lnot X; \tau \vdash \overline{P F}; \text{cp } P' \Rightarrow \tau \vdash P X
\end{align*}
\end{array}
\]
apply (drule is_FALSE_charn_local [THEN iffD2])
apply (rotate_tac -1)
apply (drule sym)
apply (simp add: OclLocalValid_def)
apply (erule_tac P = P in ocl_cp_subst)
apply auto
done

lemma subst_LJ_FALSE_fw:
\[
\begin{array}{l}
\begin{align*}
\tau \vdash \lnot X; \tau \vdash P X; \text{cp } P' \Rightarrow \tau \vdash P F
\end{align*}
\end{array}
\]
apply (drule is_FALSE_charn_local [THEN iffD2])
apply (rotate_tac -1)
apply (simp add: OclLocalValid_def)
apply (erule_tac P = P in ocl_cp_subst)
apply auto
done

lemma subst_LJ_FALSE_fw_rev:
\[
\begin{array}{l}
\begin{align*}
\tau \vdash P X; \tau \vdash \lnot X; \text{cp } P' \Rightarrow \tau \vdash P F
\end{align*}
\end{array}
\]
apply (erule subst_LJ_FALSE_fw)
apply auto
done

lemma subst_LE_FALSE:
\[
\begin{array}{l}
\begin{align*}
\tau \vdash \lnot X; \tau \vdash \overline{P F}; \text{cp } P' \Rightarrow \tau \vdash P X = P' X
\end{align*}
\end{array}
\]
apply (rule local_validity_propagation3 [simplified is_FALSE_charn_local])
back by auto

lemma subst_LE_FALSE_fw:
\[
\begin{array}{l}
\begin{align*}
\tau \vdash \lnot X; \tau \vdash \overline{P X} = P' X \tau; \text{cp } P; \text{cp } P' \Rightarrow \tau \vdash P F = P' F \tau
\end{align*}
\end{array}
\]
apply (simp only: is_FALSE_charn_local[symmetric])
apply (rule trans, rule sym)
apply (erule_tac \tau = \tau and \text{cp} = \text{cp} in cp_charn, simp)
apply (rule trans, assumption)
apply (erule_tac \tau = \tau and \text{cp} = \text{cp} in cp_charn, simp)
done

lemma subst_LE_FALSE_fw_rev:
\[
\begin{array}{l}
\begin{align*}
[\tau \vdash \lnot X; \tau \vdash \overline{P X} = P' X \tau; \text{cp } P; \text{cp } P'] \Rightarrow \tau \vdash P F = P' F \tau
\end{align*}
\end{array}
\]
by (erule subst_LE_FALSE, auto)

lemma subst_LJE_FALSE:
\[
\begin{array}{l}
\begin{align*}
\tau \vdash \lnot X; \tau \vdash \overline{P X} = (\tau \vdash P' F); \text{cp } P; \text{cp } P' \Rightarrow (\tau \vdash P X) = (\tau \vdash P' X)
\end{align*}
\end{array}
\]
by (erule subst_LE_FALSE, auto)
Appendix B. Isabelle Theories

lemma subst_LJE_FALSE-fw:
\[ \llbracket \tau \models \neg X; (\tau \models P X) = (\tau \models P' X); \text{cp } P; \text{cp } P' \rrbracket \Rightarrow (\tau \models P F) = (\tau \models P' F) \]
by (erule subst_LE_FALSE-fw, auto)

lemma subst_LJE_FALSE-fw_rev:
\[ \llbracket (\tau \models P X) = (\tau \models P' X); \tau \models \neg X; \text{cp } P; \text{cp } P' \rrbracket \Rightarrow (\tau \models P F) = (\tau \models P' F) \]
by (erule subst_LJE_FALSE-fw, auto)

lemma subst_FALSE_LJE:
\[ \llbracket \tau \models \neg X; \text{cp } P \rrbracket \Rightarrow (\tau \models P X) = (\tau \models P F) \]
apply (auto)
apply (erule subst_LJ_FALSE_fw)
done

lemma subst_LJ_undef:
\[ \llbracket \tau \models \partial X; \tau \models P \bot \rrbracket \Rightarrow \tau \models P X \]
apply (drule isUndefined_charn_local [THEN iffD2])
apply (rotate_tac \(-1\))
apply (simp add: OclLocalValid_def)
apply (erule_tac P = P in ocl_cp_subst)
done

lemma subst_LJ_undef_fw:
\[ \llbracket \tau \models \neg X; \tau \models P \bot \rrbracket \Rightarrow \tau \models P \bot \]
apply (rule_tac P = P and \tau = \tau in ocl_cp_subst)
done

lemma subst_LE_undef:
\[ \llbracket \tau \models \neg X; \tau \models P \bot \rrbracket \Rightarrow P X \tau = P' X \tau \]
apply (rule_tac P=P and \tau=\tau in ocl_cp_subst)

218
apply (simp_all)
apply (simp adj: OclLocalValid_def OclIsDefined_def strictify_def
OclNot_def o_def lift0_def lift1_def lift2_def
OclTrue_def DEF_def OclUndefined_def)
done

lemma subst_LE_undef_fw:
∀τ ⊨ ∅ X; P X τ = P' X τ ; cp P; cp P' ➞ P ⊥ τ = P' ⊥ τ
apply (rule_tac P=P and τ=τ in ocl_cp_subst3)
apply (simp_all)
apply (simp add: OclLocalValid_def OclIsDefined_def strictify_def
OclNot_def o_def lift0_def lift1_def lift2_def
OclTrue_def DEF_def OclUndefined_def)
done

lemma subst_LJE_undef:
∀τ ⊨ ∅ X; (τ ⊨ P ⊥) = (τ ⊨ P' X); cp P; cp P' ➞ (τ ⊨ P X)=(τ ⊨ P' X)
by(erule subst_LE_undef, auto)

lemma subst_LJE_undef-fw:
∀τ ⊨ ∅ X; (τ ⊨ P X)=(τ ⊨ P' X); cp P; cp P' ➞ (τ ⊨ P ⊥) = (τ ⊨ P' ⊥)
by(erule subst_LE_undef-fw, auto)

lemma subst_LJE_undef-fw_rev:
∀τ ⊨ ∅ X; (τ ⊨ P ⊥) = (τ ⊨ P X); cp P; cp P' ➞ (τ ⊨ P X)=(τ ⊨ P' X)
by(erule subst_LE_undef-fw, auto)

lemma subst_undef_LJE:
∀τ ⊨ ∅ X; cp P ➞ (τ ⊨ P X)=(τ ⊨ P ⊥)
apply(auto)
apply(erule subst_LJ_undef, auto)
done

lemmas strictEq1-fw2 = strongEq1-fw2
OF _ localvalid_strictEq_implies_localvalid_strongEq
lemmas strictEq2-fw2 = strongEq2-fw2
OF _ localvalid_strictEq_implies_localvalid_strongEq
lemmas strictEq3-fw2 = strongEq3-fw2
OF _ localvalid_strictEq_implies_localvalid_strongEq
lemmas strictEq1-fw = strongEq1-fw
OF _ localvalid_strictEq_implies_localvalid_strongEq

219
Appendix B. Isabelle Theories

lemmas strictEq2_fw = strongEq2_fw
[OF _ localvalid_strictEq_implies_localvalid_strongEq]

lemmas strictEq3_fw = strongEq3_fw
[OF _ localvalid_strictEq_implies_localvalid_strongEq]

Implementation of OCL_hyp_subst_tac

These rules give some ideas for automation:

1. OCL_norm_tac computes local norm of validities: \( \models A \implies B \) is converted
   \( \tau \models A \implies \tau \models B \ldots \) This is done forward and backward style.

2. OCL_absurd_tac should decide if a subgoal is OCL absurd.

3. OCL_hyp_subst_tac for variables or parameters \( A \), which uses:

   \[
   \begin{align*}
   A = t, t = A & \quad \text{(standard hyp_subst_tac)} \quad \text{(B.6)} \\
   A \tau = t \tau, t \tau = A \tau & \quad \text{(local congruences)} \quad \text{(B.7)} \\
   \models \partial A, \tau \models \partial A & \quad \text{(undefinednesses local and global)} \quad \text{(B.8)} \\
   \models \neg A, \tau \models \neg A & \quad \text{(falsities local and global)} \quad \text{(B.9)} \\
   \models A \triangleq t, \tau \models A \triangleq t & \quad \text{(strong equalities local and global)} \quad \text{(B.10)} \\
   \tau \models A \triangleq t, \tau \models A \triangleq t & \quad \text{(B.11)}
   \end{align*}
   \]

4. OCL_def_hyp_split_tac:
   - atomizes definedness hypotheses, e.g., \( \partial(f A B) = \partial A \wedge \partial B \ldots \)
   - case-splits undefinedness hypothesis \( \partial(f A B) = \partial(A) \lor \partial(B) \ldots \)
   - and performs safe_tac and OCL_hyp_subst_tac. (but no global simplification afterwards in order to do not too much in a black box.

5. OCL_simp and OCL_asm_simp. New rewriter that can cope with \( \_ \triangleq \_ \) and \( \_ = \_ \) and its specific format of the subst rule. Future: use standard rewriter with a fake subst rule on OCL, transform the proof objects and refeed this into the kernel?

6. integrate into simp-procedure as looper or subgoaler.

lemmas DEF_rules = subst_LE_TRUE[of _ \( \partial \) X,standard]
substitution_LJE_TRUE[of _ \( \partial \) X,standard]
subst_LJ_TRUE[of _ \( \partial \) X,standard]

lemmas DEF_few_rev_rules =
substitution_LJ_TRUE_few_rev[of _ \( \partial \) X,standard]
substitution_LJE_TRUE_few_rev[of _ \( \partial \) X,standard]
subst_LE_TRUE_few_rev[of _ \( \partial \) X,standard]
Testing OCL_hyp_subst_tac

lemma \[ \tau \emptyset A; \tau \emptyset A \rightarrow B \rightarrow \tau \emptyset A \land B \rightarrow C \]
by(tacticOCL_hyp_subst_tac (get_localclasimpset ctxt) 1, simp)

lemma \[ \tau \emptyset A; \tau \emptyset A \rightarrow B; (\tau \emptyset A = (\tau \emptyset C)) \rightarrow \tau = B \tau \]
by(tacticOCL_hyp_subst_tac (get_localclasimpset ctxt) 1, simp)

lemma !!\tau A . \[ \tau \emptyset A; \tau \emptyset A \land B ; \tau \emptyset B; (\tau \emptyset A = (\tau \emptyset C)) \rightarrow \tau = B \tau \]
by(tacticOCL_hyp_subst_tac (get_localclasimpset ctxt) 1, simp)

lemma \[ \tau \emptyset A; \tau \emptyset D \rightarrow E; \tau \emptyset D \rightarrow 1 \rightarrow \tau \emptyset A \land E \rightarrow 1 \]
by(tacticOCL_hyp_subst_tac (get_localclasimpset ctxt) 1, simp)

lemma \[ \tau \emptyset A; \tau \emptyset E \rightarrow D; \tau \emptyset 1 \rightarrow E ; \tau \emptyset 1 \rightarrow D \rightarrow \tau \emptyset A \land E \rightarrow 1 \]
by(tacticOCL_hyp_subst_tac (get_localclasimpset ctxt) 1, simp)

lemma \[ \tau \emptyset A; \tau \emptyset \neg B; \tau \emptyset 1 \rightarrow E; \tau \emptyset D \rightarrow 1 \rightarrow A \land B \tau ; P \tau = T \tau ; \tau \emptyset D \rightarrow 1 \rightarrow E \rightarrow \tau \emptyset A \land B \]
apply(tacticOCL_hyp_subst_tac (get_localclasimpset ctxt) 1)
apply(simp)
done

Testing OCL_subst_tac

lemma isDefined_cong5 : (\partial X \lor \neg \partial X) = T
apply (rule ext, case_tac x := \partial(X))
apply (tacticOCL_subst_tac(false, false, false)
apply (tacticOCL_subst_tac (false, false, false)
apply (tacticOCL_subst_tac (false, false, false)
done

Some more Absurdities

lemma not_isDefined_if_isUndefined:
\[ \tau \emptyset \partial A \equiv (\neg (\tau \emptyset \partial A)) \]
by auto
Appendix B. Isabelle Theories

**lemma non_tertium_datur_if_isDefined**:  
\[ \tau \models \partial A; \tau \models A \Rightarrow P; \tau \models \neg A \Rightarrow P \] \Rightarrow P  
by (cut_tac \tau = \tau and A = A in non_quatrium_datur, auto)

**lemma isDefined_if_valid**:  
\[ \tau \models A \Rightarrow \tau \models \partial A \]  
by (rule classical, auto)

**lemma isDefined_if_invalid**:  
\[ \tau \models \neg A \Rightarrow \tau \models \partial A \]  
by (rule classical, auto)

**Trichotomy Specialized for if**

**lemma if_trichotomy**:  
\[ \tau \models P \bot; \tau \models P Y; \tau \models P Z; \text{cp } P \] \Rightarrow \tau \models P (if X then Y else Z endif)  
apply (simp add: OclLocalValid_def)  
apply (rule_tac P = \lambda x. P (if x then Y else Z endif) in cp_distinct_core)  
apply auto  
apply (rule_tac P = P in cp_compose2)  
apply auto  
done

**lemma if_trichotomyE**:  
\[ \tau \models P (if X then Y else Z endif); \tau \models P \bot \Rightarrow R; \tau \models P Y \Rightarrow R; \tau \models P Z \Rightarrow R; \text{cp } P \] \Rightarrow R  
apply (cut_tac \tau = \tau and A = X in non_quatrium_datur)  
apply safe  
apply (drule_tac [3] subst_LJ_undef_fw)  
prefer 3  
apply (assumption)  
apply (rule_tac [3] P = P in cp_compose2, simp_all)  
apply (drule_tac [2] subst_LJ_FALSE_fw)  
prefer 2  
apply (assumption)  
apply (drule subst_LJ_TRUE_fw)  
back  
apply (assumption)  
apply auto  
apply (rule_tac [2] P = P in cp_compose2)
apply (rule_tac P = P in cp_compose2)
apply auto
done

lemmas OclIf_LJE[simp] =
  subst_TRUE_LJE[of _ _ λX. if X then Y else Z endif, simplified]
  subst_FALSE_LJE[of _ _ λX. if X then Y else Z endif, simplified]
  subst_undef_LJE[of _ _ λX. if X then Y else Z endif, simplified]

A Weak (Propositional) LJE Calculus for Propositional OCL
The weak propositional LJE gives rise to a fairly promising decision procedure for a fragment of OCL:

• blow away quantifiers
• infer from inclusions of variables the invariants
• infer from the invariants the definedness of variables
• all remaining variables occurring in formula: case-splitting over definedness, congruence closure over undefinednesses,
• simp_all: universal congruences
• simp_all: weak propositional LJE
• blast_tac.

Weak LJE Calculus means here that everything is restricted to defined expressions.

lemma localValidNot2not :
[ τ ⊨ φ A ] ⇒ (τ ⊨ ¬ A) = (¬ (τ ⊨ A))
by(auto elim!: non_tertium_datur_if_isDefined)

lemma localValidAnd2conj :
[ τ ⊨ φ A; τ ⊨ φ B ] ⇒ (τ ⊨ A ∧ B) = ((τ ⊨ A) ∧ (τ ⊨ B))
apply(auto elim!: non_tertium_datur_if_isDefined)
by(tactic.ALLGOALS(OCL_hyp_subst_tac (get_local_clasimpset ctxt)),simp_all)
Appendix B. Isabelle Theories

lemma localValidOr2disj : 
\[ (\sigma \models A \land \sigma \models B) = (\sigma \models A) \lor (\sigma \models B) \]
apply (auto elim!: non_tertium_datur_if_isDefined)
by (tacticALLGOALS (OCL_hyp_subst_tac (get_local_clasimpset ctxt)), simp_all)

lemma localValidImplies2impl : 
\[ (\sigma \models A \land \sigma \models B) = (\sigma \models A \rightarrow B) = (\sigma \models A) \rightarrow (\sigma \models B) \]
by (simp add: OclImplies_def localValidNot2not
localValidOr2disj localValidAnd2conj)

lemma isDefined_notI : 
\[ (\sigma \models \neg A) \Longrightarrow \sigma \models \neg (\neg A) \] by simp

lemma isDefined_andI : 
\[ (\sigma \models A \land \sigma \models B) \Longrightarrow \sigma \models (A \land B) \]
apply (auto elim!: non_tertium_datur_if_isDefined)
apply (tacticALLGOALS (OCL_hyp_subst_tac (get_local_clasimpset ctxt)))
apply (simp_all)
done

lemma isDefined_orI : 
\[ (\sigma \models A \lor \sigma \models B) \Longrightarrow \sigma \models (A \lor B) \]
apply (auto elim!: non_tertium_datur_if_isDefined)
apply (tacticALLGOALS (OCL_hyp_subst_tac (get_local_clasimpset ctxt)))
apply (simp_all)
done

lemma isDefined_xorI : 
\[ (\sigma \models A \lor \sigma \models B) \Longrightarrow \sigma \models (A \oplus B) \]
apply (auto elim!: non_tertium_datur_if_isDefined)
apply (tacticALLGOALS (OCL_hyp_subst_tac (get_local_clasimpset ctxt)))
apply (simp_all)
done

lemma isDefined_impliesI : 
\[ (\sigma \models A \land \sigma \models B) \Longrightarrow \sigma \models (A \rightarrow B) \]
apply (auto elim!: non_tertium_datur_if_isDefined)
apply (tacticALLGOALS (OCL_hyp_subst_tac (get_local_clasimpset ctxt)))
apply (simp_all)
done

lemma isDefined_implies2I : 
\[ (\sigma \models A \land \sigma \models B) \Longrightarrow \sigma \models (A \rightarrow B) \]
by (simp add: OclImplies2_def localValidNot2not isDefined_andI
localValidOr2disj localValidAnd2conj)

224
Isar Interface Setup

This part of the so-called thymorpher generates a number of rules and a specific setup for all operators given to him. This includes (all examples are for the case of the weak (strict) equality here.

1. context passingness rules \(cp\) for all unary and binary operators, e.g.:

\[
[cpP; cpP'] \implies cp(\lambda X.PX = P'X)
\]

These rules were used both as simplifier rules as well as safe introduction rules.

2. all undefinedness reduction rules (unary and binary case), e.g.:

\[
\bot \doteq X = \bot \quad \text{and} \quad X \doteq \bot = \bot
\]

These rules were used as global simplifier rules.

3. universal congruences for definedness, e.g.:

\[
\partial X \doteq Y = \partial X \land \partial Y
\]

4. the local judgement equivalence for definedness, e.g.:

\[
\tau \vdash \partial(X \doteq Y) = (\tau \vdash \partial X \land \tau \vdash \partial Y)
\]

This equality is used for optimized reasoning over definedness inside some OCL-tactics.

Definition of ocl_subst

In the theory files, this section contains the Isar setup code for ocl_subst.

Definition of ocl_auto

In the theory files, this section contains the Isar setup code for ocl_auto.

B.4.4. OCL Logic

theory OCL_Logic
  imports $HOLOCL_HOME/src/OCL_Logic_core
begin
Appendix B. Isabelle Theories

Testing and Demonstrations of Main Tactics

Automatic Generation of UC Variants of Global Equivalence Theorems

```ml
ML ⟨⟨
    bind_thm (strictEq_is_strongEq_UC,
    OCL_normal (the_context()) (thmstrictEq_is_strongEq_LE)) ⟩⟩
```

Theorem OCL_cp_OclStrictEq

```ml
OCL_undef_1_OclStrictEq
OCL_undef_2_OclStrictEq
OCL_is_def_OclStrictEq
OCL_is_defopt_OclStrictEq
```

Theorem OCL_isDefined_def

```ml
ocl_setup_op [OclIsDefined]
ocl_setup_op [OclStrongEq]
ocl_setup_op [OclAnd, OclNot]
ocl_setup_op [OclImplies1_alt, OclImplies2_alt]
```

Test of ocl_hypsubst

```ml
lemma \[ \Gamma \vdash A; \Gamma \vdash \neg B; \Gamma \vdash A \land B ; \Gamma \vdash \not\beta B \] \implies \Gamma \vdash A \land B
by (ocl_hypsubst [1], simp)
```

```ml
lemma \[ \Gamma \vdash A; \Gamma \vdash A \land B ; \Gamma \vdash \not\beta B; (\Gamma \vdash A) = (\Gamma \vdash C) \] \implies A = B \tau
by (ocl_hypsubst, simp)
```

```ml
lemma \[ \not\exists !\tau. A. [ \Gamma \vdash A; \Gamma \vdash A \land B ; \Gamma \vdash \not\beta B; (\Gamma \vdash A) = (\Gamma \vdash C) ] \implies A = B \tau
by (ocl_hypsubst, simp)
```

```ml
lemma \[ \Gamma \vdash A; \Gamma \vdash D \equiv E ; \Gamma \vdash D \equiv I \] \implies \Gamma \vdash A \land E \equiv I
by (ocl_hypsubst, simp)
```

```ml
lemma \[ \Gamma \vdash A; \Gamma \vdash D \equiv E ; \Gamma \vdash I \equiv D \] \implies \Gamma \vdash A \land E \equiv I
by (ocl_hypsubst, simp)
```

```ml
lemma \[ \Gamma \vdash A; \Gamma \vdash E \equiv D ; \Gamma \vdash I \equiv D \] \implies \Gamma \vdash A \land E \equiv I
by (ocl_hypsubst, simp)
```

```ml
lemma \[ \Gamma \vdash A; \Gamma \vdash \neg B; \Gamma \vdash E \equiv I; \Gamma \vdash I \equiv E;\Gamma \vdash \not\beta B; \Gamma \vdash E \equiv I;\Gamma \vdash I \equiv E;\Gamma \vdash \not\beta B \] \implies \Gamma \vdash A \land B
by (ocl_hypsubst, simp)
```

Test of ocl_subst

```ml
lemma (\not\beta X \equiv \not\beta X) = T
apply (rule ext, case_tac \x \vdash \not(X))
apply (ocl_subst (no_asm_simp), simp_all)
```

226
B.4. Library

apply (ocl_subst (no_asm_use)(no_asm_simp)(no_concl_simp) | ocl_subst)
apply (simp)
done

lemma StrongEq_valid_undef:
\[ [\tau \models x \neq y] \implies (x \triangleq y) \tau = T \tau \]
by (ocl_hypsubst,simp)

The following lemma explains strong equality entirely in terms of OCL in the sense of the standard, i.e., by using definedness and strict equality, which is the default. Due to strongEq_charn \((?x \triangleq ?y) = (?x \triangleq ?y)\), the outstanding importance of strong equality becomes obvious. The following lemma explains, why its reformulation in standard operators is possible, but tedious, and represents a post-hoc justification for the introduction strong equality.

lemma strongEq_charn2:
\[(x \triangleq y) = (if \partial x \land \partial y then x = y else if \neg \partial x \land \neg \partial y then T else F endif)\]

apply (rule ext,case_tac x \models \partial (X)
by (ocl_subst,simp_all)

Strong and Weak Definedness Calculi

lemma isDefined_cong5: (\partial X \lor \neg \partial X) = T
apply (rule ext,case_tac x \models \partial (X))
by (ocl_subst, simp_all, ocl_subst, simp_all)

lemma isDefined_cong6: (\neg \partial X \lor \partial X ) = T
apply (rule ext,case_tac x \models \partial (X))
by (ocl_subst, simp_all, ocl_subst, simp)

lemma isDefined_cong7: (\partial X \lor \neg \partial X \lor Z) = T
apply (rule ext,case_tac x \models \partial (X))
by (ocl_subst, simp_all, ocl_subst, simp)

lemma isDefined_cong8: (\neg \partial X \lor \partial X \lor Z) = T
apply (rule ext,case_tac x \models \partial (X))
Appendix B. Isabelle Theories

by (ocl_subst, simp_all, ocl_subst, simp)

lemma isDefined_cong9 : (∅ X ∧ ∅ X) = F
apply (rule ext, case_tac x ⊨ ∅(X))
by (ocl_subst, simp_all, ocl_subst, simp)

lemma isDefined_cong10 : (∅ X ∧ ∅ X) = F
apply (rule ext, case_tac x ⊨ ∅(X))
by (ocl_subst, simp_all, ocl_subst, simp)

lemma isDefined_cong11 : (∅ X ∧ ∅ X ∧ Z) = F
apply (rule ext, case_tac x ⊨ ∅(X))
by (ocl_subst, simp_all, ocl_subst, simp)

lemma isDefined_cong12 : (∅ X ∧ ∅ X ∧ Z) = F
apply (rule ext, case_tac x ⊨ ∅(X))
by (ocl_subst, simp_all, ocl_subst, simp)

lemmas core_definedness =
OclIsDefined__False OclIsDefined__True OclIsDefined__undef
isDefined_idem isDefined_notD0
isDefined_cong4 isDefined_cong3 isDefined_cong2 isDefined_cong1
isDefined_cong5 isDefined_cong6 isDefined_cong7 isDefined_cong8
isDefined_cong9 isDefined_cong10 isDefined_cong11 isDefined_cong12
isDefined_strongEq [THEN is_TRUE_charn, THEN iffD2]
isDefined_strictEq

lemmas strong_definedness = core_definedness
isDefined_ifD0 isDefined_andD0 isDefined_impliesD0 isDefined_orD0

lemmas weak_definedness = core_definedness

isDefined_if isDefined_implies
isDefined_or isDefined_and

thm weak_definedness
thm strong_definedness
thm strictEq_is_strongEq_LE

lemmas weak_prop_LJE =
OclTrue_is_Valid OclFalse_is_Invalid OclUndefined_is_Invalid
OclTrue_is_LocalValid absurd6S absurd5S

228
thm isDefined_if [simplified is_TRUE_charn global_vs_local_validity]

thm weak_prop_LJE

lemma globalValidAnd2conj:
\[ \vDash \partial A; \vDash \partial B \implies (\vDash A \land B) = ((\vDash A) \land (\vDash B)) \]
by (auto simp: global_vs_local_validity[symmetric] localValidAnd2conj)

lemma globalValidNot2localValidNot:
\[ \vDash \partial A \implies (\vDash \neg A) = (\neg (\vDash A)) \]
by (auto simp: global_vs_local_validity[symmetric] localValidNot2not[symmetric])

lemma globalValidOr2localValidDisj:
\[ \vDash \partial A; \vDash \partial B \implies (\vDash A \lor B) = (\forall \tau. (\vDash A) \lor (\vDash B)) \]
by (auto simp: global_vs_local_validity[symmetric] localValidOr2disj [symmetric])

lemma globalValidImplies2localValidimpl:
\[ \vDash \partial A; \vDash \partial B \implies (\vDash A \rightarrow B) = (\forall \tau. (\vDash A) \rightarrow (\vDash B)) \]
by (auto simp: global_vs_local_validity[symmetric] localValidImplies2impl [symmetric])

These previous equivalences demonstrate, why a “global judgement equivalence” calculus does not make sense. Desirable versions of these rules would be:

\[ \vDash \partial A \implies (\vDash \neg A) = (\neg (\vDash A)) \] (B.12)
\[ \vDash \partial A; \vDash \partial B \implies (\vDash A \lor B) = ((\vDash A) \lor (\vDash B)) \] (B.13)
\[ \vDash \partial A; \vDash \partial B \implies (\vDash A \rightarrow B) = ((\vDash A) \rightarrow (\vDash B)) \] (B.14)

This is not achievable since the logical disjunction not distribute over the universal quantifier hidden in the global validity statement.

Thus, a “global judgement equivalence” is not closed under global validity, such that any complete reasoning must be broken down on local judgements sooner or later.

end
Appendix B. Isabelle Theories

begin instance up :: (number)number ..
instance fun:: (type,number)number ..

OCL Ordering Interface

This is only a syntactic interface for the ordering operators in OCL. Concrete definitions are provided in the structures Integer and Real.

consts

OclLe :: ['a', 'b] VAL, ('a', 'b] VAL => 'a Boolean
OclLess :: ['a', 'b] VAL, ('a', 'b] VAL => 'a Boolean
OclGe :: ['a', 'b] VAL, ('a', 'b] VAL => 'a Boolean
OclGreater :: ['a', 'b] VAL, ('a', 'b] VAL => 'a Boolean

syntax

syntax _OclLe_std :: ['a', 'b] VAL, ('a', 'b] VAL => 'a Boolean
(_ _/\_< _ [63,64] [63])
_syntax _OclLess_std :: ['a', 'b] VAL, ('a', 'b] VAL => 'a Boolean
(_ _/\_< _ [63,64] [63])
_syntax _OclGe_std :: ['a', 'b] VAL, ('a', 'b] VAL => 'a Boolean
(_ _/\>_ _ [63,64] [63])
_syntax _OclGreater_std :: ['a', 'b] VAL, ('a', 'b] VAL => 'a Boolean
(_ _/\>_ _ [63,64] [63])

syntax

_syntax _OclLe_ascii :: ['a', 'b] VAL, ('a', 'b] VAL => 'a Boolean
(_ _/\_< _ [63,64] [63])
_syntax _OclLess_ascii :: ['a', 'b] VAL, ('a', 'b] VAL => 'a Boolean
(_ _/\_< _ [63,64] [63])
_syntax _OclGe_ascii :: ['a', 'b] VAL, ('a', 'b] VAL => 'a Boolean
(_ _/\>_ _ [63,64] [63])
_syntax _OclGreater_ascii :: ['a', 'b] VAL, ('a', 'b] VAL => 'a Boolean
(_ _/\>_ _ [63,64] [63])

syntax (xsymbols)

_syntax _OclLe_math :: ['a', 'b] VAL, ('a', 'b] VAL => 'a Boolean
(_ _/\_< _ [63,64] [63])
_syntax _OclLess_math :: ['a', 'b] VAL, ('a', 'b] VAL => 'a Boolean
(_ _/\_< _ [63,64] [63])
_syntax _OclGe_math :: ['a', 'b] VAL, ('a', 'b] VAL => 'a Boolean
(_ _/\>_ _ [63,64] [63])
_syntax _OclGreater_math :: ['a', 'b] VAL, ('a', 'b] VAL => 'a Boolean
(_ _/\>_ _ [63,64] [63])end
B.4.6. **OCL Integer**

```plaintext
theory OCL_Integer
imports
\$HOLOCL_HOME/src/library/basic/OCL_Numerals
begin

constdefs

  is_Integer_0 :: 'a U ⇒ bool
  is_Integer_0 ≡ sum_case (λx. False)
                      (sum_case (λx. True)(λx. False))

  get_Integer_0 :: 'a U ⇒ Integer_0
  get_Integer_0 ≡ sum_case (λx. ε x. True)
                      (sum_case (λx. ε x. True)
                        (sum_case (λx. x)(λx. ε x. True)))

  mk_Integer_0 :: Integer_0 ⇒ 'a U
  mk_Integer_0 ≡ Inr o Inr o Int

lemma get_mk_Integer_id_0:
  get_Integer_0 (mk_Integer_0 x) = x
apply (simp add: get_Integer_0_def mk_Integer_0_def)
done

lemma is_mk_Integer_0:
  is_Integer_0 (mk_Integer_0 x) = True
apply (simp add: is_Integer_0_def mk_Integer_0_def)
done

lemma mk_get_Integer_id_0:
  is_Integer_0 x =⇒ mk_Integer_0 (get_Integer_0 x) = x
apply (simp add: get_Integer_0_def mk_Integer_0_def is_Integer_0_def)
apply (case_tac x, simp, simp)
apply (case_tac b, simp,simp)
apply (case_tac ba, simp,simp)
done

constdefs

  Integer_0 :: ('st, Integer_0 Set_0) VAL
  Integer_0 ≡ lift0(Abs_Set_0 ♯ lift ' UNIV)

defs

  Zero_ocl_int_def: 0 ≡ lift0( Abs_Set_0 ♯ lift ' UNIV)
  One_ocl_int_def: 1 ≡ lift0( Abs_Set_0 ♯ lift ' UNIV)

lemma OCL_is_def_Zero_ocl_int [simp]:
  \partial (0::'a Integer) = T
apply(rule ext)
apply(simp add: OclIsDefined_def Zero_ocl_int_def OclTrue_def ss_lifting)
done

lemma OCL_is_def_One_ocl_int [simp]:
  \partial (1::'a Integer) = T
apply(rule ext)
```

231
Appendix B. Isabelle Theories

apply (simp add: OclIsDefined_def One_ocl_int_def OclTrue_def ss_lifting')
done

defs (overloaded)
  number_of_def (number_of :: 'a Integer)
  ≡ λ b. lift0 (λ a. b)

defs
  plus_def: op + ≡ lift1 (strictify (λ x::Integer. strictify (λ y. "x" +"y" )))
  minus_def: op − ≡ lift1 (strictify (λ x::Integer. strictify (λ y. "x" −"y" )))
  times_def: op * ≡ lift1 (strictify (λ x::Integer. strictify (λ y. "x" *"y" )))

ocl_setup_op [plus, minus, times]

lemma plus_zero: (X::'a Integer) + 0 = X
  by (rule ext, simp add: plus_def Zero_ocl_int_def ss_lifting')

lemma plus_commute: (X::'a Integer) + Y = Y + X
  by (rule ext, simp add: plus_def ss_lifting')

defs
  OclLe_def OclLe ≡ lift1 (strictify (λ x::Integer. strictify (λ y. "x" <="y" )))
  OclLess_def OclLess ≡ lift1 (strictify (λ x::Integer. strictify (λ y. "x" <"y" )))
  OclGe_def OclGe ≡ lift1 (strictify (λ x::Integer. strictify (λ y. 
  "x" >="y" )))
  OclGreater_def OclGreater ≡ lift1 (strictify (λ x::Integer. strictify (λ y. 
  "x" >"y" )))

ocl_setup_op [OclLe, OclLess, OclGe, OclGreater]

lemma OclGe_elim [simp]: OclGe (X::'a Integer) Y = OclLe Y X
  apply (rule ext)
  apply (case_tac x = Y, simp_all, ocl_hypsubst [2], simp_all)
  apply (case_tac x = Y, simp_all, ocl_hypsubst [2], simp_all)
  apply (simp add: OclGe_def OclLe_def
  OclIsDefined_def strictify_def OclLocalValid_def

232
B.4. Library

apply(simp add: Orderings.linorder_not_less)
done

lemma OclGreater_elim [simp]: OclGreater (X::'a Integer) Y = OclLess Y X
apply(rule ext)
apply(case_tac x ⊨ ∂(X), simp_all, ocl_hypsubst [2], simp_all)
apply(case_tac x ⊨ ∂(Y), simp_all, ocl_hypsubst [2], simp_all)
apply(simp add: OclGreater_def OclLess_def OclIsDefined_def strictify_def OclLocalValid_def
        o_def lift0_def lift1_def lift2_def
        OclTrue_def OclUndefined_def DEF_def)
apply(simp add: Orderings.linorder_not_le)
done

lemma OclLe_compute [simp]:
    OclLe((number_of:: bin ⇒ 'a Integer) a)((number_of:: bin ⇒ 'a Integer) b) =
    (if ¬ neg ((number_of:: bin ⇒ int) (bin_add b (bin_minus a)))then T else F)
apply (rule ext)
apply (simp add: number_of_def OclLe_def
        OclIsDefined_def strictify_def
        o_def lift0_def lift1_def lift2_def
        OclTrue_def OclFalse_def OclUndefined_def DEF_def)
done

lemma OclLess_compute [simp]:
    OclLess((number_of:: bin ⇒ 'a Integer) a)((number_of:: bin ⇒ 'a Integer) b) =
    (if neg ((number_of:: bin ⇒ int) (bin_add a (bin_minus b)))then T else F)
apply (rule ext)
apply (simp add: number_of_def OclLess_def
        OclIsDefined_def strictify_def
        o_def lift0_def lift1_def lift2_def
        OclTrue_def OclFalse_def OclUndefined_def DEF_def)
done

consts

OclMod :: '[a Integer, 'a Integer] ⇒ 'a Integer
        (_ -> mod(' _ ') [66,65])65

OclDiv :: '[a Integer, 'a Integer] ⇒ 'a Integer
        (_ -> div(' _ ') [66,65])65

OclRound :: 'a Integer ⇒ 'a Integer
Appendix B. Isabelle Theories

\[(\_ \rightarrow \text{round}'(\_))\]  
\(\text{OclFloor} :: \cdot a \text{ Integer} \Rightarrow \cdot a \text{ Integer} \) \(\)  
\(\text{OclFloor} \equiv \text{round}'(\_)[66]\)  
\(\text{OclFloor} \equiv \text{floor}'(\_)[66]\)  
\(\text{OclMin} :: \cdot [a \cdot \text{ Integer}, a \cdot \text{ Integer}] \Rightarrow \cdot a \text{ Integer} \)  
\(\text{OclMax} :: \cdot [a \cdot \text{ Integer}, a \cdot \text{ Integer}] \Rightarrow \cdot a \text{ Integer} \)  
\(\text{OclMax} \equiv \text{max}'(\_)[66,65]65\)  
\(\text{OclAbs} :: \cdot a \text{ Integer} \Rightarrow \cdot a \text{ Integer} \)  
\(\text{OclAbs} \equiv \text{abs}'(\_)[66]\)  
\(\text{OclNegative} :: \cdot a \text{ Integer} \Rightarrow \cdot a \text{ Integer} \)  
\(\text{OclNegative} \equiv \text{abs}'(\_)[66]\)

defs
\(\text{OclRound_def} \equiv \text{SELF} \rightarrow \text{round}() \equiv \text{SELF}\)  
\(\text{OclFloor_def} \equiv \text{SELF} \rightarrow \text{floor}() \equiv \text{SELF}\)

defs
\(\text{OclMod_def} \equiv \text{OclMod} \equiv \text{lift}_2(\text{strictify}(\lambda x::\text{Integer}_0. \text{strictify}(\lambda y. \text{if } \lceil y \rceil = 0 \text{ then down else } \lfloor \lfloor x \rfloor / \lfloor y \rfloor \rfloor))\)  
\(\text{OclDiv_def} \equiv \text{OclDiv} \equiv \text{lift}_2(\text{strictify}(\lambda x::\text{Integer}_0. \text{strictify}(\lambda y. \text{if } \lceil y \rceil = 0 \text{ then down else } \lfloor \lfloor x \rfloor / \lfloor y \rfloor \rfloor))\)  
\(\text{OclMin_def} \equiv \text{SELF} \rightarrow \text{min}(I) \equiv \text{if } (\text{SELF} \leq I) \text{ then SELF else I endif}\)  
\(\text{OclMax_def} \equiv \text{SELF} \rightarrow \text{max}(I) \equiv \text{if } (\text{SELF} \geq I) \text{ then SELF else I endif}\)  
\(\text{OclAbs_def} \equiv \text{OclAbs} \equiv \text{lift}_1(\text{strictify}(\lambda x::\text{Integer}_0. \text{abs}(\lfloor x \rfloor)))\)  
\(\text{OclNegative_def} \equiv \text{OclNegative} \equiv \text{lift}_1(\text{strictify}(\lambda x::\text{Integer}_0. \text{abs}(\lfloor x \rfloor)))\)

ocl_setup op \(\{\text{OclMod, OclDiv, OclAbs, OclNegative}\}\)

Computational Rules
Foundational rule over Integer values in OCL
B.4. Library

lemma Integer_values_defined[simp];
\partial((number_of:: bin ⇒ 'a Integer) a) = T
apply (rule ext)
by (simp add: number_of_def
OclIsDefined_def strictify_def
o_def lift0_def lift1_def lift2_def
OclTrue_def OclUndefined_def DEF_def)

These two elementary cases are required by the standard:

lemma div_zero: x \rightarrow div( 0 ) = ⊥
apply (rule ext)
apply (simp add: OclDiv_def Zero_ocl_int_def
OclIsDefined_def strictify_def
o_def lift0_def lift1_def lift2_def
OclTrue_def OclUndefined_def DEF_def)
done

lemma mod_zero: x \rightarrow mod( 0 ) = ⊥
apply (rule ext)
apply (simp add: OclMod_def Zero_ocl_int_def
OclIsDefined_def strictify_def
o_def lift0_def lift1_def lift2_def
OclTrue_def OclFalse_def OclUndefined_def DEF_def)
done

lemma plus_compute [simp];
((number_of:: bin ⇒ 'a Integer) a) + (number_of:: bin ⇒ 'a Integer) b) =
number_of (bin_add a b)
apply (rule ext)
apply (simp add: number_of_def plus_def
OclIsDefined_def strictify_def
o_def lift0_def lift1_def lift2_def
OclTrue_def OclFalse_def OclUndefined_def DEF_def)
done

lemma 3 + 4 = (7::'a Integer) by simp

lemma times_compute [simp];
((number_of:: bin ⇒ 'a Integer) a) * (number_of:: bin ⇒ 'a Integer) b) =
number_of (bin_mult a b)
apply (rule ext)
apply (simp add: number_of_def times_def
OclIsDefined_def strictify_def
o_def lift0_def lift1_def lift2_def
OclTrue_def OclFalse_def OclUndefined_def DEF_def)
done
Appendix B. Isabelle Theories

lemma minus_compute [simp]:
((number_of::bin ⇒ 'a Integer) a) − ((number_of::bin ⇒ 'a Integer) b) =
(number_of ((bin_add a (bin_minus b))))
apply (rule ext)
apply (simp add: number_of_def minus_def
       OclIsDefined_def strictify_def
       o_def lift0_def lift1_def lift2_def
       OclTrue_def OclFalse_def OclUndefined_def DEF_def)
done

lemma strong_eq_compute [simp]:
¬ iszero((number_of::bin ⇒ int) (bin_add a (bin_minus b))) ⟹
(((number_of::bin⇒'a Integer)a) ≠ ((number_of::bin⇒'a Integer)b)) = F
apply (rule ext)
apply (simp add: number_of_def OclStrongEq_def
       OclIsDefined_def strictify_def
       o_def lift0_def lift1_def lift2_def
       OclTrue_def OclFalse_def OclUndefined_def DEF_def)
done

lemma strict_eq_compute [simp]:
¬ iszero((number_of::bin ⇒ int) (bin_add a (bin_minus b))) ⟹
(((number_of::bin⇒'a Integer)a) ≡ ((number_of::bin⇒'a Integer)b)) = F
apply (rule ext)
apply (simp add: number_of_def OclStrictEq_def
       OclIsDefined_def strictify_def
       o_def lift0_def lift1_def lift2_def
       OclTrue_def OclFalse_def OclUndefined_def DEF_def)
done

lemma [[ τ ⊨ 3 ≜ C ; τ ⊨ C + 2 ≜ (7::'a Integer) ]] ⟹ τ ⊨ A ∧ B
apply ocl_hypsubst
apply simp
done

B.4.7. OCL Real

theory OCL_Real
imports

$HOLOCL_HOME/src/library/basic/OCCL_Numerals

begin constdefs
  is_Real_0 :: 'a U ⇒ bool
  is_Real_0 ≡ sum_case (λx. False) (sum_case (λx. True) (λx. False))
  get_Real_0 :: 'a U ⇒ Real_0
  get_Real_0 ≡ sum_case (λx. ϵ x. True) (sum_case (λx. ε x. True) (λx. False))
  mk_Real_0 :: Real_0 ⇒ 'a U
  mk_Real_0 ≡ Inr o Inl

lemma get_mk_Real_id_0 : get_Real_0 (mk_Real_0 x) = x
  apply (simp add: get_Real_0_def mk_Real_0_def)
done

lemma is_mk_Real_0 : is_Real_0 (mk_Real_0 x) = True
  apply (simp add: is_Real_0_def mk_Real_0_def)
done

lemma mk_get_Real_id_0 : is_Real_0 x =⇒ mk_Real_0 (get_Real_0 x) = x
  apply (simp add: get_Real_0_def mk_Real_0_def is_Real_0_def)
  apply (case_tac x, simp, simp)
done

costdef
  Real_0 :: ('st, Real_0 Set_0) VAL
  Real_0 ≡ lift0(Abs_Set_0⇧� lift ' UNIV)⇧�

defs
  OclZero_def : 0 ≡ lift0⇧� 0⇧�⇧�
  OclOne_def : 1 ≡ lift0⇧� 1⇧�

defs
  plus_def : op + ≡ lift2⇧� (strictify⇧� (λ x::Real_0. strictify⇧� (λ y.⇧�
                                  (⇧� x⇧�)⇧�+(⇧� y⇧�))⇧�))⇧�
  minus_def : op − ≡ lift2⇧� (strictify⇧� (λ x::Real_0. strictify⇧� (λ y.⇧�
                                  (⇧� x⇧�)⇧�−(⇧� y⇧�))⇧�))⇧�
  times_def : op * ≡ lift2⇧� (strictify⇧� (λ x::Real_0. strictify⇧� (λ y.⇧�
                                  (⇧� x⇧�)⇧�*(⇧� y⇧�))⇧�))⇧�

ocl_setup_op [plus, minus, times]

costs
  OclRound :: 'a Real ⇒ 'a Integer (⇧� _−>round⇧� (' ) [66])
  OclFloor :: 'a Real ⇒ 'a Integer (⇧� _−>floor⇧� ( ) [66])

  OclMin :: ['a Real, 'a Real] ⇒ 'a Real (⇧� _−>min⇧� ( ) [66,65])
  OclMax :: ['a Real, 'a Real] ⇒ 'a Real (⇧� _−>max⇧� ( ) [66,65])

237
Appendix B. Isabelle Theories

\[ OclDivide :: 'a Real, 'a Real => 'a Real \]
\[ (_ -> divide( _ )) [66,65]65 \]

\[ OclAbs :: 'a Real => 'a Real \]
\[ (_ -> abs( _ )) [66] \]

\[ OclNegative :: 'a Real => 'a Real \]
\[ (_ -> [66]) \]

defs

\[ OclDivide_def \]
\[ OclDivide \equiv lift_2 (strictify(\lambda x::Real.0. strictify(\lambda y::Real.0. (_ -> divide( _ ))))) \]

\[ OclLe_def \]
\[ OclLe \equiv lift_2 (strictify(\lambda x::Real.0. strictify(\lambda y::Real.0. (_ -> ( _ ))) \]

\[ OclLess_def \]
\[ OclLess \equiv lift_2 (strictify(\lambda x::Real.0. strictify(\lambda y::Real.0. (_ -> ( _ ))) \]

\[ OclGe_def \]
\[ OclGe \equiv lift_2 (strictify(\lambda x::Real.0. strictify(\lambda y::Real.0. (_ -> ( _ ))) \]

\[ OclGreater_def \]
\[ OclGreater \equiv lift_2 (strictify(\lambda x::Real.0. strictify(\lambda y::Real.0. (_ -> ( _ ))) \]

\[ OclMin_def \]
\[ SELF -> min(I) \equiv if (SELF <= I) then SELF else I endif \]

\[ OclMax_def \]
\[ SELF -> max(I) \equiv if (SELF >= I) then SELF else I endif \]

\[ OclAbs_def \]
\[ OclAbs \equiv lift_1 (strictify(\lambda x::Real.0. (_ -> abs( _ ))) \]

\[ OclNegative_def \]
\[ OclNegative \equiv lift_1 (strictify(\lambda x::Real.0. (_ -> ( _ ))) \]

\[ ocl_setup_op [OclMax, OclMin, OclNegative] \]
\[ ocl_setup_op [OclLe, OclLess, OclGe, OclGreater] \]

end

B.4.8. OCL String

theory OCL_String

238
B.4. Library

```plaintext
imports
$HOLOCL_HOME/src/library/basic/OCL_Integer

begin

defs

is_String_0 : 'a U ⇒ bool
is_String_0 ≡ sum_case (λ x. False)
  (sum_case (λ x. False)
    (sum_case (λ x. False)
      (sum_case (λ x. False)(λ x. True)))))

get_String_0 : 'a U ⇒ String_0
get_String_0 ≡ sum_case (λ x. ϵ x. True)
  (sum_case (λ x. ϵ x. True)
    (sum_case (λ x. ϵ x. True)
      (sum_case (λ x. ϵ x. True)(λ x. x))))

mk_String_0 : String_0 ⇒ 'a U
mk_String_0 ≡ Inr ◦ Inr ◦ Inr ◦ Inr o Inr

lemma get_mk_String_id_0: get_String_0 (mk_String_0 x) = x
apply (simp add: get_String_0_def mk_String_0_def)
done

lemma is_mk_String_0: is_String_0 (mk_String_0 x) = True
apply (simp add: is_String_0_def mk_String_0_def)
done

lemma mk_get_String_id_0: is_String_0 x =⇒ mk_String_0 (get_String_0 x) = x
apply (simp add: get_String_0_def mk_String_0_def is_String_0_def)
apply (case_tac x, simp, simp)
apply (case_tac b, simp, simp)
apply (case_tac ba, simp, simp)
apply (case_tac bb, simp, simp)
done

consts

String_0 :: ('st, String_0 Set_0) VAL
String_0 ≡ lift0(Abs_Set_0⌞UNIV⌟)

consts

OclConcat :: ['a String, 'a String] ⇒ 'a String
OclToLower :: 'a String ⇒ 'a String
OclToUpper :: 'a String ⇒ 'a String
stringsize :: 'a String ⇒ 'a Integer
OclSubstring :: ['a String, 'a Integer, 'a Integer] ⇒ 'a String

defs

OclConcat_def: OclConcat ≡ lift2(strictify(λ x::String_0.
  strictify(λ y::String_0.(x | y)))
stringsize_def: stringsize ≡ lift1(strictify(λ x::String_0. | int(size ("x")) )

OclSubstring_def:OclSubstring ≡ lift3(strictify(λ s::String_0.
```

Appendix B. Isabelle Theories

strictify(λ l::Integer_0. strictify(λ u::Integer_0. 
(List.take (nat(⌜⌜u⌟ − ⌜⌜l⌟ + 1::int⌟ ⌝))
(List.drop (nat(⌜⌜l⌟ − 1::int⌟ ⌝)) ⌝))))

end

B.4.9. OCL Collection

theory OCL_Collection
imports $HOLOCL_HOME/src/library/basic/OCL_Integer
begin

The purpose of this theory is to provide the syntactic framework for the abstract class Collection in the sense of the standard. It is defined by the Isabelle type class "collection". We require that collections must always have a bottom element.

Unfortunately, this implementation by the type class "collection" is an approximation (a type constructor class would be more appropriate, but is not available in the Isabelle type system). Consequently, in lemmas the concrete type instances for operators resulting from the abstract class must be given; the types inferred by Isabelle automatically are too coarse and can lead to dead ends in proofs (definitions can not be unfolded).

The Syntax of Abstract Operations

The core operation size (which may be undefined if and only if the collection is infinitely large), select, collect, includes and any.

Operations on all Collection Types

consts

OclSize :: (τ,α::collection) VAL ⇒ τ Integer
OclCount :: ([τ,β::collection] VAL,τ,α::bot) VAL ⇒ τ Integer
OclIncludes :: ([τ,β::collection] VAL,τ,α::bot) VAL ⇒ τ Boolean
OclExcludes :: ([τ,β::collection] VAL,τ,α) VAL ⇒ τ Boolean
OclIncluding :: ([τ,β::collection] VAL,τ,α) VAL ⇒ (τ,β) VAL
OclExcluding :: ([τ,β::collection] VAL,τ,α) VAL ⇒ (τ,β) VAL
OclFlatten :: (τ,α::collection) VAL ⇒ (τ,β::bot) VAL
OclSum :: (τ,α::collection) VAL ⇒ τ Integer
OclAsSet :: (τ,α::collection) VAL ⇒ (τ,β::collection) VAL
OclAsSequence :: (τ,α::collection) VAL ⇒ (τ,β::collection) VAL

240
B.4. Library

\begin{itemize}
  \item \texttt{OclAsBag} :: \((\alpha::\text{collection})\ VAL \Rightarrow (\beta::\text{collection})\ VAL\)
  \item \texttt{OclAsOrderedSet} :: \((\alpha::\text{collection})\ VAL \Rightarrow (\beta::\text{collection})\ VAL\)
  \item \texttt{OclExcludesAll} :: \([\alpha::\text{collection}]\ VAL, (\alpha::\text{collection}) \Rightarrow \tau \ Boolean\)
  \item \texttt{OclIsEmpty} :: \((\alpha::\text{collection})\ VAL \Rightarrow \tau \ Boolean\)
  \item \texttt{OclNotEmpty} :: \((\alpha::\text{collection})\ VAL \Rightarrow \tau \ Boolean\)
  \item \texttt{OclComplement} :: \((\beta::\text{collection})\ VAL \Rightarrow (\tau::\text{collection})\ VAL\)
  \item \texttt{OclUnion} :: \((\beta::\text{collection})\ VAL, (\beta::\text{collection}) \Rightarrow (\tau::\text{collection})\ VAL\)
  \item \texttt{OclCollectionRange} :: \((\beta::\text{collection})\ VAL, (\tau::\text{collection}) \Rightarrow (\tau::\text{collection})\ VAL\)
\end{itemize}

\textbf{syntax}

\begin{itemize}
  \item \texttt{OclSize\_std} :: \((\alpha::\text{collection})\ VAL \Rightarrow \tau \ Integer\)
    \(\rightarrow \text{size}(\cdot)(\cdot)\)[66]
  \item \texttt{OclCount\_std} :: \([\beta::\text{collection}]\ VAL, (\alpha::\text{bot})\ VAL \Rightarrow \tau \ Integer\)
    \(\rightarrow \text{count}(\cdot)(\cdot)\)[66,65]
  \item \texttt{OclIncludes\_std} :: \([\beta::\text{collection}]\ VAL, (\alpha::\text{collection}) \Rightarrow \tau \ Boolean\)
    \(\rightarrow \text{includes}(\cdot)(\cdot)\)[66,65]
  \item \texttt{OclExcludes\_std} :: \([\beta::\text{collection}]\ VAL, (\alpha::\text{collection}) \Rightarrow \tau \ Boolean\)
    \(\rightarrow \text{excludes}(\cdot)(\cdot)\)[66,65]
  \item \texttt{OclFlatten\_std} :: \((\alpha::\text{collection})\ VAL \Rightarrow (\beta::\text{collection})\ VAL\)
    \(\rightarrow \text{flatten}(\cdot)\)[66]
  \item \texttt{OclSum\_std} :: \((\alpha::\text{collection})\ VAL \Rightarrow (\beta::\text{collection})\ VAL\)
    \(\rightarrow \text{sum}(\cdot)(\cdot)\)[66]
  \item \texttt{OclAsSet\_std} :: \((\alpha::\text{collection})\ VAL \Rightarrow (\beta::\text{collection})\ VAL\)
    \(\rightarrow \text{asSet}(\cdot)(\cdot)\)[66]
  \item \texttt{OclAsSequence\_std} :: \((\alpha::\text{collection})\ VAL \Rightarrow (\beta::\text{collection})\ VAL\)
    \(\rightarrow \text{asSequence}(\cdot)(\cdot)\)[66]
  \item \texttt{OclAsBag\_std} :: \((\alpha::\text{collection})\ VAL \Rightarrow (\beta::\text{collection})\ VAL\)
    \(\rightarrow \text{asBag}(\cdot)(\cdot)\)[66]
  \item \texttt{OclAsOrderedSet\_std} :: \((\alpha::\text{collection})\ VAL \Rightarrow (\beta::\text{collection})\ VAL\)
    \(\rightarrow \text{asOrderedSet}(\cdot)(\cdot)\)[66]
  \item \texttt{OclIncludesAll\_std} :: \((\alpha::\text{collection})\ VAL, (\alpha::\text{collection}) \Rightarrow \tau \ Boolean\)
    \(\rightarrow \text{includesAll}(\cdot)(\cdot)\)[66,65]
  \item \texttt{OclExcludesAll\_std} :: \((\alpha::\text{collection})\ VAL, (\alpha::\text{collection}) \Rightarrow \tau \ Boolean\)
    \(\rightarrow \text{excludesAll}(\cdot)(\cdot)\)[66,65]
  \item \texttt{OclIsEmpty\_std} :: \((\alpha::\text{collection})\ VAL \Rightarrow \tau \ Boolean\)
    \(\rightarrow \text{isEmpty}(\cdot)(\cdot)\)[66]
\end{itemize}
Appendix B. Isabelle Theories

```plaintext
_OclNotEmpty_std :: (\tau, \alpha::collection) VAL => 'tau Boolean
(_ ->notEmpty(') [66])

_OclIncluding_std :: [(\tau, \beta::collection) VAL,(\tau,\alpha::bot) VAL] => (\tau,\beta) VAL
(_ ->including('))

_OclExcluding_std :: [(\tau, \beta::collection) VAL,(\tau,\alpha::bot) VAL] => (\tau,\beta) VAL
(_ ->excluding('))

_OclComplement_std :: (\tau, \beta::collection) VAL => (\tau, \beta) VAL
(_ ->complement('))

_OclUnion_std :: [(\tau, \beta::collection) VAL, (\tau, \beta) VAL] => (\tau, \beta) VAL
(_ ->union(')) [66,65,65]

Syntax

_OclSize_ascii :: (\tau, \alpha::collection) VAL => 'tau Integer
(_ ->size(')) [66]

_OclCount_ascii :: [(\tau, \beta::collection) VAL,(\tau,\alpha::bot) VAL] => 'tau Integer
(_ ->count(')) [66,65,65]

_OclIncludes_ascii :: [(\tau, \beta::collection) VAL,(\tau, \alpha::bot) VAL] => 'tau Boolean
(_ ->includes(')) [66,65,65]

_OclExcludes_ascii :: [(\tau, \beta::collection) VAL,(\tau, \alpha::bot) VAL] => 'tau Boolean
(_ ->excludes(')) [66,65,65]

_OclFlatten_ascii :: (\tau, \alpha::collection) VAL => (\tau, \beta::collection) VAL
(_ ->flatten(')) [66]

_OclSum_ascii :: (\tau, \alpha::collection) VAL => 'tau Integer
(_ ->sum(')) [66]

_OclAsSet_ascii :: (\tau, \alpha::collection) VAL => (\tau, \beta::collection) VAL
(_ ->asSet(')) [66]

_OclAsSequence_ascii :: (\tau, \alpha::collection) VAL => (\tau, \beta::collection) VAL
(_ ->asSequence(')) [66]

_OclAsBag_ascii :: (\tau, \alpha::collection) VAL => (\tau, \beta::collection) VAL
(_ ->asBag(')) [66]

_OclAsOrderedSet_ascii :: (\tau, \alpha::collection) VAL => (\tau, \beta::collection) VAL
(_ ->asOrderedSet(')) [66]

_OclIncludesAll_ascii :: [(\tau, \alpha::collection) VAL,(\tau, \alpha::collection) VAL] => 'tau Boolean
(_ ->includesAll(')) [66,65,65]
```

242
\_\_OclExcludesAll\_ascii\::\!(['r','\alpha::collection] \text{VAL},(['r','\alpha::collection]) \text{VAL} \Rightarrow ['r] \text{Boolean} \\
\_\_\rightarrow \text{excludesAll('')} [66,65,65]

\_\_OclIsEmpty\_ascii\::\!(['r','\alpha::collection]) \text{VAL} \Rightarrow ['r] \text{Boolean} \\
\_\_\rightarrow \text{isEmpty('')} [66]

\_\_OclNotEmpty\_ascii\::\!(['r','\alpha::collection]) \text{VAL} \Rightarrow ['r] \text{Boolean} \\
\_\_\rightarrow \text{notEmpty('')} [66]

\_\_OclIncluding\_ascii\::\!(['r','\beta::collection]) \text{VAL},(['r','\alpha::bot]) \text{VAL} \Rightarrow ('r,\beta) \text{VAL} \\
\_\_\rightarrow \text{including('')} [66]

\_\_OclExcluding\_ascii\::\!(['r','\beta::collection]) \text{VAL},(['r','\alpha::bot]) \text{VAL} \Rightarrow ('r,\beta) \text{VAL} \\
\_\_\rightarrow \text{excluding('')} [66]

\_\_OclComplement\_ascii\::\!(['r','\beta::collection]) \text{VAL} \Rightarrow ('r,\beta) \text{VAL} \\
\_\_\rightarrow \text{complement('')} [66]

\_\_OclUnion\_ascii\::\!(['r','\beta::collection]) \text{VAL},(['r','\beta::collection]) \text{VAL} \Rightarrow ('r,\beta) \text{VAL} \\
\_\_\rightarrow \text{union('')} [66,65,65]

\_\_OclIntersection\_ascii\::\!(['r','\beta::collection]) \text{VAL},(['r','\beta::collection]) \text{VAL} \Rightarrow ('r,\beta) \text{VAL} \\
\_\_\rightarrow \text{intersection('')} [71,70,70]

\textbf{syntax (xsymbols)}

\_OclSize\_math\::\!(['r','\alpha::collection]) \text{VAL} \Rightarrow ['r] \text{Integer} \\
\_\_\|\| [66]

\_OclCount\_math\::\!(['r','\beta::collection]) \text{VAL},(['r','\alpha::bot]) \text{VAL} \Rightarrow ['r] \text{Integer} \\
\_\_\rightarrow \text{count} [66,65,65]

\_OclIncludes\_math\::\!(['r','\alpha]) \text{VAL},(['r','\beta::collection]) \text{VAL} \Rightarrow ['r] \text{Boolean} \\
\_\_ \_\_ [66,65,65]

\_OclExcludes\_math\::\!(['r','\alpha::bot]) \text{VAL},(['r','\beta::collection]) \text{VAL} \Rightarrow ['r] \text{Boolean} \\
\_\_ \_\_ [66,65,65]

\_OclFlatten\_math\::\!(['r','\alpha::collection]) \text{VAL} \Rightarrow ('r,\beta::collection) \text{VAL} \\
\_\_\|\| [66]

\_OclSum\_math\::\!(['r','\alpha::collection]) \text{VAL} \Rightarrow ['r] \text{Integer} \\
\_\_\rightarrow \text{sum()} [66]

\_OclAsSet\_math\::\!(['r','\alpha::collection]) \text{VAL} \Rightarrow ('r,\beta::collection) \text{VAL} \\
\_\_\rightarrow \text{asSet()} [66]

\_OclAsSequence\_math::\!(['r','\alpha::collection]) \text{VAL} \Rightarrow ('r,\beta::collection) \text{VAL}
Appendix B. Isabelle Theories

\[\text{OclAsBag\_math} \quad : \quad (\tau, \alpha::\text{collection}) \text{ VAL} \Rightarrow (\tau, \beta::\text{collection}) \text{ VAL} \]
\[\text{OclAsOrderedSet\_math} : \quad (\tau, \alpha::\text{collection}) \text{ VAL} \Rightarrow (\tau, \beta::\text{collection}) \text{ VAL} \]
\[\text{OclIncludesAll\_math} : \quad \{\} | \{\_\} \Rightarrow (\tau, \beta::\text{collection}) \text{ VAL} \]
\[\text{OclExcludesAll\_math} : \quad \{\} | \{\_\} \Rightarrow (\tau, \beta::\text{collection}) \text{ VAL} \]
\[\text{OclNotEmpty\_math} : \quad (\tau, \alpha::\text{collection}) \text{ VAL} \Rightarrow (\tau, \beta::\text{collection}) \text{ VAL} \]
\[\text{OclIncludesAll\_math} : \quad \{\_\} \Rightarrow (\tau, \beta::\text{collection}) \text{ VAL} \]
\[\text{OclInIncluding\_math} : \quad (\tau, \alpha::\text{collection}) \text{ VAL} \Rightarrow (\tau, \beta\_\text{including}) \]
\[\text{OclExcluding\_math} : \quad (\tau, \alpha::\text{collection}) \text{ VAL} \Rightarrow (\tau, \beta\_\text{excluding}) \]
\[\text{OclComplement\_math} : \quad (\tau, \alpha::\text{collection}) \text{ VAL} \Rightarrow (\tau, \beta::\text{val}) \]
\[\text{OclUnion\_math} : \quad (\tau, \beta::\text{collection}) \text{ VAL} \Rightarrow (\tau, \beta::\text{val}) \]

Operations on Collection Types with Ordering

\[\text{OclPrepend} \quad : \quad (\tau, (\alpha::\text{collection}) \text{ VAL}, (\tau, \beta::\text{bot}) \text{ VAL}) \Rightarrow (\tau, \alpha::\text{val}) \]
\[\text{OclAppend} \quad : \quad (\tau, (\alpha::\text{collection}) \text{ VAL}, (\tau, \beta::\text{bot}) \text{ VAL}) \Rightarrow (\tau, \alpha::\text{val}) \]
\[\text{OclInsertAt} \quad : \quad (\tau, (\alpha::\text{collection}) \text{ VAL}, \tau \text{ Integer}, (\tau, \beta::\text{val}) \text{ VAL}) \Rightarrow (\tau, \alpha::\text{val}) \]
\[\text{OclAt} \quad : \quad (\tau, (\alpha::\text{collection}) \text{ VAL}, \tau \text{ Integer}) \Rightarrow (\tau, \beta::\text{val}) \]
\[\text{OclIndexOf} \quad : \quad (\tau, (\alpha::\text{collection}) \text{ VAL}, (\tau, \beta::\text{val}) \text{ VAL}) \Rightarrow (\tau, \beta::\text{val}) \]
\[\text{OclFirst} \quad : \quad (\tau, (\alpha::\text{collection}) \text{ VAL}) \Rightarrow (\tau, \beta::\text{val}) \]
\[\text{OclLast} \quad : \quad (\tau, (\alpha::\text{collection}) \text{ VAL}) \Rightarrow (\tau, \beta::\text{val}) \]

\[\text{OclPrepend\_std} \quad : \quad (\tau, (\alpha::\text{collection}) \text{ VAL}, (\tau, \beta::\text{bot}) \text{ VAL}) \Rightarrow (\tau, \alpha::\text{val}) \]
\[\text{OclAppend\_std} \quad : \quad (\tau, (\alpha::\text{collection}) \text{ VAL}, (\tau, \beta::\text{bot}) \text{ VAL}) \Rightarrow (\tau, \alpha::\text{val}) \]
B.4. Library

\[\begin{align*}
_\text{OclInsertAt_std} &:: ([\tau', \alpha::\text{collection}] \ \text{VAL}, \tau \ \text{Integer}, ([\tau', \alpha] \ \text{VAL}) \Rightarrow ([\tau', \alpha] \ \text{VAL}) \\
_\text{OclAt_std} &:: ([\tau', \alpha::\text{collection}] \ \text{VAL}, \tau \ \text{Integer}) \Rightarrow ([\tau', \beta] \ \text{VAL}) \\
_\text{OclIndexOf_std} &:: ([\tau', \alpha::\text{collection}] \ \text{VAL}, ([\tau', \beta] \ \text{VAL}) \Rightarrow \tau \ \text{Integer} \\
_\text{OclFirst_std} &:: ([\tau', \alpha::\text{collection}] \ \text{VAL}) \Rightarrow ([\tau', \beta] \ \text{VAL}) \\
_\text{OclLast_std} &:: ([\tau', \alpha::\text{collection}] \ \text{VAL}) \Rightarrow ([\tau', \beta] \ \text{VAL})
\end{align*}\]

Syntax (symbols)

\[\begin{align*}
_\text{OclPrepend_ascii} &:: ([\tau', (\alpha::\text{collection})] \ \text{VAL}, ([\tau', \beta::\text{bot}] \ \text{VAL}) \Rightarrow ([\tau', \alpha] \ \text{VAL}) \\
_\text{OclAppend_ascii} &:: ([\tau', (\alpha::\text{collection})] \ \text{VAL}, ([\tau', \beta::\text{bot}] \ \text{VAL}) \Rightarrow ([\tau', \alpha] \ \text{VAL}) \\
_\text{OclInsertAt_ascii} &:: ([\tau', (\alpha::\text{collection})] \ \text{VAL}, ([\tau', \beta::\text{bot}] \ \text{VAL}) \Rightarrow ([\tau', \alpha] \ \text{VAL}) \\
_\text{OclAt_ascii} &:: ([\tau', (\alpha::\text{collection})] \ \text{VAL}, ([\tau', \beta] \ \text{VAL}) \Rightarrow \tau \ \text{Integer} \\
_\text{OclIndexOf_ascii} &:: ([\tau', (\alpha::\text{collection})] \ \text{VAL}, ([\tau', \beta] \ \text{VAL}) \Rightarrow \tau \ \text{Integer} \\
_\text{OclFirst_ascii} &:: ([\tau', \alpha::\text{collection}] \ \text{VAL}) \Rightarrow ([\tau', \beta] \ \text{VAL}) \\
_\text{OclLast_ascii} &:: ([\tau', \alpha::\text{collection}] \ \text{VAL}) \Rightarrow ([\tau', \beta] \ \text{VAL})
\end{align*}\]

Operations on Collection Types without Ordering

\[\begin{align*}
\text{OclIntersection} &:: ([\tau', \beta::\text{collection}] \ \text{VAL}, ([\tau', \beta] \ \text{VAL}) \Rightarrow ([\tau', \beta] \ \text{VAL})
\end{align*}\]
Appendix B. Isabelle Theories

syntax

\[
\text{OclIntersection}_{\text{std}} :: [(\tau, \beta :: \text{collection}) \text{ VAL}, (\tau, \beta) \text{ VAL}] => (\tau, \beta) \text{ VAL} \\
(\_ \rightarrow \text{intersection}(\_ \_)) [71, 70, 70]
\]

syntax

\[
\text{OclIntersection}_{\text{ascii}} :: [(\tau, \beta :: \text{collection}) \text{ VAL}, (\tau, \beta) \text{ VAL}] => (\tau, \beta) \text{ VAL} \\
(\_ \rightarrow \text{intersection}(\_ \_)) [71, 70, 70]
\]

syntax

\[
(\_ \cap \_ \_ \_ \_)[71, 70, 70]
\]

The Syntax of Iterators

This section contains the operators of OCL involving binding, which as subsumed under the category “iterators” in the standard. Since we use a shallow embedding, we heavily use higher-order abstract syntax (HOAS) here.

However, quantifiers were not defined on the basis of iterate in HOL-OCL. This is since iterate is undefined for all infinite collections, in contrast to quantifiers, which do not necessarily have computational content.

consts

\[
\text{OclForAll} :: [(\tau, \beta :: \text{collection}) \text{ VAL}, (\tau, \alpha :: \text{bot}) \text{ VAL} => (\tau \text{ Boolean}) => (\tau \text{ Boolean}]
\]

\[
\text{OclSelect} :: [(\tau, \beta :: \text{collection}) \text{ VAL}, (\tau, \alpha :: \text{bot}) \text{ VAL} => (\tau \text{ Boolean})
\]

\[
\text{OclReject} :: [(\tau, \beta :: \text{collection}) \text{ VAL}, (\tau, \alpha :: \text{bot}) \text{ VAL} => (\tau \text{ Boolean})
\]

\[
\text{OclCollect} :: [(\tau, \gamma :: \text{collection}) \text{ VAL}, (\tau, \alpha :: \text{bot}) \text{ VAL} \rightarrow (\tau, \beta :: \text{bot}) \text{ VAL}]
\]

\[
\text{OclCollectNested} :: [(\tau, \gamma :: \text{collection}) \text{ VAL}, (\tau, \alpha :: \text{bot}) \text{ VAL} \rightarrow (\tau, \beta :: \text{bot}) \text{ VAL}]
\]

\[
\text{OclIterate} :: [(\tau, \gamma :: \text{collection}) \text{ VAL}, (\tau, \alpha :: \text{bot}) \text{ VAL} \rightarrow (\tau, \beta :: \text{bot}) \text{ VAL} \rightarrow (\tau, \beta :: \text{bot}) \text{ VAL}]
\]

246
OclIsUnique : [(\'\\tau, \'\\beta::collection\') VAL, (\'\\tau, \'\\alpha::bot\') VAL => (\'\\tau, \'\\gamma::bot\') VAL] => \'\\tau\ Boolean

OclOne : [(\'\\tau, \'\\beta::collection\') VAL, (\'\\tau, \'\\alpha::bot\') VAL => \'\\tau\ Boolean] => \'\\tau\ Boolean

OclAny : [(\'\\tau, \'\\beta::collection\') VAL, (\'\\tau, \'\\alpha::bot\') VAL => \'\\tau\ Boolean]
=\>(\'\\tau, \'\\alpha::bot\') VAL

constdefs

OclExists :: [(\'\\tau, \'\\beta::collection\') VAL, (\'\\tau, \'\\alpha::bot\') VAL => \'\\tau\ Boolean] => \'\\tau\ Boolean

OclExists S P \equiv \neg(OclForAll S (\lambda x. \neg(P x)))

syntax

_ OclForAll_std :: (\'\\tau, \'\\beta::collection\') VAL \Rightarrow idt \Rightarrow \'\\tau\ Boolean
(_ =>forall(_, _)) [71, 100, 70, 50]

_ OclExists_std :: (\'\\tau, \'\\beta::collection\') VAL \Rightarrow idt \Rightarrow \'\\tau\ Boolean
(_ =>exists(_, _)) [71, 100, 70, 50]

_ OclSelect_std :: [(\'\\tau, \'\\beta::collection\') VAL, (\'\\tau, \'\\alpha::bot\') VAL, \'\\tau\ Boolean]
=> (\'\\tau, \'\\beta::collection\') VAL
(_ =>select(_, _)) [71, 100, 70, 50]

_ OclReject_std :: [(\'\\tau, \'\\beta::collection\') VAL, (\'\\tau, \'\\alpha::bot\') VAL, \'\\tau\ Boolean]
=> (\'\\tau, \'\\beta::collection\') VAL
(_ =>reject(_, _)) [71, 100, 70, 50]

_ OclCollect_std :: [(\'\\tau, \'\\beta::collection\') VAL, (\'\\tau, \'\\alpha::bot\') VAL, (\'\\tau, \'\\beta::bot\') VAL]
=> (\'\\tau, \'\\delta::collection\') VAL
(_ =>collect(_, _)) [71, 100, 70, 50]

_ OclCollectNested_std :: [(\'\\tau, \'\\gamma::collection\') VAL, (\'\\tau, \'\\alpha::bot\') VAL, (\'\\tau, \'\\beta::bot\') VAL]
=> (\'\\tau, \'\\delta::collection\') VAL
(_ =>collectNested(_, _)) [71, 100, 70, 50]

_ OclIterate_std :: [(\'\\tau, \'\\beta::collection\') VAL, idt, idt, \'\\gamma\', \'\\delta\']
=> (\'\\tau, \'\\delta\') VAL
(_ =>iterate(_, _, _)) [71, 100, 70, 50]

_ OclOne_std :: [(\'\\tau, \'\\beta::collection\') VAL, \alpha, \'\\tau\ Boolean] => \'\\tau\ Boolean
(_ =>one(_, _)) [71, 100, 70, 50]

_ OclIsUnique_std :: [(\'\\tau, \'\\beta::collection\') VAL, \alpha, \'\\tau\ Boolean] => \'\\tau\ Boolean
(_ =>isUnique(_, _)) [71, 100, 70, 50]

syntax

_ OclForAll_ascii :: (\'\\tau, \'\\beta::collection\') VAL \Rightarrow idt \Rightarrow \'\\tau\ Boolean
Appendix B. Isabelle Theories

\( (_\rightarrow \text{forAll}(_ | _')) [71, 100, 70, 50] \)

\( \_\_\_\text{OclExists}_\text{ascii} :: (\tau, \text{',}:\text{collection}) \text{ VAL } \Rightarrow \text{idt } \Rightarrow '\tau \text{ Boolean } \\
(_\rightarrow \text{exists}(\_ | _')) [71, 100, 70, 50] \)

\( \_\_\_\text{OclSelect}_\text{ascii} :: ([\tau, \text{',}:\text{collection}] \text{ VAL }, (\tau, \text{'},:\text{bot}) \text{ VAL }, '\tau \text{ Boolean } ] \\
\Rightarrow (\tau, \text{'},:\text{collection}) \text{ VAL } \\
(_\rightarrow \text{select}(_ | _')) [71, 100, 70, 50] \)

\( \_\_\_\text{OclReject}_\text{ascii} :: ([\tau, \text{',}:\text{collection}] \text{ VAL }, (\tau, \text{'},:\text{bot}) \text{ VAL }, '\tau \text{ Boolean } ] \\
\Rightarrow (\tau, \text{'},:\text{collection}) \text{ VAL } \\
(_\rightarrow \text{reject}(_ | _')) [71, 100, 70, 50] \)

\( \_\_\_\text{OclCollect}_\text{ascii} :: ([\tau, \text{',}:\text{collection}] \text{ VAL }, (\tau, \text{'},:\text{bot}) \text{ VAL }, (\tau, \text{'},:\text{bot}) \text{ VAL } ] \\
\Rightarrow (\tau, \text{'},:\text{collection}) \text{ VAL } \\
(_\rightarrow \text{collect}(_ | _')) [71, 100, 70, 50] \)

\( \_\_\_\text{OclCollectNested}_\text{ascii} :: ([\tau, \text{',}:\text{collection}] \text{ VAL }, (\tau, \text{'},:\text{bot}) \text{ VAL }, (\tau, \text{'},:\text{bot}) \text{ VAL } ] \\
\Rightarrow (\tau, \text{'},:\text{collection}) \text{ VAL } \\
(_\rightarrow \text{collectNested}(_ | _')) [71, 100, 70, 50] \)

\( \_\_\_\text{OclIterate}_\text{ascii} :: ([\tau, \text{',}:\text{collection}] \text{ VAL }, \text{idt}, \text{idt}, '\gamma', '\delta'] \\
\Rightarrow (\tau, \delta) \text{ VAL } \\
(_\rightarrow \text{iterate}(_ | _ | _')) [71, 100, 70, 50] \)

\( \_\_\_\text{OclOne}_\text{ascii} :: ([\tau, \text{',}:\text{collection}] \text{ VAL }, '\alpha', '\tau \text{ Boolean }] \Rightarrow '\tau \text{ Boolean } \\
(_\rightarrow \text{one'}(_ | _')) [71, 100, 70, 50] \)

\( \_\_\_\text{OclIsUnique}_\text{ascii} :: ([\tau, \text{',}:\text{collection}] \text{ VAL }, '\alpha', '\tau \text{ Boolean }] \Rightarrow '\tau \text{ Boolean } \\
(_\rightarrow \text{isUnique'}(_ | _')) [71, 100, 70, 50] \)

\text{syntax (xsymbols)}

\( \_\_\_\text{OclForAll}_\text{math} :: \text{idt } \Rightarrow (\tau, \text{'},:\text{collection}) \text{ VAL } \Rightarrow '\tau \text{ Boolean } \\
(\forall \_ \in \_ \_ \_ \_ ) [71, 100, 70, 50] \)

\( \_\_\_\text{OclExists}_\text{math} :: \text{idt } \Rightarrow (\tau, \text{'},:\text{collection}) \text{ VAL } \Rightarrow '\tau \text{ Boolean } \\
(\exists \_ \in \_ \_ \_ \_ ) [71, 100, 70, 50] \)

\( \_\_\_\text{OclSelect}_\text{math} :: ([\tau, \text{',}:\text{collection}] \text{ VAL }, (\tau, \text{'},:\text{bot}) \text{ VAL }, '\tau \text{ Boolean } ] \\
\Rightarrow (\tau, \text{'},:\text{collection}) \text{ VAL } \\
(\_\_ : \_ | _ ) [71, 100, 70, 50] \)

\( \_\_\_\text{OclReject}_\text{math} :: ([\tau, \text{',}:\text{collection}] \text{ VAL }, (\tau, \text{'},:\text{bot}) \text{ VAL }, '\tau \text{ Boolean } ] \\
\Rightarrow (\tau, \text{'},:\text{collection}) \text{ VAL } \\
(\_\_ : \_ | _ ) [71, 100, 70, 50] \)

\( \_\_\_\text{OclCollect}_\text{math} :: ([\tau, \text{',}:\text{collection}] \text{ VAL }, (\tau, \text{'},:\text{bot}) \text{ VAL }, (\tau, \text{'},:\text{bot}) \text{ VAL } ] \\
\Rightarrow (\tau, \text{'},:\text{collection}) \text{ VAL } \\
(\_\_ : \_ | _ ) [71, 100, 70, 50] \)
B.4. Library

\( (\_ \_ \in \_ \_ \_ \) ) \quad [71,100,70\{50\] 

_\text{OclCollectNested}\_\text{math} :: \[(\tau,\gamma::\text{collection}) \text{ VAL}, (\tau,\alpha::\text{bot}) \text{ VAL}, (\tau,\beta::\text{bot}) \text{ VAL} \\
\Rightarrow (\tau,\delta::\text{collection}) \text{ VAL} \]
\( (\_\_\in\_\_\_\) \quad [71,100,70\{50\] 

_\text{OclIterate}\_\text{math} :: \[(\tau,\beta::\text{collection}) \text{ VAL}, \text{idt, idt, } \gamma, \delta \]
\Rightarrow (\tau,\delta) \text{ VAL} 
\( (\_\rightarrow\text{iterate}(\_\_\_\_\_\) \quad [71,100,70\{50\] 

_\text{OclOne}\_\text{math} :: \[(\tau,\beta::\text{collection}) \text{ VAL, } \alpha, \text{ } \tau \text{ } \text{Boolean} \Rightarrow \tau \text{ } \text{Boolean} \\
(\_\rightarrow\text{one}(\_\_\_\_\_\) \quad [71,100,70\{50\] 

_\text{OclIsUnique}\_\text{ascii} :: \[(\tau,\beta::\text{collection}) \text{ VAL, } \alpha, \text{ } \tau \text{ } \text{Boolean} \Rightarrow \tau \text{ } \text{Boolean} \\
(\_\rightarrow\text{isUnique}(\_\_\_\_\_\) \quad [71,100,70\{50\] 

Semantics: Derived Concepts

In principle, the following concepts could be defined—more or less directly as according
their specification in the standard—conservatively as constant definition, similarly to
an abbreviation. However, we refrained from this possibility due to our theory-morpher
technique, which requires as many as possible operator definitions in combinator for-
mat and derives the resulting properties from there. Consequently, we stated the
requirements following from the standard simply as unproven lemmas and prove them
later.

The following lemmas are unproven—they mirror the requirements in the standard
and are intended as typechecked equivalents. These collection are proven on the basis
of the concrete definitions in the concrete collection class theories.

One might wonder why the Isabelle-concept of an axiomatic class is not used here.
The reason is that the underlying type concept of Isabelle (Order-Sorted Polymorphism
in Isabelle) is too weak to express the connection between the type constructors to be
specified and their content; in the subsequent declaration for example, there is no
connection between \(\beta::\text{collection} \) and text \(\alpha::\text{bot} \) as there should be. These types
in the abstract class are therefore quite rough approximations. As a consequence,
axiomatic classes would easily require very general properties that can not be met by
any concrete instance.

Count

The standard requires the following definition:

\text{lemma } \text{REQ\_11\_7\_1\_4}:
\text{self } \rightarrow \text{count(obj) } \equiv
Appendix B. Isabelle Theories

```isar
def self->iterate(elem;acc := 0 | if elem △ obj then acc + 1 else acc endif)
```

Oops

Unfortunately, for the case of sets, this definition assumes that sets are “duplicate free lists” which is simply not the case in general, in particular if we admit infinite sets as in HOL-OCL.

Instead, we define the properties pointwise for each collection.

Excludes

The standard defines exclusion via `->count`, which is possible, but messy with hindsight to a standard set theory:

```isar
def lemma REQ11_7_1_3: self->excludes(obj) ≡ (self->count(obj) ≠ 0)
```

includesAll

The definition of `->includesAll` strictly follows the standard REQ11_7_1_5:

```isar
def lemma REQ11_7_1_5: (self->includesAll(obj)) ≡ (obj->forall(elem | (self->includes(elem))))
```

Unfortunately, this form of definition is not conservative for type technical reasons: `elem` is too polymorphic and contains a free type variable not occurring in the lhs of this equation. Solution: `->includesAll` must be defined individually on each concrete collection.

With the following three definitions, the situation is the same.

```isar
def lemma REQ11_7_1_6: (self->excludesAll(obj)) ≡ (obj->forall(elem | (self->excludes(elem))))
```

```isar
def lemma REQ11_7_1_7: isEmpty self ≡ (self->size() ≠ 0)
```

```isar
def lemma REQ11_7_1_8: notEmpty X ≡ not(isEmpty(X))
```

end

B.4.10. OCL Sequence

```isar
def theory OCL_Sequence
``` imports $HOLOC_HOME/src/library/collection/OCL_Collection $HOLOC_HOME/src/library/collection/HOL_Collections_ext

begin

250
Theorems needed in this theory but not belonging to it

**Very often used simplifier sets**

**lemma** `DEF_def_both`:

\[ \text{DEF} \equiv (\bot \neq X \land X \neq \bot) \]

**by**(simp add: `DEF_def neq_commute`)

**Lifting**

**lemma** `not_down_lift_drop [simp]`:

\[ \mathbf{x \neq \downarrow} \Rightarrow \llbracket \mathbf{x} \rrbracket = \mathbf{x} \]

**by**(drule `not_down_exists_lift [THEN iffD1]`, auto)

**Logic core**

**lemma** `and_equiv_sand`:

\[ \llbracket \tau \models \partial X; \tau \models \partial Y \rrbracket = \Rightarrow (X \land Y) \tau = (X \lor Y) \tau \]

**by**(simp add: `OclSand_def OclAnd_def localValidDefined2sem ss_lifting`)

**lemma** `ocl_if_totally_undefined [simp]`:

\[ (\text{if } \mathbf{P} \text{ then } \bot \text{ else } \bot \text{ endif}) = \bot \]

**by**(rule ext, simp add: `OclIf_def OclUndefined_def ss_lifting`)

**lemma** `if_distrib_defined`:

\[ \llbracket \mathbf{P} \in \text{cp } (\mathbf{P} \in (\tau, \alpha :: \bot) \text{ VAL}) \rrbracket \Rightarrow \mathbf{P}(\text{if } \mathbf{X} \text{ then } \mathbf{Y} \text{ else } \mathbf{Z} \text{ endif}) = (\text{if } \mathbf{X} \text{ then } \mathbf{P} \mathbf{Y} \text{ else } \mathbf{P} \mathbf{Z} \text{ endif}) \tau \]

**apply**(auto simp: `OclIf_def localValidDefined2sem ss_lifting`)

**apply**(rule_tac `\mathbf{P} = \mathbf{P}` in `cp_charn`, simp_all)+

done

**lemma** `if_distrib_strict_alt`:

\[ \llbracket \mathbf{P} \mathbf{Z} \rrbracket \Rightarrow \mathbf{P}(\text{if } \mathbf{X} \text{ then } \mathbf{Y} \text{ else } \mathbf{Z} \text{ endif}) = (\text{if } \mathbf{X} \text{ then } (\mathbf{P} \mathbf{Y}) \text{ else } (\mathbf{P} \mathbf{Z}) \text{ endif}) \]

**by**(simp add: `if_distrib_strict isStrict_def OclUndefined_def lift0_def`)

**lemma** `if_contract`:

\[ (\text{if } \mathbf{P} \text{ then } (\text{if } \mathbf{Q} \text{ then } \mathbf{X} \text{ else } \mathbf{Y} \text{ endif}) \text{ else } \mathbf{Y} \text{ endif}) = (\text{if } \mathbf{P} \land (\mathbf{P} \rightarrow \mathbf{Q})) \text{ then } \mathbf{X} \text{ else } \mathbf{Y} \text{ endif} \]

**by**(rule ext, simp add: `OclIf_def OclSand_def OclImplies_def o_def OclOr_def OclNot_def OclAnd_def ss_lifting`)

**lemma** `if_not`:

\[ (\text{if } \lnot \mathbf{X} \text{ then } \mathbf{Y} \text{ else } \mathbf{Z} \text{ endif}) = (\text{if } \mathbf{X} \text{ then } \mathbf{Z} \text{ else } \mathbf{Y} \text{ endif}) \]

**by**(rule ext, simp add: `OclIf_def OclNot_def ss_lifting`)

**Some cp theory to simplify the later proofwork**

Adding currently missing cp-ness results to the simpset

**lemmas** `cp_hol_simps [simp,intro!]` = `cp_lift2[simplified lift2_def, of _ _ op →]`

**lemmas** `cp_hol_and [simp,intro!]` = `cp_lift2[simplified lift2_def, of _ _ op ∧]`

251
lemmas cp_hol_or [simp, intro] = cp_lift2 [simplified lift2_def, of _ _ op ∨]
lemmas cp_hol_not [simp, intro] = cp_lift1 [simplified lift1_def, of _ _ λ x. ¬ x]

lemma cp_or [simp, intro]: [cp P; cp P’] ⇒ cp (λ X. (P X) ∨ (P’ X))
  by (simp add: OclOr_def)

declare cp_strongEq [simp, intro]

lemma cp_sor [simp, intro]: [cp P; cp P’] ⇒ cp (λ X. (P X) ∨ (P’ X))
  by (simp add: OclSor_def)

lemma cp_sand [simp, intro]: [cp P; cp P’] ⇒ cp (λ X. (P X) ∧ (P’ X))
  by (simp add: OclSor_def)

lemma cp_sxor [simp, intro]: [cp P; cp P’] ⇒ cp (λ X. (P X) ⊕ (P’ X))
  by (simp add: OclSxor_def)

A very useful substitution rule

lemma cp_subst:
  [cp P] ⇒ P x τ = P (lift0 (x τ)) τ
  by (subst (asm) cp_by_cpify, auto)

Setup of the environment

Special parameters

Properties of Datatype Adaption

The following rules are transformed versions of the automatically generated rules for
datatype adaption

lemma Abs_Sequence_0_inject_charn:
  [⊥ :: ′τ::bot ∉ set x; ⊥ :: ′τ::bot ∉ set y] ⇒ ((Abs_Sequence_0 x) = (Abs_Sequence_0 y)) = (x = y)
  by (subst Abs_Sequence_0_inject,
      simp_all add: OCL_Sequence_type.Sequence_0_def smash_def)

These are derived rules from the above one that allow the simplifier to detect false
assumptions which make a goal trivially true.

lemma Abs_Sequence_0_inject_absurd11 [simp]:
  [⊥ ∉ set x] ⇒ ((Abs_Sequence_0 down) = (Abs_Sequence_0 x)) = False
  by (subst Abs_Sequence_0_inject,
      simp_all add: OCL_Sequence_type.Sequence_0_def smash_def)

lemma Abs_Sequence_0_inject_absurd12 [simp]:
\[ \bot \notin \text{set } x \implies (\text{Abs\_Sequence\_0} (x)) = \bot = \text{False} \]

\textbf{lemma Abs\_Sequence\_0\_inject\_absurd21 [simp]:}
\[ \bot \notin \text{set } x \implies ((\text{Abs\_Sequence\_0} (x)) = \bot) = \text{False} \]
\textbf{by (simp add: UU\_Sequence\_def)}

\textbf{lemma Abs\_Sequence\_0\_inject\_absurd22 [simp]:}
\[ \bot \notin \text{set } x \implies ((\text{Abs\_Sequence\_0} (x)) = \bot) = \text{False} \]
\textbf{by (simp add: UU\_Sequence\_def)}

\textbf{lemma Abs\_Sequence\_0\_inverse\_charn1 [simp]:}
\[ (((\text{Abs\_Sequence\_0} (x)) = x) \implies \text{Rep\_Sequence\_0} (\text{Abs\_Sequence\_0} (x)) = x) \]
\textbf{by (subst Abs\_Sequence\_0\_inverse, simp all add: OCL\_Sequence\_type.Sequence\_0\_def smash\_def)}

\textbf{lemma Abs\_Sequence\_0\_inverse\_charn2 [simp]:}
\[ (((\text{Abs\_Sequence\_0} (x)) = x) \implies \text{Rep\_Sequence\_0} (\text{Abs\_Sequence\_0} (x)) = x) \]
\textbf{by (subst Abs\_Sequence\_0\_inverse, simp all add: OCL\_Sequence\_type.Sequence\_0\_def smash\_def)}

\textbf{lemma Abs\_Sequence\_0\_cases\_charn:}
\textbf{assumes bottomCase:} \[ (\bot \notin \text{set } x) \implies P \]
\textbf{assumes listCase :} \[ \forall y. (\{ x = \text{Abs\_Sequence\_0} (y); \bot \notin \text{set } y \} \implies P) \]
\textbf{shows}
\textbf{apply (rule_tac x = x in Abs\_Sequence\_0\_cases)}
\textbf{apply (case_tac y = \bot)}
\textbf{apply (rule_tac_tac y = \bot in listCase)}
\textbf{apply (auto simp: OCL\_Sequence\_type.Sequence\_0\_def smash\_def)}
\textbf{done}

\textbf{lemma Abs\_Sequence\_0\_induct\_charn:}
\textbf{assumes bottomCase :} \[ P \bot \]
\textbf{assumes stepCase :} \[ \forall y. (\{ \bot \notin \text{set } y \} \implies P (\text{Abs\_Sequence\_0} (y))) \]
\textbf{shows}
\textbf{apply (rule_tac x = x in Abs\_Sequence\_0\_induct)}
\textbf{apply (rule_tac_tac x = \bot in Abs\_Sequence\_0\_induct)}
\textbf{apply (auto intro!: bottomCase elim: stepCase)}
\textbf{done}

These rules stem from the set theory, that was the first collection theory. It should actually be decided which of them are really needed anymore because the above rules cover most cases.

\textbf{lemma inj\_on\_Abs\_Sequence\_0\_Sequence\_0\_Sequence\_0:}
\textbf{by (rule inj\_on\_inverse1, rule Sequence\_0.Abs\_Sequence\_0\_inverse, assumption)}
Appendix B. Isabelle Theories

lemma inj_Rep_Sequence : inj Rep_Sequence_0
  by (rule inj_on_inverseI, rule Sequence_0.Rep_Sequence_0_inverse)

lemma smashed_sequence_charn:
  \(|\bot| X) = ((X) \in Sequence_0)
  by (unfold smash_def Sequence_0_def UU_Sequence_def,auto)

lemma UU_in_smashed_sequence [simp]:
  \(\bot \in Sequence_0)
  by (unfold smash_def Sequence_0_def UU_Sequence_def,auto)

lemma down_in_smashed_sequence [simp]:
  down \in Sequence_0
  by (unfold smash_def Sequence_0_def UU_Sequence_def,auto)

lemma nt_in_smashed_sequence [simp]:
  \((\mid) \in Sequence_0
  by (unfold smash_def Sequence_0_def,auto)

lemma DEF_Abs_Sequence:
  \(\forall X. (\bot \notin set X) \implies DEF (Abs_sequence_0 (X))
  apply (unfold DEF_def UU_Sequence_def, simp)
  done

lemma DEF_Rep_Sequence:
  \(\forall X. DEF X \implies DEF (Rep_sequence_0 X)
  apply (unfold DEF_def UU_Sequence_def,auto)
  apply (drule_tac f = Abs_sequence_0 in arg_cong)
  apply (simp add: Rep_sequence_0_inverse)
  done

lemma not_DEF_Rep_Sequence:
  \(\forall X. \neg DEF X \implies \neg DEF (Rep_sequence_0 X)
  apply (unfold DEF_def UU_Sequence_def,auto)
  apply (simp add: Abs_sequence_0_inverse)
  done

lemma exists_lift_Sequence:
  \(\forall X. DEF X \implies \exists c. Rep_sequence_0 X = (c)
  by (drule DEF_Rep_Sequence, simp add: DEF_X_up)

lemma exists_lift_Sequence2:
  \(\forall X. DEF X \implies \exists c. (\bot \notin set c \wedge Rep_sequence_0 X = (c)
  apply (frule exists_lift_Sequence)
  apply (auto)
  apply (drule sym, rule swap)
  prefer 2 apply assumption
  apply (subst smashed_sequence_charn)
  apply (simp add: Rep_sequence_0)
  254
B.4. Library

done

lemma Rep_Sequence_cases:
\[ \text{Rep\_Sequence\_0}\ X = \bot \lor (\exists c. \bot \not\in \text{set} c \land \text{Rep\_Sequence\_0}\ X = (\langle c \rangle)) \]
apply(case_tac DEF X)
apply(drule exists_lift_Sequence2)
apply(drule_tac \[2\] not_DEF_Rep_Sequence)
apply(simp_all add: DEF_def)
done

These two DEF_X_Sequence lemmas are a central part in almost every proof that unfolds the OCL definitions. Furthermore they will also be used by the thy_morpher to automatically lift theorems about HOL lists to theorems about OCL sequences.

lemma DEF_X_Sequence_0 : DEF(X) = (\exists c. \bot \not\in \text{set} c \land \text{Rep\_Sequence\_0}\ X = (\langle c \rangle))
apply(insert Rep_Sequence_cases[of X], auto)
apply(drule exists_lift_Sequence,auto)
apply(rule swap)
prefer 2
apply(rule not_DEF_Rep_Sequence,auto)
done

Because in most cases both versions of the above theorems are used they are packed conveniently in this lemma.

lemma DEF_X_Sequence':
\[ [X \neq \bot] \implies (\exists c. (\bot \not\in \text{set} c) \land (\text{Rep\_Sequence\_0}\ X = (\langle c \rangle)) \land (\exists c. (\bot \not\in \text{set} c) \land (X = (\text{Abs\_Sequence\_0}\ (\langle c \rangle)))) \]
apply(fold DEF_def)
apply(drule DEF_X_Sequence[THEN iffD1])
apply(drule DEF_X_Sequence_0[THEN iffD1])
apply(simp)
done

And to enable a fast case splitting over (un)definedness of a specific sequence together with its semantic representation these two theorems are very useful. Note that there exists a _0 version which does not require cp-ness. This _0 version exists actually for almost all more complex theorems because not in all cases one has or can show cp-ness.

lemma Sequence_sem_cases_0:
Appendix B. Isabelle Theories

assumes defC: \( \forall c \ d \ . \ X \neq \bot; \bot \neq X \)
(\( \bot \notin \text{set} \ c \); \( \bot \notin \text{set} \ d \))
(\( \text{Rep}_\text{Sequence}_0 \ X = \bot \))
(\( X = \text{Abs}_\text{Sequence}_0 \ d \)) \implies P \ X

and undefC: \[ [ X = \bot ] \implies P X \]

shows P X

apply (rule Abs_Sequence_0_cases_charn, erule undefC)
apply (rule defC, auto)
done

lemma Sequence_sem_cases:
assumes defC: \( \forall c \ d \ . \ [ (X \tau) \neq \bot; \bot \neq (X \tau) \]
(\( \bot \notin \text{set} \ c \); \( \bot \notin \text{set} \ d \))
(\( \text{Rep}_\text{Sequence}_0 \ (X \tau) = \bot \))
(\( (X \tau) = \text{Abs}_\text{Sequence}_0 \ d \)) \implies P X \tau

and undefC: \[ [ (X \tau) = \bot ] \implies P X \tau \]

and cpP: \( \text{cp} \ P \)

shows P X \tau

apply (rule_tac P2=P in subst[OF sym[OF cp_psubst]], rule cpP)
apply (rule Sequence_sem_cases_0)
apply (rule defC, prefer 7)
apply (rule undefC, simp_all)
done

And a version for the case that we’re reasoning over equalities of functions.

lemma Sequence_sem_cases_ext:
assumes defC: \( \forall c \ d \ . \ [ (X \tau) \neq \bot; \bot \neq (X \tau) \]
(\( \bot \notin \text{set} \ c \); \( \bot \notin \text{set} \ d \))
(\( \text{Rep}_\text{Sequence}_0 \ (X \tau) = \bot \))
(\( (X \tau) = \text{Abs}_\text{Sequence}_0 \ d \)) \implies P X \tau = Q X \tau

and undefC: \( \forall \tau \ . \ [ (X \tau) = \bot ] \implies P X \tau = Q X \tau \)

and cpP: \( \text{cp} \ P \)

and cpQ: \( \text{cp} \ Q \)

shows P X = Q X

apply (rule ext)
apply (rule_tac X=X in Sequence_sem_cases)
apply (rule defC, prefer 7)
apply (rule undefC)
apply (simp_all add: cpP cpQ)
done

These four lemmas are used by ocl_setup_op to reason about definedness:

lemma lift2_strict_is_isdef_fw_Sequence_Val:
assumes f_def: \( f = \text{lift2}(\text{strictify}(\lambda x. \text{strictify}(\lambda y. \text{Abs}_\text{Sequence}_0 \ g \ \text{Rep}_\text{Sequence}_0 x \ y))) \)

and inv_g: \( \forall a b. \ [ [ \bot \notin \text{set} \ a; (\bot::\tau::\bot) \neq b ] \implies \bot \notin \text{set} \ (g a b) ] \)

shows \( \partial(f X Y) = (\partial X \land \partial Y) \)
B.4. Library

apply (rule ext)
apply (simp add: f_def OclIsDefined_def OclAnd_def o_def ss_lifting)
apply (rule_tac X = X x in Sequence_sem_cases_0)
apply (rule_tac X = Y x in inv_g)
apply (simp_all add: neq_commute)
done

lemma lift2_strict_is_isdef_fu_Sequence_Sequence:
  assumes f_def: f ≡ lift2(strictify(λx. strictify(λy. Abs_Sequence_0⌜g⌜Rep_Sequence_0 x⌝⌜Rep_Sequence_0 y⌝⌟)))
  and inv_g: ∀a b. (⊥ ∉ set a; ⊥ ∉ set b) ⇒ ⊥ ∉ set (g a b)
  shows ∂(f X Y) = (∂ X ∧ ∂ Y)
apply (rule ext)
apply (simp add: OclIsDefined_def OclAnd_def o_def ss_lifting)
apply (rule_tac X = X x in Sequence_sem_cases_0)
apply (rule_tac X = Y x in Sequence_sem_cases_0)
apply (rule_tac X = Y x and Y = Y in lift2_strict_is_isdef_fu_Sequence_Sequence)
apply (rule inv_g, assumption+)
apply (simp add: OclAnd_def OclTrue_def o_def OclIsDefined_def lift0_def lift1_def lift2_def OclLocalValid_def)
done

lemma lift2_strictify_implies_LocalValid_defined_Sequence_Val:
  assumes f_def: f ≡ lift2(strictify(λx. strictify(λy. Abs_Sequence_0⌜g⌜Rep_Sequence_0 x⌝⌜Rep_Sequence_0 y⌝⌟)))
  and inv_g: ∀a b. (⊥ ∉ set a; ⊥ ∉ set b) ⇒ ⊥ ∉ set (g a b)
  shows (τ ⊨ (∂ (f X Y)) = (∂ X ∧ ∂ Y))
apply (rule ext)
apply (simp add: OclAnd_def OclTrue_def o_def OclIsDefined_def lift0_def lift1_def lift2_def OclLocalValid_def)
done

lemma lift2_strictify_implies_LocalValid_defined_Sequence_Sequence:
  assumes f_def: f ≡ lift2(strictify(λx. strictify(λy. Abs_Sequence_0⌜g⌜Rep_Sequence_0 x⌝⌜Rep_Sequence_0 y⌝⌟)))
  and inv_g: ∀a b. (⊥ ∉ set a; ⊥ ∉ set b) ⇒ ⊥ ∉ set (g a b)
  shows (τ ⊨ (∂ (f X Y)) = (∂ X ∧ ∂ Y))
apply (rule ext)
apply (simp add: OclAnd_def OclTrue_def o_def OclIsDefined_def lift0_def lift1_def lift2_def OclLocalValid_def)
done

257
Appendix B. Isabelle Theories

Building a canonic representation of sequences

The empty sequence
constdefs
  OclMtSequence :: (′τ, ′α::bot Sequence_0) VAL
  OclMtSequence ≡ lift0(Abs_Sequence_0 ⌞⌜⌟⌟)

syntax
  _OclMtSequence_std :: (′τ, ′α::bot) VAL ⇒ ′τ Boolean (⌜⌜⌟⌟)
syntax
  _OclMtSequence_ascii :: (′τ, ′α::bot) VAL ⇒ ′τ Boolean (mtSequence)
syntax (xsymbols)
  _OclMtSequence_math :: (′τ, ′α::bot) VAL ⇒ ′τ Boolean (⌜⌜⌟⌟)

lemma OCL_is_isdef_OclMtSequence [simp]:
  ⊨ ∅ ⌞⌜⌟⌟
by (simp add: OclValid_def OclIsDefined_def OclTrue_def
  OclMtSequence_def ss_lifting)

lemma OCL_is_defopt_OclMtSequence [simp]:
  τ ⊨ ∅ ⌞⌜⌟⌟
by (simp add: valid_elim)

And the ‘cons’ operation: including
defs
  OclIncluding_def :
  OclIncluding ≡ lift2(strictify(λS. strictify (λe. Abs_Sequence_0
  (concat ["Rep_Sequence_0 S", [e] ]))))

ocl_setup_op [OclIncluding]

lemma OCL_is_def_OclIncluding:
  (τ ⊨ OclIncluding (X::(′τ, ′α::bot)Sequence) (Y::(′τ, ′α::bot)VAL)) = (τ ⊨ X ∧ τ ⊨ Y)
by (rule lift2_strict_is_isdef_fw_Sequence_Val[OF OclIncluding_def, simp])

lemma OCL_is_defopt_OclIncluding:
  (τ ⊨ OclIncluding (X::(′τ, ′α::bot)Sequence) (Y::(′τ, ′α::bot)VAL)) = (τ ⊨ X ∧ τ ⊨ Y)
by (rule lift2_strictify_implies_LocalValid_defined_Sequence_Val[OF OclIncluding_def, simp])

The syntax translation mkSequence builds now our sequences

syntax
  @OclFinSequence :: args ⇒ (′τ, ′α Sequence_0 ) VAL
  (mkSequence([_]))

translations
  mkSequence[x, xs] == OclIncluding (mkSequence[xs]) x
  mkSequence[x] == OclIncluding OclMtSequence x
B.4. Library

**Lemma** test : \( \text{mkSequence}[1,2,3,4,5] = ?X \text{ oops} \)

**Relation of mtSequence and including**

These two theorems should actually suffice to decide inequality of sequences

**Lemma** including_notstrongeq_mtSequence [simp]:
\( \neg ((X::(\tau',\alpha::\text{bot})\text{Sequence}) \Rightarrow \text{including} (a::(\tau',\alpha)\text{VAL}) \triangleq \[])) = T \)

apply(rule_tac X=X in Sequence_sem_cases_ext)
apply(simp_all add: cp_strongEq)
apply(simp_all add: OclIncluding_def OclMtSequence_def OclStrongEq_def OclTrue_def OclNot_def localValid2sem ss_lifting)
done

**Lemma** including_notstricteq_mtSequence [simp]:
\( [\tau \supset \partial (a::(\tau',\alpha)\text{VAL}); \tau \supset \partial (X::(\tau',\alpha::\text{bot})\text{Sequence})] = \Rightarrow \tau \supset (X \Rightarrow \text{including} a) \leftrightarrow \[] \)

apply(rule_tac X=X in Sequence_sem_cases)
apply(simp_all add: cp_strictEq)
apply(simp_all add: OclIncluding_def OclMtSequence_def OclStrictEq_def OclTrue_def OclNot_def localValid2sem ss_lifting)
done

**Case exhaustion over the canonic representation**

**Lemma** Sequence_cases_including_0:

assumes undefCase : \( [X::(\tau')::\text{bot Sequence}_0] = \bot] \Rightarrow P \)
and mtCase : \( [X = (\[] \tau)] \Rightarrow P \)
and stepCase : \( \forall (Y::(\tau')::\text{bot Sequence}_0) (a::(\tau')::\text{bot}). \]
\( [\tau = (\text{DEF Y; DEF a; X} = ((\text{lift0 Y}) \Rightarrow \text{including} (\text{lift0 a})) \tau] \Rightarrow P \)

shows \( P \)

apply(rule Abs_Sequence_0_cases_charn, erule undefCase)
apply(rule_tac xs=y in rev_cases)
apply(rule mtCase)
apply(simp add: OclMtSequence_def ss_lifting')
apply(rule_tac Y=Abs_Sequence_0 ys and a=ya in stepCase)
apply(simp_all add: OclIncluding_def ss_lifting' neq_commute)
done

**Lemma** Sequence_cases_including:

assumes undefCase : \( [\tau = (X::(\tau',\alpha::\text{bot})\text{Sequence}) \triangleq \bot] \Rightarrow P \tau \)
and mtCase : \( [\tau = X \triangleq \[]] \Rightarrow \Rightarrow P \tau \)
and stepCase : \( \forall (Y::(\tau',\alpha::\text{bot})\text{Sequence}) (a::(\tau',\alpha::\text{bot})\text{VAL}). \]
\( [\tau = X \triangleq (Y \Rightarrow \text{including} a); \tau = \partial Y; \tau = \partial a] \Rightarrow P \tau \)

259
Appendix B. Isabelle Theories

shows
  \( P \tau \)
apply (rule_tac X=X \( \tau \) and \( \tau = \tau \) in Sequence_cases_including_0)
apply (rule undefCase) prefer 2
apply (rule mtCase) prefer 3
apply (rule stepCase, simp_all add: localValidDefined2sem)
apply (simp_all add: localValid2sem OclStrongEq_def OclUndefined_def ss_lifting)
done

Induction over the canonic representation

To simplify reasoning there exists a \( _0 \) version which doesn’t pose any restrictions on the involved functions with regard to context-passingness.

lemma Sequence_induct_including_0:
assumes undefCase: \( P \perp \tau \)
and mtCase: \( P ([\tau]) \)
and stepCase: \( \forall (X::'\tau::bot) \, (a::'\tau::bot).
  \[ P X; DEF X; DEF a \Rightarrow (P ((OclIncluding (lift0 X) (lift0 a)) \tau)) \]
shows
  \( P ([X::'\tau::bot]) \)
apply (rule Abs_Sequence_0_induct_charn, rule undefCase)
apply (erule rev_mp)
apply (rule rev_induct)
apply (insert mtCase, simp add: OclMtSequence_def ss_lifting)
apply (auto)
apply (rotate_tac -1, drule_tac a=x and X=Abs_Sequence_0 xs in stepCase)
apply (erule DEF_Abs_Sequence)
apply (simp_all add: OclIncluding_def neq_commute ss_lifting)
done

lemma Sequence_induct_including:
assumes undefCase : \( P \perp \tau \)
and mtCase: \( P [\tau] \)
and stepCase: \( \forall (X::('\tau,'\alpha::bot))\, (a::('\tau,'\alpha::bot))\, \text{VAL}.
  \[ P X \tau; \tau \models X \tau; \tau \models a \Rightarrow (P (OclIncluding X a) \tau) \]
and cpP: \( \text{cp} \)
shows
  \( P ([X::('\tau,'\alpha::bot)]) \)
apply (insert cpP)
apply (frule \( \text{cp} \) by \( \text{cp} \) [THEN iffD1])
apply (rule subst, rule sym)
apply (rule_tac tac=x=X in cp_subst, simp)
apply (rule_tac \( \tau = \tau \) in Sequence_induct_including_0)
apply (insert undefCase)
apply (auto simp: OclUndefined_def intro!: mtCase)
apply (drule stepCase)
apply (simp_all add: localValidDefined2sem lift0_def)
done
A second sequence constructor on the OCL level

defs

OclCollectionRange_def : OclCollectionRange a b = lift2 (strictify (λ x:Integer. 0. strictify (λ y. Abs_Sequence_0.map (λ z::nat. ((int z) + (b y)).) [0..<(nat ("y" - x + 1))]).)))

oclsetup_op [OclCollectionRange]

lemma OCL_is_defopt_OclCollectionRange [simp]:
\[ τ ⊨ b \ (OclCollectionRange (a::"\tau" Integer)) b::("τ, Integer_0)Sequence) = ((τ ⊨ (\emptyset a) \land (τ ⊨ \emptyset b)) \quad apply\ (subgoal_tac \not \in set \ map \ ((\lambda x. ((\int z) + (\tau a))).) [0..<(\int ((\tau b) - (\tau a)) + 1))]).)\]
apply (auto simp: localValidDefined2sem OclCollectionRange_def ss_lifting)
done

lemma collectionRangeMtSequence_conv:
\[ τ ⊨ (b::"\tau" Integer) < (a::"\tau" Integer) \implies OclCollectionRange a b τ = ()::("\tau, Integer_0)Sequence) τ \quad apply\ (case_tac τ ⊨ \emptyset a, ocl_hypsubst, simp) \quad apply\ (case_tac τ ⊨ \emptyset b, ocl_hypsubst, simp) \quad apply\ (simp\ add: localValidDefined2sem DEF_def_both, clarify) \quad apply\ (simp_all\ add: DEF_def OclCollectionRange_def OclMtSequence_def OclLess_def localValid2sem ss_lifting) \quad done \]

lemma collectionRangeSingleton_UC [simp]:
OclCollectionRange a a = ()::("\tau, Integer_0)Sequence \implies including (a::"\tau, Integer_0 VAL) \quad apply\ (rule\ ext, case_tac x ⊨ \emptyset (a), ocl_hypsubst, simp) \quad apply\ (simp\ add: OclCollectionRange_def OclMtSequence_def OclIncluding_def localValidDefined2sem ss_lifting) \quad done \]

lemma collectionRangeExpandIncluding:
\[ τ ⊨ (a::"\tau" Integer) ≤ b \implies ((OclCollectionRange a b)::("\tau, Integer_0)Sequence) τ = (OclCollectionRange a (b - 1)) \implies including b τ \quad apply\ (subgoal_tac \not \in set \ map \ ((\lambda x. ((\int z) + (\tau a))).) [0..<(\int ((\tau b) - (\tau a)) + 1))]).) \quad apply\ (subgoal_tac \not \in set \ map \ ((\lambda x. ((\int z) + (\tau a))).) [0..<(\int ((\tau b) - (\tau a)))).]) \quad apply\ (auto\ simp: OclCollectionRange_def OclIncluding_def OclLe_def minus_def One_ocl_int_def localValid2sem ss_lifting) \quad apply\ (subst\ Abs_Sequence_0.inject_charn, auto) \quad apply\ (subgoal_tac \int ((\tau b) - (\tau a)) + 1)) = (\int ((\tau b) - (\tau a)) + (\int 1)) \quad done \]

261
Appendix B. Isabelle Theories

apply(simp, arith)
done

The conversion operators

Sequence to sequence
defs

\[ \text{OclAsSequence} \triangleq \text{id} \]

lemma asSequence_identity[simp]:
\text{OclAsSequence} (S::('r,'a::bot)Sequence) = S
by(simp add: OclAsSequence_def)

Sequence to bag
defs

\[ \text{OclAsBag} \triangleq \text{lift1} \ (\text{strictify} \ (\lambda X. \text{Abs_Bag_0} \ (\text{multiset_of} \ (\text{rep_Sequence_0} X))) \)) \]

ocl_setup_op [OclAsBag]

Sequence to ordered set
defs

\[ \text{OclAsOrderedSet} \triangleq \text{lift1} \ (\text{strictify} \ (\lambda X. \text{Abs_OrderedSet_0} \ (\text{rev} \ (\text{remdups} \ (\text{rev} \text{rep_Sequence_0} X)))) \)) \]

ocl_setup_op [OclAsOrderedSet]

Sequence to set
defs

\[ \text{OclAsSet} \triangleq \text{lift1} \ (\text{strictify} \ (\lambda X. \text{Abs_Set_0} \ (\text{set} \text{rep_Sequence_0} X)))) \]

ocl_setup_op [OclAsSet]

OclIterate
defs

\[ \text{OclIterate} \triangleq (\text{lift3'} \ (\text{lift_arg0} \ (\text{lift_arg2} \ (\text{lift_arg0})) \ (\text{strictify} \ (\lambda S P A. \ (\text{foldl} \ (\lambda x y. P \ y x) A \ (\text{Rep_Sequence_0} S)))))) \]
ocl_setup_op [OclIterate]

The characteristic behaviour of iterate on the canonical representation

lemma iterate_of_mtSequence[simp]:
(OclIterate ([]:('τ, 'α::bot)Sequence)
  (P::(('τ ⇒ 'α) ⇒ ('τ ⇒ 'c::bot) ⇒ ('τ ⇒ 'c))))
  A = A
apply(rule ext)
apply(subgoal_tac x ⊨ (∂ [])
apply(simp_all add: OclIterate_def OclIncluding_def OclMtSequence_def
  localValidDefined2sem ss_lifting)
done

lemma iterate_of_including_0:
assumes defS: τ ⊨ ∂(S::('τ,('α::bot))Sequence)
and defx: τ ⊨ ∂(x::('τ ⇒ 'α::bot))
shows ((OclIterate (OclIncluding S x) P A) τ) =
  (P (lift0 (x τ)) (lift0 ((OclIterate (S::(('τ, 'α::bot)Sequence)) P A) τ)) τ)
apply(insert defS defx)
apply(frule_tac Y1=x in conjI[THEN OCL_is_defopt_OclIncluding[THEN iffD2]], assumption)
apply(simp add: localValidDefined2sem DEF_def)
apply(frule DEF_X_Sequence', clarify)
apply(simp add: OclIterate_def OclIncluding_def ss_lifting')
apply(frule DEF_X_Sequence', clarify, simp)
apply(subst (asm) Abs_Sequence_0_inject_charn, auto)
done

lemma iterate_of_including:
assumes defS: τ ⊨ ∂(S::('τ,('α::bot))Sequence)
and defx: τ ⊨ ∂(x::('τ ⇒ 'α::bot))
and cpP_1: ∀y. cp (λ x. P x y)
and cpP_2: ∀x. cp (P x)
shows ((OclIterate (OclIncluding S x) P A) τ) =
  (P x (OclIterate (S::(('τ, 'α::bot)Sequence)) P A) τ)
apply(insert defS defx cpP_1 cpP_2)
apply(drule_tac x=x in iterate_of_including_0 and P=P and A=A in iterate_of_including)
apply(simp_all add: cp_by_cpfy)
done

lemma iterate_opt_of_including:
assumes P_undef_1: !!y. P ⊥ y = ⊥
and P_undef_2: !!x. P x ⊥ = ⊥
and cpP_1: ∀y. cp (λ x. P x y)
and cpP_2: ∀x. cp ((P::('τ, 'α::bot) VAL ⇒ ('τ,'c::bot) VAL ⇒ ('τ,'c::bot) VAL) x)
shows ((OclIterate (OclIncluding S x) P A) =
  (P x (OclIterate (S::(('τ, 'α::bot)Sequence)) P A)))
Appendix B. Isabelle Theories

```isabelle
apply(insert P_undef_1 P_undef_2 cpP_1 cpP_2)
apply(rule ext)
apply(case_tac xa \vdash S)
apply(case_tac xa \vdash x)
apply(rule iterate_of_including, simp_all)
apply(frule isUndefined_charn_local[THEN iffD2])
apply(rule syn[OF trans], erule_tac A=x in cp_charn, simp, simp)
apply(frule isUndefined_charn_local[THEN iffD2])
apply(rule trans, erule_tac A=S in cp_charn, simp, simp)
apply(rule syn[OF trans], erule_tac A=S in cp_charn)
apply(rule_tac P=P x in cp-compose2, simp_all)
done
```

The universal and fusion property

Fusion and universal theorems, as described in [25].

**Lemma** iterate_universal_0:
assumes g_cp: cp g
and g_undef: g (\bot.Persistence(\tau,\alpha::bot)Sequence) \tau = (\bot.Persistence(\tau,\alpha::bot)VAL) \tau
and g_mtSeq: g (OclMtSequence.:\tau,\alpha::bot)Sequence) \tau = A \tau
and g_incl: \forall S x. [ \tau \vdash \partial S; \tau \vdash \partial x ] \implies g (OclIncluding S x) \tau = P (lift0 (x \tau)) (lift0 (g S \tau)) \tau
shows g S \tau = OclIterate S (P::\tau,\alpha::bot)VAL \Rightarrow (\tau,\alpha::bot)VAL \Rightarrow (\tau,\alpha::bot)VAL) A \tau
apply(rule_tac X=S in Sequence_induct_including)
apply(simp_all add: g_cp g_undef g_mtSeq)
apply(subst iterate_of_including_0, simp_all add: g_incl)
done

**Lemma** iterate_universal:
assumes g_cp: cp g
and cpP_1: \A y. cp (\lambda x. P x y)
and cpP_2: \A x. cp (P x)
and g_undef: g \bot Persistence = \bot Persistence VAL \tau
and g_mtSeq: g (OclMtSequence.:\tau,\alpha::bot)Sequence) \tau = A \tau
and g_incl: \forall S x. [ \tau \vdash \partial S; \tau \vdash \partial x ] \implies g (OclIncluding S x) \tau = P (x \tau) (g S \tau)
shows g S \tau = OclIterate S (P::\tau,\alpha::bot)VAL \Rightarrow (\tau,\alpha::bot)VAL \Rightarrow (\tau,\alpha::bot)VAL) A \tau
by(insert cpP_1 cpP_2 g_cp, rule iterate_universal_0, simp_all add: g_undef g_mtSeq g_incl cp_by_cpfy)

**Lemma** iterate_universal_0_opt:
assumes g_cp: cp g
and g_mtSeq: g (OclMtSequence.:\tau,\alpha::bot)Sequence) \tau = A \tau
and g_incl: \forall S x. [ \tau \vdash \partial S; \tau \vdash \partial x ] \implies g (OclIncluding S x) \tau = P (lift0 (x \tau)) (lift0 (g S \tau)) \tau
and defS: \tau \vdash \partial S
```

264
shows $g \mathcal{S} \tau = \text{OclIterate } S (P A) \Rightarrow (\tau, c::\text{bot}) \Rightarrow (\tau, c::\text{bot}) \Rightarrow (\tau, c::\text{bot}) \Rightarrow A \tau$

apply(insert defS, rule_tac X=S in Sequence_induct_including)
apply(simp_all add: g_cp g_mtSeq)
apply(subst iterate_of_including_0, simp_all add: g_incl)
done

lemma iterate_universal_opt:
assumes $g \mathcal{S} \tau = \text{OclIterate } S (P A) \Rightarrow (\tau, c::\text{bot}) \Rightarrow (\tau, c::\text{bot}) \Rightarrow (\tau, c::\text{bot}) \Rightarrow A \tau$
and $\text{cpP} \_1 \\lambda \ y \ \text{cp} \ (\lambda \ x \ P \ x \ y)$
and $\text{cpP} \_2 \\lambda \ x \ \text{cp} \ (P \ x)$
and $\text{g} \ _{\text{incl}} \\wedge \ S \ x \ (\tau, \partial \ S; \tau, \partial \ x) \implies g \ (\text{OclIncluding } S x) \Rightarrow P \ x \ (g \ S) \tau$
and $\text{defS: } \tau, \partial \ S$

shows $g \mathcal{S} \tau = \text{OclIterate } S (P A) \Rightarrow (\tau, c::\text{bot}) \Rightarrow (\tau, c::\text{bot}) \Rightarrow (\tau, c::\text{bot}) \Rightarrow A \tau$
by(insert $\text{cpP} \_1 \text{cpP} \_2 \ g \ _{\text{cp}}, \text{rule iterate_universal_0_opt},$
  simp_all add: g_mtSeq g_incl cp_by_cpify)

lemma iterate_fusion_0_opt:

\[
\begin{proof}
  \begin{align*}
    h \ (\text{OclIterate } E A) & = A \tau; \\
    (Q = (\text{lift0 } (x \tau) \ (\text{lift0 } (y \tau))) & = (P \ (\text{lift0 } (x \tau))) \ (\text{lift0 } (h \ y \tau))) \tau = \\
    h \ (\text{OclIterate } E A) & = A \tau;
  \end{align*}
\end{proof}
\]

apply(rule_tac P = \text{cp-compose2})
apply(simp_all)
apply(rule_tac P = \text{cp-compose2})
apply(simp_all)
apply(subst iterate_of_including_0)
apply(simp_all add: \text{cp-compose2})

lemma iterate_fusion_opt:

\[
\begin{proof}
  \begin{align*}
    h \ (\text{OclIterate } E A) & = A \tau; \\
    (Q = (\text{lift0 } (x \tau) \ (\text{lift0 } (y \tau))) & = (P \ (\text{lift0 } (x \tau))) \ (\text{lift0 } (h \ y \tau))) \tau = \\
    h \ (\text{OclIterate } E A) & = A \tau;
  \end{align*}
\end{proof}
\]
Appendix B. Isabelle Theories

apply\( (\text{rule iterate\_fusion\_0\_opt}) \)
apply\( (\text{assumption}) + \)
apply\( (\text{simp add: localValidDefined2sem}) \)
apply\( (\text{assumption}) \)
apply\( (\text{subgoal\_tac h y } \tau = \text{h (lift0 (y } \tau)) \tau) \)
apply\( (\text{simp}) \)
apply\( (\text{rule trans}) \)
apply\( (\text{rule tac } P = \lambda x. \ P \ ?X \ x \ \text{in cp\_subst}) \)
apply\( (\text{simp\_all add: lift0\_def cp\_by\_cpify}) \)
done

lemma iterate\_fusion\_0:
[\text{cp h}:
 h (1::(\tau,c::bot) VAL) \tau = (1::(\tau,d::bot) VAL) \tau;
 h B \tau = A \tau;
 \forall x y. h (Q (\text{lift0 (x } \tau)) (\text{lift0 (y } \tau))) \tau =
 P (\text{lift0 (x } \tau)) (\text{lift0 (h y} \tau)) \tau]
\implies
 h (\text{OclIterate (S::(\tau,\alpha::bot)Sequence})
 (Q::(\tau,c::bot) VAL \Rightarrow (\tau,d::bot) VAL \Rightarrow (\tau,c::bot) VAL) B } \tau
 = \text{OclIterate S (P::(\tau,\alpha::bot) VAL \Rightarrow (\tau,d::bot) VAL \Rightarrow (\tau,d::bot) VAL) A } \tau
\text{apply (rule iterate\_universal\_0)}
apply\( (\text{rule tac } P = h \text{ in cp\_compose2}) \)
apply\( (\text{simp\_all}) \)
apply\( (\text{rule trans, rule tac } P = h \text{ in cp\_subst, simp}) \)
apply\( (\text{subgoal\_tac iterate\_of\_including\_0}) \)
apply\( (\text{simp\_all add: cp\_by\_cpify}) \)
done

lemma iterate\_fusion:
[\text{cp h}:
 \forall x. \text{cp (P x)};
 h (1::(\tau,c::bot) VAL) \tau = (1::(\tau,d::bot) VAL) \tau;
 h B \tau = A \tau;
 \forall x y. h (Q x y) \tau = P x (h y) \tau]
\implies
 h (\text{OclIterate (S::(\tau,\alpha::bot)Sequence})
 (Q::(\tau,c::bot) VAL \Rightarrow (\tau,d::bot) VAL \Rightarrow (\tau,c::bot) VAL) B } \tau
= \text{OclIterate S (P::(\tau,\alpha::bot) VAL \Rightarrow (\tau,d::bot) VAL \Rightarrow (\tau,d::bot) VAL) A } \tau
\text{apply (rule iterate\_fusion\_0)}
apply\( (\text{assumption}) + \)
apply\( (\text{simp}) \)
apply\( (\text{rule trans}) \)
apply\( (\text{rule tac } P = \lambda x. \ P ?X \ x \ \text{in cp\_subst}) \)
apply\( (\text{simp\_all add: lift0\_def cp\_by\_cpify}) \)
done

266
Further properties of iterate

Undeﬁnedness the second part

lemma OCL_undef_2_OclIterate [simp]:
assumes P_undef_2: \( \forall x. P x \perp = \perp \)
shows OclIterate \((S::(\tau,\alpha::\text{bot})\text{Sequence})\)
\((P::(\tau,\alpha::\text{bot})\text{VAL} \Rightarrow (\tau,\text{c::bot})\text{VAL})\)
\(= (\perp::(\tau,\text{c::bot})\text{VAL})\)
apply(insert P_undef_2, rule ext, rule sym)
apply(simp_all add: lift0_def OclUndefined_def)
done

lemma OCL_is_defopt_OclIterate:
assumes defopt_P: \( \forall x y. (\tau \satisfies (\partial (P x y))) \)
and cpP_1: \( \forall y. \text{cp}((\lambda x. P x y)) \)
and cpP_2: \( \forall x. \text{cp}(P::(\tau,\alpha::\text{bot})\text{VAL} \Rightarrow (\tau,\text{c::bot})\text{VAL} \Rightarrow (\tau,\text{c::bot})\text{VAL}) x) \)
shows \( \tau \satisfies (\partial(OclIterate (S::(\tau,\alpha::\text{bot})\text{Sequence}) \text{P A}) \)
= ((\tau \satisfies (\partial S)) \land (\tau \satisfies (\partial \text{A}))) \)
apply(insert defopt_P cpP_1 cpP_2)
apply(rule_tac X=S and \(=\) in Sequence_induct_including, simp_all)
apply(rule trans)
apply(rule_tac x=(X \rightarrow including a \rightarrow iterate(u;ua=A | (P u ua))) in cp_subst, simp)
apply(subst iterate_of_including)
apply(simp_all)
apply(subst OCL_is_defopt_OclIncluding)
apply(rule trans, rule sym)
apply(rule_tac x=(P a (OclIterate X P A)) in cp_subst, simp_all)
done

Forward rules to transfer properties about iterate to functions deﬁnable by iterate

For unary functions

lemma f1_by_iterate_cp:
assumes f_by_it: \( \forall S. (f (S::(\tau,\alpha::\text{bot})\text{Sequence})) = \)
\(\text{OclIterate S (P::(\tau,\alpha::\text{bot})\text{VAL} \Rightarrow (\tau,\text{c::bot})\text{VAL} \Rightarrow (\tau,\text{c::bot})\text{VAL}) A} \)
and cpF: \( \text{cp} F \)
and cpF’: \( \text{cp} F’ \)
shows \( \text{cp} (\lambda X. (f ((F X)::(\tau,\alpha::\text{bot})\text{Sequence})))) \)
by(simp add: f_by_it cpF)

lemma f1_by_iterate_undef_1:
\[ \forall S. f (S::(\tau,\alpha::\text{bot})\text{Sequence}) = \]
\(\text{OclIterate S (P::(\tau,\alpha::\text{bot})\text{VAL} \Rightarrow (\tau,\text{c::bot})\text{VAL} \Rightarrow (\tau,\text{c::bot})\text{VAL}) A} \]
Appendix B. Isabelle Theories

\[ f \perp = \perp \]
by (simp)

**Lemma f_1_by_iterate_defopt:**

**Assumes:** \( f \_ by \_ it \wedge S. (f \quad S:\quad (\tau,\alpha::\mathrm{bot})\quad \mathrm{Sequence}) = (\mathrm{OclIterate} \quad S \quad (P:\quad (\tau,\alpha::\mathrm{bot}) \quad \mathrm{VAL} \Rightarrow (\tau,\alpha::\mathrm{bot}) \quad \mathrm{VAL} \Rightarrow (\tau,\alpha::\mathrm{bot}) \quad \mathrm{VAL}) \quad A) \)

and \( \quad \mathrm{defopt}_P: \quad \forall x. \quad (\tau \Rightarrow (P \quad x \quad y)) \quad = ((\tau \Rightarrow (P \quad x \quad y))) \)

and \( \quad \mathrm{cpP}_1: \quad \forall y. \quad cp (\lambda \quad x \quad P \quad x \quad y) \)

and \( \quad \mathrm{cpP}_2: \quad \forall x. \quad cp ((P:\quad (\tau,\alpha::\mathrm{bot}) \quad \mathrm{VAL} \Rightarrow (\tau,\alpha::\mathrm{bot}) \quad \mathrm{VAL} \Rightarrow (\tau,\alpha::\mathrm{bot}) \quad \mathrm{VAL}) \quad x) \)

shows \( \tau \Rightarrow (\mathrm{OclIncluding} \quad S \quad x)\quad (\tau \Rightarrow (\mathrm{defS} \quad S \quad (\tau,\alpha::\mathrm{bot}) \quad \mathrm{Sequence}))) = ((\tau \Rightarrow (P \quad x \quad f \quad S \quad \tau))) \)

by (simp add: f_\_by_\_it \mathrm{OCL\_is\_defopt}_\_OclIterate \_defopt_\_P \_cpP_\_1 \_cpP_\_2)

**Lemma f_1_by_iterate_mSequence:**

\[ \quad \wedge S. \quad f \quad (S:\quad (\tau,\alpha::\mathrm{bot})\quad \mathrm{Sequence}) = (\mathrm{OclIterate} \quad S \quad (P:\quad (\tau,\alpha::\mathrm{bot}) \quad \mathrm{VAL} \Rightarrow (\tau,\alpha::\mathrm{bot}) \quad \mathrm{VAL} \Rightarrow (\tau,\alpha::\mathrm{bot}) \quad \mathrm{VAL}) \quad A) \quad \] \[ \Rightarrow \]

\[ \quad f \quad [] \quad = \quad A \quad \]

by (simp)

**Lemma f_1_by_iterate_including:**

**Assumes:** \( f \_ by \_ it \wedge S. (f \quad S:\quad (\tau,\alpha::\mathrm{bot})\quad \mathrm{Sequence}) = (\mathrm{OclIterate} \quad S \quad (P:\quad (\tau,\alpha::\mathrm{bot}) \quad \mathrm{VAL} \Rightarrow (\tau,\alpha::\mathrm{bot}) \quad \mathrm{VAL} \Rightarrow (\tau,\alpha::\mathrm{bot}) \quad \mathrm{VAL}) \quad A) \)

and \( \quad \mathrm{defS}: \quad \tau \Rightarrow (\mathrm{OclIncluding} \quad S \quad x)\quad (\tau \Rightarrow (\mathrm{defx} \quad f \quad S \quad (\tau,\alpha::\mathrm{bot}) \quad \mathrm{Sequence}))) = ((\tau \Rightarrow (\mathrm{defx} \quad f \quad S \quad (\tau,\alpha::\mathrm{bot}) \quad \mathrm{Sequence}))) \)

and \( \quad \mathrm{cpP}_1: \quad \forall y. \quad cp (\lambda \quad x \quad P \quad x \quad y) \)

and \( \quad \mathrm{cpP}_2: \quad \forall x. \quad cp ((P:\quad (\tau,\alpha::\mathrm{bot}) \quad \mathrm{VAL} \Rightarrow (\tau,\alpha::\mathrm{bot}) \quad \mathrm{VAL} \Rightarrow (\tau,\alpha::\mathrm{bot}) \quad \mathrm{VAL}) \quad x) \)

shows \( f \quad (\mathrm{OclIncluding} \quad S \quad x)\quad (\tau \Rightarrow (\mathrm{f_\_by_\_it} \quad iterate \_of \_including \quad \mathrm{defx} \quad f \quad S \quad (\tau,\alpha::\mathrm{bot}) \quad \mathrm{Sequence}))) = ((\tau \Rightarrow (\mathrm{f_\_by_\_it} \quad iterate \_of \_including \quad \mathrm{defx} \quad f \quad S \quad (\tau,\alpha::\mathrm{bot}) \quad \mathrm{Sequence}))) \)

by (simp add: f_\_by_\_it \_iterate \_of \_including \_defx \quad f \quad S \quad (\tau,\alpha::\mathrm{bot}) \quad \mathrm{Sequence}) \_cpP_\_1 \_cpP_\_2)

**Lemma f_1_by_iterate_opt_including:**

**Assumes:** \( f \_ by \_ it \wedge S. (f \quad S:\quad (\tau,\alpha::\mathrm{bot})\quad \mathrm{Sequence}) = (\mathrm{OclIterate} \quad S \quad (P:\quad (\tau,\alpha::\mathrm{bot}) \quad \mathrm{VAL} \Rightarrow (\tau,\alpha::\mathrm{bot}) \quad \mathrm{VAL} \Rightarrow (\tau,\alpha::\mathrm{bot}) \quad \mathrm{VAL}) \quad A) \)

and \( \quad P \_ \_undef \_1: \quad \forall y. \quad P \quad \perp \quad y = \perp \)

and \( \quad P \_ \_undef \_2: \quad \forall y. \quad P \quad \perp = \perp \)

and \( \quad \mathrm{cpP}_1: \quad \forall y. \quad cp (\lambda \quad x \quad P \quad x \quad y) \)

and \( \quad \mathrm{cpP}_2: \quad \forall x. \quad cp ((P:\quad (\tau,\alpha::\mathrm{bot}) \quad \mathrm{VAL} \Rightarrow (\tau,\alpha::\mathrm{bot}) \quad \mathrm{VAL} \Rightarrow (\tau,\alpha::\mathrm{bot}) \quad \mathrm{VAL}) \quad x) \)

shows \( f \quad (\mathrm{OclIncluding} \quad S \quad x)\quad (P \quad x \quad (f \quad S)) \)

by (simp add: f_\_by_\_it \_iterate \_opt \_of \_including \_P \_\_undef \_\_1 \_P \_\_undef \_\_2 \_cpP_\_1 \_cpP_\_2)

**For binary functions**

This cp-ness result holds only for binary functions whose second argument is the neutral element of iterate. Currently the only function of this style is union.
\textbf{B.4. Library}

\texttt{lemma f2\_by\_iterate\_cp:}
\texttt{\hspace{1em} assumes f\_by\_st: }\A S A. \( f(S:(\tau,\alpha::\bot)\text{Sequence}) \ A) = \text{(OclIterate}\ S\ (P:(\tau,\alpha::\bot)\text{VAL} \Rightarrow (\tau,\mathbb{c}::\bot)\text{VAL} \Rightarrow (\tau,\mathbb{c}::\bot)\text{VAL}) \ A) \)
\texttt{\hspace{1em} and \hspace{1em} cpF: cp F}
\texttt{\hspace{1em} and \hspace{1em} cpF': cp F'}
\texttt{\hspace{1em} shows \hspace{1em} f2\_by\_iterate\_mtSequence: \lambda X. \( f(((FX):(\tau,\alpha::\bot)\text{Sequence}))\ (F' A))\)}
\texttt{\hspace{1em} by(simp add: f\_by\_it \ cpF)}

\texttt{lemma f2\_by\_iterate\_undef\_1:}
\texttt{\hspace{1em} \[ \A S A. \( f(S:(\tau,\alpha::\bot)\text{Sequence}) \ A) = \text{(OclIterate}\ S\ (P:(\tau,\alpha::\bot)\text{VAL} \Rightarrow (\tau,\mathbb{c}::\bot)\text{VAL} \Rightarrow (\tau,\mathbb{c}::\bot)\text{VAL}) \ A) \]}
\texttt{\hspace{1em} \Longrightarrow \ f \ A = \bot}
\texttt{\hspace{1em} by(simp)}

\texttt{lemma f2\_by\_iterate\_undef\_2:}
\texttt{\hspace{1em} assumes f\_by\_st: }\A S A. \( f(S:(\tau,\alpha::\bot)\text{Sequence}) \ A) = \text{(OclIterate}\ S\ (P:(\tau,\alpha::\bot)\text{VAL} \Rightarrow (\tau,\mathbb{c}::\bot)\text{VAL} \Rightarrow (\tau,\mathbb{c}::\bot)\text{VAL}) \ A) \)
\texttt{\hspace{1em} and \hspace{1em} undef\_2\_P: }\A P. \( \lambda x. \ (P x \bot) \bot = \bot \)
\texttt{\hspace{1em} shows \hspace{1em} f\_by\_st: }\A S A. \( f(S A) = \bot \)
\texttt{\hspace{1em} by(simp add: f\_by\_it \ undef\_2\_P)}

\texttt{lemma f2\_by\_iterate\_defopt:}
\texttt{\hspace{1em} assumes f\_by\_st: }\A S A. \( f(S:(\tau,\alpha::\bot)\text{Sequence}) \ A) = \text{(OclIterate}\ S\ (P:(\tau,\alpha::\bot)\text{VAL} \Rightarrow (\tau,\mathbb{c}::\bot)\text{VAL} \Rightarrow (\tau,\mathbb{c}::\bot)\text{VAL}) \ A) \)
\texttt{\hspace{1em} and \hspace{1em} defopt\_P: }\forall x y. \ (\tau \equiv (\partial\ (P x y))) = ((\tau \equiv (\partial\ x) \land (\tau \equiv (\partial\ y)))
\texttt{\hspace{1em} and \hspace{1em} cpP\_T: }\A y. \ cP (\lambda x. \ P x y)
\texttt{\hspace{1em} and \hspace{1em} cpP\_E: }\A x. \ cP ((P:(\tau,\alpha::\bot)\text{VAL} \Rightarrow (\tau,\mathbb{c}::\bot)\text{VAL} \Rightarrow (\tau,\mathbb{c}::\bot)\text{VAL})\ x)
\texttt{\hspace{1em} shows \hspace{1em} f2\_by\_iterate\_defopt: \hspace{1em} f\_by\_st: }\A S A. \( f(S:\text{Sequence}) \ A) = ((\tau \equiv (\partial\ S) \land (\tau \equiv (\partial\ A)))
\texttt{\hspace{1em} by(simp add: f\_by\_defopt \ Ocl\_is\_defopt \ \text{OclIterate}\ \text{defopt}\_P \ \text{cpP}\_T \ \text{cpP}\_E)}

\texttt{lemma f2\_by\_iterate\_mSequence:}
\texttt{\hspace{1em} \[ \A S A. \( f(S:(\tau,\alpha::\bot)\text{Sequence}) \ A) = \text{(OclIterate}\ S\ (P:(\tau,\alpha::\bot)\text{VAL} \Rightarrow (\tau,\mathbb{c}::\bot)\text{VAL} \Rightarrow (\tau,\mathbb{c}::\bot)\text{VAL}) \ A) \]}
\texttt{\hspace{1em} \Longrightarrow \ f \ [\ A = A \}
\texttt{\hspace{1em} by(simp)}

\texttt{lemma f2\_by\_iterate\_including:}
\texttt{\hspace{1em} assumes f\_by\_st: }\A S A. \( f(S:(\tau,\alpha::\bot)\text{Sequence}) \ A) = \text{(OclIncluding}\ S\ x) \ A \Rightarrow (P x \ f\ S A) \ A \)
\texttt{\hspace{1em} and \hspace{1em} defS: }\tau \equiv (\partial(\text{Sequence})
\texttt{\hspace{1em} and \hspace{1em} defx: }\tau \equiv (\partial(x:\ (\tau \Rightarrow \alpha::\bot)))
\texttt{\hspace{1em} and \hspace{1em} cpP\_T: }\A y. \ cP (\lambda x. \ P x y)
\texttt{\hspace{1em} and \hspace{1em} cpP\_E: }\A x. \ cP (P x)
\texttt{\hspace{1em} shows \hspace{1em} f\_by\_st: }\A S A. \( f(OclIncluding\ S\ x) \ A \ A) = (P x \ f\ S A) \ A \)

269
Appendix B. Isabelle Theories

by(simp add: f_by_it iterate_of_including defS defx cpP_1 cpP_2)

lemma f2_by_iterate_opt_including:
  assumes f_by_it: \(\forall S A. \ (f \ S :: (\tau', \alpha::bot)\text{Sequence}) \ A = \ (OclIterate S \ P :: (\tau', \alpha::bot)\text{VAL} \Rightarrow (\tau', c::bot)\text{VAL} \Rightarrow (\tau', c::bot)\text{VAL}) \ A)\)
  and P_undef_1: \(!! y. \ P \bot y = \bot\)
  and P_undef_2: \(!! x. \ P x \bot = \bot\)
  and cpP_1: \(\forall y. \ cp (\lambda x. P x y)\)
  and cpP_2: \(\forall x. \ cp (P x)\)
  shows \(f \ (OclIncluding S x) A = P x (f S A)\)
  by(simp add: f_by_it iterate_opt_of_including P_undef_1 P_undef_2 cpP_1 cpP_2)

Union
defs

OclUnion_def : \(OclUnion \equiv \text{lift2 strictify} (\lambda X. \text{strictify} (\lambda Y. \text{Abs_Sequence_0} ((\text{Rep_Sequence_0 X} \uplus \text{Rep_Sequence_0 Y}))))\)

ocl_setup_op [OclUnion]

lemma OCL_is_def_OclUnion:
  \(\partial (OclUnion (X :: (\tau', \alpha::bot)\text{Sequence}) \ Y) = (\partial X \wedge \partial Y)\)
  apply(rule lift2_strict_is_isdef_fw_Sequence_Sequence[OF OclUnion_def])
  apply(simp)
  done

lemma OCL_is_defopt_OclUnion:
  \((\tau \ni \partial (OclUnion (X :: (\tau', \alpha::bot)\text{Sequence}) \ Y)) = ((\tau \ni \partial X) \wedge (\tau \ni \partial Y))\)
  apply(rule lift2_strictify_implies_LocalValid_defined_Sequence_Sequence[OF OclUnion_def])
  apply(simp)
  done

Its computational characterisation using iterate

lemma union_by_iterate:
  \(((X :: (\tau', \alpha::bot)\text{Sequence}) \cup (Y :: (\tau', \alpha::bot)\text{Sequence})) = (Y \rightarrow \text{iterate} (x::X | y \rightarrow \text{including} (x :: (\tau', \alpha::bot)\text{VAL})))\)
  apply(rule ext)
  apply(rule iterate_universal, simp_all)
  apply(rule_tac X=X in Sequence_sem_cases)
  apply(simp add: OclUnion_def OclMtSequence_def ss_lifting)
  apply(simp add: OclUnion_def OclMtSequence_def ss_lifting)
  apply(simp_all add: localValidDefined2sem DEF_def)
  apply(frule DEF_X_Sequence', clarify)
  apply(rule_tac X=X in Sequence_sem_cases, simp_all)
  apply(simp_all add: OclUnion_def OclIncluding_def ss_lifting' neq_commute)

270
B.4. Library

done

thm f2_by_iterate_including[OF union_by_iterate, simplified]

lemmas union_of_mtSequence1 [simp] = f2_by_iterate_mtSequence[OF union_by_iterate]

lemma union_of_mtSequence2 [simp]:
\[
\bigcup \{ A \mid A = (A::(\tau,\alpha::bot)Sequence) \} =\bigcup \{ B \mid B = (A::(\tau,\alpha::bot)Sequence) \}
\]
apply simp_all
apply (simp_all add: OclMtSequence_def OclUnion_def ss_lifting)
done

lemmas union_of_including [simp] = f2_by_iterate_opt_including[OF union_by_iterate, simplified]

lemma union_assoc [simp]:
\[(A \cup (B \cup C)) = ((A \cup B) \cup C) : (\tau,\alpha::bot)Sequence)\]
apply simp_all
apply (simp_all add: OclUnion_def ss_lifting)
done

Properties of iterate over a union of two sequences

lemma iterate_of_union:
\[
\begin{array}{c}
[ \tau \models \not\emptyset X ] \\
\OclIterate ((X::(\tau,\alpha::bot)Sequence) \cup Y) \\
\OclIterate Y P (\OclIterate X P A) \\
\end{array}
\]
apply case_tac \tau \models \not\emptyset Y
apply (ocl_hypsubst, simp)
apply (simp add: iterate_of_union)
done

lemma iterate_of_union_opt:
\[
[ \begin{array}{c}
X \cup P x \bot = \bot \\
\end{array} ] \\
\OclIterate ((X::(\tau,\alpha::bot)Sequence) \cup Y) \\
\OclIterate Y P (\OclIterate X P A) \\
\begin{array}{c}
apply (rule ext) \\
apply (case_tac x \models \not\emptyset X) \\
apply (simp add: iterate_of_union)
\end{array}
\]

271
apply(simp add: sym[OF isUndefined_charn_local])
apply(rule_tac A1=X in trans[OF cp_charn], simp_all)
apply(rule_tac A2=X in sym[OF trans[OF cp_charn]], simp_all)
apply(simp)
done

lemma iterate_distrib_union:
  assumes mtC: \( B \vdash g B A \tau = B \tau \)
  and stepC: \( B x y. g B (P x y) \tau = P x (g B y) \tau \)
  and cp2_g: \( x. \text{cp} (g x) \)
  and cp2_P: \( x. \text{cp} (P x) \)
  and defX: \( \tau \models \partial X \)
  and defY: \( \tau \models \partial Y \)
  shows \( \text{OclIterate} ((X :: (\tau,\alpha :: \text{bot}) \text{Sequence}) \cup Y) \)
  \( (P :: (\tau,\alpha :: \text{bot}) \text{VAL} \Rightarrow (\tau,\alpha :: \text{bot}) \text{VAL} \Rightarrow (\tau,\alpha :: \text{bot}) \text{VAL}) A \tau = g (\text{OclIterate} X P A) (\text{OclIterate} Y P A) \tau \)
by(simp add: iterate_of_union defX defY cp2_g cp2_P mtC stepC)

lemma iterate_distrib_union_opt:
  assumes undefC: \( B \vdash g B \bot = \bot \)
  and mtC: \( B \vdash g B A = B \)
  and stepC: \( B x y. g B (P x y) = P x (g B y) \)
  and undef_2_P: \( x. P x \bot = \bot \)
  and cp2_g: \( x. \text{cp} (g x) \)
  and cp2_P: \( x. \text{cp} (P x) \)
  and defX: \( \tau \models \partial X \)
  and defY: \( \tau \models \partial Y \)
  shows \( f (X \cup Y) \tau = g (f X) (f Y) \tau \)
by(rule ext, simp add: iterate_of_union_opt undef_2_P cp2_g cp2_P)

And the corresponding forward rules for unary functions
lemma f1_by_iterate_distrib_union:
  assumes f_by_it: \( \text{f} \_by\_it \ (S :: (\tau,\alpha :: \text{bot}) \text{Sequence}) = (\text{OclIterate} S (P :: (\tau,\alpha :: \text{bot}) \text{VAL} \Rightarrow (\tau,\alpha :: \text{bot}) \text{VAL} \Rightarrow (\tau,\alpha :: \text{bot}) \text{VAL}) A) \)
  and mtC: \( B \vdash g B A \tau = B \tau \)
  and stepC: \( B x y. g B (P x y) \tau = P x (g B y) \tau \)
  and cp2_g: \( x. \text{cp} (g x) \)
  and cp2_P: \( x. \text{cp} (P x) \)
  and defX: \( \tau \models \partial X \)
  and defY: \( \tau \models \partial Y \)
  shows \( f (X \cup Y) \tau = g (f X) (f Y) \tau \)
by(simp add: f_by_it iterate_distrib_union defX cp2_g cp2_P mtC)

lemma f1_by_iterate_distrib_union_opt:
B.4. Library

assumes \( f_{\_by\_it}: \forall S. (f (S:(\tau,\alpha::\bot)\text{Sequence})) = \)
\( \) (OclIterate S (P:(\tau,\alpha::\bot)\text{VAL}) \Rightarrow (\tau,\epsilon::\bot)\text{VAL}) \Rightarrow (\tau,\epsilon::\bot)\text{VAL}) \ A)

and undefC: \( \forall B. f B \bot = \bot \)

and stepC: \( \forall B x y. f B (P x y) = P (f B y) \)

and cp2_g: \( \forall x. cp (g x) \)

and cp2_P: \( \forall x. cp (P x) \)

shows \( f (X \cup Y) = g (f X) (f Y) \)

by (simp add: f_by_it iterate_distrib_union_opt undef_2_P cp2_g cp2_P undefC mtC stepC)

Append
defs

\[ \text{OclAppend} \equiv \text{lift}_2 (\text{strictify} (\lambda S. \text{strictify} (\lambda e. \text{Abs}\_\text{Sequence}_0
\quad ((\text{concat} [\text{Rep}\_\text{Sequence}_0 S, [e]]))))) \]

ocl_setup_op [OclAppend]

lemma append_including_UC [simp]:
\[ \text{OclAppend} (S:(\tau,\alpha::\bot)\text{Sequence}) (x:(\tau,\alpha)\text{VAL}) = S \rightarrow\text{including} x \]

by (simp add: OclAppend_def OclIncluding_def)

Prepend
defs

\[ \text{OclPrepend} \equiv \text{lift}_2 (\text{strictify} (\lambda S. \text{strictify} (\lambda e. \text{Abs}\_\text{Sequence}_0
\quad ((\text{concat} [\text{Rep}\_\text{Sequence}_0 S, [e]])))))) \]

ocl_setup_op [OclPrepend]

lemma prepend_union_UC [simp]:
\[ \text{OclPrepend} (X:(\tau,\alpha::\bot)\text{Sequence}) (a:(\tau,\alpha)\text{VAL}) = ([] \rightarrow\text{including} a) \cup X) \]

apply (rule_tac X=X in Sequence_sem_cases_ext)
apply (simp_all)
apply (simp_all add: OclPrepend_def OclIncluding_def OclUnion_def OclMtSequence_def ss_lifting neq_commute)
done

lemma test'': (OclPrepend (OclPrepend (X:(\tau,\alpha::\bot)\text{Sequence}) (a:(\tau,\alpha)\text{VAL})) (a:(\tau,\alpha)\text{VAL})) = bla
apply (simp) oops
Appendix B. Isabelle Theories

**Lemma OCL_is_def_OclPrepend:**
\[ \partial (\text{OclPrepend} (X::(\tau,:\alpha::\text{bot}) \text{Sequence}) (Y::(\tau,:\alpha::\text{bot}) \text{VAL})) = (\partial X \land \partial Y) \]
apply(rule lift2_strict_is_isdef_fw_Sequence_Val[OF OclPrepend_def])
apply(simp)
done

**Lemma OCL_is_defopt_OclPrepend:**
\[ (\tau \models \partial (\text{OclPrepend} (X::(\tau,:\alpha::\text{bot}) \text{Sequence}) (Y::(\tau,:\alpha::\text{bot}) \text{VAL}))) = (\tau \models \partial X \land (\tau \models \partial Y)) \]
apply(rule lift2_strictify_implies_LocalValid_defined_Sequence_Val[OF OclPrepend_def])
apply(simp)
done

Some sort of case exhaustion over prepend

**Lemma prepend_charn1:**
\[ \llbracket \bot \notin \text{set } xs; xs \neq [] \rrbracket \implies \exists (S::(\tau,:\alpha::\text{bot}) \text{Sequence}_0 \tau a). \text{Abs_Sequence}_0 (xs) = (\text{OclPrepend} (\text{lift0 } S) (\text{lift0} (a::\tau))) \]
apply(cases xs, simp_all)
apply(rule_tac x=\text{Abs_Sequence}_0 (list) in exI)
apply(rule_tac x=\text{lift0 } a in exI)
apply(auto simp: OclMtSequence_def OclIncluding_def OclUnion_def ss_lifting)
done

**Lemma prepend_charn2:**
\[ \llbracket \bot \notin \text{set } xs; xs \neq [] \rrbracket \implies \exists (S::(\tau,:\alpha::\text{bot}) \text{Sequence}_0 \tau a). \text{Abs_Sequence}_0 (xs) = (\text{OclPrepend} S (a::(\tau,:\alpha) \text{VAL})) \]
apply(cases xs, simp_all)
apply(rule_tac x=\text{lift0 } (\text{Abs_Sequence}_0 list) in exI)
apply(rule_tac x=\text{lift0 } a in exI)
apply(auto simp: OclMtSequence_def OclIncluding_def OclUnion_def ss_lifting)
done

Reverse induction on sequences: induction over prepend

**Lemma Sequence_induct_prepend_0:**
asumes undefCase : P \bot
and mtCase : P (\bot \tau)
and stepCase : \exists (S::\tau::\text{bot} \text{Sequence}_0 \tau a::\text{bot}). \text{Def } S \text{ DEF } \tau \implies (P ((\text{OclPrepend} (\text{lift0 } S) (\text{lift0} (a::\tau))) \tau))
shows P (S::\tau::\text{bot} \text{Sequence}_0)

274
apply (rule Abs_Sequence_0_induct_charn, rule undefCase)
apply (erule rev_mp)
apply (erule_tac x=a and S=Abs_Sequence_0_list in stepCase)
apply (erule DEF_Abs_Sequence)
apply (simp_all add: localValidDefined2sem lift0_def)
done

lemma Sequence_induct_prepend:
assumes undefCase: P \bot \tau
and mtCase: P [] \tau
and stepCase: \(\lambda x. P x A\) \tau
shows \((\text{OclIterate} (\text{OclPrepend} S (x:\!(\tau,\alpha::\bot)\!\text{VAL})) P A) \tau\) =
((\text{OclIterate} S (x:\!(\tau,\alpha::\bot)\!\text{VAL})) P (P x A)) \tau)
apply (insert undefCase mtCase)
apply (simp_all add: iterate_of_including)
done

iterate on prepend

lemma iterate_of_prepend:
assumes defS: \(\tau \vdash \partial (S::(\tau,\alpha::\bot))\!\text{Sequence}\)
and defx: \(\tau \vdash \partial (x::(\tau \Rightarrow \alpha::\bot))\!\text{Sequence}\)
and cpP_1: \(\lambda y. \text{cp} (\lambda x. P x y)\)
and cpP_2: \(\lambda x. \text{cp} (P x)\)
shows \(((\text{OclIterate} (\text{OclPrepend} S (x:\!(\tau,\alpha::\bot)\!\text{VAL})) P A) \tau) =
((\text{OclIterate} S (x:\!(\tau,\alpha::\bot)\!\text{VAL})) P (P x A)) \tau)\)
apply (insert defS defx) simp
apply (subst iterate_of_union)
apply (simp add: OCL_is_defopt_OclIncluding)
apply (rule_tac A=(\text{OclIterate} [] \Rightarrow \text{including} x P A) in cp_charn)
done

Includes and excludes

defs
Appendix B. Isabelle Theories

\[ OclIncludes_{\text{def}} : OclIncludes \equiv \text{lift}_2(\text{strictify}(\lambda X. \text{strictify}(\lambda x. \{x \in \text{set} \ Rep\_Sequence_0 X\})) \]  

\[ OclExcludes_{\text{def}} : OclExcludes \equiv \text{lift}_2(\text{strictify}(\lambda X. \text{strictify}(\lambda x. \{x \notin \text{set} \ Rep\_Sequence_0 X\})) \]  

\text{ocl\_setup\_op} [OclIncludes, OclExcludes]

\text{lemma} \ includes\_charm1 :  
\( (\tau \vdash x \in S) \Rightarrow (\{x \in S\} \in \text{set} \ Rep\_Sequence_0 (S \tau)) \)  
apply(case_tac \( S \tau \neq \bot \))  
apply(auto simp: OclIncludes_def localValid2sem ss_lifting)  
done

\text{lemma} \ includes\_charm2 :  
[ DEF(S \tau); \{x \in S\} \in \text{set} \ Rep\_Sequence_0 (S \tau) ] \Rightarrow \tau \vdash x \in S  
apply(unfold DEF_def, frule DEF_X_Sequence')  
apply(auto simp: OclIncludes_def localValid2sem ss_lifting)  
done

Higher Properties of includes and excludes

\text{lemma} \ exludes\_not\_includes[simp]:  
x \notin X = \neg (x:(\tau, \alpha::\text{bot})\text{VAL}) \in (X:(\tau, \alpha::\text{bot})\text{Sequence})  
apply(rule ext)  
apply(simp add: OclExcludes_def OclIncludes_def OclNot_def ss_lifting')  
done

Currently a non-strict body is taken although a strict one is possible. This should be corrected as the experience has shown that strict functions are much nicer to work with.

\text{lemma} \ includes\_by\_iterate:  
\( a \in (X:(\tau, \alpha::\text{bot})\text{Sequence}) = \)  
OclIterate X (\( \lambda x:(\tau, \alpha::\text{bot})\text{VAL} \) y. (x = a) \lor y)  
(\( \text{if } a \text{ then } \bot \text{ else } \bot \text{ endif} \))  
apply(rule ext)  
apply(rule iterate_universal, simp_all)  
apply(simp add: OclIncludes_def OclMtSequence_def OclFalse_def OclIf_def OclIsDefined_def OclUndefined_def ss_lifting')  
apply(simp add: localValidDefined2sem DEF_def)  
apply(frule DEF_X_Sequence', clarify)  
apply(auto simp: OclIncludes_def OclIncluding_def OclTrue_def OclSor_def OclNot_def OclSand_def OclStrictEq_def)
SS lifting

\[\text{done}\]

**lemmas** includes_mtSequence [simp] = f1_by_iterate_mtSequence[OF includes_by_iterate]
**thm** f1_by_iterate_opt_including[OF includes_by_iterate, simplified]

**lemma** excludes_by_iterate:
\[a \notin (\mathbf{X} :: (\tau \cdot \alpha :: \bot) \mathbf{Sequence}) =\]
\[\text{OclIterate } \lambda (x :: (\tau \cdot \alpha :: \bot) \text{VAL} \cdot (x < > ' a \land y) \cdot (\text{if } (\partial a) \text{ then } \top \text{ else } \bot \text{ endif})\]
\[\text{apply(\text{rule ext, simp add: includes_by_iterate);}\]
\[\text{apply(\text{rule iterate_fusion, simp_all);}\]
\[\text{apply(\text{simp_all add: OclNot_def OclTrue_def OclFalse_def OclIf_def OclSand_def OclSor_def OclUndefined_def ss_lifting');}\]
\[\text{done}\]

containment in the empty list can be reduced to definedness

**lemma** includes_of_mtSequence [simp]:
\[-(\tau \models (x :: (\tau \cdot \alpha :: bot) \mathbf{Sequence}))\]
\[\text{by(simp add: OclIncludes_def OclMtSequence_def OclNot_def localValid2sem ss\_lifting')}\]

**lemma** includes_charn1_including [simp]:
\[\text{\llbracket} \tau \models (a :: (\tau \cdot \alpha :: \text{VAL})) \land (X :: (\tau \cdot \alpha :: \bot) \mathbf{Sequence}) \text{\rrbracket} \implies\]
\[\tau \models a \in (X ->including a)\]
\[\text{apply(simp only: localValidDefined2sem DEF_def);}\]
\[\text{apply(frule DEF_X_Sequence', clarify);}\]
\[\text{apply(simp add: localValid2sem OclIncludes_def OclIncluding_def ss\_lifting' neq\_commute);}\]
\[\text{done}\]

**lemma** includes_charn2_including [simp]:
\[\text{\llbracket} \tau \models (a :: (\tau \cdot \alpha :: \text{VAL})) \land (b :: (\tau \cdot \alpha :: \text{VAL})) \land a \in (X :: (\tau \cdot \alpha :: \bot) \mathbf{Sequence}) \text{\rrbracket} \implies\]
\[\tau \models a \in (X ->including b)\]
\[\text{apply(case_tac X \tau \neq \bot);}\]
\[\text{apply(frule DEF_X_Sequence', clarify);}\]
\[\text{apply(simp_all only: localValidDefined2sem DEF_def);}\]
\[\text{apply(simp_all add: localValid2sem OclIncludes_def OclIncluding_def ss\_lifting' neq\_commute);}\]
\[\text{apply(drule neq\_commute[THEN iffD1], simp);}\]
\[\text{done}\]

**lemma** includes_of_collectionRange:
\[(x :: (\tau \cdot \text{Integer})) \in ((\text{OclCollectionRange } a b) :: (\tau \cdot \text{Integer}_0) \mathbf{Sequence}) =\]
\[((a :: (\tau \cdot \text{Integer})) \leq x) \land (x \leq (b :: (\tau \cdot \text{Integer})))\]
\[\text{apply(rule ext);}\]

277
Appendix B. Isabelle Theories

apply (case_tac DEF (x xa), case_tac DEF (a xa), case_tac DEF (b xa))
apply (simp add: DEF_def_both, clarify)
apply (subgoal_tac \ι \notin set (map (\z. ((int z) + \i((a xa)\i)) + \j)))
apply (auto simp add: DEF_def_both, clarify)
apply (rule_tac x = nat ((\i((b xa)\i) - \i((a xa)\i)) + \j) in image_eqI, auto)
done

lemma includes_of_union [simp]:
\[ \tau \models x \in (X \cup Y) \quad \Rightarrow \quad \tau = (x \in X) \lor (x \in Y) \]
apply (simp add: localValidDefined2sem DEF_def)
apply (frule DEF_X_Sequence)
apply (rotate_tac 1, frule DEF_X_Sequence, clarify)
apply (simp add: OclIncludes_def OclUnion_def OclOr_def OclAnd_def OclNot_def ss_lifting)
done

lemma test’’: \(1::'\tau Integer) \in (mkSequence[(1::'\tau Integer)]:('\tau, Integer_0)Sequence)
oops

lemma not_includes_of_mtSequence:
\(\tau \models - (x::('\tau, 'a::bot)VAL) \in (((::('\tau, 'a::bot)Sequence) \cup Y) \quad \Rightarrow \quad \tau = ((a \in X) \lor (a \in Y)) + \j))
by (simp add: localValidNot2sem localValidDefined2sem OclIncludes_def OclMtSequence_def ss_lifting)

lemma mtSequence_includes_of_OLD:
\(\tau \models x \Rightarrow \quad \tau \models (x::('\tau, 'a::bot)VAL) \notin (\cdot::('\tau, 'a::bot)Sequence)
by (simp add: ocl_subst, simp)

Note: this will be subsumed by OCL_is_defopt_OclIncludes if ocl_setup_op is working correctly on collection types

Emptiness
defs

\text{OclIsEmpty}_\text{def} : 
\text{OclIsEmpty} \equiv \text{lift} I (\lambda X. (\text{Rep\_Sequence\_0} X = []))
\[ \text{OclNotEmpty}_{\text{def}} : \]
\[ \text{OclNotEmpty} \equiv \text{lift1} \left( \text{strictify} \left( \lambda X. \{ \text{Rep\_Sequence}_0 X \neq [] \} \right) \right) \]

\text{ocl\_setup\_op} \{ \text{OclIsEmpty, OclNotEmpty} \}

\text{The normal form for emptiness testing:}

we prefer a canonical form were notEmpty is written using isEmpty

\text{lemma notEmpty\_not\_isEmpty\_conv[simp]:}
\[ \not\emptyset (X:\{\tau, \alpha::\text{bot}\} \text{Sequence}) = \neg (\not\emptyset X) \]
\text{apply(rule ext)}
\text{apply(simp add: OclNotEmpty\_def OclIsEmpty\_def OclNot\_def ss\_lifting')}
\text{done}

further on the emptiness test is reduced to an equality to a constant

\text{lemma isEmpty\_stricteq\_mtSequence\_conv[simp]:}
\[ (\not\emptyset X) = ((X:\{\tau, \alpha::\text{bot}\} \text{Sequence}) \not\equiv []) \]
\text{apply(rule_tac X y \neq \bot)}
\text{apply(erule DEF_X\_Sequence', clarify)}
\text{apply(simp all add: OclIsEmpty\_def OclMtSequence\_def OclStrictEq\_def Abs\_Sequence_0_inject\_charn ss\_lifting')}
\text{done}

\text{Properties of emptiness:}

\text{lemma isEmpty\_of\_mtSequence:}
\[ \tau \models \not\emptyset [] \]
\text{by (simp)}

\text{lemma isEmpty\_of\_including:}
\[ \models (\tau \models \emptyset (a:(\tau,\alpha::\text{bot}) \text{VAL})) \Rightarrow (\tau \models \not\emptyset (X \rightarrow \text{including } a)) \]
\text{by(simp)}

\text{lemma isEmpty\_union [simp]:}
\[ ((X:\{\tau,\alpha::\text{bot}\} \text{Sequence}) \cup Y) \equiv [] = (X \equiv []) \land (Y \equiv []) \]
\text{apply(rule_tac X=X in Sequence\_sem\_cases\_ext)}
\text{apply(rule_tac X=Y in Sequence\_sem\_cases, simp\_all)}
\text{apply(simp all add: OclUnion\_def OclStrictEq\_def OclMtSequence\_def OclSand\_def ss\_lifting' Abs\_Sequence_0_inject\_charn)}
\text{done}

\text{lemma notEmpty\_of\_mtSequence:}
\[ \tau \models \neg (\not\emptyset []) \]
\text{by(simp)}

279
Appendix B. Isabelle Theories

lemma notEmpty_of_including:
\[ \tau \models \emptyset \quad (a::(\tau',\alpha::\bot)VAL) ; \quad \tau \models \emptyset \quad (X::(\tau',\alpha::\bot)Sequence) \] \implies 
\[ \tau \models \not\emptyset \quad (X \rightarrow including \ a) \]
by (simp)

Size
defs

\[ OclSize_def : OclSize \equiv lift1 (strictify(\lambda X. 
\text{int}(\text{Nat.size} \circ \text{Rep.Sequence}_0 X)) ) \]

ocl_setup_op [OclSize]

lemma size_by_iterate:
\((self::(\tau',\alpha::\bot)Sequence) \rightarrow size)\) = 
OclIterate self \((\lambda x: (\tau',\alpha::\bot)VAL) \quad (y: \tau \text{ Integer}) \quad y + 1) \emptyset\)
apply (rule ext)
apply (rule iterate_universal)
apply (simp_all)
apply (simp_all add: OclSize_def OclIterate_def plus_def OclMtSequence_def
OclUndefined_def OclIncluding_def localValidDefined2sem
OCL_Integer.Zero_ocl_int_def OCL_Integer.One_ocl_int_def
ss_lifting')
apply (rule DEF_X_Sequence')
apply (auto simp: neq_commute)
done

lemmas size_of_mtSequence[simp] = f1_by_iterate_mtSequence[OF size_by_iterate]

lemma size_of_including[simp]:
\[ \tau \models \emptyset \quad (S::(\tau',\alpha::\bot)Sequence) \] \implies 
\[ (\| (\| S::(\tau',\alpha::\bot)Sequence) \rightarrow including \ x \|) \tau = (\| S \| + 1) \tau \]
apply (case_tac \tau \not\models \emptyset \ S)
apply (ocl_hypsubst, simp)
apply (simp add: f1_by_iterate_including[OF size_by_iterate, simplified])
done

lemma size_eq_zero_isEmpty_UC[simp]:
\[ (\| S::(\tau',\alpha::\bot)Sequence) \| = 0) = (S = \|) \]
apply (rule ext)
apply (case_tac S x \neq \bot)
apply (frule DEF_X_Sequence', clarify)
apply (simp_all add: OclSize_def OclStrictEq_def OclMtSequence_def)
B.4. Library

OCL_Integer.Zero_ocl_int_def Abs_Sequence_0_inject_charn
localValid2sem ss_lifting')
done

lemma size_of_union_UC [simp]:
(∥((X::(τ,α::bot)Sequence) U Y))∥ = (∥X∥ + ∥Y∥)
apply(rule f1_by_iterate_distrib_union_opt[OF size_by_iterate, of op +, simplified])
apply(rule ext, simp add: plus_def Zero_ocl_int_def ss_lifting')
done

Induction over two lists

lemma Sequence_induct2_including_0:
assumes undefCase: P ⊥ ⊥ τ
and equalSize: ∥(lift0 X)∥ τ = ∥(lift0 Y)∥ τ
and mtCase: P [] [] τ
and stepCase: ⋀ (X::(τ,α::bot)Sequence) (a::(τ,α::bot)VAL) (Y::(τ,c::bot)Sequence) (b::(τ,c::bot)VAL).
 show P (OclIncluding (lift0 X) (lift0 a) τ) (OclIncluding (lift0 Y) (lift0 b) τ))
apply(insert equalSize, erule rev_mp)
apply(rule Abs_Sequence_0_induct_charn)
apply(rule_tac x=X in Abs_Sequence_0_induct_charn)
apply(rule impI)+
apply(erule rev_mp)+
prefer 2 apply(assumption, rotate_tac 1)
apply(erule rev_mp)+
apply(erule rev_mp)+
apply(erule list_rev_induct2)
apply(erule mtCase)
apply(auto simp: OclSize_def OclMtSequence_def ss_lifting')
apply(rotate_tac -1, drule_tac a=x and X=Abs_Sequence_0 xs, and b=yb and Y=Abs_Sequence_0 ys, in stepCase)
apply(simp_all add: OclIncluding_def neq_commute ss_lifting')
done

lemma Sequence_induct2_including:
assumes equalSize: τ ⊨ ∥X∥ ⇔ ∥Y∥
and undefCase: P ⊥ ⊥ τ
and mtCase: P ⊥ ⊥ τ
and stepCase: ⋀ (X::(τ,α::bot)Sequence) (α::(τ,α::bot)VAL) (Y::(τ,c::bot)Sequence) (b::(τ,c::bot)VAL).

281
Appendix B. Isabelle Theories

\[ P X Y \tau; \tau \vdash \partial X; \tau \vdash \partial a; \tau \vdash \partial Y; \tau \vdash \partial b \] \implies P (OclIncluding X a) (OclIncluding Y b) \tau

and \( \text{cp}_1 \cdot P \ \land \ \exists y. \ \text{cp} (\lambda x. P x y) \)

and \( \text{cp}_2 \cdot P \ \land \ \exists x. \ \text{cp} (P x) \)

defines

\[ P (X::(\tau',\alpha::\bot)\text{Sequence}) (Y::(\tau',\alpha::\bot)\text{Sequence}) \tau \]

apply\( (\text{insert} \ \text{cp}_1 \cdot P \ \text{cp}_2 \cdot P, \ \text{simp add:}\ \text{cp\_by\_cpify}) \)

apply\( (\text{insert} \ \text{cp}_1 \cdot P \ \text{cp}_2 \cdot P) \)

apply\( (\text{rule subst, rule sym, rule_tac x=X in cp\_subst, simp}) \)

apply\( (\text{rule subst, rule sym, rule_tac x=Y in cp\_subst, simp}) \)

apply\( (\text{rule_tac} \ \tau=\tau \text{ in Sequence\_induct2\_including\_0}) \)

apply\( (\text{simp add: lift0\_def}) \)

oops

Count
defs

\[ \text{OclCount\_def} : \text{OclCount} \equiv \text{lift2} (\text{strictify} \ (\lambda x. \text{strictify}(\lambda x. \text{int}(\text{Nat.size}(\text{List.filter}(\text{op}=x) ~\text{Rep\_Sequence\_0 X)))))) \]

ocl\_setup\_op [OclCount]

Its computational characterisation by iterate

lemma count\_by\_iterate:

\[(S::(\tau',\alpha::\bot)\text{Sequence}) \rightarrow \text{count}(\text{a}::(\tau',\alpha)\text{VAL})) = (\text{OclIterate} S (\lambda x::(\tau',\alpha::\bot)\text{VAL}) (y::\tau'\text{Integer}).\text{if} \ ((x = a) \text{ then} \ (y+1) \text{ else} \ y) \text{ endif} ))\]

apply\( (\text{rule ext}) \)

apply\( (\text{rule iterate\_universal}) \)

apply\( (\text{simp\_all}) \)

apply\( (\text{simp\_all add: OclMtSequence\_def OclCount\_def OCL\_Integer.Zero\_ocl\_int\_def OCL\_Integer.\_oocl\_int\_def OclIf\_def OclIsDefined\_def OclUndefined\_def OclStrictEq\_def OclIncluding\_def OclLocalValid\_def plus\_def OclTrue\_def ss\_lifting}) \)

apply\( (\text{frule DEP\_X\_Sequence}) \)

apply\( (\text{auto\ simp: neg\_commute}) \)

done

thm f1\_by\_iterate\_including[OF count\_by\_iterate, simplified]

lemma count\_of\_mtSequence [simp]:

\[ \tau \vdash (\partial (x::(\tau',\alpha)\text{VAL})) \] \implies

\[ (\text{count} x) \tau = 0 \tau \]

by\( (\text{simp add: count\_by\_iterate, ocl\_subst, simp}) \)

thm f1\_by\_iterate\_including[OF count\_by\_iterate, simplified]

thm f1\_by\_iterate\_opt\_including[OF count\_by\_iterate, simplified]

282
B.4. Library

thm f1_by_iterate_distrib_union[OF count_by_iterate, of op +, simplified]
thm f1_by_iterate_distrib_union_opt[OF count_by_iterate, of op +, simplified]

lemma plus_assoc1: \( (X :: \tau Integer) + Y) + Z = X + (Y + Z) \)
  by (rule ext, simp add: plus_def ss_lifting')
lemma plus_assoc2: \( (X :: \tau Integer) + (Y + Z) = Y + (X + Z) \)
  by (rule ext, simp add: plus_def ss_lifting')
lemmas plus_AC = plus_commute plus_assoc1 plus_assoc2

lemma plus_zero': simp]: \( (X :: \tau Integer) + 0 = X \land 0 + (X :: \tau Integer) = X \)
  by (rule conjI, (rule ext, simp add: plus_def ss_lifting' Zero_ocl_int_def)+)

lemma count_of_union [simp]:
  \( ((X :: (\tau, \alpha::bot) Sequence) \cup Y) \rightarrow count (a :: (\tau, \alpha) VAL)) = ((X \rightarrow count a) + (Y \rightarrow count a)) \)
  apply (rule ext, case_tac x \neq a, ocl_hypsubst, simp)
  apply (case_tac x \neq a X, ocl_hypsubst, simp)
  apply (case_tac x \neq a Y, ocl_hypsubst, simp)
  apply (simp add: count_by_iterate)
  apply (ocl_subst)
  apply (simp add: iterate_distrib_union plus_AC if_distrib_strict_alt if_distrib_strict_alt'

lemma includes_by_count_UC[simp]:
  \( S \rightarrow count x > 0 = (x :: (\tau, \alpha) VAL \in (S :: (\tau, \alpha::bot) Sequence)) \)
  apply (auto simp: OclIncludes_def OclCount_def OclGreater_def filter_empty_conv
    OclLess_def OclNot_def OclNot_def Zero_ocl_int_def ss_lifting)
  done

lemma excludes_by_count_UC[simp]:
  \( S \rightarrow count x \leq 0 = \neg ((x :: (\tau, \alpha) VAL \in (S :: (\tau, \alpha::bot) Sequence)) \)
  apply (rule Sequence_sem_cases_ext, simp_all)
  apply (auto simp: OclIncludes_def OclCount_def OclStrictEq_def OclNot_def
    OclNot_def Zero_ocl_int_def ss_lifting filter_empty_conv)
  done

Subsequence
constdefs
Appendix B. Isabelle Theories

\textit{OclSubSequence} :: \(\langle \tau', \alpha::\bot \rangle \text{Sequence}, \tau \text{ Integer}, \tau \text{ Integer}\rangle = \langle \tau', \alpha \rangle \text{ Sequence} \)

\(\text{OclSubSequence} \equiv \text{lift3 (strictify (} \lambda S. \text{strictify (} \lambda \alpha. \text{Abs}_{\text{Sequence}} 0
\begin{align*}
&\text{(_take (nat (\langle u \rangle - T + 1\rangle))}
\end{align*}
\begin{align*}
&\text{(_drop (nat (T - 1\rangle))}
\end{align*}
\begin{align*}
&\langle \text{Rep}_{\text{Sequence}} 0 S \rangle))))))
\)

\textit{ocl_setup_op} [OclSubSequence]

Because the \textit{ocl_setup_op} method doesn’t deal with \textit{lift3} we have to setup the simple properties by ourselves.

\textbf{lemma} OCL\_undef\_1\_OclSubsequence[simp]:
\begin{align*}
\text{OclSubSequence (} \bot::\langle \tau', \alpha::\bot \rangle \text{Sequence}\rangle a b = \bot
\end{align*}
\begin{align*}
\text{by (simp add: OclUndefined_def OclSubSequence_def ss_lifting')}
\end{align*}

\textbf{lemma} OCL\_undef\_2\_OclSubsequence[simp]:
\begin{align*}
\text{OclSubSequence (} S::\langle \tau', \alpha::\bot \rangle \text{Sequence}\rangle \bot b = \bot
\end{align*}
\begin{align*}
\text{by (rule ext, simp add: OclUndefined_def OclSubSequence_def ss_lifting')}
\end{align*}

\textbf{lemma} OCL\_undef\_3\_OclSubsequence[simp]:
\begin{align*}
\text{OclSubSequence (} S::\langle \tau', \alpha::\bot \rangle \text{Sequence}\rangle a \bot = \bot
\end{align*}
\begin{align*}
\text{by (rule ext, simp add: OclUndefined_def OclSubSequence_def ss_lifting')}
\end{align*}

\textbf{lemma} OCL\_cp\_OclSubsequence[simp,intro!]:
\begin{align*}
\llbracket cp \ P; cp \ F; cp \ F' \rrbracket = \Rightarrow cp (\lambda X. \text{OclSubSequence (} (P X)::\langle \tau', \alpha::\bot \rangle \text{Sequence}\rangle (F X) (F' X))
\end{align*}
\begin{align*}
\text{by (simp add: OclSubSequence_def)}
\end{align*}

Higher properties of subsequence

Note: \textit{subsequence} is in our case not undefined except for undefined arguments. To avoid undefinedness is quite favorable in general as it makes proofs simpler and reduces the side conditions to check. But our implementation still fulfills the requirements of the standard.

\textbf{lemma} size\_of\_subsequence:
\begin{align*}
\llbracket \tau \models 1 \leq a; \tau \models a \leq b - 1; \tau \models b \leq \| S \| \rrbracket \Rightarrow
\end{align*}
\begin{align*}
\| (\text{OclSubSequence (} S::\langle \tau', \alpha::\bot \rangle \text{Sequence}\rangle a b) \| \models \tau = (b - a + 1) \tau
\end{align*}
\begin{align*}
\text{apply (rule_tac X=S in Sequence_sem_cases)}
\end{align*}
\begin{align*}
\text{apply (simp_all add: localValid2sem)}
\end{align*}
\begin{align*}
\text{apply (case_tac b \tau \neq \bot, frule neq_commute[THEN iffD1])}
\end{align*}
\begin{align*}
\text{apply (case_tac a \tau \neq \bot, rotate_tac -1, frule neq_commute[THEN iffD1])}
\end{align*}
\begin{align*}
\text{apply (auto simp: OclIncluding_def OclSubSequence_def OclSize_def}
\end{align*}
\begin{align*}
\text{One_ocl_int_def plus_def minus_def ss_lifting'\text{OclLe_def OclLess_def notin_set_take notin_set_drop})}
\end{align*}
\begin{align*}
\text{apply (arith, case_tac ((b \tau) = down), simp_all)}
\end{align*}
\textbf{done}

284
lemma subsequence_mtSequence_conv:
\[ \tau \vdash \emptyset \land \tau \vdash b < a \quad \Rightarrow \\
(\text{OclSubSequence } (S::('\tau', 'a::bot)\text{Sequence}) \ a \ b) \ \tau = [] \ \tau \] 
apply\ (\text{rule_tac } X=S \ \text{in } \text{Sequence}_{\ \text{sem}}_{\ \text{cases}}) 
apply\ (\text{simp}_{\ \text{all}} \ \text{add: } \text{localValidDefined2sem}_{\ \text{DEF}_{\ \text{def}}}) 
apply\ (\text{case_tac } b \ \tau \ \neq \bot, \ \text{frule neq_commute}[\ \text{THEN } \text{iffD1}]) 
apply\ (\text{case_tac } a \ \tau \ \neq \bot, \ \text{rotate_tac } -1, \ \text{frule neq_commute}[\ \text{THEN } \text{iffD1}]) 
apply\ (\text{auto } \text{simp}_{\ \text{add: } \text{OclIncluding}_{\ \text{def}} \ \text{OclSubSequence}_{\ \text{def}} \ \text{OclMtSequence}_{\ \text{def}} \ \text{localValid2sem} \ \text{OclLess}_{\ \text{def}} \ \text{notin_set}_{\ \text{take}} \ \text{notin_set}_{\ \text{drop}} \ \text{ss}_{\ \text{lifting}}]) 
done

InsertAt 
defs

OclInsertAt_{\ \text{def}} 
OclInsertAt \equiv \text{lift3}(\text{strictify}(\lambda S. \text{strictify}(\lambda i. \text{strictify}(\lambda x. \text{Abs}_{\ \text{Sequence}}_0(\text{if } (1 \leq i \land i < (1+\text{int}(\text{Nat.size} \text{Rep}_{\ \text{Sequence}}_0 S))) 
\quad \text{then } (\text{concat} \ [\text{take} \ (\text{op } -i \ 1)] (\text{Rep}_{\ \text{Sequence}}_0 S), [x], \ \text{List.drop} \ (\text{op } -i \ 1)) (\text{Rep}_{\ \text{Sequence}}_0 S)) \ [x], \ \text{List.drop} \ (\text{op } -i \ 1)) (\text{Rep}_{\ \text{Sequence}}_0 S))))) 
\quad \text{else } \bot)))))

ocl_setup_op [OclInsertAt] 
Because the ocl_setup_op method doesn’t deal with \text{lift3} we have to setup the simple properties by ourselves.

lemma OCL_cp_OclInsertAt[simp,intro!]: 
\[ \text{cp } P, \ \text{cp } F, \ \text{cp } F' \quad \Rightarrow \\
\text{cp } (\lambda X. \text{OclInsertAt}(P X)) :: ((\tau, \alpha::\text{bot})\text{Sequence}) \ (F X) :: ((\tau, \alpha)\text{VAL}) \] 
by(simp add: OclInsertAt_{\ \text{def}})

lemma OCL_undef_1_OclInsertAt[simp]: 
OclInsertAt ((\tau::(\tau', \alpha::\text{bot})\text{Sequence}) \ (x::(\tau', \alpha)\text{VAL})) = \bot 
by(simp add: OclUndefined_def OclInsertAt_{\ \text{def}} ss_{\ \text{lifting}})

lemma OCL_undef_2_OclInsertAt[simp]: 
OclInsertAt ((\tau, \alpha::\text{bot})\text{Sequence}) \ i \ (x::(\tau, \alpha)\text{VAL}) = \bot 
by(rule ext, simp add: OclUndefined_def OclInsertAt_{\ \text{def}} ss_{\ \text{lifting}})

lemma OCL_undef_3_OclInsertAt[simp]: 
OclInsertAt ((\tau::(\tau', \alpha::\text{bot})\text{Sequence}) \ i \ (\bot::(\tau', \alpha)\text{VAL}) = \bot 
by(rule ext, simp add: OclUndefined_def OclInsertAt_{\ \text{def}} ss_{\ \text{lifting}})

Higher properties of insertAt 
lemma includes_of_insertAt: 
\[ \tau \vdash \emptyset \land \text{(S::(\tau', \alpha::\text{bot})\text{Sequence})} \land \tau \vdash \emptyset \land \text{(x::(\tau', \alpha)\text{VAL})} \]
Appendix B. Isabelle Theories

\[
\begin{align*}
\tau \vdash 1 \leq i; \tau \vdash i \leq (|S| + 1) & \implies \\
\tau \vdash x \in (OclInsertAt S i x)
\end{align*}
\]

apply (rule_tac X=S in Sequence_sem_cases)
apply (simp_all add: localValidDefined2sem DEF_def)
apply (case_tac i \neq \bot, frule neq_commute [THEN iffD1])
apply (case_tac x \neq \bot, rotate_tac -1, frule neq_commute [THEN iffD1])
apply (auto simp: OclInsertAt_def OclIncludes_def OclLe_def OclSize_def plus_def One_ocl_int_def ss_lifting)
done

lemma size_of_insertAt:
\[
\begin{align*}
\tau \vdash \partial\left( x :: (\tau, \alpha :: \bot) VAL \right); \tau \vdash 1 \leq i; \tau \vdash i \leq (|S| + 1) & \implies \\
\|OclInsertAt S i x\| \tau = (\|S :: (\tau, \alpha) Sequence\| + 1) \tau
\end{align*}
\]

apply (rule_tac X=S in Sequence_sem_cases)
apply (simp_all add: localValidDefined2sem DEF_def)
apply (case_tac i \neq \bot, frule neq_commute [THEN iffD1])
apply (case_tac x \neq \bot, rotate_tac -1, frule neq_commute [THEN iffD1])
apply (auto simp: OclInsertAt_def OclIncludes_def OclLe_def OclSize_def plus_def One_ocl_int_def ss_lifting)
done

at

defs

\[OclAt \text{ def :}
OclAt \equiv \text{lift2(\text{strictify}(\lambda X. \text{strictify}(\lambda x. \text{if } \neg x \leq 0 \lor \neg (\text{Nat.size}(\text{Rep_Sequence_0 X})) < \neg x \text{ then } \bot \text{ else } \neg (\text{Rep_Sequence_0 X}!\text{nat}(\neg x) - 1))))}\]

ocl_setup_op [OclAt]

lemma at_lower_bound:
\[
\begin{align*}
\tau \vdash \partial\left( S :: (\tau, \alpha :: \bot) Sequence \right) i \tau = (\bot :: (\tau, \alpha) VAL) \tau & \implies \\
\tau \vdash \partial\left( (OclAt \left( S :: (\tau, \alpha :: \bot) Sequence \right) i) :: (\tau, \alpha) VAL \right) = (\tau \vdash \partial S)
\end{align*}
\]

B.4. Library

apply (case_tac i \( \tau = \bot \))
apply (simp add: localValid2sem OclLe_def One_ocl_int_def ss_lifting)
apply (rule_tac X = S in Sequence_sem_cases, simp_all add: localValid2sem)
apply (auto simp: OclIsDefined_def OclAt_def OclLe_def OclSize_def ss_lifting)
apply (simp add: in_set_conv_nth)
apply (erule_tac x = \( \lfloor (i \tau) \rfloor - 1 \) in allE, arith)
done

lemma at_of_mtSequence[simp]:
OclAt ([\( \tau, \alpha :: \bot \)] Sequence) i = (\( \bot :: \alpha \) VAL)
by (rule ext, auto simp: OclAt_def OclMtSequence_def)

lemma at_last_of_including:
\[ \tau \models i = (\| S \| + 1) \] \( \Rightarrow \)
OclAt (S;:: Sequence) \( \Rightarrow \) including x i \( \tau \) =
\( x :: (\tau, \alpha :: VAL) \) \( \tau \)
apply (case_tac i \( \tau = \bot \))
apply (simp add: localValid2sem OclStrictEq_def ss_lifting)
apply (case_tac \( \tau \neq \emptyset \) x)
apply (ocl_hypsubst, simp)
apply (rule_tac X = S in Sequence_sem_cases)
apply (simp_all add: localValidDefined2sem DEF_def)
apply (frule neq_commute \[ THEN iffD1 \])
apply (rotate_tac 2, frule neq_commute \[ THEN iffD1 \])
apply (auto simp: OclAt_def OclIncluding_def OclSize_def localValid2sem
OclStrictEq_def plus_def One_ocl_int_def ss_lifting)
done

lemma at_inside_of_including:
\[ \tau \models i \leq \| S \| \] \( \Rightarrow \)
OclAt (S \Rightarrow including \( x :: (\tau, \alpha :: VAL) \) \( \tau \) =
((OclAt S :: (\tau, \alpha :: Sequence) i \( \tau \));:: \( \tau \))
apply (case_tac i \( \tau = \bot \))
apply (simp add: localValid2sem OclLe_def ss_lifting)
apply (rule_tac X = S in Sequence_sem_cases)
apply (simp_all add: localValidDefined2sem DEF_def)
apply (frule neq_commute \[ THEN iffD1 \])
apply (auto simp: OclAt_def OclIncluding_def OclSize_def localValid2sem
OclLe_def plus_def One_ocl_int_def ss_lifting)
apply (simp add: nth_append, arith)
done

lemma at_of_union1:
\[ \tau \models \emptyset \ S2, \tau \models i \leq \| S1 \| \] \( \Rightarrow \)
Appendix B. Isabelle Theories

\[ \text{OclAt} (S_1 \cup S_2) i \tau = \]
\[(\text{OclAt} (S_1; (\tau, \alpha::bot)\text{Sequence}) i \tau);\alpha) \]
apply(rule_tac X=S1 in Sequence_sem_cases)
apply(rule_tac X=S2 in Sequence_sem_cases)
apply(simp_all add: localValidDefined2sem DEF_def)
apply(auto simp: OclAt_def OclUnion_def OclSize_def localValid2sem
OclLe_def plus_def ss_lifting)
apply(simp add: nth_append, arith)
done

lemma at_of_union2:
\[
\llbracket \tau \models \|S_1\| < i; \tau \models i \leq (\|S_1\| + \|S_2\|) \rrbracket \implies
\text{OclAt} (S_1 \cup S_2) i \tau =
\]
apply(rule_tac X=S1 in Sequence_sem_cases)
apply(rule_tac X=S2 in Sequence_sem_cases)
apply(simp_all add: localValidDefined2sem DEF_def)
apply(auto simp: OclAt_def OclUnion_def OclSize_def minus_def localValid2sem
OclLe_def OclLess_def plus_def ss_lifting)
apply(auto simp: nth_append, arith)
done

lemma at_first_of_prepend:
\[
\llbracket \tau \models \partial S \rrbracket \implies
\text{OclAt} (\text{OclPrepend} (S; (\tau, \alpha::bot)\text{Sequence}) x) 1 \tau = (x; (\tau, \alpha)\text{VAL}) \tau
\]
apply(case_tac \tau \models \emptyset x)
apply(ocl_hypsubst, simp)
apply(rule_tac X=S in Sequence_sem_cases)
apply(simp_all add: localValidDefined2sem DEF_def)
apply(ulem_neg_commute[THEN iffD1])
apply(auto simp: OclAt_def OclIncluding_def OclUnion_def OclSize_def localValid2sem
OclMtSequence_def One_ocl_int_def ss_lifting)
done

lemma at_inside_of_prepend:
\[
\llbracket \tau \models \emptyset x; \tau \models i < i \rrbracket \implies
\text{OclAt} (\text{OclPrepend} (S; (\tau, \alpha::bot)\text{Sequence}) x; (\tau, \alpha)\text{VAL}) i \tau =
\]
apply(case_tac \tau \models \emptyset i)
apply(ocl_hypsubst, simp_all)
apply(case_tac \tau \models \emptyset i)
apply(ocl_hypsubst, simp_all)
apply(rule_tac X=S in Sequence_sem_cases)

288
B.4. Library

apply(simp_all add: localValidDefined2sem DEF_def)
apply(erule neq_commute[THEN iffD1])
apply(rotate_tac 2, erule neq_commute[THEN iffD1])
apply(case_tac i τ = ∥S∥ + I < i)
apply(auto simp: OclAt_def OclIncluding_def OclUnion_def OclSize_def localValid2sem
        OclMtSequence_def OclLess_def minus_def plus_def
        One_ocl_int_def ss_lifting)
apply(subgoal_tac ((x τ) ≠ c) = ([x τ] ∩ c))
apply(simp only: nth_append)
apply(subgoal_tac ((nat (∑(i τ) − I) − (Suc 0)) = (nat (−2 + (∑(i τ)))))
apply(auto, arith, arith)
done

lemma at_of_subsequence:
[| τ ⊨ I ⊆ a; τ ⊨ a < b; τ ⊨ b ≤ ∥S∥ |] ⊨
τ ⊨ I ⊆ i; τ ⊨ i ≤ (a − b + I) ⊨
OclAt (OclSubSequence (S::('τ,α::bot)Sequence) a b) i τ =
((OclAt S (a + i − I) τ)::α)
apply(rule_tac X=S in Sequence_sem_cases)
apply(simp_all add: localValid2sem)
apply(case_tac i τ ≠ ⊥, erule neq_commute[THEN iffD1])
apply(case_tac a τ ≠ ⊥, rotate_tac −I, erule neq_commute[THEN iffD1])
apply(case_tac b τ ≠ ⊥, rotate_tac −I, erule neq_commute[THEN iffD1])
apply(simp_all add: OclIncluding_def OclAt_def OclSize_def
        One_ocl_int_def plus_def minus_def ss_lifting
        OclLe_def OclLess_def notin_set_take notin_set_drop)
done

lemma at_of_insertAt_before:
[| τ ⊨ ∅ (x::('τ,α)VAL); τ ⊨ i ≤ (∥S∥ + I); τ ⊨ j < i |] ⊨
OclAt (OclInsertAt (S::('τ,α::bot)Sequence) i x) j τ =
((OclAt S j τ)::α)
apply(case_tac i τ = ∅ S, ocl_hypsubst, simp)
apply(case_tac τ ⊨ ∅ i, ocl_hypsubst, simp)
apply(case_tac τ ⊨ ∅ j, ocl_hypsubst, simp)
apply(simp add: localValidDefined2sem DEF_def both)
apply(clarify, erule frule DEF_X_Sequence', clarify)
apply(case_tac (I ≤ (∑(j τ))))
apply(subgoal_tac ((nat (∑(j τ)) − I) < (length c)))
apply(auto simp: OclInsertAt_def OclAt_def OclLess_def OclSize_def plus_def
        One_ocl_int_def ss_lifting localValid2sem
        OclLe_def notin_set_take notin_set_drop nth_append)
apply(arith, arith)
done

lemma at_of_insertAt:
[| τ ⊨ I ⊆ i; τ ⊨ i ≤ (∥S∥ + I) |] ⊨
OclAt (OclInsertAt (S::('τ,α::bot)Sequence) i (x::('τ,α)VAL)) i τ = x τ

289
Appendix B. Isabelle Theories

apply (case_tac \( \tau = \emptyset \) \( S \))
apply (ocl_hypsubst, simp)
apply (rule_tac \( X=S \) in Sequence_sem_cases)
apply (simp_all add: localValidDefined2sem DEF_def)
apply (case_tac \( i \neq \bot \), frule neq_commute[THEN iffD1])
apply (case_tac \( x \neq \bot \), rotate_tac 1, frule neq_commute[THEN iffD1])
apply (auto simp: OclInsertAt_def OclAt_def OclLe_def OclSize_def plus_def
  One_ocl_int_def ss_lifting localValid2sem
  notin_set_take notin_set_drop
  nth_append_length)
done

lemma at_of_insertAt_after: 
  \[[ \tau = \emptyset \) \( x:(\tau,\alpha){\mathrm{VAL}} \); \tau \ni I \leq i \); \tau \ni i \leq (|S| + I); \tau \ni i < j \] \implies
OclAt (OclInsertAt (S:(\tau,\alpha::bot)Sequence) i x) j \( \tau = ((\text{OclAt} S \ (j - I) \ \tau):::\alpha) \)
apply (case_tac \( \tau = \emptyset \) \( S, ocl_hypsubst, simp)\)
apply (case_tac \( \tau = \emptyset \) \( i, ocl_hypsubst, simp)\)
apply (case_tac \( \tau = \emptyset \) \( j, ocl_hypsubst, simp)\)
apply (simp add: localValidDefined2sem DEF_def_both)
apply (clarify, frule DEF_X_Sequence', clarify)
apply (subgoal_tac \( \text{min} (\text{length} c) \ (\text{nat} (\lceil (i \ \tau) \rceil - I)) = \text{nat} (\lceil (i \ \tau) \rceil - I) \))
apply (case_tac \( (\text{int} (\text{length} c) + I) < (\lceil (j \ \tau) \rceil) \))
apply (auto simp: OclInsertAt_def OclAt_def OclLess_def OclSize_def plus_def
  One_ocl_int_def ss_lifting localValid2sem
  minus_def
  OclLe_def
  notin_set_take
  notin_set_drop
  nth_append_length)
apply (thin_tac \( \?P \neq \?Q \) \( \text{thin_tac} \ ?P = \?Q\))
apply (thin_tac \( \?P \neq \?Q \) \( \text{thin_tac} \ ?P = \?Q\))
apply (thin_tac \( \?P \neq \?Q \) \( \text{thin_tac} \ ?P = \?Q \) \( \text{thin_tac} \ ?P \neq \?Q\))
apply (subgoal_tac \( \text{min} (\text{length} c) \ (\text{nat} (\lceil (i \ \tau) \rceil - I)) = (\text{nat} (\lceil (i \ \tau) \rceil - I)) = \text{Suc} (\text{nat} (\lceil (j \ \tau) \rceil - 2)) - (\text{nat} (\lceil (i \ \tau) \rceil - I)))\))
apply (simp)
apply (subgoal_tac \( \text{nth_drop} \ ?P = \?Q \) \( \text{arith} \))
apply (subgoal_tac \( (\text{nth} \ ?P = \?Q \) \( \text{thin_tac} \ ?P = \?Q \) \( \text{thin_tac} \ ?P \neq \?Q\))
apply (subgoal_tac \( (\text{nat} (\lceil (i \ \tau) \rceil - I)) + (\text{nat} (\lceil (j \ \tau) \rceil - 2)) - (\text{nat} (\lceil (i \ \tau) \rceil - I)))\) =
(\text{nat} (-2 + (\lceil (j \ \tau) \rceil))\))
apply (simp)
apply (rotate_tac 2, thin_tac \?P, thin_tac \?P, thin_tac \?P, arith)
apply (rotate_tac 2, thin_tac \?P, thin_tac \?P, arith)
apply (thin_tac \( \?P \neq \?Q \) \( \text{thin_tac} \ ?P = \?Q \) \( \text{thin_tac} \ ?P \neq \?Q\))
apply (rotate_tac 2, thin_tac \?P, thin_tac \?P, arith)
done

IndexOf

defs

290
\[
\begin{align*}
\text{OclIndexOf}\_\text{def} : \\
\text{OclIndexOf} & \equiv \text{lift}2(\text{strictify}(\lambda X. \text{strictify}(\lambda x. \text{if } x \in \text{set } [\text{Rep\_Sequence}_0 X] \text{ then } \text{init}(\text{length (takeWhile } (\lambda y. y \neq x) \text{ Rep\_Sequence}_0 X) + 1) \text{ else } \bot)))
\end{align*}
\]

\text{ocl\_setup\_op} [\text{OclIndexOf}]

\text{lemma} indexOf\_excludes[simp]:
\[\tau \vdash \neg ((x:('\tau,\alpha) \text{ VAL}) \in (S:('\tau,\alpha::\text{bot}) \text{ Sequence})) \implies \text{OclIndexOf} S x \tau = \bot\]
apply(rule_tac X=S in \text{Sequence\_sem\_cases}, simp\_all)
apply(case_tac x \tau = \bot)
apply(simp\_all add: OclIndexOf\_def OclIncludes\_def OclNot\_def o\_def
\text{OclUndefined\_def ss\_lifting'} localValid2sem)
done

\text{lemma}notin\_conv\_all\_noteq: a \notin S = (\forall x \in S. x \neq a)
by (auto)

\text{lemma} in\_set\_conv\_first\_decomp:
(x \in \text{set} ys) = (\exists z s z s = z y x \neq z s x \notin \text{set} ys)
apply(auto, erule rev_mp)
apply(rule length\_induct, auto)
apply(frule in\_set\_conv\_decomp[THEN iffD1], clarify)
apply(case_tac x \in \text{set} ys)
applyerule_tac x = ys in allE, auto
apply(rule_tac x = ysa in exI, simp)
done

In the standard self->at(i) = obj is written where it should be written like
self->at(result) = obj. Furthermore the standard doesn’t specify that the
returned index is the index of the first object that is equal to the searched one. This
disables the combined use of indexOf and subsequence to visit all occurrences of a
specific object.

\text{lemma} at\_indexOf:
\[\tau \vdash (x:('\tau,\alpha) \text{ VAL}) \in (S:('\tau,\alpha::\text{bot}) \text{ Sequence}) \implies \text{OclAt} S (\text{OclIndexOf} S x) \tau = x \tau\]
apply(rule_tac X=S in \text{Sequence\_sem\_cases}, simp\_all)
apply(case_tac x \tau \neq \bot)
apply(frule neq\_commute[THEN iffD1])
apply(simp\_all add: OclIndexOf\_def OclIncludes\_def OclNot\_def OclAt\_def
\text{OclUndefined\_def ss\_lifting'} localValid2sem
o\_def)
apply(frule in\_set\_conv\_first\_decomp[THEN iffD1])
apply(clarify, simp add: notin\_conv\_all\_noteq takeWhile\_tail)
done
Appendix B. Isabelle Theories

First and Last
defs

OclFirst_def :
OclFirst ≡ lift1 (strictify (λS. if ⌦ Rep_Sequence_0 S = [] then ⊥ else hd (⌜ Rep_Sequence_0 S ⌝)))

OclLast_def :
OclLast ≡ lift1 (strictify (λS. if ⌦ Rep_Sequence_0 S = [] then ⊥ else List.last (⌜ Rep_Sequence_0 S ⌝)))

ocl_setup_op [OclFirst, OclLast]

lemma first_at_UC :
OclFirst (S::(′τ,′α::bot)Sequence) = ((OclAt S 1)::′τ,′α :: bot)VAL
apply(rule Sequence_sem_cases, simp_all)
done

lemma first_of_mtSequence[simp]:
OclFirst ([]::(′τ,′α::bot)Sequence) = (⊥::′τ,′α :: bot)VAL
by(simp add: first_at_UC)

lemma first_empty_of_including:
[ τ = (S = []) ]] ⇒
OclFirst ((S::(′τ,′α::bot)Sequence)→including (x::′τ,′α :: bot)VAL) τ =
((x τ)::′α)
apply(rule_tac X=S in Sequence_sem_cases)
apply(simp_all, localValid2sem)
apply(case_tac c)
apply(auto simp: OclIncluding_def OclFirst_def OclNot_def OclStrictEq_def
OclMtSequence_def ss_lifting _ neq_commute)
done

Note: strange writing because of normal form of emptiness test

lemma first_not_empty_of_including:
[ τ = ⌈ x : τ ⌉ | τ = ~ (S = []) ] ⇒
OclFirst ((S::(′τ,′α::bot)Sequence)→including (x::′τ,′α :: bot)VAL) τ =
((OclFirst S τ)::′α)
apply(rule_tac X=S in Sequence_sem_cases)
apply(simp_all, localValidDefined2sem DEF_def)
apply(case_tac c)
apply(auto simp: OclIncluding_def OclFirst_def OclNot_def OclStrictEq_def
OclMtSequence_def ss_lifting _ neq_commute localValid2sem)
done

292
lemma last_at_UC:
OclLast (S::('τ,'α::bot)Sequence) = ((OclAt S ||S)||::('τ,'α)VAL)
apply(rule Sequence_sem_cases_ext, simp_all)
apply(auto simp: OclLast_def OclAt_def OclSize_def
Zero_ocl_int_def One_ocl_int_def
ss_lifting' last_conv_nth neq_Nil_conv)
done

lemma last_of_mtSequence[simp]:
OclLast ( ||::('τ,'α::bot)Sequence) = (⊥::('τ,'α)VAL)
by(simp add: last_at_UC)

lemma last_empty_of_including:
[[ τ ⊨ ∅ S ]] ⊢
OclLast ( (S::('τ,'α::bot)Sequence) ->including (x:('τ,'α)VAL) ) τ = ((x τ)::'α)
apply(rule_tac X=S in Sequence_sem_cases)
apply(simp_all add: localValidDefined2sem DEF_def)
apply(simp add: OclIncluding_def OclLast_def
ss_lifting' neq_commute)
done

Forall
defs

OclForAll_def: OclForAll ≡ (lift2' lift_arg0 lift_arg1) (strictify( λ S. (λ P. if ∀ x ∈ set 'Rep_Sequence_0 S. P x = 'True,
then 'True,
else if ∃ x ∈ set 'Rep_Sequence_0 S. P x = 'False,
then 'False,
else '⊥))))

ocl_setup_op [OclForAll]

Its computation characterisation by iterate
lemma forall_by_iterate:
(∀ x ∈ (S::('τ,'α::bot)Sequence) . (P::('τ,'α::bot)VAL ⇒ 'τ Boolean) x) =
(S->iterate(x;y= T | (P x) ∧ y))
apply(rule ext)
apply(rule iterate_universal_0, simp_all)
apply(simp add: cp_by_cpify OclForAll_def lift0_def)
apply(simp_all add: OclForAll_def OclUndefined_def OclMtSequence_def)
Appendix B. Isabelle Theories

```
OclTrue_def localValidDefined2sem ss_lifting')
apply(frule DEF_X_Sequence', clarify)
apply(frule neq_commute[THEN iffD1])
apply(rotate_tac 1, frule neq_commute[THEN iffD1])
apply(rule_tac x=P (λs. (xa x)) x in boolean_cases_sem)
apply(simp_all add: OclIncluding_def OclAnd_def ss_lifting) done

Derived properties from the relation to iterate
lemma OCL_undef_2_OclForAll[simp]:
[τ ⊨ ¬ ((S::(τ::bot)Sequence) ⊦ [])]  ⟹
(∀ x ∈ S. (λ x: (τ::bot)VAL. ⊥) x) τ = ⊥ τ
apply(rule_tac X=S in Sequence_sem_cases, simp_all)
apply(simp_all add: OclForAll_def OclUndefined_def OclMtSequence_def
OclStrictEq_def OclNot_def localValid2sem
OclTrue_def localValidDefined2sem ss_lifting')
done

lemmas forall_of_mtSequence[simp] = f1_by_iterate_mtSequence[OF forall_by_iterate]

lemma forall_including[simp]:
[τ ⊨ ∂ ((X::(τ::bot)Sequence); τ ⊨ ∂ ((a::(τ::bot))VAL); cp P)]  ⟹
(∀ x ∈ (X→including a) • P (x::(τ::bot))) τ = (((P a) ∧ (∀ x ∈ X • P x)) τ
by(simp add: forall_by_iterate iterate_of_including)

thm f1_by_iterate_distrib_union[OF forall_by_iterate, of OclAnd, simplified]

lemma forall_union[simp]:
[τ ⊨ ∂ ((X::(τ::bot)Sequence); τ ⊨ ∂ Y)]  ⟹
(∀ x ∈ (X ∪ Y) • P (x::(τ::bot))) τ = (((∀ x ∈ X • P x) ∧ (∀ x ∈ Y • P x)) τ
apply(simp add: forall_by_iterate)
apply(rule iterate_distrib_union)
apply(simp_all add: OCL_logic_ACI)
done

Introduction rules for ForAll
lemma ForAll:
assumes defS : τ ⊨ ∂((S::(τ::bot)Sequence)
and allP : ∃ x. τ ⊨ x ∈ S  ⟹ τ ⊨ P x
shows τ ⊨ (∀ x ∈ S • P(x::'a::bot))
apply(insert defS)
apply(rule_tac X=S in Sequence_sem_cases)
apply(simp_all add: boolean_cases_sem DEF_def)
apply(subgoal_tac (∀ x ∈ set c. P (λτ. x) τ = (True))
apply(auto simp: OclForAll_def localValid2sem ss_lifting')

294
```
B.4. Library

apply \(\text{rule_tac } \tau \mathbf{I} = \tau \text{ in } \text{localValid2sem\{THEN iffD\}, rule allP}\)
apply (auto simp: localValid2sem OclIncludes_def ss_lifting )
done

lemma notForAllI:
assumes isin : \(\tau \models x \in (S::(\tau,(\alpha::\text{bot}))\text{Sequence})\)
and not : \(\tau \models \neg(P \ x)\)
and cpP : cp P
shows \(\tau \models \neg(\forall x \in S \cdot P(x; \tau \Rightarrow \alpha::\text{bot}))\)
apply (insert isin not cpP)
apply (rule_tac X = S in Sequence_sem_cases)
apply (simp_all add: localValidDefined2sem DEF_def)
apply (rule_tac x = xa in bexI, simp_all)
prefer 2 apply (rule_tac x = x \(\tau\) in bexI, simp_all)
apply (rule_tac \(\tau\) = \(\tau\) and P = P in cp_charn, simp add: cpP, rule cpP)

apply (rule_tac \(\tau\) = \(\tau\) and P = P in cp_charn, simp add: cpP)

Appendix B. Isabelle Theories

apply (erule_tac Q = P (λτ. xa) τ = (False in contrapos_pp)
apply (rule_tac τ1 = τ in localValidToNotFalse2sem [THEN iffD1]) back
apply (rule allP)
apply (simp add: includes_charn2 DEF_def)
done

lemma isdefForAllI2:
assumes exNot1: τ ⊨ x ∈ (S::(τ, (α::bot))Sequence)
and exNot2: τ ⊨ ¬ (P x)
and cpP : cp P
shows τ ⊨ (∀ x ∈ S. P(x; τ ⇒ α::bot))
apply (insert exNot1 exNot2 cpP)
apply (drule_tac P = P in notForAllI,
simp_all add: isDefined_if_invalid)
done

lemmas isdefForAllI3 = ForAllI[THEN isDefined_if_valid]

Exists

Alternative definition for the use with ocl_setup_op
lemma OclExists_alt_def:
OclExists ≡ (lift2' lift_arg0 lift_arg1) (strictify(λ S. (λ P.
  if ∃ x ∈ set Rep_Sequence_0 S. P x = True
  then True
  else if ∀ x ∈ set Rep_Sequence_0 S. P x = False
  then False
  else ⊥))
apply (rule eq_reflection, (rule ext)+)
apply (simp add: OclExists_def forall_by_iterate
apply (auto)
oops

The relation to iterate

lemma exists_by_iterate:
(∃ x ∈ (source::(τ,α::bot)Sequence) • (P::(τ,α::bot)VAL ⇒ τ Boolean) x) =
(source ->iterate(x;y= F | (P x) ∨ y))
apply (simp add: OclExists_def forall_by_iterate)
apply (rule ext, rule iterate_fusion_0, simp_all)
apply (simp add: OclNot_def OclOr_def OclAnd_def DEF_def
lift0_def lift1_def lift2_def strictify_def)
done

Derived properties from the relation to iterate
lemma OCL_cp_OclExists [simp,introl]:
[ ∃ x. cp (P x); cp S ]
B.4. Library

\[ \Rightarrow cp(\lambda X. \exists Y \in ((\mathcal{S} X):((\tau, \alpha)::\text{bot})\text{Sequence}) \cdot P (Y:(\tau, \alpha)\text{VAL}) X) \]
by(simp add: exists_by_iterate)

lemmas exists_undef [simp] = f1_by_iterate_undef [OF exists_by_iterate]
lemmas exists_empty [simp] = f1_by_iterate_mtSequence [OF exists_by_iterate]
thm f1_by_iterate_including [OF exists_by_iterate, simplified]
thm f1_by_iterate_opt_including [OF exists_by_iterate, simplified]

thm f1_by_iterate_undef [OF exists_by_iterate]
thm f1_by_iterate_mtSequence [OF exists_by_iterate]
thm f1_by_iterate_including [OF exists_by_iterate, simplified]
thm f1_by_iterate_opt_including [OF exists_by_iterate, simplified]

Introduction rules for exist

lemma ExistsI:
assumes isin : \( \tau \vdash x \in (\mathcal{S}:(\tau, (\alpha::\text{bot})\text{Sequence}) \]
and not : \( \tau \not\vdash (P x) \]
and cpP : \( cp P \)
shows \( \tau \vdash (\exists x \in \mathcal{S} /s\sqsubseteq\text{P}(x::\tau \Rightarrow \tau ::\alpha ::\text{bot})) \]
by (auto simp: OclExists_def intro!: notForAllI isin not cpP)

lemma NotExistsI:
assumes defS : \( \tau \equiv \partial((\mathcal{S}:(\tau, (\alpha::\text{bot})\text{Sequence}) \]
and allP : \( \forall x. \tau \vdash x \in \mathcal{S} \Rightarrow \tau \vdash \neg P x \)
shows \( \tau \vdash \neg(\exists x \in \mathcal{S} /\neg\text{P}(x::\tau \Rightarrow \tau ::\alpha::\text{bot})) \]
by (auto simp: OclExists_def intro!: ForAllI defS allP)

IncludesAll

defs

\[ OclIncludesAll \_ def. OclIncludesAll \equiv \text{tiff}(\lambda X. \text{strictify}(\lambda Y. \text{set} \text{Rep\_Sequence}\_0 Y \subseteq \text{set} \text{Rep\_Sequence}\_0 X)) \]

ocl_setup_op [OclIncludesAll]

Its computational characterisation by iterate

This formulation should be replaced by one with a strict body to simplify reasoning over it.

lemma includesAll_by_iterate:
OclIncludesAll X (Y:(\tau, \alpha::\text{bot})\text{Sequence}) = \( (Y \rightarrow\text{iterate}(x,y => \text{if} (\partial X) \text{ then } Y \text{ else } \bot \text{ endif} | ((x:(\tau, \alpha)\text{VAL}) \in X) \land y)) \]
apply(rule ext)
Appendix B. Isabelle Theories

apply (rule iterate_universal_0, simp_all)
apply (simp add: OclIncludesAll_def OclMtSequence_def OclIsDefined_def OclIf_def OclTrue_def OclUndefined_def ss_lifting)
apply (frule DEF_X_Sequence, clarify)
apply (rotate_tac 1, frule neq_commute[THEN iffD1])
apply (simp add: localValidDefined2sem DEF_def)
apply (frule DEF_X_Sequence, clarify)
apply (rotate_tac 1, frule neq_commute[THEN iffD1])
apply (simp add: OclIncludesAll_def OclIncluding_def OclIncludes_def OclSand_def ss_lifting)
done

thm f1_by_iterate_undef_1 [OF includesAll_by_iterate]
lemmas includesAll_empty [simp] = f1_by_iterate_mtSequence [OF includesAll_by_iterate]
lemmas includesAll_including [simp] = f1_by_iterate_opt_including [OF includesAll_by_iterate, simplified]

thm includesAll_empty includesAll_including

lemma includesAll_union [simp]:
  \(\text{OclIncludesAll } Z \ (X : (\tau, \alpha::\bot)\text{Sequence)} \cup Y) = \)
  \(\text{OclIncludesAll } Z X \land \text{OclIncludesAll } Z Y\)
apply (rule ext)
apply (case_tac x, ocl_hypsubst, simp)
apply (case_tac x, ocl_hypsubst, simp)
apply (case_tac x, ocl_hypsubst, simp)
apply (subst f1_by_iterate_distrib_union[OF includesAll_by_iterate])
apply (simp_all)
apply (ocl_subst, simp)
done

lemma includesAll_by_forall:
  \[ \tau = \partial (X : (\tau, \alpha::\bot)\text{Sequence}) \] \[\Rightarrow\]
  \(\text{OclIncludesAll } X Y = (\forall a \in Y . ((a: (\tau, \alpha)\VAL) \in X)) \tau\)
apply (rule_tac X=X in Sequence_sem_cases)
apply (rule_tac X=Y in Sequence_sem_cases)
apply (simp_all add: localValidDefined2sem DEF_def)
apply (auto simp: OCL_Sequence OclForAll_def OCL_Sequence OclStrongEq_def)
done

ExcludesAll
defs

OclExcludesAll_def:
OclExcludesAll \equiv \text{lift2 (strictify } (\lambda X. \text{strictify } (\lambda Y. \text{.)})}
\[ \text{set } \text{Rep\_Sequence\_0 } X \cap \text{ set } \text{Rep\_Sequence\_0 } Y = \{ \} \]

**ocl\_setup\_op** [OclExcludesAll]

**Its computational characterisation by iterate**

This formulation should be replaced by one with a strict body to simplify reasoning over it.

**lemma excludesAll\_by\_iterate:**

\[
\text{OclExcludesAll } X (Y:\{\tau, \alpha:\text{bot}\}\text{Sequence}) = \\
(Y \rightarrow \text{iterate}(x, y = \begin{cases} \partial X & \text{if } \emptyset X \text{ then } \top \text{ else } \bot \end{cases} | ((x:\{\tau, \alpha\}\text{VAL}) \not\in X) \land y))
\]

apply(rule ext)
apply(rule iterate\_universal\_0, simp\_all)
apply(simp add: OclExcludesAll\_def OclMtSequence\_def OclIsDefined\_def OclIf\_def OclTrue\_def OclUndefined\_def ss\_lifting)
apply(simp add: localValidDefined2sem DEF\_def)
apply(frule DEF\_X\_Sequence', clarify)
apply(rotate_tac 1, frule neq\_commute[THEN iffD1])
apply(simp add: OclExcludesAll\_def OclIncluding\_def OclIncludes\_def OclSand\_def OclNot\_def ss\_lifting)
done

**thm f1\_by\_iterate\_undef\_l[OF excludesAll\_by\_iterate]**

**lemmas excludesAll\_empty\_simp[simp] = f1\_by\_iterate\_mtSequence[OF excludesAll\_by\_iterate]**

**lemmas excludesAll\_including\_simp[simp] = f1\_by\_iterate\_opt\_including[OF excludesAll\_by\_iterate, simplified]**

**thm excludesAll\_empty excludesAll\_including**

**lemma excludesAll\_union\_simp[Ocl]:**

\[
(\text{OclExcludesAll } Z ((X:\{\tau, \alpha:\text{bot}\}\text{Sequence}) \cup Y)) = \\
(\text{OclExcludesAll } Z X) \land (\text{OclExcludesAll } Z Y)
\]

apply(rule ext)
aply(case_tac \(x \not\in \emptyset X\), ocl\_hyps\_subst, simp)
aply(case_tac \(x \not\in \emptyset Y\), ocl\_hyps\_subst, simp)
aply(case_tac \(x \not\in \emptyset Z\), ocl\_hyps\_subst, simp)
aply(subst f1\_by\_iterate\_distrib\_union[OF excludesAll\_by\_iterate])
aply(simp\_all)
aply(ocl\_subst, simp)
aply(auto simp: OclIncludesAll\_def OclSand\_def OclIncludes\_def OclTrue\_def localValidDefined2sem ss\_lifting)
done

**lemma excludesAll\_by\_forall[\(\forall\tau \vdash \partial(X:\{\tau, \alpha:\text{bot}\}\text{Sequence})\)]**
Appendix B. Isabelle Theories

(OclExcludesAll X Y) τ = (∀ a ∈ Y . ((a::('τ,'α)VAL) \notin X)) τ
apply(rule_tac X=X in Sequence_sem_cases)
apply(rule_tac X=Y in Sequence_sem_cases)
apply(simp_all add: localValidDefined2sem DEF_def)
apply(auto simp: OCL_Sequence.OclForAll_def OCL_Sequence.OclExcludesAll_def OclStrongEq_def)
apply(rule_tac X=X in Sequence_sem_cases)
done

OCL_Sequence.OclIncludes_def ss_lifting

Its computational characterisation by iterate

lemma select_by_iterate:
(∀ x . (source::('τ,'α::bot) Sequence) | ((P::('τ,'α::bot)VAL ⇒ 'τ Boolean) x)) =
(source \rightarrow iterate(iterator; result = [] | if (P iterator) then (OclIncluding result iterator) else (result) endif))
apply(rule ext, rule iterate_universal_0, simp_all)
apply(simp add: OclSelect_def cp_by_cpify lift0_def)
apply(simp_all add: OclUndefined_def OclMtSequence_def OclSelect_def
localValidDefined2sem ss_lifting')
apply(frule DEF_X_Sequence', clarify)
apply(frule neq_commute[THEN iffD1])
apply(rotate_tac 1, frule neq_commute[THEN iffD1])
apply(rule_tac x=P (λ s. (xa x)) x in boolean_cases_sem)
apply(simp_all add: OclIf_def OclIncluding_def ss_lifting')
done

Properties derived from the correspondence to iterate
lemmas select_of_mtSequence [simp] = f1_by_iterate_mtSequence[OF select_by_iterate]

lemma select_including [simp]:
[ τ \in \partial (x::('τ,'α::bot)VAL); cp P ] \Rightarrow
OclSelect (S::('τ,'α)Sequence) \rightarrow including x) P τ =
(if (P x) then (OclSelect S P) \rightarrow including x else (OclSelect S P) endif) τ
apply(case_tac τ \in \partial S, ocl_hypsubst, simp)
apply(simp add: f1_by_iterate_including[OF select_by_iterate, simplified])
done

lemma select_including_strict [simp]:

300
\( P \perp = \perp; cp \perp = \Rightarrow \) 

\[ \text{OclSelect} \left( \text{S} \left( \text{\': \tau, \alpha::bot} \right) \text{Sequence} \rightarrow \text{including} \left( \text{x::\': \tau, \alpha::bot} \right) \text{VAL} \right) \] 

\( \text{if} \left( \text{P} \text{x} \right) \text{then} \left( \text{OclSelect} \; \text{S} \; \text{P} \right) \rightarrow \text{including} \; \left( \text{x} \right) \; \text{else} \left( \text{OclSelect} \; \text{S} \; \text{P} \right) \) \text{endif} \) \\( \tau \) 

\( \text{by} \left( \text{simp add: \text{f1_by_iterate_opt_including(OF select_by_iterate, simplified)} } \right) \)

**lemma** \text{select_union} \left[ \text{simp} \right]: 
\( \left( \text{OclSelect} \left( \left( \text{X} \left( \text{\': \tau, \alpha::bot} \right) \text{Sequence} \right) \cup \left( \text{Y} \left( \text{\': \tau, \alpha::bot} \right) \text{Sequence} \right) \right) \left( \text{P} \left( \text{\': \tau, \alpha::bot} \right) \text{VAL} \Rightarrow \text{\': \tau} \text{Boolean} \right) \right) \) 

\( \left( \text{OclSelect} \; \text{X} \; \text{P} \right) \cup \left( \text{OclSelect} \; \text{Y} \; \text{P} \right) \) 

\( \text{apply} \left( \text{subst f1_by_iterate_distrib_union_opt(OF select_by_iterate, simp_all)} \right) \)

\( \text{apply} \left( \text{simp_all} \right) \)

\( \text{apply} \left( \text{rule_tac \text{X}=B in Sequence_sem_cases_ext, simp_all} \right) \)

\( \text{apply} \left( \text{rule_tac \text{X}=y in Sequence_sem_cases, simp_all} \right) \)

\( \text{apply} \left( \text{case_tac \text{x \tau \neq \perp, drule neq_commute[THEN iffD1]} } \right) \)

\( \text{apply} \left( \text{simp_all add: \text{OclIf_def OclUnion_def OclIncluding_def ss_lifting} } \right) \)

\( \text{done} \)

**lemma** \text{iterate_of_select}: 
\( \left( \text{assumes \text{undef_1_Q}} : \left( \text{\': \tau, \alpha::bot} \right) \text{VAL} \Rightarrow \text{\': \tau \text{Boolean}} \right) \) 

\( \left( \text{OclIf_def OclUnion_def OclIncluding_def ss_lifting} \right) \)

\( \text{shows} \left( \text{OclIterate} \; \left( \text{OclSelect} \; \text{\left( \text{\': \tau, \alpha::bot} \right) Sequence} \right) \; \text{P} \; \text{Q} \; \text{A} \right) \) 

\( \left( \text{apply\left( \text{insert \text{cp1_Q cp2_Q cp_P undef_1_P undef_2_Q undef_1_Q}} \right) } \right) \)

\( \left( \text{apply\left( \text{rule \text{ext}} \right) } \right) \)

\( \left( \text{apply\left( \text{rule iterate_universal, simp_all add: select_by_iterate} \right) } \right) \)

\( \left( \text{apply\left( \text{subst iterate_opt_of_including, simp_all} \right) } \right) \)

\( \left( \text{apply\left( \text{rule_tac \text{P}=\lambda \text{t}. \text{t x = ?Q in subst}} \right) } \right) \)

\( \left( \text{apply\left( \text{rule sym[OF if_distrib_strict_all]} \right) } \right) \)

\( \left( \text{apply\left( \text{simp_all add: iterate_opt_of_including} \right) } \right) \)

\( \text{done} \)

**lemma** \text{select_of_select}: 
\( \left( \text{assumes \text{undef_1_P}} : \text{P} \; \text{\perp = \perp} \right) \)

\( \left( \text{and \text{undef_1_Q}} : \text{Q} \; \text{\perp = \perp} \right) \)

\( \left( \text{and \text{cp_P}} \right) \)

\( \left( \text{and \text{cp_Q}} \right) \)

\( \text{shows} \left( \text{OclSelect} \; \left( \text{OclSelect} \; \left( \text{\': \tau, \alpha::bot} \right) \text{Sequence} \right) \; \text{P} \; \text{\text{\': \tau} \text{Boolean}} \right) \) 

\( \left( \text{Q} \; \left( \text{\': \tau, \alpha::bot} \right) \text{VAL} \Rightarrow \text{\': \tau} \text{Boolean} \right) \) 

\( \left( \text{by\left( \text{simp add: select_by_iterate iterate_of_select[simplified select_by_iterate]} \right) } \right) \)

\( \left( \text{undef_1_Q undef_1_P cp_P cp_Q if_contract} \right) \)

\( \text{done} \)

**Reject**

defs
Appendix B. Isabelle Theories

\[ \text{OclReject}_\text{def} \triangleq (\text{lift2}' \text{lift}_\text{arg0} \text{lift}_\text{arg1}) (\text{strictify} (\lambda S P. \text{if } \forall x \in \text{set} \Repsquence_0 S. \text{DEF} (P x) \text{ then } \text{AbsSequence}_0 (\lambda e. x = (\text{False}))) (\text{RepSequence}_0 S).) \]

\[ \text{ocl_setup_op \{OclReject\}} \]

\[ \text{lemma reject_by_select: } \]

\[ \text{OclReject } S \ P = (x : (S :: (\tau', \alpha :: bot) \text{Sequence}) | (\neg (P :: (\tau', \alpha :: bot) \text{VAL} \Rightarrow 'tau \text{ Boolean})))) \]

\[ \text{apply(rule ext, auto simp: OclReject_def OclNot_def OclSelect_def ss_lifting') } \]

\[ \text{apply(subgoal_tac (\lambda xa. ((P (\lambda s. xa) x) \neq False)))) = (\lambda xa. (((P (\lambda s. xa) x) \neq down) \wedge (((P (\lambda s. xa) x) \neq down) \rightarrow (\neg (P (\lambda s. xa) x))))) ) \]

\[ \text{apply(simp, rule ext) } \]

\[ \text{apply(}\text{rule_tac x = P (\lambda s. xa) x in boolean_cases_sem, simp_all)} \]

\[ \text{done} \]

\[ \text{lemma reject_by_iterate: } \]

\[ \text{OclReject (source :: (\tau', \alpha :: bot) \text{Sequence}) (P :: (\tau', \alpha :: bot) \text{VAL} \Rightarrow 'tau \text{ Boolean}) = } \]

\[ \text{(source } \rightarrow \text{iterate(iterator; result = [] | } \}

\[ \text{if } (\neg P \text{ iterator) then (OclIncluding result iterator) else (result endif ))} \]

\[ \text{by(simp add: reject_by_select select_by_iterate)} \]

\[ \text{Properties derived from the correspondence to iterate } \]

\[ \text{lemmas reject_of_union [simp] = f1_by_iterate_union [OF reject_by_iterate]} \]

\[ \text{Excluding } \]

\[ \text{defs } \]

\[ \text{OclExcluding}_\text{def} : \]

\[ \text{OclExcluding } \triangleq \text{lift2'(strictify}(\lambda X. \text{strictify} (\lambda x. \text{AbsSequence}_0 \text{List.filter} (\lambda e. x \neq e) (\Repsquence_0 X))).) \]

\[ \text{ocl_setup_op \{OclExcluding\}} \]

\[ \text{Its computational characteristic by iterate } \]

\[ \text{lemma excluding_by_iterate: } \]

\[ \text{(OclExcluding (S :: (\tau', \alpha :: bot) \text{Sequence}) (x :: (\tau', \alpha) \text{VAL})) = } \]

\[ \text{OclIterate S (\lambda a :: (\tau', \alpha) \text{VAL}) y. if (a \equiv x) then y else y } \rightarrow \text{including a endif} \]

\[ \text{(if (x \equiv a) then [] else down endif)} \]

302
apply(rule ext)
apply(rule iterate_universal, simp_all add: localValidDefined2sem DEF_def)
apply(simp add: OclMtSequence_def OclExcluding_def ss_lifting'
OclIf_def OclIsDefined_def OclUndefined_def)
apply(case_tac xa ⊭∂x)
apply(ocl_hypsubst, simp_all add: localValidDefined2sem)
apply(case_tac xa ⊭∂X)
apply(case_tac xa ⊭∂Y)
apply(simp add: OclExcluding_def OclIterate_def OclIf_def OclIncluding_def OclStrictEq_def ss_lifting'
)
done

lemmas excluding_empty [simp] = f1_by_iterate_opt_including[OF excluding_by_iterate, simplified]

lemma excluding_union [simp]:
((x::(τ,α::bot)Sequence) ∪ Y)→excluding x =
(X→excluding x) ∪ (Y→excluding (x::(τ,α)VAL))
apply(rule ext, case_tac xa ⊭∂x, ocl_hypsubst, simp)
apply(case_tac xa ⊭∂X, ocl_hypsubst, simp)
apply(case_tac xa ⊭∂Y, ocl_hypsubst, simp)
apply(simp add: localValidDefined2sem DEF_def)
apply(frule DEF_X_Sequence', clarify)
apply(rotate_tac 2, frule DEF_X_Sequence', clarify)
apply(simp add: OclExcluding_def OclUnion_def ss_lifting'
)
done

thm excluding_empty excluding_including excluding_union

Further properties of excluding

lemma includes_of_excluding:
[ τ ⊨ ∂(x::(τ,α)VAL); τ ⊨ ∂(S::(τ,α::bot)Sequence) ] =
(x ∈ (S→excluding x)) τ = F τ
apply(simp add: localValidDefined2sem DEF_def both, clarify)
apply(frule DEF_X_Sequence', clarify)
apply(simp add: OclIncludes_def OclExcluding_def OclFalse_def ss_lifting'
)
done

lemma size_of_excluding:
∥(S::(τ,α::bot)Sequence)→excluding (x::(τ,α)VAL)∥ =
∥S∥ − OclCount S x
apply(rule Sequence_sem_cases_ext, simp_all)
apply(case_tac x τ ≠⊥)
apply(frule neq_commute[THEN iffD1])
apply(simp_all add: OclSize_def OclExcluding_def OclCount_def

303
Appendix B. Isabelle Theories

apply (induct_tac c, auto)
done

lemma excluding_by_select:
   [ τ ⊨ δ x ] ⇒
   ( (S::(τ,α::bot)Sequence) ->excluding (x:(τ,α)VAL)) τ =
   (\{ a : S | (¬ (a = x)) \}) τ
apply (rule_tac X=S in Sequence_induct_including)
apply simp
apply (simp, ocl_subst, simp)
apply (simp_all add: if_not)

apply (rule trans, rule_tac A=X->excluding x in cp_charn)
apply simp_all
done

Sum

Like collect sum is specified as mapping to other [OCL] functions ...this is because for every type supporting plus, zero the sum function must be available and it make no sense to add a new definition for every one of these

defs

OclSum_def : OclSum S ≡ OclIterate (S::(τ, Integer_0)Sequence) (λ x y. x + y) 0

thm OclSum_def
thm OclIterate_def

lemma sum_by_iterate: OclSum S = OclIterate (S::(τ, Integer_0)Sequence) (λ x y. x + y) 0
by (simp add: OclSum_def)

lemma OCL_cp_OclSum [simp, intro!]:
   cp P ⇒ cp(λ S. (OclSum (P S::(τ,Integer_0)Sequence)))
by (simp add: OclSum_def)

lemmas OCL_undef_1_OclSum [simp] = f1_by_iterate_undef_1[OF sum_by_iterate]
lemmas sum_empty [simp] = f1_by_iterate_mtSequence[OF sum_by_iterate]
thm f1_by_iterate_including[OF sum_by_iterate, simplified]
lemmas sum_including [simp] = f1_by_iterate_opt_including[OF sum_by_iterate, simplified]
lemmas sum_union [simp] = f1_by_iterate_distrib_union_opt[OF sum_by_iterate, of op (+, simplified plus_AC, simplified)]
thm OCL_undef_1 OclSum sum_empty sum_including sum_union

One
defs

OclOne_def:
OclOne ≡ (lift2' lift_arg0 lift_arg1) (strictify (λ S P.
if ∀ x ∈ set 'Rep_Sequence_0 S' . DEF (P x)
then size (List.filter (λ x . P x = True) 'Rep_Sequence_0 S') = 1,
else ⊥))

ocl_setup_op [OclOne]

Note that contrary to many other OCL functions it is not possible to represent the one operation directly by an iterate. The reason for it is, that the internal state space that is used has cardinality three whereas the state space of the output values has cardinality two.

lemma one_by_size_of_select:
(OclOne (S::('τ,'α::bot)Sequence) P) =
∥{x : S | P x:(τ,α)VAL})∥ = 1
apply(rule_tac X=S in Sequence_sem_cases_ext)
apply(auto simp : OclSelect_def OclSize_def OclStrictEq_def cp_by_cpify
OclOne_def One_ocl_int_def ss_lifting)
done

lemma one_of_mtSequence [simp]:
OclOne ([]::('τ,'α::bot)Sequence) (P::('τ,'α)VAL ⇒ ('τ Boolean)) = F
apply(simp add: one_by_size_of_select)
apply(simp add : OclStrongEq_def OclFalse_def ss_lifting'
One_ocl_int_def Zero_ocl_int_def)
done

lemma one_singleton [simp]:
[τ ⊨ ∂ (x:(τ,α)VAL); cp P ] ==>
OclOne ([]::('τ,'α::bot)Sequence ->including x) (P::('τ,'α)VAL ⇒ ('τ Boolean)) τ
= P x τ
apply(simp only : one_by_size_of_select)
apply(rule trans)
apply(rule_tac A=[x : [] ->including x | (P x)] in cp_charn)
apply(simp_all)
apply(rule_tac A=P x τ in boolean_cases_sem)
apply(simp_all add : OclSize_def OclStrictEq_def OclIncluding_def OclMtSequence_def
OclIf_def One_ocl_int_def neq_commute
ss_lifting' localValidDefined2sem)
done
Appendix B. Isabelle Theories

lemma one_union [simp]:
OclOne ((X::('τ,'α::bot)Sequence) ∪ Y) P =
(OclOne X P) ⊕ (OclOne Y (P::('τ, 'α)VAL ⇒ 'τ Boolean))
apply(rule ext)
apply(case_tac x ⊨ ∅ X, ocl_hypsubst, simp)
apply(simp add: OclSxor_def ss_lifting)
apply(case_tac x ⊨ ∅ Y, ocl_hypsubst, simp)
apply(simp add: one_by_size_of_select)
oops

IsUnique
defs

OclIsUnique_def :
OclIsUnique ≡ (lift2' lift_arg0 lift_arg1) (strictify (λ S P.
if ∀ x ∈ set "Rep_Sequence_0 S". DEF (P x)
then distinct (map (λx. P x) "Rep_Sequence_0 S")
else ⊥))

ocl_setup_op [OclIsUnique]

lemma isUnique_by_forall:
OclIsUnique (S::('τ,'α::bot)Sequence) (P::('τ, 'α)VAL ⇒ ('τ, 'c::bot)VAL) =
OclIterate S (λ x y. ((OclCount (OclCollectNested S P) (P x)) ≡ 1) ∧ y) T
oops

Any
defs

OclAny_def :
OclAny ≡ (lift2' lift_arg0 lift_arg1) (strictify (λ S P.
if (∀ x ∈ set "Rep_Sequence_0 S". DEF (P x)) ∧
(∃ x ∈ set "Rep_Sequence_0 S". P x = T_True)
then hd (List.filter (λ x. P x = T_True) "Rep_Sequence_0 S")
else ⊥))

ocl_setup_op [OclAny]

lemma any_by_first_of_select:
OclAny (S::('τ,'α::bot)Sequence) (P::('τ, 'α)VAL ⇒ 'τ Boolean) =
OclFirst ((OclAsSequence (OclSelect S P)::('τ,'α)Sequence))
apply(rule_tac X=S in Sequence_sem_cases_ext, simp_all)
apply(auto simp: OclAny_def OclSelect_def OclFirst_def
cp_by_cpify ss_lifting' filter_empty_conv)
done
lemma any_of_mtSequence[simp]:
  OclAny (λ_x: ('τ', 'α::bot)Sequence) (P::('τ, 'α) VAL ⇒ ('τ Boolean)) = ⊥
by(simp add: any_by_first_of_select)

collectNested_def:

OclCollectNested_def ≡ (lift2' lift_arg0 lift_arg1) (strictify (λ S P.
  if ∀ x ∈ set 'Rep_Sequence_0 S. DEF (P x)
  then Abs_Sequence_0 (map (λ x: P x) 'Rep_Sequence_0 S)
  else ⊥))

ocl_setup_op [OclCollectNested]

lemma collectNested_by_iterate:
  ((OclCollectNested (X::('τ, 'α::bot)Sequence) (P::('τ, 'α) VAL ⇒ ('τ, 'c) VAL)::('τ, 'c::bot)Sequence)
   = (X ->iterate x, y=[] | y->including (P x)))
apply(rule ext, rule iterate_universal_0, simp_all)
apply(simp add: OclCollectNested_def cp_by_cpify lift0_def)
apply(simp_all add: OclUndefined_def OclMtSequence_def OclCollectNested_def
                   localValidDefined2sem ss_lifting)
apply(frule DEF_X_Sequence', clarify)
apply(frule neq_commute[THEN iffD1])
apply(rotate_tac 1, frule neq_commute[THEN iffD1])
apply(auto simp: OclIncluding_def ss_lifting)
apply(subst (asm) Abs_Sequence_0_inject_absurd22, auto)
apply(subst Abs_Sequence_0_inverse_charn2, auto)
done

Properties derived from the correspondence to iterate

lemmas collectNested_of_mtSequence [simp] = f1_by_iterate_mtSequence[OF collectNested_by_iterate]
thm f1_by_iterate mtSequence[of collectNested_by_iterate, simplified]

lemmas collectNested_of_including [simp] = f1_by_iterate_opt_including[OF collectNested_by_iterate, simplified]

lemmas collectNested_of_union [simp] = f1_by_iterate_distrib_union_opt[OF collectNested_by_iterate, of OclUnion, simplified]

thm OCL_undef_1_OclCollectNested collectNested_of_mtSequence collectNested_of_including collectNested_of_union

Checking a general property of iterate... on foldl

This lemma could actually be made stronger because the two assumptions undef_1_P
and cp_P are not needed. But they make the proof much simpler.
Appendix B. Isabelle Theories

lemma iterate_of_collectNested:
assumes undef_1_Q: ∀ y. Q ⊥ y = ⊥
and undef_2_Q: ∀ x. Q x = ⊥
and cp1_Q: ∀ y. cp (λ x. Q x y)
and cp2_Q: ∀ x. cp (Q x)
and undef_1_P: P ⊥ = ⊥
and cp_P: cp P
shows OclIterate ((OclCollectNested (S::('τ,'α::bot)Sequence)
(P::('τ,'α::bot)VAL ⇒ ('τ,'c::bot)VAL)::('τ,'Sequence)) Q A = OclIterate S (λ x y. Q (P x) y) A)
apply(insert undef_1_Q undef_2_Q cp1_Q cp2_Q undef_1_P cp_P)
apply(rule ext)
apply(rule iterate_universal, simp_all)
apply(rule_tac P = P in cp_compose2)
apply(simp_all add: collectNested_by_iterate iterate_opt_of_including)
done

Definedness rules
lemma undefCollectNestedI:
assumes exUndef1: τ ⊨ x ∈ (C::('τ,'α::bot)Sequence)
and exUndef2: τ ⊨ x ∉ (P::('τ,'α::bot)VAL ⇒ ('τ,'c::bot)VAL) x
and cp_P: cp P
shows τ ⊨ x ∉ ((OclCollectNested C P)::('τ,'c::bot)Sequence)
apply(insert exUndef1 exUndef2)
apply(frule isDefined_if_valid)
apply(subst (asm) OCL_is_defopt_OclIncludes)
apply(clarify, simp add: localValidDefined2sem DEF_def)
apply(frule DEF_X_Sequence', clarify)
apply(simp add: localValid2sem OclIsDefined_def OclNot_def OclCollectNested_def OclIncludes_def ss_lifting')
apply(clarify)
apply(erule_tac x = x τ in ballE)
apply(insert cp_P, simp_all add: cp_by_cpify lift0_def)
done

Flatten
This definition is currently not complete as it deals only with the type sequence of sequences of some type. There must be written a tactic that defines all needed instances of flatten on the different types. See the semester thesis for further information about the problems around flatten.

defs

OclFlatten_def: OclFlatten ≡ lift1(strictify (λ S.
Abs_Sequence_0 ‹concat (map (λ x. ‹Rep_Sequence_0 x›) ‹Rep_Sequence_0 S›)›))
B.4. Library

ocl_setup_op [OclFlatten]

lemma OCL_is_defopt_OclFlatten [simp]:
\((\tau \models \emptyset ((\llceil X \rrceil):((\tau,\alpha:\text{bot})(\text{Sequence})))) = (\tau \models \emptyset (X:((\tau,\alpha)(\text{Sequence})))\))
apply(rule_tac X=X in Sequence_sem_cases, simp_all)
apply(simp_all add: localValidDefined2sem OclFlatten_def ss_lifting)
apply(clarify)
apply(subst (asm) Abs_Sequence_0_inject_absurd22, auto)
apply(rule_tac x=a in Abs_Sequence_0_cases_charn, simp_all)
done

Its computational characteristic by iterate

lemma flatten_by_iterate:
\((\text{OclFlatten } (X:((\tau,\alpha:\text{bot})(\text{Sequence}))))::((\tau,\alpha)\text{Sequence}) = (X \rightarrow_{\text{iterate}} x = [] | y \cup ((x:((\tau,\alpha)(\text{Sequence}))))))\)
apply(rule ext, rule iterate_universal_0, simp_all)
apply(simp_all add: OclUndefined_def OclMtSequence_def OclFlatten_def localValidDefined2sem ss_lifting)
apply(rotate_tac 1, frule DEF_X_Sequence', clarify)
apply(rotate_tac 1, frule neq_commute[THEN iffD1])
apply(rotate_tac 1, frule DEF_X_Sequence', clarify)
apply(simp add: OclUnion_def OclIncluding_def ss_lifting)
apply(subgoal_tac \(\bot \in \text{set} (\text{concat} (\text{map} (\lambda x:((\tau,\alpha)(\text{Sequence}))). (\text{Rep_Sequence_0} x))))\))
apply(auto)
apply(rule_tac x=a in Abs_Sequence_0_cases_charn, auto)
done

Deriving the rules that follow from the correspondence to iterate

lemmas flatten_of_mtSequence [simp] = f1_by_iterate_mtSequence[OF flatten_by_iterate]
lemmas flatten_of_including [simp] = f1_by_iterate_opt_including[OF flatten_by_iterate, simplified]
lemmas flatten_of_union [simp] = f1_by_iterate_distrib_union_opt[OF flatten_by_iterate, of OclUnion, simplified]

thm OCL_undef_1_OclFlatten flatten_of_mtSequence flatten_of_including flatten_of_union

functions applied to flatten

lemma f_of_flatten:
assumes g_by_iterate: \(g\ f\ X\ \tau = \text{OclIterate } X\ (\lambda x.\ y.\ h\ (f\ x))\ A\ \tau\)
and f_union: \(\bigwedge\ X\ Y.\ [\ \tau \models \emptyset\ X;\ \tau \models \emptyset\ Y] \implies f\ (X \cup Y) = h\ (f\ X)\ (f\ Y)\ \tau\)
and f_empty: \(\bot\ \tau = (A::((\tau,\alpha:\text{bot})(\text{VAL})\))\ \tau\)
and f_undef: \(\bot\ \tau = \bot\ \tau\)
and \(\text{cp}\_f\_f\_cp\ f\)
and \(\text{cp}\_f\_h:\ \bigwedge\ y.\ \text{cp}\ (\lambda x.\ h\ x\ y)\)
Appendix B. Isabelle Theories

and 

shows f ((\(X:\{\tau,'\alpha\}Sequence_0\)Sequence)):(\(\tau,'\alpha\)bot)Sequence) \tau

= ((g f X):(\(\tau,'\alpha\)\bot)VAL) \tau

apply(insert cp2_h cp2_h cp_f subst g_by_iterate)

apply(rule_tac X=X and \tau=\tau in Sequence_induct_including)

apply(simp_all add: f_undef f_empty f_union)

apply(subst iterate_of_including, simp_all)

apply(rule_tac P=f in cp_compose2, simp_all)

apply(rule_tac A=(f \[[X]] in cp_charn, simp_all)

apply(rule_tac P_eq rule_tac P=f in cp_compose2, simp_all)

done

lemma isEmpty_flatten:

((\(\exists x\in X. (x:\{\tau,'\alpha\}Sequence) \Rightarrow [\tau=\emptyset] = [a:\{\tau,'\alpha\}Sequence=\emptyset])

apply(rule ext, rule_tac X=X and h=OclAnd in f_of_flatten)

apply(simp_all add: forall_by_iterate OCL_logic_ACI

and_equiv_sand[symmetric] OCL_is_defopt_OclStrictEq

deb: OCL_is_def_opt_OclStrictEq)

done

lemma includes_flatten:

[ \tau = \emptyset \Rightarrow (a:\{\tau,'\alpha\}bot)VAL] \Longrightarrow

(a \in ((\(\exists x\in X. (x:\{\tau,'\alpha\}Sequence) \Rightarrow [\tau=\emptyset] = [a:\{\tau,'\alpha\}Sequence=\emptyset])

apply(rule_tac X=X and h=OclOr in f_of_flatten)

apply(simp_all add: includes_of_union exists_by_iterate)

apply(ocl_subst, simp add: OCL_logic_ACI)

done

lemma excludes_flatten:

[ \tau = \emptyset \Rightarrow (a:\{\tau,'\alpha\}bot)VAL] \Longrightarrow

(a \notin ((\(\exists x\in X. (x:\{\tau,'\alpha\}Sequence) \Rightarrow [\tau=\emptyset] = [a:\{\tau,'\alpha\}Sequence=\emptyset])

apply(rule_tac X=X and h=OclAnd in f_of_flatten)

apply(simp_all add: includes_of_union forall_by_iterate)

apply(ocl_subst, simp add: OCL_logic_ACI)

apply(rule_tac A=(a \in (X \cup Y)) in cp_charn)

apply(rule includes_of_union, simp_all)

apply(simp add: OclAnd_def OclOr_def OclNot_def ss_lifting)

done

The current formulation is NOT true. But what is an elegant solution to it?

lemma includesAll_flatten:

(OclIncludesAll X ((\(\exists x\in X. (x:\{\tau,'\alpha\}Sequence) \Rightarrow [\tau=\emptyset] = [a:\{\tau,'\alpha\}Sequence=\emptyset])

\forall Z \in Y. (OclIncludesAll X (Z:\{\tau,'\alpha\}bot)Sequence))

oops

310
The current formulation is NOT true. But what is an elegant solution to it?

**Lemma** `excludesAll_flatten`:

\[
\text{OclExcludesAll} \ X \ (\forall \ Z \in \ Y \cdot \text{OclExcludesAll} \ X \ (Z \cdot (\tau, \alpha :: \text{bot})))
\]

**Lemma** `forall_flatten`:

\[
(\forall \ y \in (\forall \ X : (\tau, \alpha \cdot \text{Sequence}_0)) : ((\tau, \alpha) \cdot \text{Sequence}) \cdot P \ y) =
(\text{OclForAll} \ X \ (\lambda \ Y : (\tau, \alpha) \cdot \text{Sequence}. \ (\forall \ y \in Y \cdot P \ (y : (\tau, \alpha :: \text{bot}))))))
\]

**Lemma** `exists_flatten`:

\[
(\exists \ y \in (\forall \ X : (\tau, \alpha \cdot \text{Sequence}_0)) : ((\tau, \alpha) \cdot \text{Sequence}) \cdot P \ y) =
(\text{OclExists} \ X \ (\lambda \ Y : (\tau, \alpha) \cdot \text{Sequence}. \ (\exists \ y \in Y \cdot P \ (y : (\tau, \alpha :: \text{bot}))))))
\]

**Lemma** `select_flatten`:

\[
\text{OclSelect} (\forall \ X : (\tau, \alpha \cdot \text{Sequence}_0) : ((\tau, \alpha :: \text{bot}) \cdot \text{Sequence}) \ (P :: ((\tau, \alpha) \cdot \text{VAL} \Rightarrow (\tau \cdot \text{Boolean}))) =
\text{OclCollectNested} \ X \ (\lambda \ Y : (\tau, \alpha \cdot \text{Sequence}). \ \text{OclSelect} \ Y \ P) : ((\tau, \alpha \cdot \text{Sequence}_0))
\]

**Lemma** `reject_flatten`:

\[
\text{OclReject} (\forall \ X : (\tau, \alpha \cdot \text{Sequence}_0) : ((\tau, \alpha :: \text{bot}) \cdot \text{Sequence}) \ (P :: ((\tau, \alpha) \cdot \text{VAL} \Rightarrow (\tau \cdot \text{Boolean}))) =
\text{OclCollectNested} \ X \ (\lambda \ Y : (\tau, \alpha \cdot \text{Sequence}). \ \text{OclReject} \ Y \ P) : ((\tau, \alpha \cdot \text{Sequence}_0))
\]

By rewriting any to `first_of_select` it is clear what the result of any on a flattened sequence will be.

Currently there is no simplification rule for the equalities. I just can’t see a way how to represent the iterated comparison of a prefix of both sequences with its intertwined concatenations.

By rewriting one to `size_of_select` it is clear what the result of one on a flattened sequence will be.

**Lemma** `size_flatten`:

\[
\text{OclSum} (\text{OclCollectNested} \ X (\text{OclSize} :: (\tau, \alpha) \cdot \text{Sequence} \Rightarrow \tau \cdot \text{Integer})) : ((\tau, \alpha :: \text{bot}) \cdot \text{Sequence})
\]

apply (rule ext)
Appendix B. Isabelle Theories

apply (rule_tac X=X and \( \tau = x \) in Sequence_induct_including, simp_all)
apply (rule trans)
apply (rule_tac A=\( \llbracket X \rrbracket \) and \( \tau = x \) in cp_charn)
apply (simp_all add: plus_AC)
done

lemma count_flatten:
\[
\llbracket \tau = a \rrbracket \implies (\llbracket (\llbracket X : (\llbracket (\tau, \alpha \ Sequence_0)Sequence_0) \rrbracket) : (\llbracket (\tau, \alpha :: bot)Sequence_0) \rrbracket) : (\llbracket (\tau, \alpha :: VAL) \rrbracket) : (\llbracket (\tau, \alpha : \text{count} (\alpha : (\llbracket (\tau, \alpha) \rrbracket)) \rrbracket) : (\llbracket (\tau, \alpha : (\tau, \alpha :: bot)Sequence_0) \rrbracket) \rrbracket = OclSum (OclCollectNested X (\lambda (X : (\tau, \alpha)Sequence_0). X \rightarrow \text{count} a)) : (\llbracket (\tau, \alpha :: VAL) \rrbracket)
\]
apply (rule_tac X=X and \( \tau = \tau \) in Sequence_induct_including, simp_all)
apply (rule trans)
apply (rule_tac A=\( \llbracket X \rrbracket \rightarrow \text{count} a \) and \( \tau = \tau \) in cp_charn)
apply (simp_all add: plus_AC)
done

lemma sum_flatten:
\[
OclSum ((\llbracket (\llbracket X : (\llbracket (\tau, \alpha \ Sequence_0)Sequence_0) \rrbracket) : (\llbracket (\tau, \alpha :: bot)Sequence_0) \rrbracket) : (\llbracket (\tau, \alpha :: VAL) \rrbracket) : (\llbracket (\tau, \alpha : \text{count} (\alpha : (\llbracket (\tau, \alpha) \rrbracket)) \rrbracket) : (\llbracket (\tau, \alpha : (\tau, \alpha :: bot)Sequence_0) \rrbracket) \rrbracket = OclSum (OclCollectNested X (OclSum : (\llbracket (\tau, \alpha :: VAL) \rrbracket) \rightarrow \tau \ rrbracket) : (\llbracket (\tau, \alpha :: VAL) \rrbracket) : (\llbracket (\tau, \alpha : \text{count} (\alpha : (\llbracket (\tau, \alpha) \rrbracket)) \rrbracket) : (\llbracket (\tau, \alpha :: bot)Sequence_0) \rrbracket)
\]
apply (rule ext)
apply (rule_tac X=X and \( \tau = \tau \) in Sequence_induct_including, simp_all)
apply (rule trans)
apply (rule_tac A=\( \llbracket X \rrbracket \rightarrow \text{sum} (\llbracket X \rrbracket) \) and \( \tau = \tau \) in cp_charn)
apply (simp_all add: plus_AC)
done

lemma excluding_flatten:
\[
\llbracket \tau = a \rrbracket \implies OclExcluding ((\llbracket (\llbracket X : (\llbracket (\tau, \alpha \ Sequence_0)Sequence_0) \rrbracket) : (\llbracket (\tau, \alpha :: bot)Sequence_0) \rrbracket) : (\llbracket (\tau, \alpha :: VAL) \rrbracket) = (\llbracket (\tau, \alpha :: VAL) \rrbracket) : (\llbracket (\tau, \alpha :: bot)Sequence_0) \rrbracket) : (\llbracket (\tau, \alpha :: VAL) \rrbracket) : (\llbracket (\tau, \alpha : \text{count} (\alpha : (\llbracket (\tau, \alpha) \rrbracket)) \rrbracket) : (\llbracket (\tau, \alpha :: bot)Sequence_0) \rrbracket)
\]
apply (rule_tac X=X and \( h = \text{OclUnion} f \) of flatten)
apply (simp_all)
apply (ocl_subst)
apply (simp add: flatten_by_iterate iterate_of_collectNested)
done

lemma collectNested_flatten:
\[
(OclCollectNested (\llbracket (\llbracket X : (\llbracket (\tau, \alpha \ Sequence_0)Sequence_0) \rrbracket) : (\llbracket (\tau, \alpha :: bot)Sequence_0) \rrbracket) : (\llbracket (\tau, \alpha :: VAL) \rrbracket) = (\llbracket (\tau, \alpha :: bot)Sequence_0) \rrbracket) : (\llbracket (\tau, \alpha : \text{count} (\alpha : (\llbracket (\tau, \alpha) \rrbracket)) \rrbracket) : (\llbracket (\tau, \alpha :: bot)Sequence_0) \rrbracket)
\]
apply (rule ext)
apply (rule_tac X=X and \( \tau = \tau \) in Sequence_induct_including, simp_all)
apply (rule trans)
apply (rule_tac A=\( \llbracket X \rrbracket \) \ and \( \tau = \tau \) in cp_charn)

312
apply(simp_all)
done

lemma test':
\[(\{(X::\langle\tau,\alpha Sequence_0 Sequence_0\rangle Sequence)\}::\langle\tau,\alpha Sequence_0\rangle Sequence)\} = h\]
apply(simp add: size_flatten collectNested_flatten sum_flatten)
oops

Sequence specific functions

These are: first, last, at, insertAt, index0f, subSequence

lemma last_empty [simp]:
OclLast (\{\}::\langle\tau,\alpha ::\bot\rangle Sequence) = (\bot::\langle\tau,\alpha\rangle VAL)
by(simp add: last_at_UC)

lemma last_including [simp]:
\[\tau \models \partial S\implies OclLast (\{S::\langle\tau,\alpha ::\bot\rangle Sequence\} \rightarrow including (x::\langle\tau,\alpha\rangle VAL))\]
apply(rule_tac X=S in Sequence_sem_cases)
apply(simp_all add: localValidDefined2sem DEF_def)
apply(simp add: OclIncluding_def OclLast_def ss_lifting neq_commute)
done

lemma last_union:
\[\tau \models \partial X;\tau \models (Y \langle\rangle \{\})\implies OclLast ((X::\langle\tau,\alpha ::\bot\rangle Sequence) \cup Y)\]
apply(rule_tac X=X in Sequence_sem_cases)
apply(rule_tac X=Y in Sequence_sem_cases, simp_all)
apply(auto simp: localValid2sem OclNot_def OclStrictEq_def OclMtSequence_def
OclUnion_def OclLast_def OclIsDefined_def ss_lifting')
done

lemma ForAllD:
assumes allP: \[\forall x \in C \cdot P(x::\langle\tau \Rightarrow \alpha ::\bot\rangle)\]
and cpP : \[cp P\]
shows \[\lambda x.\tau \models x \in (C::\langle\tau,\alpha ::\bot\rangle Sequence) \implies \tau \models P \ x\]
apply (insert allP cpP)
apply (simp add: localValid2sem OclForAll_def OclNot_def OclIncludes_def
ss_lifting')
apply (simp add: strictify_def
split: split_if_asm)
apply (erule_tac x=x in ballE)
apply (simp_all add: cp_by_cpyfy lift0_def)
Appendix B. Isabelle Theories

done

lemma isdefSelectE:
assumes localDef: \(\forall x \in C \cdot (P (x::\tau::\alpha::bot) VAL))\)
and \(cp_P::cp P\)
shows \(\sigma \models (OclSelect (C::\tau::\alpha::bot)Sequence) P)\)
apply(insert localDef)
apply(erule rev_mp)
apply(rule_tac X=C in Sequence_induct_including)
apply(simp_all, clarify)
oops

lemma last_flatten:
\((OclLast (\{\{X::\tau::\alpha Sequence_0 Sequence\}\})::\tau::\alpha VAL) =
OclLast (OclLast (OclSelect X (\lambda Y::\tau::\alpha Sequence. Y '<<' [])::\tau::\alpha Sequence)\))::\tau::\alpha Sequence\)
apply(rule ext)
apply(rule_tac X=X in Sequence_induct_including, simp_all)
apply(case_tac x \(\vDash\) (a '<<' []))
apply(rule trans[symmetric])
apply(rotate_tac -1, subst (asm) is_TRUE_charn_local[symmetric])
apply(simp_all)
apply(simp add: last_union)
apply(rule_tac A=a '<<' [] in cp_charn)
apply(rotate_tac 1, thin_tac ?P)
apply(rule_tac A=a and B=[] in cp_charn)
apply(simp add: OclLast_def OclUnion_def localValid2sem OclStrictEq_def OclFlatten_def
OclIsDefined_def OclMtSequence_def ss_lifting)
apply(simp_all)
oops
Collect

Note: Currently collectNested is only defined here but it should actually be polymorphically defined in the theory OCL_Collection. Furthermore the collect operation could be defined right there because it should be defined on the [OCL] level to avoid having to define collect over all different types like we have to do it for flatten.

Defining collect based on collectNested according to the standard defs

\[
\begin{align*}
\text{OclCollect_def} & : \left( \text{OclCollect} : (\tau, \alpha : \text{bot}) \text{Sequence}, (\tau, \alpha) \text{VAL} \Rightarrow (\tau, c : \text{bot}) \text{Sequence} \right) \Rightarrow (\tau, c : \text{bot}) \text{Sequence} \right) \Rightarrow (\tau, c : \text{Sequence_0}) \right) \\
S P & \equiv \left( \text{OclCollectNested} S P \right) \left( \tau, c : \text{Sequence_0} \right) \\
\text{OclCollect_Integer_def} & : \left( \text{OclCollect} : (\tau, \alpha : \text{bot}) \text{Sequence}, (\tau, \alpha) \text{VAL} \Rightarrow \tau \text{Integer} \right) \Rightarrow (\tau, \text{Integer_0}) \text{Sequence} \\
S P & \equiv \left( \text{OclCollectNested} S P \right) \\
\end{align*}
\]

lemma collect_by_iterate:
\[
\begin{align*}
\text{\text{lbracket} P \bot = \bot; \text{cp} P \text{\text{rbracket}} \Rightarrow (\text{OclCollect} (X : (\tau, \alpha : \text{bot}) \text{Sequence}) (P : (\tau, \alpha) \text{VAL} \Rightarrow (\tau, c : \text{bot}) \text{Sequence})) : (\tau, c \text{Sequence})} \\
= (X \rightarrow \text{iterate}(x \mapsto []; (y \cup (P x)))) \\
\text{by (simp add: OclCollect_def flatten_by_iterate iterate_of_collectNested)} \\
\end{align*}
\]

lemma OCL_cp_OclCollect[simp, intro!]:
\[
\begin{align*}
\text{\text{lbracket} \lambda y. \text{cp} (\lambda x. P x y) : \text{cp} S \text{\text{rbracket}} \Rightarrow \text{cp}(\lambda X. \text{OclCollect} ((S X) : (\tau, \alpha : \text{bot}) \text{Sequence}) \\
(P X) : (\tau, \alpha) \text{VAL} \Rightarrow (\tau, c : \text{Sequence})) : (\tau, c : \text{Sequence})} \\
\text{by (simp add: OclCollect_def)} \\
\end{align*}
\]

lemma OCL_undef_1_OclCollect[simp]:
\[
\begin{align*}
\text{OclCollect (\bot : (\tau, \alpha : \text{bot}) \text{Sequence}) (P : (\tau, \alpha) \text{VAL} \Rightarrow (\tau, c \text{Sequence})} \\
= (\bot : (\tau, c : \text{bot}) \text{Sequence}) \\
\text{by (simp add: OclCollect_def)} \\
\end{align*}
\]

end
Appendix B. Isabelle Theories

B.4.11. OCL Bag

theory OCL_Bag
imports
  $HOLOCL_HOME/src/library/collection/smashed/OCL_Sequence
begin

Properties of the the Datatype Adaption

The following rules are transformed versions of the automatically generated rules for datatype adaption

lemma Abs_Bag_0_inject_absurd11 [simp]:
  \[\text{count } x \bot = 0 \implies (\text{Abs}_\text{Bag}_0 \text{down} = (\text{Abs}_\text{Bag}_0 \text{abs} x)) = \text{False}\]
by (subst Abs_Bag_0_inject, simp_all add: OCL_Bag_type.Bag_0_def smash_def)

lemma Abs_Bag_0_inject_absurd12 [simp]:
  \[\text{count } x \bot = 0 \implies (\text{Abs}_\text{Bag}_0 \text{abs} x = (\text{Abs}_\text{Bag}_0 \text{down})) = \text{False}\]
by (subst Abs_Bag_0_inject, simp_all add: OCL_Bag_type.Bag_0_def smash_def)

lemma Abs_Bag_0_inject_absurd21 [simp]:
  \[\text{count } x \bot = 0 \implies (\bot = (\text{Abs}_\text{Bag}_0 \text{abs} x)) = \text{False}\]
by (simp add: UU_Bag_def)

lemma Abs_Bag_0_inject_absurd22 [simp]:
  \[\text{count } x \bot = 0 \implies ((\text{Abs}_\text{Bag}_0 \text{abs} x) = \bot) = \text{False}\]
by (simp add: UU_Bag_def)

lemma Abs_Bag_0_inject_charn [simp]:
  \[\text{count } x \bot = 0 \implies ((\text{Abs}_\text{Bag}_0 \text{charn} x) = (\bot = (\text{Abs}_\text{Bag}_0 \text{abs} x))) = \text{False}\]
by (subst Abs_Bag_0_inject, simp_all add: OCL_Bag_type.Bag_0_def smash_def)

lemma Abs_Bag_0_inverse_charn [simp]:
  \[(\text{count } x \bot = 0) \implies (\text{Rep}_\text{Bag}_0 (\text{Abs}_\text{Bag}_0 \text{abs} x) = x)\]
by (subst Abs_Bag_0_inverse, simp_all add: OCL_Bag_type.Bag_0_def smash_def)

lemma Abs_Bag_0_inverse_charn2 [simp]:
  \[(\text{count } x \bot = 0) \implies (\text{Rep}_\text{Bag}_0 (\text{Abs}_\text{Bag}_0 \text{abs} x)^\text{rev} = x)\]
by (subst Abs_Bag_0_inverse, simp_all add: OCL_Bag_type.Bag_0_def smash_def)

lemma Abs_Bag_0_cases_charn:
  assumes bottomCase: \[x = \bot \implies P\]
  assumes listCase : \[\forall y. (\{x = Abs_\text{Bag}_0 \text{abs} y; \text{count } x \bot = 0 \implies P\)}
  shows P
apply (rule_tac x=x in Abs_Bag_0_cases)
apply (case_tac y = ⊥)
apply (rule_tac x=Abs_Bag_0 y in Abs_Bag_0_cases_charn)
adds: OCL_Bag_type.Bag_0_def smash_def DEF_def
apply (auto simp: OCL_Bag_type.Bag_0_def smash_def DEF_def)
done

lemma Abs_Bag_0_induct_charn:
assumes bottomCase: P ⊥
assumes stepCase: y. (count y ⊥ = 0) =⇒ P (Abs_Bag_0 y)
shows P x
apply (rule_tac x=x in Abs_Bag_0_induct)
apply (auto intro: bottomCase stepCase)
done

lemma inj_on_Abs_Bag_0: inj_on Abs_Bag_0 Bag_0
by (rule inj_on_inverseI, rule Bag_0.Abs_Bag_0_inverse, assumption)

lemma inj_Rep_Bag_0: inj Rep_Bag_0
by (rule inj_on_inverseI, rule Bag_0.Rep_Bag_0_inverse)

lemma smashed_bag_charn: (count X ⊥ = 0) = ((X) ∈ Bag_0)
by (unfold smash_def Bag_0_def UU_Bag_def, auto)

lemma UU_in_smashed_bag [simp]: ∉ Bag_0
by (unfold smash_def Bag_0_def UU_Bag_def, auto)

lemma down_in_smashed_bag [simp]: down ∈ Bag_0
by (unfold smash_def Bag_0_def UU_Bag_def, auto)

lemma mt_in_smashed_bag [simp]: (#{#}) ∈ Bag_0
by (unfold smash_def Bag_0_def, auto)

lemma DEF_Abs_Bag: (∃X. (count X ⊥ = 0) =⇒ DEF (Abs_Bag_0 (X)))
apply (unfold DEF_def UU_Bag_def, simp)
done

lemma DEF_Rep_Bag:
(∃X. DEF X) =⇒ DEF (Rep_Bag_0 X)
apply (unfold DEF_def UU_Bag_def, auto)
apply (drule_tac f = Abs_Bag_0 in arg_cong)
apply (simp add: Rep_Bag_0_inverse)
done

lemma not_DEF_Rep_Bag:
\( \forall X. \neg DEF X \implies \neg DEF (Rep_Bag_0 X) \)
apply (unfold DEF_def UU_Bag_def, auto)
apply (simp add: Abs_Bag_0_inverse)
done

lemma exists_lift_Bag:
\( \forall X. DEF X = \implies \exists c. \ Rep_Bag_0 X = (\langle c, \rangle) \)
by (drule DEF_Rep_Bag)
apply (auto simp: DEF_def)
apply (erule not_DEF_Rep_Bag)
done

lemma Rep_Bag_cases:
\( Rep_Bag_0 X = \bot \lor (\exists c. \ count c \bot = 0 \land \ Rep_Bag_0 X = (\langle c, \rangle)) \)
apply (case_tac DEF X)
apply (drule exists_lift_Bag)
apply (drule_tac \[2\] not_DEF_Rep_Bag)
apply (simp_all add: DEF_def)
done

lemma DEF_X_Bag_0: 
\( \exists c. \ count c \bot = 0 \land \ X = Abs_Bag_0 (\langle c, \rangle) \)
apply (auto simp: DEF_Abs_Bag)
apply (simp add: DEF_X_Bag_0)
apply (erule exE, rule exI, auto)
apply (rule injD [OF inj_Rep_Bag_0])
done

The following lemma is crucial for thy_morpher:

lemma DEF_X_Bag: 
\( DEF X = (\exists c. \ count c \bot = 0 \land X = Abs_Bag_0 (\langle c, \rangle)) \)
apply (auto simp: DEF_Abs_Bag)
apply (simp add: DEF_X_Bag_0)
apply (erule exE, rule exI, auto)
apply (rule injD [OF inj_Rep_Bag_0])
apply (auto simp: smashed_bag_charn Abs_Bag_0_inverse)
done

This lemma is very convenient in simplifying unfolded definitions

lemma DEF_X_Bag':
\[
\begin{align*}
[X \neq \bot] \implies \\
(\exists c. (\text{count } c \bot = 0) \land (\text{Rep_Bag}_0 X = \langle c, \rangle)) \land \\
(\exists c. (\text{count } c \bot = 0) \land (X = (\text{Abs_Bag}_0 \langle c, \rangle)))
\end{align*}
\]
apply (fold DEF_def)
apply (frule DEF_X_Bag [THEN iffD1])
apply (drule DEF_X_Bag_0 [THEN iffD1])
apply (simp)
done

lemma Bag_sem_cases_0:
assumes defC: \(\forall c d. \llbracket X \neq \bot; \bot \neq X;\)
\((\text{count } c \bot = 0); (\text{count } d \bot = 0);\)
\(\text{Rep_Bag}_0 X = \langle c, \rangle;\)
\(X = \text{Abs_Bag}_0 \langle d, \rangle \implies P X\)
and undefC: \(\llbracket X = \bot \rrbracket = \implies P X\)
shows P X
apply (rule Abs_Bag_0_cases_charn, erule undefC)
apply (rule defC, auto)
done

lemma Bag_sem_cases:
assumes defC: \(\forall c d \tau. \llbracket (X \tau) \neq \bot; \bot \neq (X \tau);\)
\((\text{count } c \bot = 0); (\text{count } d \bot = 0);\)
\(\text{Rep_Bag}_0 (X \tau) = \langle c, \rangle;\)
\((X \tau) = \text{Abs_Bag}_0 \langle d, \rangle \implies P X \tau\)
and undefC: \(\llbracket (X \tau) = \bot \rrbracket = \implies P X \tau\)
and cpP\( cp P\)
shows P X \tau
apply (rule_tac P2=P in subst[OF sym[OF cp_subst]], rule cpP)
apply (rule Bag_sem_cases_0)
apply (rule_tac P1=P in subst[OF cp_subst], rule cpP)
apply (rule defC) prefer 7
apply (rule_tac P1=P in subst[OF cp_subst], rule cpP)
apply (rule undefC, simp_all)
done

lemma Bag_sem_cases_ext:
assumes defC: \(\forall c d \tau. \llbracket (X \tau) \neq \bot; \bot \neq (X \tau);\)
\((\text{count } c \bot = 0); (\text{count } d \bot = 0);\)
\(\text{Rep_Bag}_0 (X \tau) = \langle c, \rangle;\)
\((X \tau) = \text{Abs_Bag}_0 \langle d, \rangle \implies P X \tau = Q X \tau\)
and undefC: \(\forall \tau. \llbracket (X \tau) = \bot \rrbracket = \implies P X \tau = Q X \tau\)
and cpP: cp P
shows P X \tau = Q X \tau
apply (auto simp: smashed_bag_charn Abs_Bag_0_inverse)
done
Appendix B. Isabelle Theories

and \( cpQ : cp Q \)
shows \( P X = Q X \)
apply(rule ext)
apply(rule_tac X=X in Bag_sem_cases)
apply(rule_tac X=X in Bag_sem_cases)
apply(apply(rule_tac X=X in Bag_sem_cases))
apply(apply(rule_tac X=X in Bag_sem_cases))
apply(apply(rule defC prefer 7)
apply(apply(define undefC)
apply(apply(simp_all add: cpP cpQ)
done

These four lemmas are used by ocl_setup_op to reason about definedness:

lemma lift2_strict_is_isdef-fw_Bag_Val:
assumes f_def: \( f \equiv \text{lift2}(\text{strictify}(\lambda x. \text{strictify}(\lambda y. \text{Abs_Bag}_0 g \text{Rep_Bag}_0 x y))) \)
and inv_g: \( \forall a b. (\{ \text{count a} = 0 ; (\bot :: ':a:@bot) \neq b \} \Longrightarrow \text{count}(g a b) \bot = 0) \)
shows \( \partial(f X Y) = (\partial X \land \partial Y) \)
apply(simp add: f_def OclIsDefined_def OclAnd_def
apply(insert f_def)
prefer 2
apply(case_tac DEF (X τ))
apply(case_tac DEF (Y τ))
apply(drule DEF_X_Bag_0 [THEN iffD1])
apply(auto simp: strictify_def DEF_def
apply(drule_tac b=(Y τ) in inv_g)
apply(drule DEF_Abs_Bag, simp add: DEF_def, auto)
done

lemma lift2_strictify_implies_LocalValid_defined_Bag_Val:
assumes f_def: \( f \equiv \text{lift2}(\text{strictify}(\lambda x. \text{strictify}(\lambda y. \text{Abs_Bag}_0 g \text{Rep_Bag}_0 x y))) \)
and inv_g: \( \forall a b. (\{ \text{count a} = 0 ; (\bot :: ':a:@bot) \neq b \} \Longrightarrow \text{count}(g a b) \bot = 0) \)
shows \( (\tau \models f X Y) = ((\tau \models \partial X) \land (\tau \models \partial Y)) \)
apply(insert f_def)
B.4. Library

apply(drule_tac X=X and Y=Y in lift2_strict_is_isdef_fw_Bag_Val)
apply(rule inv_g, assumption+)
apply(simp add: OclAnd_def OclTrue_def o_def
  OclIsDefined_def lift0_def lift1_def lift2_def
  OclLocalValid_def)
done

lemma lift2_strictify_implies_LocalValid_defined_Bag_Bag:
  assumes f_def: f ≡ lift2 (strictify (λx. strictify(λy. Abs_Bag_0 ⌜Rep_Bag_0 x⌝⌜Rep_Bag_0 y⌝)))
  and inv_g: !!a b. [[ count a ⊥ = 0; count b ⊥ = 0 ]] ===> count (g a b) ⊥ = 0
  shows ((τ ⋵∂ (f X Y)) = ((τ ⋵∂ X) ∧ (τ ⋵∂ Y))
apply(insert f_def)
apply(drule_tac X=X and Y=Y in lift2_strict_is_isdef_fw_Bag_Bag)
apply(rule inv_g, assumption+)
apply(simp add: OclAnd_def OclTrue_def o_def
  OclIsDefined_def lift0_def lift1_def lift2_def
  OclLocalValid_def)
done

The conversion operators

From bags to sequences

Informally speaking the AsSequence operation returns just some sequence that has
exactly the same elements like the given bag.

Note: With the current definition it does not hold that bag->seq->oset = bag->set->oset.
To get this behaviour one has to define AsSequence as some sequence that has the el-
ements in the same order like the associated set converted to a sequence.
defs

  OclAsSequence_def : 
  OclAsSequence ≡ lift1 (strictify (λX. 
    Abs_Sequence_0 ⌜list_of_multiset⌜Rep_Bag_0 X⌝⌟))

ocl_setup_op [OclAsSequence]

lemma OCL_is_defopt_OclAsSequence:
  ((τ ⋵∂ ((self::(′a,b::bot)Bag) -> asSequence))::((′a, ′b::bot)Sequence)) = (τ ⋵∂ self)
apply(simp add: localValidDefined2sem OclAsSequence_def ss_lifting)
apply(rule impl, frule DEF_X_Bag', auto)
done

lemma OCL_is_def_OclAsSequence:
  ((∂ ((self::(′a,b::bot)Bag) -> asSequence))::((′a, ′b::bot)Sequence)) = (∂ self)
apply(rule Bag_sem_cases_ext, simp_all)
apply(simp_all add: OclsDefined_def OclAsSequence_def ss_lifting)
done
Appendix B. Isabelle Theories

Characterisation theorems of the conversion from bags to sequences

lemma OclAsSequence_charn_0:
  assumes undefCase : \( P \bot \bot \tau \)
  and defCase : \( \forall \text{xs}. [\lceil \text{Rep_Bag_0 } B \rceil = \text{multiset_of } xs ; \bot \notin \text{set } xs ] \implies P (\lambda s. (\text{Abs_Sequence_0 } \text{xs})) (\lambda s. B) \tau \)
  shows \( P ((\text{OclAsSequence } (\lambda s. (B;':b::bot)Bag_0)));:('a,'b::bot)Sequence) (\lambda s. B) \tau \)
apply(insert undefCase)
apply(simp add: localValidDefined2sem OclIsDefined_def OclAsSequence_def OclUndefined_def ss_lifting)
apply(rule impI)
apply(frule DEF_X_Bag_0 [THEN iffD1, simplified DEF_def, clarify])
apply(rule list_of_multisetI)
apply(rule defCase)
apply(simp_all add: localValidDefined2sem DEF_def mem_set_multiset_eq)
done

lemma OclAsSequence_charn:
  assumes undefCase : \( P \bot \bot \tau \)
  and defCase : \( \forall \text{xs}. [\lceil \text{Rep_Bag_0 } (B \tau) \rceil = \text{multiset_of } xs ; \bot \notin \text{set } xs ; \tau \vDash \emptyset] \implies P (\lambda s. (\text{Abs_Sequence_0 } \text{xs})) (\lambda s. B) \tau \)
  and cp1_P : \( \forall y. \text{cp} (\lambda x. P x y) \)
  and cp2_P : \( \forall x. \text{cp} (P x) \)
  shows \( P ((\text{OclAsSequence } (B;(':a,'b::bot)Bag));(':a,'b::bot)Sequence) B \tau \)
apply(rule subst, rule_tac x1 = B in sym[OF cp_subst])
apply(simp_all add: cp2_P)
done

Showing the distributivity over the equivalences

Note: Actually it should be possible to develop the theory about the inverse of multiset_of in a more general way by combining the results of Hilbert_Choice.thy and Multiset.thy. But I’m currently not using this approach because only few specifically tailored theorems are needed.

lemma count_filter_inv_multiset_of:
  count M x = length (List.filter (\lambda y. y = x) (list_of_multiset M))
by(auto intro: list_of_multisetI simp: count_filter[symmetric])

322
B.4. Library

**lemma bag2seq_holEq:**
\[(\langle a, b : \text{bot} \rangle \text{Bag}) = Y = (\langle a, b : \text{Sequence} \rangle = (\text{Sequence}) Y)\]
apply(auto simp: OclAsSequence_def ss_lifting' expand_fun_eq)
apply(erule_tac x = x in allE)
apply(case_tac X x \neq \bot, frule DEF_X_Bag')
apply(case_tac Y x \neq \bot, rotate_tac -1, frule DEF_X_Bag')
apply(auto)
apply(subst (asm) Abs_Sequence_0_inject_charn, auto)
apply(subst multiset_eq_conv_count_eq)
apply(subst count_filter_inv_multiset_of)+
apply(simp)
apply(case_tac Y x \neq \bot, rotate_tac -1, frule DEF_X_Bag')
apply(auto)
done

**lemma bag2seq_strictEq:**
\[(\langle a, b : \text{bot} \rangle \text{Bag}) = Y = (\langle a, b : \text{Sequence} \rangle = (\text{Sequence}) Y)\]
apply(rule ext)
apply(simp add: OclStrictEq_def OclAsSequence_def ss_lifting')
apply(rule impI)+
apply(frule DEF_X_Bag', rotate_tac 1)
apply(frule DEF_X_Bag', clarify, simp)
apply(subst Abs_Sequence_0_inject_charn, auto)
apply(subst multiset_eq_conv_count_eq)
apply(subst count_filter_inv_multiset_of)+
apply(simp)
done

**lemma bag2seq_strongEq:**
\[(\langle a, b : \text{bot} \rangle \text{Bag}) \approx Y = (\langle a, b : \text{Sequence} \rangle \approx (\text{Sequence}) Y)\]
apply(rule ext)
apply(auto simp: OclStrongEq_def OclAsSequence_def ss_lifting')
apply(frule DEF_X_Bag', clarify, simp)
apply(frule DEF_X_Bag', clarify, simp)
apply(frule DEF_X_Bag', rotate_tac 1)
apply(frule DEF_X_Bag', clarify, simp)
apply(subst (asm) Abs_Sequence_0_inject_charn, auto)
apply(subst multiset_eq_conv_count_eq)
apply(subst count_filter_inv_multiset_of)+
apply(simp)
done

**lemmas ss_bag2seq = bag2seq_strictEq bag2seq_strongEq**

**Simplifying multiple conversions**

asBag is the inverse of asSequence

**lemma seq2bag_of_bag2seq_UC:**
Appendix B. Isabelle Theories

\[(\text{OclAsBag}((\text{OclAsSequence}(B;'(a',b::bot)\text{Bag}));'(a',b::bot)\text{Sequence})) = B\]
apply(rule Bag_sem_cases_ext, simp_all)
apply(simp_all add: OclAsBag_def OclAsSequence_def ss_lifting')
done

This blowup theorem can be quite useful for conversion proofs about functions that map from a collection type to the same collection type.

lemmas blowup_bag2seq = seq2bag_of_bag2seq_UC[symmetric]

From bags to bags
defs
\(\text{OclAsBag}_\text{def} : \text{OclAsBag}(\text{self};'(a',b::bot)\text{Bag}_0) \text{VAL} \equiv \text{self}\)

lemma bag2bag_id [simp]:
\((\text{OclAsBag}(B;'(a',b::bot)\text{Bag}));'(a',b)\text{Bag}) = B\)
by(simp add: OclAsBag_def)

From bags to ordered sets
defs
\(\text{OclAsOrderedSet}_\text{def} : \text{OclAsOrderedSet} \equiv \text{lift1} (\text{strictify} (\lambda X. \text{Abs_OrderedSet}_0 \text{list_of_set} \text{set_of } \text{Rep_Bag}_0 X'))\)

ocl_setup_op [OclAsOrderedSet]

From bags to sets
defs
\(\text{OclAsSet}_\text{def} : \text{OclAsSet} \equiv \text{lift1} (\text{strictify} (\lambda X. \text{Abs_Set}_0 \text{set_of } \text{Rep_Bag}_0 X'))\)

ocl_setup_op [OclAsSet]

Building a canonic representation of bags

The empty bag
constdefs
\(\text{OclMtBag} :: (a',b::bot)\text{Bag}_0 \text{VAL} \equiv \text{lift0}(\text{Abs_Bag}_0\{\#\})\)

syntax
_\text{OclMtBag}_\text{std} :: (\text{a'},\text{b::bot})\text{VAL} => \text{a Boolean} (\text{Bag}\{\})
syntax
_\text{OclMtBag}_\text{ascii} :: (\text{a'},\text{b::bot})\text{VAL} => \text{a Boolean} (\text{mtBag})
syntax (xsymbols)
_\text{OclMtBag}_\text{math} :: (\text{a'},\text{b::bot})\text{VAL} => \text{a Boolean} (\emptyset)
parse_translation \(\text{\S{ flat(map (emb trans_const) [OclMtBag])}}\)
print_translation \(\text{\S{ map emb_print [OclMtBag]}}\)

324
Foundational Properties of MtBag

lemma OCL_is_isdef_OclMtBag [simp]:
⊢ ∅ 
by (simp add: OclValid_def OclIsDefined_def OclTrue_def OclMtBag_def ss_lifting)

lemma OCL_is_defopt_OclMtBag [simp]:
τ ⊨ ∅ 
by (simp add: valid_elim)

Relating the empty bag to its counterpart on Sequences

lemma bag2seq_mtSequence_mtBag_UC:
(\[] :: ('a', 'b::bot)Sequence) = ((OclAsSequence (\[] :: ('a', 'b::bot)Bag)))
by (simp add: OclAsSequence_def OclMtBag_def OclMtSequence_def ss_lifting)

lemmas ss_bag2seq = ss_bag2seq bag2seq_mtSequence_mtBag_UC [symmetric]
and vice versa

lemma seq2bag_mtBag_mtSequence_UC:
(\[] :: ('a', 'b::bot)Bag) = ((OclAsBag (\[] :: ('a', 'b::bot)Sequence)))
by (simp add: OclAsSequence_def OclMtBag_def OclMtSequence_def ss_lifting)

lemmas ss_seq2bag = seq2bag_mtBag_mtSequence_UC [symmetric]

And the 'cons' operation: including

defs

OclIncluding_def : 
OclIncluding ≡ lift2 (strictify (λX. strictify (λY. Abs_Bag_0 
(\[] (\#Y\#) + Rep_Bag_0 X\}))))

ocl_setup_op [OclIncluding]

lemma OCL_is_isdef_OclIncluding:
(\)(OclIncluding (X :: ('a', 'b::bot)Bag) (Y :: ('a', 'b::bot)VAL)) = (∅ X ∧ ∅ Y)
by (rule lift2_strict_is_isdef_fw_Bag_Val[OF OclIncluding_def],
simp add: neq_commute)

lemma OCL_is_defopt_OclIncluding:
(τ ⊨ ∅ (OclIncluding (X :: ('a', 'b::bot)Bag) (Y :: ('a', 'b::bot)VAL))) = ((τ ⊨ ∅ X) ∧ (τ ⊨ ∅ Y))
by (rule lift2_strictify_implies_LocalValid_defined_Bag_Val[OF OclIncluding_def],
simp add: neq_commute)

The syntax translation mkBag builds now our bags
syntax
@OclFinBag :: args ⇒ ('a, 'b Bag_0)VAL (mkBag{()})
Appendix B. Isabelle Theories

translations

\begin{align*}
mkBag(x, xs) & \equiv \text{OclIncluding}(\text{mkBag}(xs)) x \\
\text{mkBag}(x) & \equiv \text{OclIncluding}\ \text{OclMtBag}\ x
\end{align*}

lemma \(\text{mkBag}\{1,1,1,0,1\} = ?X\) oops

Relating including on Bags to its counterpart on sequences

\begin{align*}
\text{seq2bag\_including\_UC} & : \\
(\text{(OclAsBag}\ S::\ ('a', 'b': bot):\ Bag) \rightarrow \text{including} \ x = \\
(\text{OclAsBag}\ ((S::\ ('a', 'b': bot):\ Sequence) \rightarrow \text{including} \ (x::\ ('a', 'b):\ VAL)))
\end{align*}

apply (rule_tac X=S in Sequence\_sem\_cases\_ext, simp\_all)
apply (case_tac x τ \(\neq \bot\), frule neq_commute[THEN iffD1])
apply (simp\_all add: OCL\_Sequence.OclAsBag\_def OCL\_Sequence.OclIncluding\_def OclIncluding\_def ss\_lifting' mem\_set\_multiset\_eq union\_ac)
done

lemmas ss\_seq2bag = ss\_seq2bag seq2bag\_including\_UC

Singletons can be converted in any direction and don’t lose information

\begin{align*}
\text{bag2seq\_singleton} & : \\
(\text{([]}::\ ('a', 'b': bot):\ Sequence) \rightarrow \text{including} \ x = \\
(\Rightarrow \text{asSequence}) (\text{([]}::\ ('a', 'b):\ Bag) \rightarrow \text{including} \ (x::\ ('a', 'b):\ VAL)))
\end{align*}

apply (rule ext)
apply (simp add: OclMtSequence\_def OclMtBag\_def OclIncluding\_def OclAsSequence\_def OCL\_Sequence.OclIncluding\_def\ ss\_lifting')
done

lemmas ss\_bag2seq = ss\_bag2seq bag2seq\_singleton[ symmetric ]

Further properties of including

\begin{align*}
\text{including\_ord\_Indep} & : \\
((B::\ ('a', 'b': bot):\ Bag) \rightarrow \text{including} \ a \rightarrow \text{including} \ b = \\
(B \rightarrow \text{including} \ (b::\ ('a', 'b):\ VAL)) \rightarrow \text{including} \ (a::\ ('a', 'b):\ VAL))
\end{align*}

apply (rule_tac X=B in Bag\_sem\_cases\_ext, simp\_all)
apply (case_tac DEF (a x), case_tac DEF (a x))
apply (simp add: localValidDefined\_2sem DEF\_def\_both, clarify)
apply (simp\_all add: OclIncluding\_def union\_ac ss\_lifting')
done

A second bag constructor on the OCL level

defs

\begin{align*}
\text{OclCollectionRange\_def} : \\
\text{OclCollectionRange} & \equiv \text{lift2 (strictify (λ x::Integer\_0. strictify (λ y.} \\
\text{Abs\_Bag\_0\ \_multiset\_of (map (λ z::nat. (\text{(int} z + \ l))) [0..<(\text{nat} (\text{nat} y - \ l) + 1)])))})
\end{align*}
B.4. Library

lemma OCL_is_defopt_OclCollectionRange [simp]:
\[
\tau \models \partial ((OclCollectionRange (a::\langle a \text{ Integer} \rangle) b)::\langle a, \text{ Integer}_0 \rangle Bag) = \\
((\tau \models \partial a) \land (\tau \models \partial b))
\]
apply(subgoal_tac count (multiset_of (map (\lambda z. (\langle (\text{int} z) + (\langle a \tau \rangle) \rangle) \langle 0..<(\text{nat (\langle (b \tau \rangle) - (\langle a \tau \rangle) + 1) \rangle) \rangle) | | set (map (\lambda z. (\langle (\text{int} z) + (\langle a x \rangle) \rangle) \langle 0..<(\text{nat (\langle (b x) \rangle - (\langle a x \rangle) + 1) \rangle) \rangle)))
apply(auto simp: OCL_Sequence.OclAsBag_def OclCollectionRange_def ss_lifting)
done

Relating collectionRange on Bags to its counterpart on Sequences

lemma seq2bag_collectionRange_UC:
\[
((OclCollectionRange a b)::\langle a, \text{ Integer}_0 \rangle Bag) = \\
OclAsBag ((OclCollectionRange (a::\langle a \text{ Integer} \rangle) (b::\langle a \text{ Integer} \rangle))::\langle a, \text{ Integer}_0 \rangle Sequence)
\]
apply(rule ext)
apply(case_tac x | \tau \models (b\text{::}\langle a \text{ Integer} \rangle) < (a\text{::}\langle a \text{ Integer} \rangle) \])
apply(auto simp: OCL_Sequence.OclAsBag_def OclCollectionRange_def ss_lifting)
done

lemmas ss_seq2bag = ss_seq2bag seq2bag_collectionRange_UC

Further properties of collectionRange

Having a cp-aware rewriter would be that nice...

lemma collectionRange_mtbag_conv:
\[
[ \tau \models (b::\langle a \text{ Integer} \rangle) < (a::\langle a \text{ Integer} \rangle) ] \implies \\
OclCollectionRange a b = (\langle a, \text{ Integer}_0 \rangle Bag) \tau
\]
apply(simp add: ss_seq2bag)
apply(rule trans, rule_tac P=OclAsBag in cp_charn)
apply(rule OCL_Sequence.collectionRange_mtbSequence_conv)
apply(simp_all add: ss_seq2bag)
done

lemma collectionRange_singleton_UC[simp]:
\[
OclCollectionRange a a = (\langle a, \text{ Integer}_0 \rangle Bag) \text{ including} (a::\langle a, \text{ Integer}_0 \rangle VAL)
\]
by(simp add: seq2bag_collectionRange_UC seq2bag_including_UC[symmetric] seq2bag_mtbag_mtbSequence_UC)

lemma collectionRange_expand_including:
Appendix B. Isabelle Theories

\[ \tau \models (a : 'a Integer) \leq b \quad \rightarrow \quad ((\text{OclCollectionRange } a \ b) :: (\text{`a,Integer}_{\text{Bag}})) \tau = \quad (\text{OclCollectionRange } a \ (b - 1)) \rightarrow \text{including } b \tau \]

apply(simp add: ss_seq2bag)
apply(rule trans, rule_tac P=\text{OclAsBag} in \text{cp_charn})
apply(rule OCL_Sequence.collectionRange_expand_including, simp_all)
done

Iterate
defs

\text{OclIterate_def}:
\quad \text{OclIterate} \equiv (\text{lift3 } \text{lift_arg0} \text{lift_arg2} \text{lift_arg0}) \ (\text{strictify } (\lambda S P A. (\text{fold_multiset } (\lambda x y. P y x) A (\text{⌜Rep_Bag_0} S⌝))))

\text{ocl_setup_op} [\text{OclIterate}]

Its characterisation on the empty bag

lemma \text{iterate_charn_mtBag}[simp]:
\quad (\text{OclIterate } (\text{OclIncluding } B x) P A) \tau = A
by (rule ext, simp add: ss_lifting \text{OclIterate_def} \text{OclMtBag_def})

lemma \text{iterate_charn_including_0}:
\quad \text{assumes ordIndep}: \forall a b c. (P a (\text{lift0} (P b c \tau))) \tau = (P b (\text{lift0} (P a c \tau))) \tau
\quad and \text{defB}: \tau \models \partial (B::(\text{`a,`b:bot})\text{Bag})
\quad and \text{defx}: \tau \models \partial (x::(\text{`a \Rightarrow `b:bot}))
\quad shows ((\text{OclIterate} (\text{OclIncluding } B x) P A) \tau) = (P (\text{lift0} (x \tau)) (\text{lift0} ((\text{OclIterate} (B::(\text{`a,`b:bot})\text{Bag}) P A) \tau)) \tau)
apply(insert defx defB ordIndep)
apply(simp add: localValidDefined2sem)
apply(frule DEF_X_Bag[THEN iffD1])
apply(frule DEF_X_Bag_0[THEN iffD1])
apply(clarify)
apply(simp add: OclIncluding_def \text{OclIterate_def} \text{ss_lifting’} \text{fold_multiset_union_rev} \text{fold_multiset_union_rev}
dek: \text{fold_multiset_union})
done

lemma \text{iterate_charn_including_2}:
\quad \text{assumes ordIndep}: \forall a b c. (P a (P b (P a c))) = (P b (P a c))
\quad and \text{cpP_1}: \lambda y. cp (\lambda x. P x y)
\quad and \text{cpP_2}: \lambda x. cp (P x)
\quad and \text{defB}: \tau \models \partial (B::(\text{`a,`b:bot})\text{Bag})
\quad and \text{defx}: \tau \models \partial (x::(\text{`a \Rightarrow `b:bot}))
\quad shows ((\text{OclIterate} (\text{OclIncluding } B x) P A) \tau) =
\[(P \ x \ (\text{OclIterate} \ (B::('a,'b::bot)\text{Bag}) \ P \ A) \ x)\]
\[\text{apply} \ (\text{rule sym}[\text{OF trans}])\]
\[\text{apply} \ (\text{rule_tac} \ x=x \ \text{in} \ \text{cp} _ \ \text{subst}, \ \text{simp add: cpP} _ 1)\]
\[\text{apply} \ (\text{rule trans})\]
\[\text{apply} \ (\text{rule_tac} \ x=(B \rightarrow \text{iterate}(u;u\text{=}A \ | \ (P \ u \ ua))) \ \text{in} \ \text{cp} _ \ \text{subst})\]
\[\text{apply} \ (\text{insert} \ cpP \ 2, \ \text{simp add: cp_by_cpify \ lift0_def})\]
\[\text{apply} \ (\text{rule sym}[\text{OF iterate_charn_including_0}])\]
\[\text{apply} \ (\text{simp add: ordIndep cp_by_cpify})\]
\[\text{apply} \ (\text{rule defB}, \ \text{rule defx})\]
\[\text{done}\]

\textbf{lemma} iterate_opt_charn_including:
\[\text{assumes} \ \text{ordIndep: \ } \forall \ a \ b \ c. \ (P \ a \ (P \ b \ c)) = (P \ b \ (P \ a \ c))\]
\[and \ P\_\text{undef} _1: \ !!y. \ P \bot = \bot\]
\[and \ P\_\text{undef} _2: \ !!!x. \ P \bot = \bot\]
\[and \ \text{cpP} _1: \ \forall \ y. \ \text{cp} \ (\lambda \ x. \ P \bot \ y) = \bot\]
\[and \ \text{cpP} _2: \ \forall \ x. \ \text{cp} \ ((\text{OclIterate} \ (\text{OclIncluding} \ B \ x) \ P \ A) = (P \ x \ (\text{OclIterate} \ (B::('a,'b::bot)\text{Bag}) \ P \ A))\]
\[\text{shows} \ ((P\_\text{undef} _1 \ P\_\text{undef} _2 \ \text{cpP} _1 \ \text{cpP} _2 \ \text{ordIndep})\]
\[\text{apply} \ (\text{rule exit})\]
\[\text{apply} \ (\text{case_tac} \ xa = \partial B)\]
\[\text{apply} \ (\text{case_tac} \ xa = \partial x)\]
\[\text{apply} \ (\text{rule iterate_charn_including}, \ \text{simp_all})\]
\[\text{apply} \ (\text{rule isUndefined_charn_local[THEN iffD2])}\]
\[\text{apply} \ (\text{rule trans}, \ \text{erule_tac} \ A=x \ \text{in} \ \text{cp} _ \ \text{charn}, \ \text{simp}, \ \text{simp})\]
\[\text{apply} \ (\text{rule sym}[\text{OF trans}], \ \text{erule_tac} \ A=x \ \text{in} \ \text{cp} _ \ \text{charn}, \ \text{simp}, \ \text{simp})\]
\[\text{apply} \ (\text{rule isUndefined_charn_local[THEN iffD2])}\]
\[\text{apply} \ (\text{rule trans}, \ \text{erule_tac} \ A=B \ \text{in} \ \text{cp} _ \ \text{charn}, \ \text{simp}, \ \text{simp})\]
\[\text{apply} \ (\text{rule sym}[\text{OF trans}], \ \text{erule_tac} \ A=B \ \text{in} \ \text{cp} _ \ \text{charn})\]
\[\text{apply} \ (\text{rule_tac} \ P=P \ x \ \text{in} \ \text{cp} _ \ \text{compose2}, \ \text{simp_all})\]
\[\text{done}\]

Forward rules for functions defined by iterate

\textbf{lemma} f_by_iterate_undef_1:
\[\text{\begin{array}{l}
\forall \ A. \ \text{f} \ (B::('a,'b::bot)\text{Bag}) = \\
\quad \text{OclIterate} \ B \ ((P::('a,'b::bot)\text{VAL}) = ('a,'c::bot)\text{VAL} \Rightarrow ('a,'c::bot)\text{VAL}) \ A \\
\text{implies} \\
\quad \text{f} \bot = \bot \\
\quad \text{by(simp)}
\end{array}}\]

\textbf{lemma} f_by_iterate_mtBag:
\[\begin{array}{l}
\forall \ A. \ \text{f} \ (B::('a,'b::bot)\text{Bag}) = \\
\quad \text{OclIterate} \ B \ ((P::('a,'b::bot)\text{VAL}) = \forall \ a \ b. \ (P \ a \ (P \ b \ c)) = (P \ b \ (P \ a \ c)); \\
\quad \forall \ x. \ \text{cp} \ (P \ x) \\
\text{implies} \\
\quad \text{f} \bot = A
\end{array}\]

329
Appendix B. Isabelle Theories

by (simp)

lemma f_by_iterate_including:
  assumes f_by_it: \( \exists B. (f (B::{a,b::bot}Bag)) = (\text{OclIterate } B (('a,b::bot)VAL \Rightarrow ('a,c::bot)VAL \Rightarrow ('a,c::bot)VAL) A) \)
  and ordIndep: \( \forall a\ b\ c. (P a \ (P b \ c)) = (P b \ (P a \ c)) \)
  and defB: \( \tau \models \delta(B::{a,b::bot}Bag) \)
  and defx: \( \tau \models \delta(x::{a \Rightarrow b::bot}) \)
  and cpP_1: \( \land y. \ cp (\lambda x. P x y) \)
  and cpP_2: \( \land x. \ cp (P x) \)
  shows \( (f \ (\text{OclIncluding } B \ x) \tau) = (P x \ (f \ B) \ \tau) \)
  by (simp add: f_by_it iterate_charn_including ordIndep defB defx cpP_1 cpP_2)

lemma f_by_iterate_opt_including:
  assumes f_by_it: \( \exists S. (f (S::{a,b::bot}Bag)) = (\text{OclIterate } S (('a,b::bot)VAL \Rightarrow ('a,c::bot)VAL \Rightarrow ('a,c::bot)VAL) A) \)
  and ordIndep: \( \forall a\ b\ c. (P a \ (P b \ c)) = (P b \ (P a \ c)) \)
  and P_undef_1: \( \forall y. \ P \bot \ y = \bot \)
  and P_undef_2: \( \forall x. \ P x \bot = \bot \)
  and cpP_1: \( \land y. \ cp (\lambda x. P x y) \)
  and cpP_2: \( \land x. \ cp (P x) \)
  shows \( f \ (\text{OclIncluding } S \ x) \tau) = (P x \ (f \ S) \ \tau) \)
  by (simp add: f_by_it iterate_opt_charn_including ordIndep P_undef_1 P_undef_2 cpP_1 cpP_2)

Conversion of iterate from bags to sequences

For iterates mapping from bags to a non-sequence

lemma iterate_bag2seq:
  assumes ordIndep: \( \forall a\ b\ c. (P a \ (P b \ c)) = (P b \ (P a \ c)) \)
  shows \( (\text{OclIterate } (B::{a,b::bot}Bag)) \)
    \( (P::{a,b::bot}VAL \Rightarrow ('a,c::bot)VAL \Rightarrow ('a,c::bot)VAL) A) \)
  = \( (\text{OclIterate } ((\text{OclAsSequence } B)::(a,b::bot))\text{Sequence) } P A) \)
  apply (insert ordIndep, rule ext)
  apply (simp add: OclAsSequence_def OclIterate_def OCL_Sequence.OclIterate_def ss_lifting)
  apply (case_tac B x \( \bot \))
  apply (frule DEF_X_Bag)
  apply (auto simp: fold_multiset_def)
  done

lemma iterate_seq2bag:
  assumes ordIndep: \( \forall a\ b\ c. (P a \ (P b \ c)) = (P b \ (P a \ c)) \)
  and cpP_2: \( \land x. \ cp (P x) \)
  shows \( (\text{OclIterate } (S::{a,b::bot}Sequence)) \)
    \( (P::{a,b::bot}VAL \Rightarrow ('a,c::bot)VAL \Rightarrow ('a,c::bot)VAL) \)
  = 330
\( A ) = \\
( \text{OclIterate} \ ( ( \text{OclAsBag } S ) :: ( 'a, 'b ) \text{Bag} ) \ P A ) \\
\text{apply}(\text{insert ordIndep cpP\_2, rule ext}) \\
\text{apply}(\text{simp add: OCL\_Sequence.OclAsBag\_def OclIterate\_def} \\
\text{OCL\_Sequence.OclIterate\_def ss\_lifting cp\_by\_cpify}) \\
\text{apply}(\text{case\_tac } S x \neq \bot) \\
\text{apply}(\text{frule DEF\_X\_Sequence', clarify}) \\
\text{apply}(\text{auto simp: mem\_set\_multiset\_eq fold\_multiset\_def}) \\
\text{apply}(\text{rule list\_of\_multisetI}) \\
\text{apply}(\text{rule foldl\_right\_order\_independent}) \\
\text{apply}(\text{simp}) \\
\text{done} \\

and for iterates mapping from bags to bags. This lemma characterises the same 
property on lists and multisets and serves as a guide through the proof of the theorem on bags and sequences. Because there the view is a little bit obfuscated by the contextpassingness and undefinedness cases.

\text{lemma}\ foldl\_multiset2list2multiset: \\
\text{assumes}\ mtC\_conv: e = \text{multiset\_of } e' \\
\text{and}\ \text{stepC\_conv}: f = ( \lambda M a. \text{multiset\_of} ( f' (\text{SOME} xs. M = \text{multiset\_of} xs) a)) \\
\text{and}\ \text{ordInsens\_f'}: \forall xs ys a. [ \text{multiset\_of} xs = \text{multiset\_of} ys ] \implies \\
\text{multiset\_of} ( f' xs a) = \text{multiset\_of} ( f' ys a) \\
\text{shows}\ foldl f e xs = \text{multiset\_of} (foldl f' e' xs) \\
\text{apply}(\text{simp add: mtC\_conv stepC\_conv}) \\
\text{apply}(\text{rule sym}) \\
\text{apply}(\text{rule_tac xs=xs in rev\_induct}) \\
\text{apply}(\text{simp\_all}) \\
\text{apply}(\text{rule ordInsens\_f', simp}) \\
\text{apply}(\text{rule someI2\_ex}) \\
\text{apply}(\text{rule surj\_multiset\_of[THEN surjD], simp}) \\
\text{done} \\

\text{lemma}\ iterate\_bag2seq2bag\_0: \\
\text{assumes}\ \text{ordIndep}\ P: \forall a b c. ( P a (\text{lift0} ( P b c )) \tau) = ( P b (\text{lift0} ( P a c )) \tau) \\
\text{and}\ \text{mitC\_conv}: A = \text{OclAsBag } A' \\
\text{and}\ \text{stepC\_conv}: \forall B x B' = \text{OclAsBag} (P' x (\text{OclAsSequence} B)) \\
\text{and}\ \text{ordInsens\_P'}: \forall S S' x \tau. [ (\text{OclAsBag} S \tau) = (( (\text{OclAsBag} S' \tau) :: 'c\::\bot) \text{Bag} \_0) ] \\
\implies \\
\text{OclAsBag} (P' x S) \tau = (( (\text{OclAsBag} (P' x S') \tau) :: 'c\::\bot) \text{Bag} \_0) \\
\text{shows}\ (\text{OclIterate} (B :: ( 'a, 'b ::\bot) \text{Bag} \\
(P :: ( 'a, 'b ::\bot) \text{VAL} \Rightarrow ( 'a, 'c ::\bot) \text{Bag} \Rightarrow ( 'a, 'c ::\bot) \text{Bag} \\
A ) ) \\
= \text{OclAsBag} (\text{OclIterate} ((\text{OclAsSequence} B) :: ( 'a, 'b ::\bot) \text{Sequence} \\
(P' :: ( 'a, 'b ::\bot) \text{VAL} \Rightarrow ( 'a, 'c ::\bot) \text{Sequence} \Rightarrow ( 'a, 'c ::\bot) \text{Sequence} \\
A ') \\
\text{apply}(\text{rule ext}) \\
\text{apply}(\text{rule trans[symmetric]}) \\
\text{apply}(\text{rule_tac P=\text{OclAsBag in cp\_subst, simp add: OCL\_cp\_OclAsBag})
Appendix B. Isabelle Theories

apply(simp add: OclIterate_def OCL_Sequence.OclIterate_def ordIndepP)

apply(simp add: lift3′_def lift_arg0_def lift_arg2_def strictify_def)
apply(rule_tac P=λ t. (?A −→ (?B t) ∧ ?C) ∧ ?Z)
apply(subgoal_tac P= (λ x. OclAsBag (P′ x (OclAsSequence B))))
apply(simp add: mtC_conv)
prefer 2
apply(simp add: stepC_conv[symmetric])
apply(rule_tac P= λ t. ?A −→ (?B t) ∧ ?C)
apply(simp add: OclAsSequence_def OCL_Sequence.OclAsBag_def)
apply(frule DEF_X_Bag′, clarify, simp)
oops

lemma iterate_bag2seq2bag:
assumes ordIndepP : ∃ a b c. (P a (P b c)) = (P b (P a c))
and cpP_2 : ∃ a. cp (P a)
and mtC_conv: A = OclAsBag A′
and stepC_conv: ∃ B. P x B = OclAsBag (P′ x (OclAsSequence B))
and ordInsens_P: ∃ S S′ x. (OclAsBag S x). (OclAsBag S′ x). ('c::bot Bag_0) =⇒
OclAsBag (P′ x S) x. (OclAsBag S x). (OclAsBag S′ x). ('c::bot Bag_0)
shows
(OclIterate (B :: ('a,'b::bot)Bag)
(P :: ('a,'b::bot)VAL ⇒ ('a,'c::bot)Bag ⇒ ('a,'c::bot)Bag)
A)
= OclAsBag (OclIterate ((OclAsSequence B)::('a,'b::bot)Sequence)
(P′ :: ('a,'b::bot)VAL ⇒ ('a,'c::bot)Sequence ⇒ ('a,'c::bot)Sequence)
A′)

Note: If one looks at the conversion theorems about union, including and so on one
sees that asBag can be taken inside. The same theorem should hold on iterate. But
currently I’m not sure about its use, because in all proofs about a conversion of iterate
the “reformulation” theorems (the form: bag2seq2bag) are used.
I don’t know either if the “distributive” version is equally or more powerful. Therefore
the following lemma after the lemmas illustrating the idea on multisets is currently
left unproven.

lemma projection_of_foldl:
assumes projInsens_f: ∃ xs ys a. [ P xs = P ys ] =⇒
P (fold f e xs) = P (fold f e ys)

332
foldl \(\lambda a b. P (\text{SOME} y. a = P y) b\)) (P e) xs

apply (rule_tac zs=xs in rev_induct)
apply (simp_all)
apply (projInsens_f, simp)
apply (rule someI2, rule sym, simp_all)
done

lemma multiset_of_foldl:
assumes ordInsens_f: \(\forall xs ys a. [\text{multiset_of} \ x = \text{multiset_of} \ y] \implies \text{multiset_of} (f \ x = \text{multiset_of} \ y)\)
and ordIndep_f: \(\forall a b c. ((f (f a) b) = (f a (f a) c))\)
shows multiset_of (foldl f e xs) =
  \(\text{SOME} y. \text{multiset_of} (f \ x = \text{multiset_of} \ y)\)

oops

lemma iterate_seq2bag':
assumes ordIndepP: \(\forall a b c. (P' a (P' b c)) = (P' b (P' a c))\)
and P'_def: \(P' = (\lambda x. (B::'(a,c)\text{::bot} Bag)\text{::}(\lambda \text{OclAsSequence}\ B::'(a,c)\text{::bot} Bag))\)
and ordInsens_P: \(\forall S S' x. \text{OclAsBag} S = \text{OclAsBag} S' \implies \text{OclAsBag} (P S x) = \text{OclAsBag} (P S' x)\)
shows \(\text{OclIterate} (\text{OclAsBag} S::'(a,c)\text{::bot} Bag)\)

oops

Union
defs

\text{OclUnion_def} : \text{OclUnion} \equiv \text{lift2} (\lambda X. \text{strictify} (\lambda Y. \text{Abs_Bag_0} X) \text{::}(\lambda \text{Rep_Bag_0} Y X))

ocl_setup_op [OclUnion]

lemma OCL_is_def_OclUnion:
\(\partial (\text{OclUnion} X::'(a,b,c)\text{::bot} Bag) Y = (\partial X \land \partial Y)\)
apply (rule lift2_strict_is_isdef-fw_Bag_Bag [OF OclUnion_def])

333
Appendix B. Isabelle Theories

apply (simp)
done

lemma OCL_is_defopt_OclUnion:
(τ ⊨ ∅ (OclUnion (X::('a,'b::bot)Bag) Y)) = ((τ ⊨ ∅ X) ∧ (τ ⊨ ∅ Y))
apply (rule lift2_strictify_implies_LocalValid_defined_Bag_Bag[OF OclUnion_def])
apply (simp)
done

lemma union_seq2bag:
→asBag() ((→asSequence() (X::('a,'b::bot)Bag))::('a,'b)Bag) ∪ (→asSequence() Y)::('a,'b)Bag
apply (rule_tac X=X in Sequence_sem_cases, simp_all)
apply (simp_all add: OCL_Sequence.OclAsBag_def OclUnion_def mem_set_multiset_eq)
done

lemma union_bag2seq2bag:
→asBag() ((→asSequence() (X::('a,'b::bot)Bag))::('a,'b)Bag) ∪ (→asSequence() Y)
= (X ∪ Y)
by (simp add: union_seq2bag seq2bag_of_bag2seq_UC)

lemma union_by_iterate:
((X::('a,'b::bot)Bag) ∪ (Y::('a,'b::bot)Bag)) =
(Y→iterate(x;y=X | y→including (x::('a,'b::bot)VAL) ))
apply (simp add: union_bag2seq2bag[symmetric] union_by_iterate)
oops

Intersection
defs

OclIntersection_def: OclIntersection ≡ lift2 (strictify (λX. strictify (λY. Abs_Bag_0
(_⌜Rep_Bag_0 X⌝ #∩ _⌜Rep_Bag_0 Y⌜)))

ocl_setup_op [OclIntersection]

We’ll need a second definition for the mixed case of intersection between a set and
a bag ... the same case holds for union.

Includes and excludes
defs

OclIncludes_def : OclIncludes ≡ lift2(strictify (λX. strictify (λx.
(0 < count "Rep_Bag_0 X" x)))

334
**OclExcludes_def**: \( OclExcludes \equiv \text{lift2} (\text{strictify} (\lambda X. \text{strictify} (\lambda x. \text{count} \text{Rep}_\text{Bag}_0 X' x = 0))) \)

**ocl_setup_op** \[ OclIncludes, OclExcludes \]

We prefer a normal form where excludes is written as not includes

**lemma** includes_not_includes [simp]:
\( x \not\in X = \neg (x::(a', b)::\text{bot}) \in (X::(a', b)::\text{bot}) \text{Bag} \)

apply (rule Bag_sem_cases_ext, simp_all)

apply (simp_all add: OclExcludes_def OclNot_def OclIncludes_def ss_lifting)

done

### Relating includes and excludes on Bags to their counterparts on Sequences

**lemma** includes_bag2seq:
\( (a::(a', b)::\text{VAL}) \in (X::(a', b)::\text{bot}) \text{Sequence} = a \in ((\text{OclAsSequence} X):(a', b)::\text{bot}) \text{Sequence} \)

apply (rule Sequence_sem_cases_ext, simp_all)

apply (simp_all add: OclIncludes_def OCL_Sequence.OclIncludes_def OCL_Sequence.OclAsBag_def ss_lifting mem_set_multiset_eq)

done

**lemma** excludes_bag2seq:
\( (a::(a', b)::\text{VAL}) \not\in (X::(a', b)::\text{bot}) \text{Sequence} = a \not\in ((\text{OclAsSequence} X):(a', b)::\text{bot}) \text{Sequence} \)

by (simp add: includes_bag2seq)

building a simplifier set to expand all functions on bags to the ones on sets, if it is possible.

**lemmas** ss_bag2seq = ss_bag2seq includes_bag2seq excludes_bag2seq

and vice versa

**lemma** includes_seq2bag:
\( (a::(a', b)::\text{VAL}) \in (X::(a', b)::\text{bot}) \text{Sequence} = a \in ((\text{OclAsBag} X):(a', b)::\text{bot}) \text{Bag} \)

apply (rule Sequence_sem_cases_ext, simp_all)

apply (simp_all add: OclIncludes_def OCL_Sequence.OclIncludes_def OCL_Sequence.OclAsBag_def ss_lifting mem_set_multiset_eq)

done

**lemma** excludes_seq2bag:
\( (a::(a', b)::\text{VAL}) \not\in (X::(a', b)::\text{bot}) \text{Sequence} = \)

335
Appendix B. Isabelle Theories

\[ a \not\in ((OclAsBag X)::\langle'a,b\rangle Bag) \]

by(simp add: includes_seq2bag

lemmas ss_seq2bag = ss_seq2bag includes_seq2bag excludes_seq2bag

Characterisation of includes on the semantic part

lemma includes_charn1 :
\[ (\tau \models x \in S) \implies (0 < \text{count}\ Rep\ Bag\ 0\ (S\ \tau)\ (x\ \tau)) \]

apply(frule isDefined_if_valid)

apply(drule OCL_is_defopt_OclIncludes[THEN iffD1])

apply(simp add: OclLocalValid_def OclIsDefined_def strictify_def
               OclNot_def OclIncludes_def
               o_def lift0_def lift1_def lift2_def
               OclTrue_def OclFalse_def OclUndefined_def DEF_def)

done

lemma includes_charn2 :
\[ [\ \tau \models DEF(S\ \tau); 0 < \text{count}\ Rep\ Bag\ 0\ (S\ \tau)\ (x\ \tau) ] \implies \tau \models x \in S \]

apply(simp add: OclLocalValid_def OclIsDefined_def strictify_def

OclNot_def OclIncludes_def

o_def lift0_def lift1_def lift2_def

OclTrue_def OclFalse_def OclUndefined_def DEF_def

apply(auto)

done

Higher Properties of includes and excludes

lemma includes_of_mtBag [simp]:
\[ \neg (\tau \models (x::\langle'a, 'b\rangle VAL) \in (\langle a::\langle a, b::\bot\rangle Bag) ) \]

by(simp only: ss_bag2seq, rule includes_of_mSequence)

lemma includes_charn1_including[simp]:
\[ [\ \tau \models \partial (a::\langle'a, 'b\rangle VAL); \tau \models \partial (X::\langle a, b::\bot\rangle Bag) ] \implies \tau \models a \in (X\rightarrow\text{including}\ a) \]

apply(simp only: localValidDefined2sem DEF_def

OclNot_def OclIncludes_def

o_def lift0_def lift1_def lift2_def

OclTrue_def OclFalse_def OclUndefined_def DEF_def

apply(frule DEF_X_Bag', clarify)

apply(simp add: localValid2sem OclIncludes_def OclIncluding_def

ss_lifting\ neq_commute)

done

lemma includes_charn2_including[simp]:
\[ [\ \tau \models \partial (a::\langle'a, 'b\rangle VAL); \tau \models \partial (b::\langle'a, 'b\rangle VAL); \tau \models a \in (X::\langle a, b::\bot\rangle Bag) ] \implies \tau \models a \in (X\rightarrow\text{including}\ b) \]

apply(case_tac X \ \tau \neq \bot)

apply(frule DEF_X_Bag', clarify)
It would be more favourable to have this theorem proven on sets first and lift it then to all other collections. Because it doesn’t work to lift it in the other direction, from sequences to bags, sets and ordered sets.

lemma includes_of_collectionRange:

\[
(x::('a Integer)) \in ((OclCollectionRange a b)::('a, Integer) Bag) = \\
((a::('a Integer)) \leq x) \land (x \leq (b::('a Integer)))
\]

apply(rule ext)
apply(case_tac DEF (x xa))
apply(case_tac DEF (a xa))
apply(case_tac DEF (b xa))
apply(simp add: DEF_def_both, clarify)
apply(subgoal_tac count (multiset_of (map (λ z. ([](z + (a xa)))) [0..<nat ((b xa) - (a xa))]))) ⊥ = 0)
apply(arith)
apply(rule_tac x = nat((a xa) - (a xa)) in image_eqI, auto)
apply(simp add: count_filter)

done

lemma cp_sor [simp,intro!]: \[ cp P; cp P' \implies cp (λ X. P' X) \]
by(simp add: OclSor_def)

lemma includes_of_union [simp]:

\[
((a::('a,'b::bot) VAL) \in ((X::('a,'b::bot) Bag) \cup Y))= \\
((a \in X) \lor (a \in Y))
\]

apply(rule_tac X=X in Bag_sem_cases_ext, simp_all)
apply(rule_tac X=Y in Bag_sem_cases, simp_all)
apply(case_tac τ ⊨ ≠ a, ocl_hypsubst, simp add: OclSor_def)
apply(simp add: localValidDefined2sem DEF_def_both, clarify)
apply(simp_all add: OclIncludes_def OclUnion_def OclSor_def
OclSand_def OclNot_def ss_lifting)

done

lemma not_includes_of_mtBag:

\[
(τ \not\models (x::('a,'b::bot) Bag)) = (τ \models ∅)
\]

by(simp only: ss_bag2seq not_includes_of_mtSequence)

Emptiness
defs
Appendix B. Isabelle Theories

\textbf{OclIsEmpty\_def} : \texttt{OclIsEmpty} ≡ \texttt{lift1 (strictify (λX. \{#\} = \texttt{Rep\_Bag\_0 X'))})

\textbf{OclNotEmpty\_def} : \texttt{OclNotEmpty} ≡ \texttt{lift1 (strictify (λX. \{#\} ≠ \texttt{Rep\_Bag\_0 X'))})

\textbf{ocl\_setup\_op} \{\texttt{OclIsEmpty, OclNotEmpty}\}

we prefer a canonical form were notEmpty is written using isEmpty

\textbf{lemma notEmpty\_not\_isEmpty\_conv\{simp\}}:
\texttt{¬ ∅ (X::\{}\′a,\′b::\texttt{bot}\texttt{)} Bag = ¬ (\texttt{∅ X})

apply\(\texttt{rule ext})

apply\(\texttt{simp add: OclNotEmpty\_def OclIsEmpty\_def OclNot\_def ss\_lifting})

\texttt{done}

\textbf{Relating isEmpty and isNotEmpty on Bags to their counterparts on Sequences}

\textbf{lemma isNotEmpty\_bag2seq}:
\texttt{OclNotEmpty \texttt{(X::\{}\′a,\′b::\texttt{bot}\texttt{)} Bag}) = \texttt{OclNotEmpty \texttt{(OclAsSequence X)::\{}\′a,\′b::\texttt{bot}\texttt{)}}

apply\(\texttt{rule Sequence\_sem\_cases\_ext})

apply\(\texttt{simp all add: OclAsBag\_def ss\_lifting})

\texttt{done}

\textbf{lemma isNotEmpty\_bag2seq}:
\texttt{OclNotEmpty \texttt{(X::\{}\′a,\′b::\texttt{bot}\texttt{)} Bag}) = \texttt{OclNotEmpty \texttt{(OclAsSequence X)::\{}\′a,\′b::\texttt{bot}\texttt{)}}

apply\(\texttt{rule Sequence\_sem\_cases\_ext})

apply\(\texttt{simp all add: OclAsBag\_def ss\_lifting})

\texttt{done}

\textbf{lemmas ss\_bag2seq = ss\_bag2seq isNotEmpty\_bag2seq isNotEmpty\_bag2seq}

from sequences to bags

\textbf{lemma isEmpty\_seq2bag}:
\texttt{OclIsEmpty \texttt{(X::\{}\′a,\′b::\texttt{bot}\texttt{)} Sequence}) = \texttt{OclIsEmpty \texttt{(OclAsBag X)::\{}\′a,\′b\texttt{)}}

apply\(\texttt{rule Sequence\_sem\_cases\_ext})

apply\(\texttt{simp all add: OclAsBag\_def ss\_lifting})

\texttt{done}

338
apply (auto simp: mem_set_multiset_eq)
done

lemma isNotEmpty_seq2bag:
OclNotEmpty (X::(′a,′b::bot)Sequence) = OclNotEmpty ((OclAsBag X)::(′a,(′b::bot))Bag)
by (simp only: isNotEmpty_seq2bag notEmpty_not_isEmpty_conv OCL_Sequence.notEmpty_not_isEmpty_conv)

lemmas ss_seq2bag = ss_seq2bag isEmpty_seq2bag isNotEmpty_seq2bag

Further properties of the emptiness test

lemma isEmpty_stricteq_mtBag_conv[simp]:
(= ∅ X) = ((X::(′a,′b::bot)Bag) = (\))
apply (rule ext)
apply (case_tac X x ≠ ⊥)
apply (frule DEF_X_Bag', clarify)
apply (simp_all add: OclIsEmpty_def OclMtBag_def OclStrictEq_def Abs_Bag_0_inject_charn ss_lifting)
apply (blast)
done

lemma isEmpty_of_mtBag:
τ ⊨ = ∅ by (simp)

lemma isEmpty_of_including:
[ τ ⊨ a::(′a,′b)VAL; τ ⊨ X::(′a,′b::bot)Bag ] ⇒
τ ⊨ ¬ (= ∅ (X→including a))
oops

lemma notEmpty_of_mtBag:
τ ⊨ ≠ ∅ by (simp)

lemma notEmpty_of_including:
[ τ ⊨ a::(′a,′b)VAL; τ ⊨ X::(′a,′b::bot)Bag ] ⇒
τ ⊨ ≠ ∅ (X→including a)
oops

Size
defs

OclSize_def ::(OclSize::(′a, ′b::bot Bag_0) VAL =⇒ ′a Integer) ≡
lift1 (strictify(λX. ′int (size(" Rep_Bag_0 X")), ))

ocl_setup_op [OclSize]
Appendix B. Isabelle Theories

Relating size on bags to its counterpart on sequences

lemma size_multiset_of: (size (multiset_of xs)) = (length xs)
by (induct xs, auto)

lemma size_bag2seq:
  ((X :: ((′a,′b::bot)Bag))) = 
  (((OclAsSequence X)::′a,′b::bot)Sequence))
apply (rule ext)
apply (rule_tac B=X in OclAsSequence_charn, simp_all)
apply (frule DEF_X_Bag', clarify)
apply (auto simp: OclSize_def OCL_Sequence size_multiset_of)
done

lemmas ss_bag2seq = ss_bag2seq size_bag2seq

lemma size_seq2bag:
  ((X :: ((′a,′b::bot)Sequence))) = 
  (((OclAsBag X)::′a,′b::bot)Bag))
apply (rule Sequence_sem_cases_ext, simp_all)
apply (auto simp: OclSize_def OCL_Sequence size_multiset_of)
done

lemmas ss_seq2bag = ss_seq2bag size_seq2bag

Further properties of size

The real power of the proofs by conversion

lemma size_by_iterate:
  ((X :: ((′a,′b::bot)Bag)))) = 
  OclIterate X (λ (x::′a,′b::bot) VAL) (y::′a Integer). y + 1) 0
by (simp add: ss_bag2seq OCL_Sequence.size_by_iterate iterate_bag2seq)

Count
defs

OclCount_def : OclCount ≡ lift2 (strictify (λX. strictify (λx.
  int (count Rep_Bag_0 X' x))))

ocl_setup_op [OclCount]
B.4. Library

Relating count on bags to its counterpart on sequences

**lemma** count_bag2seq:

\[(X::(\text{'a,\text{'b}:\text{bot})Bag) \rightarrow count (a::(\text{'a,\text{'b})VAL}) =
\]  
\[(\rightarrow\text{asBag()} X)::(\text{'a,\text{'b}:\text{bot})Bag) \rightarrow count a
\]

apply\(\text{rule ext})

apply\(\text{auto simp: localValidDefined2sem DEF_def})

apply\(\text{induct_tac xs, simp_all})

done

**lemmas** ss_bag2seq = ss_bag2seq count_bag2seq

**lemma** count_seq2bag:

\[(X::(\text{'a,\text{'b}:\text{bot})Sequence) \rightarrow count (a::(\text{'a,\text{'b})VAL}) =
\]  
\[\rightarrow\text{asBag()} X)::(\text{'a,\text{'b}:\text{bot})Bag) \rightarrow count a
\]

apply\(\text{auto simp: OclCount_def OCL_Sequence.OclCount_def
}

\[\text{OclAsBag_def ss_lifting' mem_set_multiset_eq})
\]

apply\(\text{induct_tac c, simp_all})

done

**lemmas** ss_seq2bag = ss_seq2bag count_seq2bag

Further properties of count

**lemma** count_by_iterate:

\[(X::(\text{'a,\text{'b}:\text{bot})Bag) \rightarrow count ((x::(\text{'a,\text{'b})VAL})) =
\]  
\[\rightarrow\text{OclIterate X} \lambda (x::(\text{'a,\text{'b}:\text{bot})VAL) (y::\text{'a Integer}).if (x \oplus a) then (y+1) else (y) endif
\]

\[\rightarrow\text{endif})
\]

apply\(\text{simp add: ss_bag2seq count_by_iterate})

apply\(\text{subst iterate_bag2seq, simp_all})

apply\(\text{rule allI, rule ext})

apply\(\text{simp add: OclIf_def plus_def One_ocl_int_def ss_lifting'})

done

**lemma** includes_by_count_UC\(\text{simp}):

\[(X \rightarrow count a) > \emptyset \rightarrow (a::(\text{'a,\text{'b})VAL}) \in (X::(\text{'a,\text{'b}:\text{bot})Bag)
\]

by\(\text{simp only: ss_bag2seq OCL_Sequence.includes_by_count_UC})

**lemma** excludes_by_count_UC\(\text{simp}):

\[\text{OclCount S x \oplus a = \rightarrow ((x::(\text{'a,\text{'b})VAL}) \in (S::(\text{'a,\text{'b}:\text{bot})Sequence))
\]

by\(\text{simp only: ss_bag2seq OCL_Sequence.excludes_by_count_UC})
Appendix B. Isabelle Theories

Forall
defs

\( OclForAll \text{ _def}: OclForAll \equiv (\text{lift}_2' \text{ lift}_0' \text{ lift}_1' \text{ lift}_0') \circ \text{strictify}(\land S, (\lambda P. \\
  \text{if } \forall x \in \text{set_of} \ Rep\_Bag\_0 S. P x = \bot \text{ then } \bot, \\
  \text{else if } \exists x \in \text{set_of} \ Rep\_Bag\_0 S. P x = \bot \text{ then } \bot, \\
  \text{else } \bot)) \)

ocl_setup_op [OclForAll]

Relating forall on bags to its counterpart on sequences
lemma forall_bag2seq:
\( (\forall x \in (X :: (\{a, b\} :: \text{Bot}) \text{Bag}) \circ P (x) :: (\{a, b\} :: \text{Bot}) \text{Seq} \Rightarrow \\text{Boolean}) x = \\
(\forall x \in ((\text{OclAsSequence} X) :: (\{a, b\} :: \text{Bot}) \text{Seq}) \circ P x) \)
apply(rule ext)
apply(rule_tac B = X in OclAsSequence_charn, simp_all)
apply(simp_all add: OclForAll_def OCL_Sequence.OclForAll_def OclUndefined_def
OclAsSequence_def mem_set_multiset_eq cp_by_cpify
ss_lifting
\text{localValidDefined2sem})
done

lemmas ss_bag2seq = ss_bag2seq forall_bag2seq

Further properties of forall
lemma forall_by_iterate:
\( (\forall x \in (B :: (\{a, b\} :: \text{Bot}) \text{Bag}) \circ (P :: (\{a, b\} :: \text{Bot}) \text{VAL} \Rightarrow \\text{Boolean}) x) = \\
(B \rightarrow \text{iterate}(x,y \leftarrow T | (P x) \land y)) \)
apply(simp add: forall_by_iterate ss_bag2seq)
apply(subst iterate_bag2seq, simp_all add: and_assoc)
apply(simp add: and_commute)
done

Exists
Relating exists on bags to its counterpart on sequences
lemma exists_bag2seq:
\( (\exists x \in (B :: (\{a, b\} :: \text{Bot}) \text{Bag}) \circ P (x) :: (\{a, b\} :: \text{Bot}) \text{VAL} \Rightarrow \\text{Boolean}) x = \\
(\exists x \in ((\text{OclAsSequence} B) :: (\{a, b\} :: \text{Bot}) \text{Seq}) \circ P x) \)
by(simp add: OclExists_def forall_bag2seq)

lemmas ss_bag2seq = ss_bag2seq exists_bag2seq

Further properties of exists
lemma exists_by_iterate:
\( (\exists x \in (B :: (\{a, b\} :: \text{Bot}) \text{Bag}) \circ (P :: (\{a, b\} :: \text{Bot}) \text{VAL} \Rightarrow \\text{Boolean}) x) = \\
(B \rightarrow \text{iterate}(x,y \leftarrow F | (P x) \lor y)) \)
apply(simp add: exists_by_iterate ss_bag2seq)
apply (subst iterate_bag2seq, simp_all)
apply (subst or_commute)
apply (simp add: or_assoc)
apply (simp add: or_commute)
done

IncludesAll

defs

OclIncludesAll_def: OclIncludesAll ≡ lift2 (strictify (λX. strictify (λY.
set_of "Rep_Bag_0 Y" ⊆ set_of "Rep_Bag_0 X")))

ocl_setup_op [OclIncludesAll]

Relating includesAll on bags to its counterpart ond sequences

lemma includesAll_bag2seq:
  OclIncludesAll (X::(′a::bot)Bag) (Y::(′a,′b::bot)Bag) =
  OclIncludesAll ((OclAsSequence X)::(′a,′b)Sequence) ((OclAsSequence Y)::(′a,b)Sequence)
  apply (rule ext)
  apply (rule_tac B = X in OclAsSequence_charn, simp_all)
  apply (rule_tac B = Y in OclAsSequence_charn, simp_all)
  apply (simp add: OclIncludesAll_def OCL_Sequence.oclIncludesAll_def
    OclAsSequence_def mem_set_multiset_eq
    localValidDefined2sem ss_lifting)
done

lemmas ss_bag2seq = ss_bag2seq includesAll_bag2seq

Further properties of includesAll

lemma includesAll_by_iterate:
  OclIncludesAll X (Y::(′a::bot,′b)Bag) =
  (Y->iterate (x,y= if (∅ X) then T else undefined | ((x::(′a,b)VAL) ∈ X) ∧ y))
  apply (simp add: includesAll_by_iterate ss_bag2seq)
  apply (subst iterate_bag2seq)
  apply (simp_all add: OCL_is_def_OclAsSequence)
  apply (auto, rule ext, auto simp: OclSand_def ss_lifting)
done

ExcludesAll

defs

OclExcludesAll_def: OclExcludesAll ≡ lift2 (strictify (λX. strictify (λY.
set_of "Rep_Bag_0 X" ∩ set_of "Rep_Bag_0 Y" = {})))

ocl_setup_op [OclExcludesAll]
Appendix B. Isabelle Theories

Relating excludesAll on bags to its counterpart on sequences

**Lemma** excludesAll_bag2seq

\( OclExcludesAll \left( X :: (\forall a, b :: bot) Bag \right) \Rightarrow \exists Y :: (\forall a, b :: bot) Bag \) =

\( OclExcludesAll \left( \left( OclAsSequence X \right) :: (\forall a, b :: bot) Sequence \right) \Rightarrow \exists Y :: (\forall a, b :: bot) Sequence \) =

apply (rule ext)

apply (rule_tac B = X in OclAsSequence_charn, simp_all)

apply (rule_tac B = Y in OclAsSequence_charn, simp_all)

apply (simp add: OclExcludesAll_def OCL_Sequence.

OclAsSequence_def mem_set_multiset_eq

localValidDefined2sem ss_lifting)

done

**Lemmas** ss_bag2seq = ss_bag2seq excludesAll_bag2seq

Further properties of excludesAll

**Lemma** excludesAll_by_iterate

\( OclExcludesAll X \Rightarrow \exists Y :: (\forall a, b :: bot) Bag \) =

\( \forall x \in \text{set_of} \left\{ \text{Rep_Bag}_0 S \right\} . \text{DEF} \left( P x \right) \land \left( x :: (\forall a, b :: bot) VAL \Rightarrow \top \right) \) =

apply (simp add: excludesAll_by_iterate ss_bag2seq)

apply (subst iterate_bag2seq)

apply (simp_all add: OCL_is_def_OclAsSequence)

apply (auto, rule ext, auto simp: OclSand_def ss_lifting)

done

Select

**Defs**

\( OclSelect \equiv (\text{lift2'} lift_arg0 lift_arg1) \left( \text{strictify} \left( \lambda S P. \right. \right.

\left. \begin{array}{ll}
\forall x \in \text{set_of} \left\{ \text{Rep_Bag}_0 S \right\} . \text{DEF} \left( P x \right) \\
\text{then Abs_Bag_0 MCollect \text{Rep_Bag}_0 S \left( \lambda x. P x = \top \right) \right) \\
\text{else } \bot
\end{array} \right) ) \)

ocl_setup_op [OclSelect]

Relating select on bags to its counterpart on sequences

**Lemma** select_bag2seq2bag

\( OclSelect \left( X :: (\forall a, b :: bot) Bag \right) \Rightarrow \exists Y :: (\forall a, b :: bot) VAL \Rightarrow 'a Boolean \) =

\( \left( OclAsBag \left( OclSelect \left( \left( OclAsSequence X \right) :: (\forall a, b :: bot) Sequence \right) \right) \right) \)

apply (rule ext)

apply (case_tac X \( x \neq \bot \))

apply (frule DEF_X_Bag', clarify)

apply (auto simp: OclAsSequence_def OCL_Sequence. OclAsBag_def

OclSelect_def OCL_Sequence. OclSelect_def

ss_lifting)

done

**Lemmas** ss_bag2seq2bag = select_bag2seq2bag

344
Further properties of select lemma select_by_iterate:

\[(\forall x : (B := \text{\texttt{\textquoteleft\textquoteleft}a,\texttt{\textquoteleft\textquoteleft}b::bot\texttt{\textquoteleft\textquoteleft}}))bag | ((P := \text{\texttt{\textquoteleft\textquoteleft}a,\texttt{\textquoteleft\textquoteleft}b::bot\texttt{\textquoteleft\textquoteleft}}))\text{\texttt{\textquoteleft\textquoteleft}true\texttt{\textquoteleft\textquoteleft}})) = (B \rightarrow iterate (x, y = \text{\texttt{\{\{\}}} | if (P x) then (y \rightarrow including x) else y endif ))\]

oops

Reject
defs

\[\text{OclReject}_\text{def} : \text{OclReject} \equiv (lift2 lift_\text{arg0} lift_\text{arg1}) (\text{strictify} (\lambda S P. if \forall x \in \text{set of } \overline{\text{Rep_Bag_0}} S. \text{DEF} (P x) then Abs_Bag_0 (\text{MCollect} \overline{\text{Rep_Bag_0}} S (\lambda x. P x = \text{\textquoteleft\textquoteleft}false\texttt{\textquoteleft\textquoteleft}))) else \bot))\]

ocl\_setup\_op [OclReject]

Excluding
defs

\[\text{OclExcluding}_\text{def} : \text{OclExcluding} \equiv lift2 (\text{strictify} (\lambda X. \text{strictify} (\lambda Y. \text{Abs_Bag_0} (\text{MCollect} \overline{\text{Rep_Bag_0}} X (\lambda a. a \neq Y)))))\]

ocl\_setup\_op [OclExcluding]

Collect

Note currently this collect corresponds to a collectNested. This will be fixed later.
defs

\[\text{OclCollect}_\text{def} : \text{OclCollect} S P \equiv \lambda \tau. if \text{DEF} (S \tau) then if \forall x \in \text{set of } \overline{\text{Rep_Bag_0}} S. \text{DEF} (P (\lambda x. \tau)) then Abs_Bag_0 (\text{Abs_multiset} ((\text{count} \overline{\text{Rep_Bag_0}} S) o (\text{inv} (\lambda x. P (\text{lift0} x) \tau)))) else \bot else \bot\]

ocl\_setup\_op [OclCollect]

One
defs

\[\text{OclOne}_\text{def} : \text{OclOne} (S := \text{\texttt{\textquoteleft\textquoteleft}a,\texttt{\textquoteleft\textquoteleft}b::bot\texttt{\textquoteleft\textquoteleft}}))bag_0 VAL) P \equiv\]
Appendix B. Isabelle Theories

\[ (((x : S \mid P (x::'a::type \Rightarrow 'b::bot))) \| = 1 \]

ocl_setup_op [OctOne]

Sum
Like collect sum is specified as mapping to other ocl functions . . . this is because for every type supporting plus, zero the sum function must be available and it make no sense to add a new definition for every one of these

defs

OclSum_def : 
OclSum S \equiv OclIterate (S::('a,'b::{bot,plus,zero})Bag) (\lambda x y. x + y) 0

Complement
defs

OclComplement_def:OclComplement \equiv lift1 (strictify(\lambda X. Abs_Bag_0 ((\_,-(\langle\text{Rep_Bag_0 X}\rangle - \{\#\bot\#\})))))

ocl_setup_op[OclComplement]

Stuff that should be moved into the right theory
end

B.4.12. OCL OrderedSet
theory OCL_OrderedSet
imports
$\text{HOLOCL_HOME}/$src/library/collection/smashed/OCL_Sequence
begin

Properties of the Congruence inside the Datatype Adaption
The following rules are transformed versions of the automatically generated rules for datatype conversions.

lemma Abs_OrderedSet_inject_charn:
[ (\bot::'a::bot) \notin set x; (\bot::'a::bot) \notin set y; distinct x; distinct y ] \implies
(((Abs_OrderedSet_0 x) = (Abs_OrderedSet_0 y)) \Rightarrow (x = y))
by(subst Abs_OrderedSet_0_inject,
    simp_all add: OrderedSet_0_def smash_def)
lemma Abs_OrderedSet_0_inject_absurd11 [simp]:
\[\bot \notin \text{set } x; \text{distinct } x \implies ((\text{Abs}\_\text{OrderedSet}_0 \downarrow x) = (\text{Abs}\_\text{OrderedSet}_0 \downarrow x)) = False\]
by\{(subst Abs_OrderedSet_0_inject, simp \_ all add: OrderedSet_0_def smash_def)\}

lemma Abs_OrderedSet_0_inject_absurd12 [simp]:
\[\bot \notin \text{set } x; \text{distinct } x \implies ((\text{Abs}\_\text{OrderedSet}_0 \downarrow x) = (\text{Abs}\_\text{OrderedSet}_0 \downarrow x)) = False\]
by\{(subst Abs_OrderedSet_0_inject, simp \_ all add: OrderedSet_0_def smash_def)\}

lemma Abs_OrderedSet_0_inject_absurd21 [simp]:
\[\bot \notin \text{set } x; \text{distinct } x \implies ((\text{Abs}\_\text{OrderedSet}_0 \downarrow x) = \bot) = False\]
by\{(simp add: UU_OrderedSet_def)\}

lemma Abs_OrderedSet_0_inject_absurd22 [simp]:
\[\bot \notin \text{set } x; \text{distinct } x \implies ((\text{Abs}\_\text{OrderedSet}_0 \downarrow x) = \bot) = False\]
by\{(simp add: UU_OrderedSet_def)\}

lemma Abs_OrderedSet_0_inject_charn:
\[
\begin{align*}
\{\bot::'a::bot \notin \text{set } x; \text{distinct } x; \text{distinct } y \}\implies ((\text{Abs}\_\text{OrderedSet}_0 \downarrow x) = (\text{Abs}\_\text{OrderedSet}_0 \downarrow y)) = (x = y)
\end{align*}
\]
by\{(subst Abs_OrderedSet_0_inject, simp \_ all add: OrderedSet_0_def smash_def)\}

lemma Abs_OrderedSet_0_inverse_charn1 [simp]:
\[
\begin{align*}
\{\bot \notin \text{set } x; \text{distinct } x \}\implies \text{Rep}_\text{OrderedSet}_0 (\text{Abs}\_\text{OrderedSet}_0 \downarrow x) = x
\end{align*}
\]
by\{(subst Abs_OrderedSet_0_inverse, simp \_ all add: OrderedSet_0_def smash_def)\}

lemma Abs_OrderedSet_0_inverse_charn2 [simp]:
\[
\begin{align*}
\{\bot \notin \text{set } x; \text{distinct } x \}\implies \text{Rep}_\text{OrderedSet}_0 (\text{Abs}\_\text{OrderedSet}_0 \downarrow x) = x
\end{align*}
\]
by\{(subst Abs_OrderedSet_0_inverse, simp \_ all add: OrderedSet_0_def smash_def)\}

lemma Abs_OrderedSet_0_cases_charn:
assumes bottomCase: \{\{x = \bot\}\}\implies P
assumes listCase : \forall y. (\{x = \text{Abs}\_\text{OrderedSet}_0 \downarrow y; \bot \notin \text{set } y; \text{distinct } y\}\implies P)
shows P
apply\{rule_tac x=x in Abs_OrderedSet_0_cases\}
apply\{case_tac \_ y = \bot\}
apply\{rule bottomCase, simp add: UU_OrderedSet_def\}
apply\{frule not_down_exists_lift2\[THEN iffD1\]\}
apply\{rule_tac y=y\} in listCase
apply\{auto simp: OrderedSet_0_def smash_def\}
done
Appendix B. Isabelle Theories

lemma Abs_OrderedSet_0_induct_charn:
  assumes bottomCase : P ⊥
  assumes stepCase : ∀y. (⊥ ∉ set y; distinct y) ⇒ P (Abs_OrderedSet_0 (y))
  shows P x
  apply (rule_tac x=x in Abs_OrderedSet_0_induct)
  apply (rule_tac x=Abs_OrderedSet_0 y in Abs_OrderedSet_0_cases_charn)
  apply (simp_all add: bottomCase stepCase)
  done

lemma inj_on_Abs_OrderedSet: inj_on Abs_OrderedSet_0 OrderedSet_0
  by (rule inj_on_inverseI, rule OrderedSet_0.Abs_OrderedSet_0_inverse, assumption)

lemma inj_Rep_OrderedSet: inj Rep_OrderedSet_0
  by (rule inj_on_inverseI, rule OrderedSet_0.Rep_OrderedSet_0_inverse)

lemma smashed_orderedSet_charn:
  (⊥ ∉ set X ∧ distinct X) = ((X) ∈ OrderedSet_0)
  by (unfold smash_def OrderedSet_0_def UU_OrderedSet_def, auto)

lemma UU_in_smashed_orderedSet [simp]: ⊥ ∈ OrderedSet_0
  by (unfold smash_def OrderedSet_0_def UU_OrderedSet_def, auto)

lemma down_in_smashed_orderedSet [simp]: down ∈ OrderedSet_0
  by (unfold smash_def OrderedSet_0_def UU_OrderedSet_def, auto)

lemma mt_in_smashed_orderedSet[simp]:
  ([]) ∈ OrderedSet_0
  by (unfold smash_def OrderedSet_0_def UU_OrderedSet_def, auto)

lemma DEF_Abs_OrderedSet:
  ∀X. (⊥ ∉ set X; distinct X) ⇒ DEF (Abs_OrderedSet_0 (X))
  apply (unfold DEF_def UU_OrderedSet_def, simp)
  done

lemma DEF_Rep_OrderedSet:
  ∀X. DEF X ⇒ DEF (Rep_OrderedSet_0 X)
  apply (unfold DEF_def UU_OrderedSet_def, auto)
  apply (drule_tac f = Abs_OrderedSet_0 in arg_cong)
  apply (simp add: Rep_OrderedSet_0_inverse)
  done

lemma not_DEF_Rep_OrderedSet:
  ∀X. ¬ DEF X ⇒ ¬ DEF (Rep_OrderedSet_0 X)
  apply (unfold DEF_def UU_OrderedSet_def, auto)
  apply (simp add: Abs_OrderedSet_0_inverse)
  done
B.4. Library

lemma exists_lift_OrderedSet:
\( \forall X. \text{DEF } X \implies \exists \ c. \ \text{Rep\_OrderedSet\_0 } X = (\langle c, c \rangle) \)
by (drule DEF_Rep_OrderedSet, simp add: DEF_X up)

lemma exists_lift_OrderedSet2:
\( \forall X. \text{DEF } X \implies \exists \ c. \ \bot \notin \text{set } c \land \text{distinct } c \land \text{Rep\_OrderedSet\_0 } X = (\langle c, c \rangle) \)
apply (frule exists_lift_OrderedSet)
apply (clarify, simp)
apply (erule smashed_orderedSet_charn \[ THEN iffD2 \])
apply (erule sm4, simp add: Rep_OrderedSet_0)
done

lemma Rep_OrderedSet_cases:
\( \text{Rep\_OrderedSet\_0 } X = \bot \lor (\exists \ c. \ \bot \notin \text{set } c \land \text{distinct } c \land \text{Rep\_OrderedSet\_0 } X = (\langle c, c \rangle)) \)
apply (case_tac \text{DEF } X)
apply (drule exists_lift_OrderedSet2)
apply (drule_tac \[2\] not_DEF_Rep_OrderedSet)
apply (simp_all add: DEF_def)
done

lemma DEF_X_OrderedSet_0:
\( \text{DEF } (\text{X}) = (\exists \ c. \ \bot \notin \text{set } c \land \text{distinct } c \land \text{X} = \text{Abs\_OrderedSet\_0 } (\langle c, c \rangle)) \)
apply (auto simp: DEF_Abs_OrderedSet)
apply (simp add: DEF_X_OrderedSet_0)
apply (erule exE, rule exI, auto)
apply (rule injD \[OF inj_Rep_OrderedSet\])
apply (auto simp: smashed_orderedSet_charn Abs_OrderedSet_0_inverse)
done

The following lemma is crucial for the theory morpher:

lemma DEF_X_OrderedSet:
\( \text{DEF } (\text{X}) = (\exists \ c. \ \bot \notin \text{set } c \land \text{distinct } c \land \text{X} = \text{Abs\_OrderedSet\_0 } (\langle c, c \rangle)) \)
apply (auto simp: DEF_Abs_OrderedSet)
apply (simp add: DEF_X_OrderedSet_0)
apply (erule exE, rule exI, auto)
apply (rule injD \[OF inj_Rep_OrderedSet\])
apply (auto simp: smashed_orderedSet_charn Abs_OrderedSet_0_inverse)
done

This lemma is very convenient in simplifying unfolded definitions

lemma DEF_X_OrderedSet':
\( \land [X \neq \bot] \implies (\exists \ c. \ \bot \notin \text{set } c \land \text{distinct } c \land \text{Rep\_OrderedSet\_0 } X = (\langle c, c \rangle)) \land (\exists \ c. \ \bot \notin \text{set } c \land \text{distinct } c \land \text{X} = (\text{Abs\_OrderedSet\_0 } (\langle c, c \rangle))) \)
apply (fold DEF_def)

349
Appendix B. Isabelle Theories

apply (frule DEF_X_OrderedSet_THEN iffD1)
apply (drule DEF_X_OrderedSet_0_THEN iffD1)
apply (simp)
done

lemma OrderedSet_sem_cases_0:
assumes defC: \( \forall c\ d. \ X \neq \bot; X \neq \bot \)
and undefC: \( X = \bot \) \( \Longrightarrow \ P X \)
shows \( P X \)
apply (rule Abs_OrderedSet_0_cases_charn, erule undefC)
apply (rule defC, auto)
done

lemma OrderedSet_sem_cases:
assumes defC: \( \forall c\ d. \ (X \tau) \neq \bot; X \tau \neq \bot \)
and undefC: \( (X \tau) = \bot \) \( \Longrightarrow \ P X \tau \)
shows \( P X \tau \)
apply (rule_tac P2=P in subst[OF sym[OF cp_subst]], rule cpP)
apply (rule OrderedSet_sem_cases_0)
apply (rule defC)
prefer 9
apply (rule undefC)
done

lemma OrderedSet_sem_cases_ext:
assumes defC: \( \forall c\ d\ \tau. \ (X \tau) \neq \bot; X \tau \neq \bot \)
and cpP: \( P X = \tau \) \( \Longrightarrow \ Q X \tau \)
shows \( P X = \tau \)
apply (rule_tac X=X in OrderedSet_sem_cases)
apply (rule defC) prefer 9
apply (rule undefC)

350
apply(simp_all add: cpP cpQ)
done

These four lemmas are used by ocl_setup_op to reason about definedness:

**Lemma lift2_strict_is_isdef_fu_OrderedSet_Val**

**Assumptions**

\( f \_def \equiv \text{lift} \soft2(\text{strictify}(\lambda x. \text{strictify}(\lambda y. \text{Abs}\_\text{OrderedSet}\_0 \ g \ Rep\_\text{OrderedSet}\_0 \ x \ y))) \)

and

- \( \text{inv}_1 \equiv \forall a. b. [ \bot \notin \text{set} \ a; \text{distinct} \ a; \bot \notin \text{set} \ (g \ a \ b)] \implies \bot \notin \text{set} \ (g \ a \ b) \)
- \( \text{inv}_2 \equiv \forall a. b. [ \bot \notin \text{set} \ a; \bot \notin \text{set} \ b; \text{distinct} \ b \implies \bot \notin \text{set} \ (g \ a \ b)] \implies \text{distinct} \ (g \ a \ b) \)

**Shows**

\( \partial(f X Y) = (\partial X \land \partial Y) \)

case_a apply(rule ext)
case_a apply(simp add: f_def OclIsDefined_def OclAnd_def a_def ss_lifting)
case_a apply(rule_tac X=X x in OrderedSet_sem_cases_0)
case_a apply(rule_tac x in inv_g1)
case_a apply(drule_tac (Y x) in inv_g2)
case_a apply(simp_all add: neq_commute)
done

**Lemma lift2_strict_is_isdef_fu_OrderedSet_OrderedSet**

**Assumptions**

\( f \_def \equiv \text{lift} \soft2(\text{strictify}(\lambda x. \text{strictify}(\lambda y. \text{Abs}\_\text{OrderedSet}\_0 \ g \ Rep\_\text{OrderedSet}\_0 \ x \ y))) \)

and

- \( \text{inv}_1 \equiv \forall a. b. [ \bot \notin \text{set} \ a; \text{distinct} \ a; \bot \notin \text{set} \ (g \ a \ b)] \implies \bot \notin \text{set} \ (g \ a \ b) \)
- \( \text{inv}_2 \equiv \forall a. b. [ \bot \notin \text{set} \ a; \bot \notin \text{set} \ b; \text{distinct} \ b \implies \bot \notin \text{set} \ (g \ a \ b)] \implies \text{distinct} \ (g \ a \ b) \)

**Shows**

\( \partial(f X Y) = (\partial X \land \partial Y) \)

case_a apply(rule ext)
case_a apply(simp add: f_def OclIsDefined_def OclAnd_def a_def ss_lifting)
case_a apply(rule_tac X=X x in OrderedSet_sem_cases_0)
case_a apply(rule_tac x in inv_g1)
case_a apply(drule_tac (Y x) in inv_g2)
case_a apply(simp_all add: neq_commute)
done

**Lemma lift2_strictify_implies_LocalValid_defined_OrderedSet_Val**

**Assumptions**

\( f \_def \equiv \text{lift} \soft2(\text{strictify}(\lambda x. \text{strictify}(\lambda y. \text{Abs}\_\text{OrderedSet}\_0 \ g \ Rep\_\text{OrderedSet}\_0 \ x \ y))) \)

and

- \( \text{inv}_1 \equiv \forall a. b. [ \bot \notin \text{set} \ a; (\bot \bot' \bot \bot) \notin \text{set} \ (g \ a \ b)] \implies \bot \notin \text{set} \ (g \ a \ b) \)
- \( \text{inv}_2 \equiv \forall a. b. [ \bot \notin \text{set} \ a; (\bot \bot' \bot \bot) \notin \text{set} \ (g \ a \ b)] \implies \text{distinct} \ (g \ a \ b) \)

**Shows**

\( \text{(}\tau \equiv \partial(f X Y) = (\tau \equiv \partial X) \land (\tau \equiv \partial Y)) \)

case_a apply(insert f_def)
case_a apply(drule_tac X=X Y=Y in lift2_strict_is_isdef_fu_OrderedSet_Val)
case_a apply(assumption)+
case_a apply(simp_all add: OclAnd_def OclTrue_def o_def OclIsDefined_def lift0_def lift1_def lift2_def)
done
Appendix B. Isabelle Theories

**Lemma** lift2_strictify_implies_LocalValid_defined_OrderedSet_OrderedSet:

**Assumes**

1. \( f \equiv \text{lift2}(\text{strictify}(\lambda x. \text{strictify}(\lambda y. \text{Abs}_{\text{OrderedSet}_0}(\text{Rep}_{\text{OrderedSet}_0} x) \text{Rep}_{\text{OrderedSet}_0} y))) \)

2. \( \text{inv}_{g1}: ! a b. (\| \bot \notin \text{set} \ a; \text{distinct} \ a; \bot \notin \text{set} \ b; \text{distinct} \ b \) \implies \( \bot \notin \text{set} \ (g \ a \ b) \)

3. \( \text{inv}_{g2}: ! a b. (\| \bot \notin \text{set} \ a; \text{distinct} \ a; \bot \notin \text{set} \ b; \text{distinct} \ b \) \implies \text{distinct} \ (g \ a \ b) \)

**Shows**

\( (\tau \models \partial (f \ X \ Y)) = ((\tau \models \partial \ X) \land (\tau \models \partial \ Y)) \)

**Apply**

1. \( \text{insert} \ f \ _def \)

2. \( \text{drule_tac} \ X = X \ \text{and} \ Y = Y \ \text{in} \ \text{lift2_strict_is_isdef_fw_OrderedSet_OrderedSet} \)

3. \( \text{assumption} \)\+

4. \( \text{simp add:} \ \text{OclAnd_def OclTrue_def o_def OclIsDefined_def lift0_def lift1_def lift2_def OclLocalValid_def} \)

**Done**

**Building a canonic representation of orderedSets**

**The empty orderedSet**

**Constdefs**

\( \text{OclMtOrderedSet} :: (\forall a, b. \text{bot OrderedSet}_0) \ \text{VAL} \)

\( \text{OclMtOrderedSet} \equiv \text{lift0}(\text{Abs}_{\text{OrderedSet}_0}(\text{[]})) \)

**Syntax**

1. \( \text{OclMtOrderedSet}_\text{std} :: (\forall a, b. \text{bot}) \ \text{VAL} \Rightarrow \text{a Boolean} \) (OrderedSet\{\})

2. \( \text{OclMtOrderedSet}_\text{ascii} :: (\forall a, b. \text{bot}) \ \text{VAL} \Rightarrow \text{a Boolean} \) (mtOrderedSet)

3. \( \text{OclMtOrderedSet}_\text{math} :: (\forall a, b. \text{bot}) \ \text{VAL} \Rightarrow \text{a Boolean} \) (\{\})

**Parse Translation**

\( \\text{\text{}flat(map (emb trans const) [OclMtOrderedSet])} \)

**Print Translation**

\( \text{\text{}map emb_print [OclMtOrderedSet]} \)

**Lemma** OCL_is_isdef_OclMtOrderedSet [simp]:

\( \tau \models \partial (\text{\{\}}) \)

**By**\( \text{simp add: OclValid_def OclIsDefined_def OclTrue_def OclMtOrderedSet_def ss_lifting} \)

**Lemma** OCL_is_defopt_OclMtOrderedSet [simp]:

\( \tau \models \partial (\text{\{\}}) \)

**By**\( \text{simp add: valid_elim} \)

**And the 'cons' operation: including def**

\( \text{OclIncluding_def} \)

\( \text{OclIncluding} \equiv \text{lift2}(\text{strictify}(\lambda x. \text{strictify}(\lambda y. \text{Abs}_{\text{OrderedSet}_0}(\text{Rep}_{\text{OrderedSet}_0} S, [e])))) \)

352
B.4. Library

ocl_setup_op [OclIncluding]

**Lemma OCL_is_def_OclIncluding:**
\[ \partial(\text{OclIncluding} (X::(a,b::\text{bot})\text{OrderedSet}) (Y::(a,b::\text{bot})\text{VAL})) = (\partial X \land \partial Y) \]
by (rule lift2_strict_is_isdef_fw_OrderedSet_Val [OF OclIncluding_def], simp_all)

**Lemma OCL_is_defopt_OclIncluding:**
\[ (\tau \models \partial(\text{OclIncluding} (X::(a,b::\text{bot})\text{OrderedSet}) (Y::(a,b::\text{bot})\text{VAL}))) = (\tau \models \partial X) \land (\tau \not\models \partial Y) \]
by (rule lift2_strictify_implies_LocalValid_defined_OrderedSet_Val [OF OclIncluding_def], simp_all)

The syntax translation mkOrderedSet builds now our orderedSets syntax

\[ \text{@OclFinOrderedSet :: args} \Rightarrow (a,b\text{OrderedSet}_0)\text{VAL} \]

**Translators**

- \[ \text{mkOrderedSet}[x, xs] \Rightarrow \text{OclIncluding} (\text{mkOrderedSet}[xs]) x \]
- \[ \text{mkOrderedSet}[x] \Rightarrow \text{OclIncluding} \text{OclMtOrderedSet} x \]

**Lemma test : mkOrderedSet[1, 2, 3, 4, 5] = ?X oops**

**Relation of mtOrderedSet and including**

These two theorems should actually suffice to decide inequality of orderedSets

**Lemma including_notstrongeq_mtOrderedSet[simp]:**
\[ \neg ((X::(a,b::\text{bot})\text{OrderedSet})\rightarrow\text{including} (a::(a,b)\text{VAL}) \triangleq []) = \top \]
apply (rule_tac X=X in OrderedSet_sem_cases_ext)
apply (simp_all add: cp_strongEq)
apply (simp_all add: OclIncluding_def OclMtOrderedSet_def OclStrongEq_def OclTrue_def OclNot_def localValid2sem ss_lifting Abs_OrderedSet_inject_charn neq_commute)
done

**Lemma including_notstricteq_mtOrderedSet[simp]:**
\[ \begin{array}{c} \tau \models \partial (a::(a,b)\text{VAL}); \tau \not\models \partial (X::(a,b::\text{bot})\text{OrderedSet}) \end{array} \] \[
\Rightarrow \tau \models (X\rightarrow\text{including} a) \not\models () \]
apply (rule_tac X=X in OrderedSet_sem_cases)
apply (simp_all add: cp_strongEq cp_strictEq localValidDefined2sem)
apply (simp_all add: OclIncluding_def OclMtOrderedSet_def OclStrictEq_def OclTrue_def OclNot_def localValid2sem ss_lifting Abs_OrderedSet_inject_charn neq_commute)
done
Appendix B. Isabelle Theories

A second orderedSet constructor on the OCL level
defs

\[ OclCollectionRange \equiv \text{lift2} (\text{strictify} (\lambda x::\text{Integer}_0. \text{strictify} (\lambda y. \text{Abs}\_\text{OrderedSet}_0 \text{map} (\lambda z::\text{nat}. (\text{int } z + \lceil x \rceil)) [0..<(\text{nat } (\lceil y \rceil - \lceil x \rceil + 1))])))
\]

ocl_setup_op [OclCollectionRange]

lemma \(OCL\_is\_defopt\_OclCollectionRange\) [simp]:
\[ \tau \vdash \emptyset ((\text{OclCollectionRange } a::(a \text{ Integer} ) ) b::(a, \text{Integer}_0)\text{OrderedSet}) = ((\tau \vdash \emptyset ) \land (\tau \vdash \emptyset ))\]

oops

lemma collectionRange_mtOrderedSet_conv:
\[ [ \tau \vdash (b::a \text{ Integer}) < (a::a \text{ Integer}) ] \Longrightarrow \text{OclCollectionRange } a\ b\ \tau = (\langle\rangle::(a,\text{Integer}_0)\text{OrderedSet}\rangle\ \tau\]

apply (case_tac \(\tau\ \emptyset\ \not\vdash a\), ocl_hypsubst, simp)
apply (case_tac \(\tau\ \emptyset\ \not\vdash b\), ocl_hypsubst, simp)
apply (simp add: localValidDefined2sem DEF_def both, clarify)
apply (simp_all add: DEF_def OclCollectionRange_def OclMtOrderedSet_def OclIncluding_def localValid2sem ss_lifting)
done

lemma collectionRange_singleton_UC [simp]:
\(\text{OclCollectionRange } a\ a\ \tau\ =\ \langle\rangle::(a,\text{Integer}_0)\text{OrderedSet}\rightarrow\text{including} (a::(a,\text{Integer}_0)\text{VAL})\]

apply (rule ext, case_tac \(\tau\ \not\vdash a\), ocl_hypsubst, simp)
apply (simp add: OclCollectionRange_def OclMtOrderedSet_def OclIncluding_def localValidDefined2sem ss_lifting)
done

lemma collectionRange_expand_including:
\[ [ \tau \vdash (a::a \text{ Integer}) \leq b ] \Longrightarrow (\text{OclCollectionRange } a\ b::(a,\text{Integer}_0)\text{OrderedSet})\ \tau = (\text{OclCollectionRange } a\ (b - 1))\rightarrow\text{including} b\ \tau\]

oops

The conversion operators

OrderedSet to sequence
defs

\[ OclAsSequence \equiv \text{lift1} (\text{strictify} (\lambda X.\).

354
Abs_Sequence_0 (Rep_OrderedSet_0 X))

ocl_setup_op [OclAsSequence]

OrderedSet to bag
defs

OclAsBag_def :
  OclAsBag ≡ lift1 (strictify (λX. Abs_Bag_0 \multiset_of\ Rep_OrderedSet_0 X))

ocl_setup_op [OclAsBag]

OrderedSet to orderedSet
defs

OclAsOrderedSet_def :
  OclAsOrderedSet ≡ id

lemma asOrderedSet_identity[simp]:
  OclAsOrderedSet (S::('a,'b::bot)OrderedSet) = S
  by(simp add: OclAsOrderedSet_def)

OrderedSet to set
defs

OclAsSet_def :
  OclAsSet ≡ lift1 (strictify (λX. Abs_Set_0 \set\ Rep_OrderedSet_0 X))

ocl_setup_op [OclAsSet]

end

B.4.13. OCL Set

theory OCL_Set
imports...

begin

Is OCL Set Faithful to the OCL Standard?

Unlike the basic library theories, for this first and foundational "Collection"-type theory the question of faithfulness is not merely a syntactic matter (like fusing various
Appendix B. Isabelle Theories

semantic interpretation functions into one). In fact, it is not even trivial to answer. This is mainly for two reasons:

1. We generalize the concept of Set as required by OCL 2.0 to infinite sets.

2. We interpret the standard with respect to the question, if OclUndefind can be included in a set or not: we deny this question in this version of a set theory.

The first decision has the advantage that general Sets can be used to represent the syntactic category of "types" in the standard as "characteristic sets" in HOL and enables the possibility to reason over them. From the pragmatics point of view, this also allows for quantifications of sets of "values" in the sense of OCL 2.0. For example, it is possible in HOL-OCL to express the commutativity law on Integers inside OCL.

The second decision is due to the fact that the standard avoids a clear statement to the nature of undefinedness in OCL. This led to two forms of interpretations by users and tool-developers: is OclUndefined just "null", so the null-reference, or is it a semantic construct denoting non-termination or exceptional behaviour? In the former sense, it is possible to include OclUndefined in a set, in the latter not. Admitting undefinedness in sets also has consequences in the inclusion operation and the elementhood test (->including and ->includes). The operational behaviour of Java-implementations reflects the latter view, which we chose therefore as default. Semantically, this leads to a quotient construction (called "smashing" in the literature) which identifies \{x, ⊥\} with ⊥.

With respect to the question of faithfulness of our OCL semantics we have the following answers:

1. the required properties of OCL 2.0 should hold in our setting for finite sets, and

2. the subtle issue of ⊥s in sets is not mentioned at all in them.

Therefore, we show compliance in this sense by providing the individual proof for these "required properties". However, we will only use them in rare cases in definitions and provide a number of other properties for them that are more suited for automated reasoning.

Foundational Properties of the smash-construction

Global OCL type constructor for OCL Set types:

types ('τ,'α) Set = ('τ,'α Set_0) VAL

Properties of the Congruence inside the Datatype Adaptation

lemma Abs_Set_inject_absurd11 [simp]:
[ ⊥ ≠ x ] \implies ((Abs_Set_0 down) = (Abs_Set_0 ,x)) = False
by(subst Abs_Set_0_inject,
   simp_all add: OCL_Set_type.Set_0_def smash_def)

lemma Abs_Set_0_inject_absurd12 [simp]:

356
B.4. Library

\[
\begin{align*}
\text{lemma } & \text{Abs_Set_0_inject_absurd21 \ [simp]:} \\
& \quad [\bot \notin x] \implies (\bot = (\text{Abs_Set_0} (x) = \bot) = False \\
& \quad \text{by (simp add: UU_Set_def)}
\end{align*}
\]

\[
\begin{align*}
\text{lemma } & \text{Abs_Set_0_inject_absurd22 \ [simp]:} \\
& \quad [\bot \notin x] \implies ((\text{Abs_Set_0} (x) = \bot) = False \\
& \quad \text{by (simp add: UU_Set_def)}
\end{align*}
\]

\[
\begin{align*}
\text{lemma } & \text{Abs_Set_0_cases_charn:} \\
& \quad \text{assumes bottomCase: } [\bot = \bot] \implies P \\
& \quad \text{assumes listCase : } \forall y. ([x = \text{Abs_Set_0} (y); \bot \notin y] \implies P) \\
& \quad \text{shows} \\
& \quad \text{apply (rule_tac x=x in Abs_Set_0_cases)} \\
& \quad \text{apply (case_tac y = \bot)} \\
& \quad \text{apply (rule bottomCase, simp add: UU_Set_def)} \\
& \quad \text{apply (frule not_down_exists_lift2 \ [THEN iffD1])} \\
& \quad \text{apply (rule_tac y="\bot" in listCase)} \\
& \quad \text{apply (auto simp: OCL_Set_type.Set_0_def smash_def)} \\
& \quad \text{done}
\end{align*}
\]

\[
\begin{align*}
\text{lemma } & \text{Abs_Set_0_induct_charn:} \\
& \quad \text{assumes bottomCase : } P \bot \\
& \quad \text{assumes stepCase : } \forall y. ([x \notin y] \implies P (\text{Abs_Set_0} (y))) \\
& \quad \text{shows} \\
& \quad \text{apply (rule_tac x=x in Abs_Set_0_induct)} \\
& \quad \text{apply (rule_tac x=Abs_Set_0 y in Abs_Set_0_cases_charn)} \\
& \quad \text{apply (auto intro!: bottomCase elim: stepCase)} \\
& \quad \text{done}
\end{align*}
\]
Appendix B. Isabelle Theories

the rest

lemma inj_on_Abs_Set: inj_on Abs_Set_0 OCL_Set_type.Set_0
by (rule inj_on_inverseI, rule Set_0.Abs_Set_0.inverse, assumption)

lemma inj_Rep_Set: inj Rep_Set_0
by (rule inj_on_inverseI, rule Set_0.Rep_Set_0.inverse)

lemma smashed_set_charn: (⊥ ∉ X) = (X ∈ OCL_Set_type.Set_0)
by (unfold smash_def OCL_Set_type.Set_0_def UU_Set_def, auto)

lemma UNIV_in_smashed_set[simp]: ⊥ : OCL_Set_type.Set_0
by (unfold smash_def OCL_Set_type.Set_0_def UU_Set_def, auto)

lemma down_in_smashed_set[simp]: down : OCL_Set_type.Set_0
by (unfold smash_def OCL_Set_type.Set_0_def UU_Set_def, auto)

lemma mt_in_smashed_set: ⊥ ∈ OCL_Set_type.Set_0
by (unfold smash_def OCL_Set_type.Set_0_def, auto)

Universal sets are never in OCL_Set_type.Set_0, since they must contain bottom elements (HOL's type discipline forces UNIV to be of type 'α::bot set) and therefore their smashed value is ⊥.

lemma UNIV_notin_smashed_set: ¬ UNIV ∉ OCL_Set_type.Set_0
by (simp add: smash_def OCL_Set_type.Set_0_def)

In contrast, the set consisting of all 1-lifted elements is in OCL_Set_type.Set_0.
This justifies the construction of characteristic sets for Boolean, Integer, Real, and String.

lemma liftUNIV_in_smashed_set: lift ' UNIV ∈ OCL_Set_type.Set_0
by (auto simp: smash_def OCL_Set_type.Set_0_def)

lemma DEF Abs_Set: ⊥ ∉ X ⇒ DEF (Abs_Set_0 (X))
by (simp add: DEF_def)

lemma DEF Rep_Set: DEF X ⇒ DEF (Rep_Set_0 X)
apply (unfold DEF_def UU_Set_def,auto)
apply (drule_tac f = Abs_Set_0 in arg_cong)
apply (simp add: Rep_Set_0_inverse)
done

lemma not_DEF Rep_Set: ¬ DEF X ⇒ ¬ DEF (Rep_Set_0 X)
apply (unfold DEF_def UU_Set_def,auto)
apply (simp add: Abs_Set_0_inverse)
done

358
lemmas \begin{align*}
\text{exists\_lift\_Set} & : \text{DEF } X \implies \exists c. \text{Rep\_Set\_0 } X = (c) \\
\text{exists\_lift\_Set2} & : \text{DEF } X \implies \exists c. \bot \notin c \land \text{Rep\_Set\_0 } X = (c) \\
\text{Rep\_Set\_cases} & : \text{Rep\_Set\_0 } X = \bot \lor (\exists c. \bot \notin c \land \text{Rep\_Set\_0 } X = (c)) \\
\text{DEF\_X\_Set\_0} & : \text{DEF } X = (\exists c. \bot \notin c \land \text{Rep\_Set\_0 } X = (c)) \\
\text{RepAbs\_UNIV\_simp} & : \text{Rep\_Set\_0 } (\text{Abs\_Set\_0 } (\text{lift } ' \text{UNIV})) = (\text{lift } ' \text{UNIV})
\end{align*}

The following lemma is crucial for the theory morpher:

\text{lemma } \text{DEF\_X\_Set} : \text{DEF } X = (\exists c. \bot \notin c \land \text{X = Abs\_Set\_0 } (c)) 

The following lemma is crucial for the theory morpher:

\text{lemma } \text{Rep\_Set\_cases: Rep\_Set\_0 } X = \bot \lor (\exists c. \bot \notin c \land \text{Rep\_Set\_0 } X = (c)) 

The following lemma is crucial for the theory morpher:

\text{lemma } \text{DEF\_X\_Set\_0} : \text{DEF } X = (\exists c. \bot \notin c \land \text{Rep\_Set\_0 } X = (c)) 

The following lemma is crucial for the theory morpher:

\text{lemma } \text{RepAbs\_UNIV\_simp: Rep\_Set\_0 } (\text{Abs\_Set\_0 } (\text{lift } ' \text{UNIV})) = (\text{lift } ' \text{UNIV})
Appendix B. Isabelle Theories

lemma Abs_Set_rangelift_DEF[simp]:
  DEF(Abs_Set_0, range lift)
by(auto simp: DEF_\\_X_Set_0)

lemma DEF_\\_X_Set':
[ X ≠ ⊥ ] ⇒
  (∃ c. (⊥ ≤ c) ∧ (Rep_Set_0 X = \{c,\})) ∧
  (∃ c. (⊥ ≤ c) ∧ (X = (Abs_Set_0 \{c,\})))
apply(fold DEF_def)
apply(frule DEF_\\_X_Set[THEN iffD1])
apply(drule DEF_\\_X_Set_0[THEN iffD1])
apply(simp)
done

lemma Set_sem_cases_0:
assumes defC: \(\exists\ c\ d\ .\ \[X ≠ ⊥; \bot ≠ X;\]∀ c d. \[X ≠ ⊥; \bot ≠ X;\]
\[X = Abs_Set_0 \{c\} \]⇒ P X
and undefC: \[X = ⊥ \]⇒ P X
shows P X
apply(rule Abs_Set_0_cases_charn, erule undefC)
apply(rule defC, auto)
done

lemma Set_sem_cases:
assumes defC: \(\exists\ c\ d\ .\ \[X(τ) ≠ ⊥; \bot ≠ (X τ);\]∀ c d. \[X(τ) ≠ ⊥; \bot ≠ (X τ);\]
\[X (τ) = Abs_Set_0 \{d\} \]⇒ P X τ
and undefC: \[X (τ) = ⊥ \]⇒ P X τ
and cpP: cp P
shows P X τ
apply(rule_tac P2=P in subst[of sym[of cp_subst]], rule cpP)
apply(rule Set_sem_cases_0)
apply(rule_tac P1=P in subst[of cp_subst], rule cpP)
apply(rule defC) prefer 7
apply(rule_tac P1=P in subst[of cp_subst], rule cpP)
apply(rule undefC, simp_all)
done

lemma Set_sem_cases_ext:
assumes defC: \(\exists\ c\ d\ .\ \[X(τ) ≠ ⊥; \bot ≠ (X τ);\]∀ c d. \(X (τ) = Abs_Set_0 \{d\} \)⇒ P X τ = Q X τ
and undefC: \[ (X \tau) = \bot \] \implies P X \tau = Q X \tau
and cpP: cp P
and cpQ: cp Q
shows P X = Q X
apply(rule ext)
apply(\text{rule_tac X=X in Set_sem_cases})
apply(\text{rule_tac X=Q in Set_sem_cases})
apply(\text{rule_tac X=Q in Set_sem_cases})
apply(\text{rule_def C})
prefer 7
apply(\text{rule_def C})
apply(simp_all add: cpP cpQ)
done

These four lemmas are used by ocl_setup_op to reason about definedness:

\textbf{lemma lift2_strict_is_isdef_fw_Set_Val:}
\textit{assumes f_def:} \( f \equiv \text{lift2}(\text{strictify}(\lambda x. \text{strictify}(\lambda y. \text{Abs_Set_0}(\text{Rep_Set_0} x \text{Rep_Set_0} y)))) \)
\textit{and inv_g:} \( !a b.( \| \bot \notin a; \bot :: \alpha :: \text{bot} \| \neq b \| \Rightarrow \bot \notin (g a b) ) \)
\textit{shows} \( \partial(f X Y) = (\partial X \wedge \partial Y) \)
apply(simp add: f_def OclIsDefined_def OclAnd_def)
apply(rule ext)
apply(case_tac DEF (X \tau))
apply(auto simp add: DEF_def)
done

\textbf{lemma lift2_strictify_implies_LocalValid_defined_Set_Val:}
\textit{assumes f_def:} \( f \equiv \text{lift2}(\text{strictify}(\lambda x. \text{strictify}(\lambda y. \text{Abs_Set_0}(\text{Rep_Set_0} x \text{Rep_Set_0} y)))) \)
\textit{and inv_g:} \( !a b.( \| \bot \notin a; \bot :: \alpha :: \text{bot} \| \neq b \| \Rightarrow \bot \notin (g a b) ) \)
\textit{shows} \( \partial(f X Y) = (\partial X \wedge \partial Y) \)
apply(simp add: f_def OclIsDefined_def OclAnd_def)
apply(rule ext)
apply(case_tac DEF (X \tau))
apply(auto simp add: DEF_def)
done
Appendix B. Isabelle Theories

shows \((\tau \models \partial (f \, X \, Y)) = ((\tau \models \partial X) \land (\tau \models \partial Y))\)  
apply(insert f_def)  
apply(drule_tac X=X and Y=Y in lift2_strict_is_isdef_fw_Set_Val)  
apply(rule inv_g, assumption+)  
apply(simp add: OclAnd_def OclTrue_def o_def  
OclIsDefined_def lift0_def lift1_def lift2_def  
OclLocalValid_def)  
done

lemma lift2_strictify_implies_LocalValid_defined_Set_Set:  
assumes f_def: \(f \equiv \text{lift2} (\text{strictify}(\lambda x. \text{strictify}(\lambda y. \text{Abs_Set_0} \, \text{list_of_set} \, \text{Rep_Set_0} \, X \, y)))\)  
and inv_g: \(!a \, b. \left[ \bot \notin a; \bot \notin b \right] \implies \bot \notin (g \, a \, b)\)  
shows \((\tau \models \partial (f \, X \, Y)) = ((\tau \models \partial X) \land (\tau \models \partial Y))\)  
apply(insert f_def)  
apply(drule_tac X=X and Y=Y in lift2_strict_is_isdef-fw_Set_Set)  
apply(rule inv_g, assumption+)  
apply(simp add: OclAnd_def OclTrue_def o_def  
OclIsDefined_def lift0_def lift1_def lift2_def  
OclLocalValid_def)  
done

The conversion operators

From sets to sequences

Because there is no ordering on the elements of the sets the \textit{AsSequence} operator chooses some sequence which has exactly the same elements as the set.

Because this is just some sequence the proof about statements containing the \textit{AsSequence} operator, have to show the same properties as one has to show for the general Hilbert Choice operator.

defs

\[
\text{OclAsSequence} \text{ def} : \\
\text{OclAsSequence} \equiv \text{lift1} (\text{strictify}(\lambda X. \\
\text{Abs_Sequence}_0 \, \text{list_of_set} \, \text{Rep_Set_0} \, X))
\]
ocl_setup_op [OclAsSequence]

From sets to bags

defs

\[
\text{OclAsBag} \text{ def} : \\
\text{OclAsBag} \equiv \text{lift1} (\text{strictify}(\lambda X. \\
\text{Abs_Bag}_0 \, \text{multiset_of_set} \, \text{Rep_Set_0} \, X))
\]
ocl_setup_op [OclAsBag]

From sets to ordered sets

defs
B.4. Library

\textit{OclAsOrderedSet\_def:}
\[\textit{OclAsOrderedSet} \equiv \texttt{lift1 (strictify (\lambda X.} \texttt{Abs\_OrderedSet\_0 (\textit{list\_of\_set} \texttt{\{Rep\_Set\_0 \ X\})})}\]
\[\texttt{ocl\_setup\_op}\ [\textit{OclAsOrderedSet}]\]

\textbf{lemma} \textit{bag2seq2oset\_eq\_bag2set2oset:}
\[\texttt{\textit{asOrderedSet}()} \texttt{((\texttt{asSequence}()) (B::\texttt{\{\`\tau,\`\alpha::bot\}Bag}))::\texttt{\{\`\tau,\`\alpha\}Sequence}) =}
\[\texttt{((\texttt{asOrderedSet}()) \texttt{(\texttt{asSet}()) B::\texttt{\{\`\tau,\`\alpha\}Set})::\texttt{\{\`\tau,\`\alpha\}OrderedSet})}\]
\[\texttt{apply (rule ext)}\]
\[\texttt{apply (case\_tac x :\texttt{\notin B, ocl\_hypsubst, simp)}}\]
\[\texttt{apply (simp add: localValidDefined2sem \texttt{DEF\_def\_both, clarify)}}\]
\[\texttt{apply (rule \texttt{DEF\_X\_Bag}, clarify)}\]
\[\texttt{apply (simp add: OCL\_Bag.OclAsSequence\_def OCL\_Bag.OclAsSet\_def OCL\_Sequence.OclAsOrderedSet\_def OCL\_Set.OclAsOrderedSet\_def ss\_lifting)}\]
\[\texttt{apply (subst Abs\_OrderedSet\_0\_inject\_charn)}\]
\[\texttt{apply (simp\_all add: rev\_remdups rev\_remstutter [symmetric])}\]
\texttt{done}\]

\textbf{lemma} \textit{oset2seq2bag\_eq\_oset2set2bag:}
\[\texttt{\textit{asBag}()} \texttt{((\texttt{asSequence}()) (X::\texttt{\{\`\tau,\`\alpha::bot\}OrderedSet})::\texttt{\{\`\tau,\`\alpha\}Sequence}) =}
\[\texttt{((\texttt{asBag}()) \texttt{(\texttt{asSet}()) X::\texttt{\{\`\tau,\`\alpha\}Set})::\texttt{\{\`\tau,\`\alpha\}Bag})}\]
\[\texttt{apply (rule OrderedSet\_sem\_cases\_ext, simp\_all)}\]
\[\texttt{apply (simp\_all add: OCL\_OrderedSet.OclAsSequence\_def OCL\_OrderedSet.OclAsSet\_def OCL\_Sequence.OclAsBag\_def OCL\_Set.OclAsBag\_def ss\_lifting)}\]
\texttt{done}\]

\textbf{From sets to sets}
\textbf{defs}
\[\textit{OclAsSet\_def:}\]
\[\textit{OclAsSet} \equiv \texttt{id}\]

\textbf{lemma} \textit{set2set\_id [simp]}:
\[\texttt{(\textit{OclAsSet} (B::\texttt{\{\`\tau,\`\alpha::bot\}Set}))::\texttt{\{\`\tau,\`\alpha\}Set}) = B}\]
\texttt{by (simp add: OclAsSet\_def)}\]

\textbf{The Definition of the Set Operators}

\textbf{The Powerset Operator}
\textbf{constdefs}
\[\textit{Set} \equiv \texttt{\{\`st, (\`\alpha::bot\) Set\_0 \ VAL \Rightarrow \texttt{\{\`st, \texttt{\`\alpha Set\_0 Set\_0} \ VAL}}\]
\[\texttt{Set} \equiv \texttt{lift1 (strictify (\lambda X.} \texttt{Abs\_Set\_0 (Abs\_Set\_0 o lift)} \texttt{\{x. x \subseteq \texttt{Rep\_Set\_0 X\})})})\]
\texttt{ocl\_setup\_op [Set]}\]
Appendix B. Isabelle Theories

lemma Set_defined: 
  \( \tau \models \partial X \Rightarrow \tau \models \partial (Set X) \) 
apply(simp add: Set_def localValidDefined2sem)
apply (auto simp: strictify_def 
  o_def lift0_def lift1_def lift2_def image_def 
  OclTrue_def OclFalse_def OclUndefined_def)
apply (rule DEF_Abs_Set, simp)
apply (rule allI, rule impI)
apply (rule neq_commute [THEN iffD1])
apply (simp only: DEF_def [symmetric])
apply (rule DEF_Abs_Set, simp)
apply (rule exists_lift_Set2)
done

lemma Set_definedR: 
  \((\tau \models \partial (Set X)) = (\tau \models \partial X)\) 
apply (auto simp: Set_defined)
oops

The Empty Set

constdefs 
  OclMtSet :: ('\tau, '\alpha::bot Set_0) VAL
  OclMtSet \equiv lift0 (Abs_Set_0 (\{\}))

syntax 
  _OclMtSet_std :: ('\tau, '\alpha::bot) VAL \Rightarrow '\tau Boolean ('{})
syntax 
  _OclMtSet_ascii :: ('\tau, '\alpha::bot) VAL \Rightarrow '\tau Boolean (mtSet)
syntax (xsymbols) 
  _OclMtSet_math :: ('\tau, '\alpha::bot) VAL \Rightarrow '\tau Boolean (\emptyset)

Foundational Properties of MtSet

lemma Abs_Set_Mt_DEF[simp]: 
  DEF(Abs_Set_0 {\{\}})
by(simp add: DEF_X_Set_0)

lemma mtSet_Defined [simp]:
  \( \partial \emptyset = T \)
by(simp add: localValidDefined2sem Abs_Set_Mt DEF OclMtSet_def 
  OclIsDefined_def lift1_def lift0_def OclTrue_def)

Foundational Properties of includes and excludes
defs 
  OclIncludes_def :: OclIncludes \equiv lift2 (strictify (\lambda X. strictify (\lambda x.

364
B.4. Library

\( x \in \text{Rep}\_Set\_\emptyset \ X \))

\( \text{OclExcludes}\_\text{def} : \text{OclExcludes} \equiv \text{lift2} (\text{strictify} (\lambda X. \text{strictify} (\lambda x. \nonumber \\
( x \notin \text{Rep}\_Set\_\emptyset \ X \))) )

\text{ocl}\_\text{setup}\_\text{op} \{ \text{OclIncludes, OclExcludes} \}

\text{lemma includes\_charn1} : (\tau \models x \in S) \implies (x \tau \in \text{Rep}\_\text{Set}\_\emptyset (S \tau))
\text{apply (frule isDefined\_if\_valid,}
\text{drule OCL\_is\_deopt\_OclIncludes\{THEN iffD1\})}
\text{apply (simp add: OclLocalValid\_def OclIsDefined\_def strictify\_def}
\text{ OclNot\_def OclIncludes\_def}
\text{ o\_def lift0\_def lift1\_def lift2\_def}
\text{ OclTrue\_def OclFalse\_def OclUndefined\_def DEF\_def)}
\text{done}

\text{lemma includes\_charn2} : \{ \text{DEF\{(S \tau): x \tau \in \text{Rep}\_\text{Set}\_\emptyset (S \tau) \}} \} \implies \tau \models x \in S
\text{apply (frule DEF\_X\_Set \{THEN iffD1\}, erule exE)}
\text{apply (simp add: OclLocalValid\_def OclIsDefined\_def strictify\_def}
\text{ OclNot\_def OclIncludes\_def}
\text{ o\_def lift0\_def lift1\_def lift2\_def}
\text{ OclTrue\_def OclFalse\_def OclUndefined\_def DEF\_def)}
\text{apply auto}
\text{done}

\text{Higher Properties of includes and excludes}
\text{lemma excludes\_not\_includes\{simp\}:}
\text{x \notin X = \neg (x:(\tau, \alpha) VAL) \in (X:(\tau, \alpha):bot) Set}
\text{apply (rule ext)}
\text{apply (simp add: OclExcludes\_def OclIncludes\_def)}
\text{apply (simp only: OclNot\_def)}
\text{apply (simp only: OclIncludes\_def OclExcludes\_def lift1\_def lift2\_def}
\text{ strictify\_def o\_def not\_def)}
\text{apply (simp\_all \{no\_asm\_use\} add: UU\_fun\_def DEF\_def strictify\_def}
\text{ split add: split\_if up.split)}
\text{done}

\text{lemma mtSet\_Includes\_charn :}
\text{\tau \models \emptyset \models}
\text{\tau \models (x:(\tau, \alpha) VAL) \notin (\emptyset:(\tau, \alpha):bot) Set}
\text{apply (simp add: localValidDefined\{2sem\})}
\text{apply (simp only: excludes\_not\_includes OclMtSet\_def OclLocalValid\_def}
\text{ OclIncludes\_def lift0\_def lift1\_def lift2\_def)}

365
Appendix B. Isabelle Theories

\[ \text{apply}(\text{simp add: Abs_Set_Mt_DEF [simplified DEF_def] strictify_def}) \]
\[ \text{done} \]

\textbf{lemma X in Set X:}
\[ \tau \models S :: (\tau', \alpha :: \text{bot}) \text{Set} \implies \tau \models S \in (\text{Set } S) \]
\textit{oops}

\textbf{lemma mtSet_in_Set:}
\[ \tau \models S :: (\tau', \alpha :: \text{bot}) \text{Set} \implies \tau \models (\emptyset :: (\tau', \alpha :: \text{bot}) \text{Set}) \in (\text{Set } S) \]
\textit{oops}

\section*{Consequences of Smashedness of Sets}

This is the characteristic property of smashed sets: if $X \rightarrow \text{includes}(x)$ is valid, we know that $x$ must be defined.

\textbf{lemma smashed_sets_charn:}
\[ \tau \models (x :: (\tau', \alpha :: \text{bot}) \text{VAL}) \in (X :: (\tau', \alpha :: \text{bot}) \text{Set}) \implies \tau \models \partial x \]
\[ \text{apply}(\text{simp add: OclIncludes_def lift2_def lift1_def lift0_def}
\[ \text{strictify_def OclTrue_def OclIsDefined_def o_def OclLocalValid_def}) \]
\[ \text{apply}(\text{case_tac } S = \bot, \text{auto}) \]
\[ \text{apply}(\text{simp add: strictify_def DEF_def}) \]
\[ \text{apply}(\text{case_tac } x = \bot, \text{auto}) \]
\[ \text{done} \]

\section*{Fundamental Theorem on (infinite) Sets}

\textbf{lemma Set_Ext0:}
\[ (\forall x :: (\tau', \alpha :: \text{bot}) \text{VAL}). \tau \models \partial x \implies x \in (S :: (\tau', \alpha :: \text{bot}) \text{Set}) \tau = (x \in T)\tau \]
\[ \text{apply}(\text{simp add: OclIncludes_def lift2_def lift1_def lift0_def}
\[ \text{strictify_def OclTrue_def OclIsDefined_def o_def OclLocalValid_def}) \]
\[ \text{apply}(\text{case_tac } S = \bot, \text{auto}) \]
\[ \text{apply}(\text{case_tac } T = \bot, \text{auto}) \]
\[ \text{apply}(\text{insert exists_DEF, auto}) \]
\[ \text{apply}(\text{case_tac } T = \bot, \text{auto}) \]
\[ \text{apply}(\text{simp add: DEF_def[symmetric]}) \]
\[ \text{apply}(\text{frule DEF_Rep_Set}) \]
\[ \text{apply}(\text{frule DEF_Rep_Set}) \text{ back} \]
\[ \text{apply}(\text{simp add: DEF_X_Set}) \]
\[ \text{apply auto} \]
\[ \text{apply}(\text{rule_tac f = } \lambda c. \text{Abs_Set_0 c} \in \text{arg_cong}) \]
\[ \text{apply}(\text{rule set_ext}) \]
\[ \text{apply}(\text{erule_tac x = } \lambda r. xa \in \text{allE}) \]
\[ \text{apply}(\text{case_tac } x = \bot, \text{simp_all add: DEF_def[symmetric]}) \]

366
apply (auto simp: DEF_def)
done

lemma Set_Ext :
assumes H :\(\forall x::(\langle \tau, \alpha \rangle \text{VAL}) \cdot \tau \models \varnothing \ x \Rightarrow (x \in (S::(\langle \tau, \alpha::\text{bot} \rangle\text{Set}))\tau = (x \in T)\tau
shows \(S \tau = T \tau
apply (rule Set_Ext0) back
by (auto intro: H)

Semantic Definitions of the Standard Set Operations
defs

\textit{OclSize}_def \quad :\langle \text{OclSize::(\langle \tau, \alpha::\text{bot} \rangle \text{Set}) VAL} = \tau \text{ Integer} \rangle \equiv \\
(\text{lift1 (strictify (} \lambda X. \\
\text{if infinite } \langle \text{Rep\_Set\_0} X \rangle \text{ then } \bot \\
\text{ else } (\langle \text{int (card(} \langle \text{Rep\_Set\_0} X \rangle)\rangle)\rangle)))

\textit{OclCount}_def \quad :\langle \text{OclCount} \equiv \text{lift2 (strictify (} \lambda X. \\
\text{strictify (} \lambda x. \\
\langle \text{if } x \in \langle \text{Rep\_Set\_0} X \rangle \text{ then int 1} \\
\text{ else } \text{int 0} \rangle) \rangle)))

\textit{OclIncludesAll}_def : \text{OclIncludesAll} \equiv \text{lift2 (strictify (} \lambda X. \\
\text{strictify (} \lambda Y. \\
\langle \text{\langle \text{Rep\_Set\_0} Y \rangle \subseteq \langle \text{Rep\_Set\_0} X \rangle} \rangle)))

\textit{OclExcludesAll}_def : \text{OclExcludesAll} \equiv \text{lift2 (strictify (} \lambda X. \\
\text{strictify (} \lambda Y. \\
\langle \text{\langle \text{Rep\_Set\_0} X \rangle \cap \langle \text{Rep\_Set\_0} Y \rangle} = \{\} \rangle)))

\textit{OclIsEmpty}_def : \text{OclIsEmpty} \equiv \text{lift1 (strictify (} \lambda X. \\
\langle \{\} = \langle \text{Rep\_Set\_0} X \rangle \rangle))

\textit{OclNotEmpty}_def : \text{OclNotEmpty} \equiv \text{lift1 (strictify (} \lambda X. \\
\langle \{\} \neq \langle \text{Rep\_Set\_0} X \rangle \rangle))

defs

\textit{OclSum}_def \quad : \text{OclSum (self::(\langle \tau, \alpha::\text{bot} \rangle \text{Set}) VAL} \equiv \bot

\textit{ocl\_setup\_op \quad [OclSize, OclCount, OclIncludesAll, OclExcludesAll, OclIsEmpty, OclNotEmpty]}

\textbf{Warning:} The \textit{sum} operation is wrongly defined in the standard: idempotence

\textit{sum wrongly defined in the standard: idempotence}

367
Appendix B. Isabelle Theories

```isabelle
defs
OclFlatten_def: (OclFlatten :: ('τ,'α::bot Set_0 Set_0) VAL ⇒ ('τ,'α Set_0) VAL) ≡ 
  lift1 (strictify (λX. Abs_Set_0 (⋃ (Lifting.drop o Rep_Set_0)) "Rep_Set_0 X"))

ocl_setup_op [OclFlatten]

defs

OclIncluding_def:OclIncluding ≡ lift2 (strictify (λX. strictify (λY. Abs_Set_0 (insert Y "Rep_Set_0 X")))

OclExcluding_def:OclExcluding ≡ lift2 (strictify (λX. strictify (λY. Abs_Set_0 ("Rep_Set_0 X" − {Y})))

OclComplement_def:OclComplement ≡ lift1 (strictify (λX. Abs_Set_0 ("Rep_Set_0 X" − {⊥})))

OclUnion_def : OclUnion ≡ lift2 (strictify (λX. strictify (λY. Abs_Set_0 ("Rep_Set_0 X" ∪ "Rep_Set_0 Y")))

OclIntersection_def: OclIntersection ≡ lift2 (strictify (λX. strictify (λY. Abs_Set_0 ("Rep_Set_0 X" ∩ "Rep_Set_0 Y")))

ocl_setup_op [OclIncluding, OclExcluding, OclComplement, OclUnion]

ocl_setup_op [OclIntersection]

declare OCL_cp_OclIntersection[simp,intro!]

Note that also sequencing of declarations are possible.

syntax
@OclFinSet :: args ⇒ ('τ,'α Set_0) VAL (mkSet{(_)})

lemma mkSet{1,1,1,0,1} = ?X oops
```

368
Properties of the Operations

Useful Lemmas (necessary in this theory)

**Lemma** \( \text{OclIntersection}_\text{idem} \): \((\{\tau, \alpha::\text{bot}\}\cap S) = S\)

```plaintext
apply (rule ext)
apply (case_tac x ⊩ (∂ S))
apply (simp_all, ocl_hypsubst [2], simp_all)
apply (simp only: OclIntersection_def OclIsDefined_def strictify_def
          OclNot_def o_def lift0_def lift1_def lift2_def
          OclTrue_def DEF_def)
apply (simp_all add: strictify_def lift_drop_idem DEF_def symmetric
                       DEF_Rep_Set Rep_Set_0_inverse)
```
done

**Lemma** \( \text{not_empty} \): \(\text{OclNotEmpty}(X) = \neg (\text{OclIsEmpty}(X))\)

```plaintext
apply (rule ext)
apply (case_tac x ⊩ (∂ X))
apply (simp_all)
apply (ocl_hypsubst [2], simp_all)
apply (rule sym, rule trans)
apply (rule strictEq_is_strongEq_LE)
apply (simp_all)
apply (simp only: OclIsDefined_def OclTrue_def OclLocalValid_def OclMtSet_def
          OclIsEmpty_def OclStrongEq_def lift0_def lift1_def lift2_def
          strictify_def o_def not_def)
apply (auto simp: DEF_def)
apply (auto simp: DEF_def Rep_Set_0_inverse)
```
done

**Lemma** \( \text{OclIsEmpty}_\text{charn} : \text{OclIsEmpty}(X) = (X = \emptyset)\)

```plaintext
apply (rule ext)
apply (case_tac x ⊩ (∂ X))
apply (simp_all)
apply (ocl_hypsubst [2], simp_all)
apply (rule sym, rule trans)
apply (rule strictEq_is_strongEq_LE)
apply (simp_all)
apply (simp only: OclIsDefined_def OclTrue_def OclLocalValid_def OclMtSet_def
          OclIsEmpty_def OclStrongEq_def lift0_def lift1_def lift2_def
          strictify_def o_def not_def)
apply (auto simp: DEF_def)
apply (rule_tac t = {} in subst, assumption)
apply (rule DEF_Rep_Set)
apply (auto simp: DEF_def Rep_Set_0_inverse)
```
done
lemma OclIsNotEmpty_local_charn_LJE : 
\( \tau \models \partial (X::(\tau, \alpha::\text{bot})\text{Set}) \implies (\tau \models (\not= \emptyset(X))) = \exists a::(\tau, \alpha::\text{bot})\text{VAL} \ \tau \models \partial a \wedge \tau \models a \in X) \)

apply (simp only. OclIsDefined_def OclTrue_def OclLocalValid_def 
OclNotEmpty_def OclIncludes_def 
OclIsEmpty_def OclStrongEq_def 
lift0_def lift1_def lift2_def 
strictify_def o_def not_def DEF_def)

apply (simp)
apply (subst neq_commute, subst ex_in_conv[symmetric])
apply auto
apply (rule_tac x = \( \lambda c. x \) in exI)
apply (simp add: strictify_def DEF_def [symmetric] DEF_X_Set)
apply auto
apply (subgoal_tac Rep_Set_0 (Abs_Set_0 (\( c \)) = \( c \)) )
apply (auto simp: DEF_def)
done

Sets with defined size must be finite.

lemma finite_Sets : \( \tau \models \partial (self::(\tau, \alpha::\text{bot})\text{Set}) \implies \exists a::(\tau, \alpha::\text{bot})\text{VAL} \ \tau \models \partial a \wedge \tau \models a \in X) \)

apply (simp only: OclSize_def)
apply (simp add: lift0_def lift1_def lift2_def strictify_def DEF_def UU_up_def)
apply (cases self \( \tau = \bot \) )
apply (auto simp: OclLocalValid_def OclTrue_def lift0_def)
apply (cases finite (Lifting.drop (Rep_Set_0 (self \( \tau \) ))) )
apply (auto simp: OclLocalValid_def)
done

Sets with defined size must be defined.

lemma def_if_size_def : \( \tau \models \partial (self::(\tau, \alpha::\text{bot})\text{Set}) \implies \exists a::(\tau, \alpha::\text{bot})\text{VAL} \ \tau \models \partial a \wedge \tau \models a \in X) \)

apply (simp add: OclIsDefined_def OclTrue_def lift0_def lift1_def lift2_def 
strictify_def o_def not_def)
apply (simp only: OclSize_def)
apply (simp add: lift0_def lift1_def lift2_def strictify_def DEF_def UU_up_def)
apply (cases self \( \tau = \bot \) )
apply (auto simp: OclLocalValid_def)
done

lemma def_if_size_def_global : \( \exists a::(\tau, \alpha::\text{bot})\text{VAL} \ \tau \models \partial a \wedge \tau \models a \in X) \)

apply (rule valid_intro)
apply (drule_tac \( \tau = \tau \) in valid_elim)
done

lemmas fin1 = OclIsNotEmpty_local_charn_LJE 
lemmas fin2 = finite_Sets 
lemmas def_if_size = def_if_size_def 
lemmas def_if_size_global = def_if_size_def_global
apply (simp add: def_if_size_def)
done

lemma REQ_11_7_1_4_alt2:
  (self::′τ,′α::bot) Set) −> count((obj::′τ, ′α) VAL)) =
  (if obj ∈ self then 1 else 0 endif)
apply (rule ext)
apply (simp add:
  OclCount_def OclIncludes_def OclIf_def OclStrictEq_def
OCL_IntegerZero_ocl_int_def OCL_IntegerOne_ocl_int_def)
apply (simp only:
lift0_def lift1_def lift2_def lift3_def
  strictify_def o_def not_def OclsDefined_def OclTrue_def)
apply (simp_all (no_asm_use)
  add: UU_fun_def DEF_def strictify_def split
  add: split_if up.split)
done

Semantic Definitions of the Iterators

With "iterator", the standard attempts to subsume several vaguely related higher-order
concepts such as fold, map, filter and comprehensions. The quantifier ForAll is in fact
be defined via fold. Since this approach only works for finite sets, we do not follow the
standard here and give a more general definition, but show that this coincides with
the standards requirements for finite sets.

Universal Quantifiers
defs

\[ OclForAll \equiv \text{(lift}^2\text{ lift}_{\lambda 0} \text{ lift}_{\lambda 1}) (\text{strictify}(\lambda S. (\lambda P. (\forall x \in \text{Rep Set}_0 S). P x = \text{True} \text{ if } \exists x \in \text{Rep Set}_0 S. P x = \text{False} \text{ else } \bot))) \]

ocl_setup_op[OclForAll]

lemma OclExists_UU1 [simp]:
  (\exists x \in (\bot::′τ, ′α::bot) Set) • P (x::′τ, ′α) VAL)) = ⊥
by(simp add: OclExists_def)

lemma OclForAllMt [simp]:
  (\forall x \in (\bot::′τ, ′α::bot) Set) • P (x::′τ, ′α::bot) VAL)) = T
by(simp add: OclForAll_def OclTrue_def OclMtSet_def ss_lifting)
Appendix B. Isabelle Theories

lemma OclExistsMt [simp]:
\[(\exists\ x \in (\emptyset::('\tau,'\alpha::bot) Set) . P (x::('\tau, '\alpha::bot) VAL)) = F\]
by(simp add: OclExists_def)

lemma ForAllI:
assumes defS : τ ⊨ ∂(S::('τ,'α::bot) Set)
and allP : \( \forall x. τ ⊨ x \in S \implies τ ⊨ P x \)
sshows τ ⊨ (∀x ∈ S . P(x::'τ ⇒ 'α::bot))
apply(insert defS)
apply(rule_tac X=S in Set_sem_cases)
apply(simp_all add: localValidDefined2sem DEF_def)
apply(subgoal_tac (∀x ∈ c. P (λτ. x)) τ = (True))
apply(auto simp: OclForAll_def localValid2sem ss_lifting)
done

lemma notForAllI:
assumes isin : τ ⊨ x ∈ (S::('τ,'α::bot) Set)
and not : τ ⊨ \neg (P x)
and cpP : cp P
shows τ ⊨ (∀x ∈ S . P(x::'τ ⇒ 'α::bot))
apply(insert isin not cpP)
apply(rule_tac X=S in Set_sem_cases)
apply(simp_all add: localValidDefined2sem DEF_def)
apply(subgoal_tac (P x τ) = (P (λs. (x τ)) τ))
apply(case_tac (P x τ) ≠ ⊥, frule neq_commute)
apply(case_tac x τ ≠ ⊥, rotate_tac -1, frule neq_commute)
apply(subgoal_tac (∃x ∈ c. ((P (λs. x)) τ) = (False)))
apply(auto simp: OclNot_def OclIncludes_def OclNot_def
  ss_lifting localValid2sem o_def)
apply(rule_tac x=x in bexI)
apply(rule_tac x=P (λs. (x τ)) τ in boolean_cases_sem)
apply(simp_all)
apply(rule_tac x=x τ in bexI)
apply(rule_tac x=P (λs. (x τ)) τ in boolean_cases_sem)
apply(simp_all)
apply(rule trans, rule_tac x=x in cp_subst)
apply(simp_all add: lift0_def)
done

lemma ForAllD:
assumes allP : τ ⊨ (∀x ∈ S . P(x::'τ ⇒ 'α::bot))
and cpP : cp P
shows \( \forall x. τ ⊨ x \in (S::('τ,'α::bot) Set) \implies τ ⊨ P x \)
apply (insert allP cpP)
apply (simp add: OclForAll_def OclNot_def OclIncludes_def ss_lifting)
apply (simp add: strictify_def
split: split_if_asm)
apply (erule_tac x = x τ in ballE)
apply (simp_all add: cp_by_cpify lift0_def)
done

lemma localValidForall2forall :
assumes defS : τ ⊨ ∂(S::(′τ, ′α::bot))Set and cpP : cp P
shows (τ ⊨ ∀x ∈ S. P(x::′τ ⇒ ′α::bot)) = (∀ x. τ ⊨ x ∈ S → τ ⊨ P x)
apply (safe)
apply (drule ForAllD, simp_all add: cpP)
apply (rule ForAllI, simp_all add: defS)
done

lemma undefForAllI1:
assumes exUndef1 : τ ⊨ x ∈ (S::(′τ, ′α::bot))Set and exUndef2 : τ ⊨ ¬∂(P x)
and allP : ∀ x. τ ⊨ x ∈ S → τ ⊨ (P x) ∨ ¬(P x) and cpP : cp P
shows τ ⊨ ¬∂(∀x ∈ S/squaresmall P(x::′τ ⇒ ′α::bot))
apply (insert exUndef1 exUndef2)
apply (frule isDefined_if_valid)
apply (drule OCL_is_defopt_OclIncludes[THEN iffD1])
apply (simp add: OclLocalValid_def OclIsDefined_def OclTrue_def OclForAll_def OclNot_def OclIncludes_def ss_lifting)
apply (erule conjE)
apply (frule DEF_X_Set', clarify)
apply (simp, safe)
apply (rule_tac x = x in bexI, simp_all prefer 2)
apply (rule_tac x = x τ in bexI, simp_all)
apply (subgoal_tac P (λτ a. x τ) τ = P x τ, simp)
apply (rule_tac τ = τ and and P = P in cp_charn, simp add: cpP, rule cpP)
apply (frule_tac Q = P (λτ. xa) τ = False in contrapos_pp)
apply (rule_tac τ = τ in localValidToNotFalse2sem [THEN iffD1])
apply (rule allP)
apply (simp add: includes_charn2 DEF_def)
done

lemma undefForAllI2:
assumes exUndef1 : τ ⊨ x ∈ (S::(′τ, ′α::bot))Set and exUndef2 : τ ⊨ ¬∂(P x)
and allP : ∀ x. τ ⊨ x ∈ S → τ ⊨ (P x) ∨ ¬(P x) and cpP : cp P
shows τ ⊨ ¬∂(∀x ∈ S/squaresmall P(x::′τ ⇒ ′α::bot))
apply (insert exUndef1 exUndef2)
apply (frule isDefined_if_valid)
apply (drule OCL_is_defopt_OclIncludes[THEN iffD1])
apply (simp add: OclLocalValid_def OclIsDefined_def OclTrue_def OclForAll_def OclNot_def OclIncludes_def ss_lifting)
apply (erule conjE)
apply (frule DEF_X_Set', clarify)
apply (simp, safe)
apply (rule_tac x = x in bexI, simp_all prefer 2)
apply (rule_tac x = x τ in bexI, simp_all)
apply (subgoal_tac P (λτ a. x τ) τ = P x τ, simp)
apply (rule_tac τ = τ and and P = P in cp_charn, simp add: cpP, rule cpP)
apply (frule_tac Q = P (λτ. xa) τ = False in contrapos_pp)
apply (rule_tac τ = τ in localValidToNotFalse2sem [THEN iffD1])
apply (rule allP)
apply (simp add: includes_charn2 DEF_def)
done

373
Appendix B. Isabelle Theories

lemma isdefForAllI2:
  assumes exNot1: \( \vDash \exists x \in (S\cdot(\tau, (\alpha::\text{bot})))\text{Set} \)
  and exNot2: \( \vDash \neg (P x) \)
  and cpP : cp P
  shows \( \vDash (\forall x \in S \cdot \alpha) \Rightarrow (P x) \)
apply(insert exNot1 exNot2 cpP)
apply(drule_tac P=P in notForAllI)
apply(simp_all add: isDefined_if_invalid)
done

lemma isDefinedForall_UC:
  assumes cpP : cp P
  shows \( (\forall x \in S \cdot \alpha) \Rightarrow (P x)) = \) 
((\forall x \in S \cdot \alpha) \Rightarrow (P x)) \land
((\exists x \in S \cdot \neg (P x)) \lor
(\forall x \in S \cdot (\alpha)))
oops

lemma isDefinedForall_LJE:
  assumes cpP : cp P
  shows \( (\forall x \in S \cdot \alpha) \Rightarrow (P x)) = \) 
((\forall x \in S \cdot \alpha) \Rightarrow (P x)) \land
((\exists x \in S \cdot (\alpha) \Rightarrow (P x)) \lor
(\forall x \in S \cdot (\alpha) \Rightarrow (P x)))
oops

lemmas isdefForAllI3 = ForAll[THEN isDefined_if_valid]

lemma ExistsI:
  assumes isin : \( \vDash x \in (S\cdot(\tau, (\alpha::\text{bot})))\text{Set} \)
  and not : \( \vDash (P x) \)
  and cpP : cp P
  shows \( \vDash (\exists x \in S \cdot \alpha) \Rightarrow (P x)) \)
by (auto simp:OclExists_def intro: notForAllI isin not cpP)

lemma NotExistsI:
  assumes defS : \( \vDash (S\cdot(\tau, (\alpha::\text{bot})))\text{Set} \)
  and allP : \( \forall x. \vDash x \in S \Rightarrow \alpha \Rightarrow (P x) \)
  shows \( \vDash (\exists x \in S \cdot \alpha) \Rightarrow (P x)) \)
by (auto simp:OclExists_def intro: ForAll defS allP)

lemma ExistsD:

374
B.4. Library

assumes $ExP : \tau \models (\exists x \in S, P(x) : (\gamma : \alpha : \text{bot}))$
shows $\exists x. \tau \models x \in (S :: (\gamma, (\alpha : \text{bot})) \text{Set}) \wedge \tau \models (P x)$
apply (insert $ExP$)
apply (simp add: localValid2sem OclForAll_def OclNot_def OclIncludes_def
OclExists_def o_def ss_lifting)
apply (simp add: strictify_def
split: split_if_asm)
apply (frule DEF_X_Set', clarify)
apply (rule_tac x = lift0 x in exI)
apply (auto simp: lift0_def not_down_exists_lift)
done

lemma localValidExists2exists:
assumes $cpP : cp P$
assumes $defS : \tau \models \partial (S :: (\gamma, (\alpha : \text{bot})) \text{Set})$
shows $\tau \models (\exists x \in S /s\square \text{uaresmallsolid} P (x :: \gamma \Rightarrow (\alpha : \text{bot}))) =
(\exists x. \tau \models x \in S \wedge \tau \models (P x))$
apply (safe)
apply (drule ExistsD, simp)
apply (auto intro!: ExistsI $cpP$)
done

lemmas weak_pred_LJE = weak_prop_LJE
localValidForall2forall
localValidExists2exists

thm weak_pred_LJE

Select and Collect
def

$OclSelect_\text{def} : OclSelect \equiv (\text{lift2'} \text{lift_arg0} \text{lift_arg1}) \text{strictify} (\lambda S P.
if \quad \forall x \in "\text{Rep_Set_0} S". \text{DEF} (P x)
then \text{Abs_Set_0} (\{x \in "\text{Rep_Set_0} S". P x = True\})
else \bot)$

ocl_setup_op [OclSelect]

lemma SelectPartialUndef:
assumes $ainS : \tau \models a \in (S :: (\gamma, (\alpha : \text{bot})) \text{Set})$
and $allP : \tau \models \emptyset (P a)$
and $cpP : cp P$
shows $((x : S \mid P (x : (\gamma \Rightarrow (\alpha : \text{bot})))) \tau = \bot \tau$
apply (insert $ainS$ allP)
apply (frule isDefined_if_valid,
drule OCL_is_defopt_OclIncludes[THEN iffD1])
apply (simp add: OclSelect_def OclIncluding_def OclUndefined_def
OclLocalValid_def OclIsDefined_def OclTrue_def
OclForAll_def OclNot_def OclIncludes_def)
Appendix B. Isabelle Theories

apply auto
apply (erule_tac x = a in ballE, simp_all)
apply (subgoal_tac P (λx'. a τ) τ = P a τ, simp)
apply (rule_tac τ = τ and P = P in cp_charn, simp add: cpP, rule cpP)
done

lemma SelectPartialUndef2 : assumes aInS : τ ⊨ (a ::'τ ⇒ 'α ::'bot) ∈ (S ::('τ,('α::bot))Set) shows (λx : S | (λx. ⊥))((λx::'τ⇒'α::bot)) τ = ⊥ τ
by (rule SelectPartialUndef, auto intro!: aInS)

lemma mem_OclSelect_eq : assumes defS : τ ⊨ ∂(S ::('τ,('α::bot))Set) and defa : τ ⊨ ∂(a ::('τ ⇒ 'α::bot)) and aInS : τ ⊨ a ∈ S and allP : ∀x. τ ⊨ x ∈ S =⇒ τ ⊨ ∂(P x) and cpP : cp P
shows (τ ⊨ a ∈ ((λx : S | P (x ::'τ⇒'α::bot)))
= (τ ⊨ P a))
by (insert defS defa aInS)

apply (simp add: OclSelect_def OclIncluding_def OclUndefined_def
OclLocalValid_def OclIsDefined_def OclTrue_def OclForAll_def OclNot_def OclIncludes_def
ss_lifting', simplified)
apply (auto simp: strictify_def)
done

lemma nonmem_OclSelect : assumes defS : τ ⊨ ∂(S ::('τ,('α::bot))Set) and defa : τ ⊨ ∂(a ::('τ ⇒ 'α::bot)) and aInS : ¬ (τ ⊨ a ∈ S)
and cpP : cp P
shows ¬ (τ ⊨ a ∈ ((λx : S | P (x ::'τ⇒'α::bot))))
by (insert defS defa aInS)
apply (simp add: OclSelect_def OclIncluding_def
376
B.4. Library

apply (frule DEF_X_Set', clarify, simp)
done

lemma mem_Select_charn:
assumes defS : \( \tau \models \partial (S :: (\tau \Rightarrow \alpha :: \text{bot}))\text{Set} \)
and defa : \( \tau \models \partial (a :: (\tau \Rightarrow \alpha :: \text{bot})) \)
and allP : \( \forall x. \tau \models x \in S \Rightarrow \tau \models \partial (P x) \)
and cpP : cpP
shows \( \tau \models a \in (\langle x : S | P(x :: (\tau \Rightarrow \alpha :: \text{bot}))\rangle) = \)
\( (\tau \models \tau \wedge \tau \models a \in S) \)
apply (case_tac \( \tau \models a \in S \), simp_all)
by (intro mem_OclSelect_eq nonmem_OclSelect, simp_all add: allP defS defa cpP)+

lemma select_mem_charn:
assumes defT : \( \tau \models \partial (T :: (\tau \Rightarrow \alpha :: \text{bot}))\text{Set} \)
shows \( (\langle x : S | ((x :: (\tau \Rightarrow \alpha :: \text{bot})) \in T)\rangle) = \)
\( ((S :: (\tau, \alpha)\text{Set}) \cap T) \)
apply (rule defT)
apply (simp add: localValidDefined2sem DEF_def)
apply (frule DEF_X_Set', clarify)
apply (rule_tac X=S in Set_sem_cases, simp_all)
apply (simp_all add: OclSelect_def OclIncludes_def OclIntersection_def ss_lifting)
apply (safe)
apply (subst Abs_Set_0_inject_charn)
apply (auto)
done

lemma select_mem_charn_global[simp]:
assumes defT : \( \tau \models \partial (T :: (\tau \Rightarrow \alpha :: \text{bot}))\text{Set} \)
shows \( (\langle x : S | ((x :: (\tau \Rightarrow \alpha :: \text{bot})) \in T)\rangle) = \)
\( ((S :: (\tau, \alpha)\text{Set}) \cap T) \)
apply (rule ext, insert defT)
apply (rule select_mem_charn)
apply (auto simp: global_vs_local_validity[symmetric])
done

lemma select_mem_charn_universal[simp]:
shows \( (\langle x : S | ((x :: (\tau \Rightarrow \alpha :: \text{bot})) \in S)\rangle) = \)
\( ((S :: (\tau, \alpha)\text{Set})) \)

377
Appendix B. Isabelle Theories

apply (rule ext)
apply (case_tac x )
apply (simp_all, ocl_hypsubst [\A, simp_all])
apply (rule trans)
apply (rule select_mem_charn)
apply (simp_all)
done

defs
OclCollect_def: OclCollect S P ≡ λ τ.
  if DEF (S τ) then if \( \forall x \in \mathit{Rep_Set_0} (S \tau) \). DEF (P (\lambda x. \tau) x)
  then Abs_Set_0 (\lambda x. P (\lambda \tau. x) \tau) \{ Rep_Set_0 (S \tau) \} else ⊥
  else ⊥
Any and One
defs
OclAny_def :
OclAny ≡ (lift2' lift_arg0 lift_arg1) (strictify (λ S P.
  if (\forall x \in \mathit{Rep_Set_0} S). DEF (P x) \and
  (\exists x \in \mathit{Rep_Set_0} S).
  P x = \{\True\}) then SOME x.(x \in \mathit{Rep_Set_0} S \and P x = \{\True\})
  else ⊥))
ocl_setup_op [OclAny]
defs
OclOne_def :
OclOne ≡ (lift3' lift_arg0 lift_arg2 lift_arg0) (strictify (λ S P.
  if (\forall x \in \mathit{Rep_Set_0} S). DEF (P x)
  then \card { x \in \mathit{Rep_Set_0} S . P x = \{\True\}} = 1
  else ⊥))
ocl_setup_op [OclOne]
The iterator for Sets is not adequately defined in the standard (idempotence). This

OclIterate_def :
OclIterate ≡ (lift3' lift_arg0 lift_arg2 lift_arg0) (strictify (λ S P A.
  if (\finite \mathit{Rep_Set_0} S) then (fold_set (\lambda x y. P y x) A (\mathit{Rep_Set_0} S))

378
Properties of iterate on sets

lemma iterate_infinite:
\[ \tau \models \neg \exists \partial (\| (X::(\tau',\alpha::\text{bot})\text{Set}) \|) \Rightarrow \tau \models \neg \partial (\text{OclIterate} X (P::(\tau',\alpha)\text{VAL} \Rightarrow (\tau',\beta::\text{bot})\text{VAL} \Rightarrow (\tau',\beta)\text{VAL}) A) \]

apply (rule_tac X=X in Set_sem_cases, simp_all)
apply (case_tac finite c)
apply (simp_all add: localValidUnDefined2sem OclSize_def OclIterate_def ss_lifting)
done

Using this theorem all requirements proven about iterate on bags will carry over to iterate on finite sets quite easily. Because they are actually equivalent in this case.

lemma iterate_set2bag:
\[ \tau \models \partial (\| (X::(\tau',\alpha::\text{bot})\text{Set}) \|) \Rightarrow \tau \models (\text{OclIterate} X (P::(\tau',\alpha)\text{VAL} \Rightarrow (\tau',\beta::\text{bot})\text{VAL} \Rightarrow (\tau',\beta)\text{VAL}) A) \triangleq (\text{OclIterate} ((\text{asBag()} X)::(\tau',\alpha::\text{bot})\text{Bag}) P A) \]

apply (rule_tac X=X in Set_sem_cases, simp_all)
apply (case_tac finite c)
apply (simp_all add: OclSize_def OclIterate_def OclLocalValid_def OclStrongEq_def OCL_Bag.OclIterate_def fold_set_fold_multiset_conv OclAsBag_def OclTrue_def ss_lifting)
done

B.4.14. OclAny

theory OCL_OclAny
imports $\text{HOLOCL_HOME}/\text{src/library/collection}/\text{COLLECTION}/\text{OCL_Set}$
begin

Level 0
setup ⟨⟨ Theory.add_path UML_OCL.level0 ⟩⟩

constdefs
mk_OclAny :: 'α OclAny_0 ⇒ 'α U
mk_OclAny ≡ Inl
get_OclAny :: 'α U ⇒ 'α OclAny_0
get_OclAny ≡ sum_case (λx. x) (λx. ε x. True)
is_OclAny :: 'α OclAny ⇒ bool
is_OclAny obj ≡ DEF(obj)
is_OclAny_univ :: 'α U ⇒ bool

is_OclAny_univ ≡ false
Appendix B. Isabelle Theories

\[ \text{is}_\text{OclAny}_\text{univ} \equiv \text{sum_case}(\lambda x. \text{True})(\lambda x. \text{False}) \]

**lemma** \text{get\_mk\_OclAny\_id}:
\[ \text{get\_OclAny}(\text{mk\_OclAny}\ x) = x \]
by (simp add: \text{get\_OclAny\_def} \text{mk\_OclAny\_def})

**lemma** \text{mk\_get\_OclAny\_id}:
\[ \text{is}_\text{OclAny}_\text{univ}\ x \Rightarrow \text{mk\_OclAny}(\text{get\_OclAny}\ x) = x \]
apply (simp add: \text{get\_OclAny\_def} \text{mk\_OclAny\_def} \text{is}_\text{OclAny}_\text{univ\_def})
done

**lemma** \text{is\_OclAny\_univ\_implies\_is\_get}:
\[ \lbrack DEF(\text{get\_OclAny}\ x) ; \text{is\_OclAny}_\text{univ}\ x \rbrack \Rightarrow \text{is\_OclAny}(\text{get\_OclAny}\ x) \]
apply (simp add: \text{get\_OclAny\_def} \text{is\_OclAny\_univ\_def} \text{is\_OclAny\_def} \text{get\_OclAny\_def} \text{mk\_OclAny\_def})
done

**constdefs**
\[ \text{mk\_OclAny} :: (\tau \times \alpha \text{OclAny}_0)\text{VAL} \Rightarrow (\tau \times \alpha \text{U})\text{VAL} \]
\[ \text{mk\_OclAny} \equiv \text{lift1 level0.mk\_OclAny} \]
\[ \text{get\_OclAny} :: (\tau \times \alpha \text{U})\text{VAL} \Rightarrow (\tau \times \alpha \text{OclAny}_0)\text{VAL} \]
\[ \text{get\_OclAny} \equiv \text{lift1 level0.get\_OclAny} \]
\[ \text{is\_OclAny} :: (\tau \times \alpha \text{OclAny}_0)\text{VAL} \Rightarrow (\tau)\text{Boolean} \]
\[ \text{is\_OclAny} \equiv \text{lift1(strictify(lift o level0.is\_OclAny))} \]
\[ \text{is\_OclAny\_univ} :: (\tau \times \alpha \text{U})\text{VAL} \Rightarrow (\tau)\text{Boolean} \]
\[ \text{is\_OclAny\_univ} \equiv \text{lift1(lift o level0.is\_OclAny\_univ)} \]
\[ \text{OclAny} :: (\tau \times \alpha \text{OclAny}_0\text{Set}_0)\text{VAL} \]
\[ \text{OclAny} \equiv \text{lift0(Abs\_Set}_0\text{lift 'UNIV)} \]

**lemma** \text{get\_mk\_OclAny\_id}:
\[ \tau\equiv\text{level1.is\_OclAny\_obj} \Rightarrow \text{level1.get\_OclAny}(\text{level1.mk\_OclAny}\ \text{obj}) = \text{obj} \]
apply (unfold level1.is\_OclAny\_univ\_def level1.is\_OclAny\_def level1.get\_OclAny\_def
level1.mk\_OclAny\_def OclLocalValid\_def)
apply (rule ext)
apply (simp add: lift1\_def level0.get\_mk\_OclAny\_id)
done

**lemma** \text{mk\_get\_OclAny\_id}:

380
B.4. Library

\[ \tau \models level1.is_OclAny_univ univ \]
\[ \implies level1.mk_OclAny (level1.get_OclAny univ) \tau \models univ \tau \]
apply (frule isDefined_if_valid)
apply (frule localValidDefined2sem [THEN iffD1])
apply (unfold level1.is_OclAny_univ_def level1.is_OclAny_def level1.get_OclAny_def level1.mk_OclAny_def OclLocalValid_def)
apply (simp add: OclLocalValid_def lift1_def lift0_def level0.mk_OclAny_id OclTrue_def)
done

lemma is_OclAny_univ_implies_is_get:
\[ [ \text{DEF}(level1.get\_OclAny\_univ \tau); (\tau \models level1.is\_OclAny\_univ univ) ] \implies (\tau \models level1.is\_OclAny (level1.get\_OclAny univ)) \]
apply (frule isDefined_if_valid)
apply (frule localValidDefined2sem [THEN iffD1])
apply (unfold level1.is_OclAny_univ_def level1.is_OclAny_def level1.get_OclAny_def level1.mk_OclAny_def OclLocalValid_def)
apply (simp add: OclLocalValid_def lift1_def lift0_def level0.is_OclAny_univ_implies_is_get
OclTrue_def strictify_def DEF_def)
done

lemma is_mk_OclAny:
\[ \tau \models level1.is\_OclAny\_obj \]
\[ \implies (\tau \models level1.is\_OclAny\_univ (level1.mk\_OclAny \text{obj})) \]
apply (frule isDefined_if_valid)
apply (frule localValidDefined2sem [THEN iffD1])
apply (unfold level1.is_OclAny_univ_def level1.is_OclAny_def level1.get_OclAny_def level1.mk_OclAny_def OclLocalValid_def)
apply (simp add: OclLocalValid_def lift1_def lift0_def OclTrue_def strictify_def DEF_def)
apply (case_tac \text{obj} \tau = \text{down})
apply (auto simp: level0.is_mk_OclAny)
done

setup [Theory.add_path / ]

Level 2
setup [Theory.add_path UML_OCL]

constdefs
mk_OclAny :: \( '\tau \to '\alpha\ OclAny_0\ VAL \Rightarrow ('\tau \to '\alpha\ U)\ VAL \)
mk_OclAny \equiv level1.mk_OclAny
get_OclAny :: \( '\tau \to 'U\ VAL \Rightarrow ('\tau \to '\alpha\ OclAny_0)\ VAL \)
get_OclAny \equiv level1.get\_OclAny
is_OclAny :: \( '\tau \to '\alpha\ OclAny_0\ VAL \Rightarrow ('\tau \to Boolean \)
is_OclAny \equiv level1.is\_OclAny
is_OclAny_univ :: \( '\tau \to 'U\ VAL \Rightarrow ('\tau \to Boolean \)

381
Appendix B. Isabelle Theories

\[ \text{is}_\text{OclAny}_\text{univ} \equiv \text{level1.is}_\text{OclAny}_\text{univ} \]
\[ \text{OclAny} :: (\tau, \alpha \text{OclAny}_0 \text{Set}_0) \text{ VAL} \]
\[ \text{is}_\text{OclAny} \equiv \text{level1.OclAny} \]

**lemma** get.mk.OclAny.id:
\[ \tau \models \text{UML}_\text{OCL.is}_\text{OclAny} \text{ obj} \]
\[ \Rightarrow \text{UML}_\text{OCL.get}_\text{OclAny} (\text{UML}_\text{OCL.mk}_\text{OclAny} \text{ obj}) = \text{ obj} \]
\[ \text{apply} (\text{unfold} \text{UML}_\text{OCL.is}_\text{OclAny}_\text{def} \text{UML}_\text{OCL.get}_\text{OclAny}_\text{def} \text{UML}_\text{OCL.mk}_\text{OclAny}_\text{def}) \]
\[ \text{apply} (\text{simp add: level1.get.mk.OclAny.id}) \]
\[ \text{done} \]

**lemma** mk.get.OclAny.id:
\[ \tau \models \text{UML}_\text{OCL.is}_\text{OclAny}_\text{univ} \text{ univ} \]
\[ \Rightarrow \text{UML}_\text{OCL.mk}_\text{OclAny} (\text{UML}_\text{OCL.get}_\text{OclAny}_\text{univ}) \tau = \text{ univ} \tau \]
\[ \text{apply} (\text{unfold} \text{UML}_\text{OCL.is}_\text{OclAny}_\text{def} \text{UML}_\text{OCL.get}_\text{OclAny}_\text{def} \text{UML}_\text{OCL.mk}_\text{OclAny}_\text{def}) \]
\[ \text{apply} (\text{simp add: level1.mk.get.OclAny.id}) \]
\[ \text{done} \]

**lemma** is.OclAny.univ.implies.is.get:
\[ [ \text{DEF}(\text{UML}_\text{OCL.get}_\text{OclAny}_\text{univ} \tau); (\tau \models \text{UML}_\text{OCL.is}_\text{OclAny}_\text{univ} \text{ univ}) ] \]
\[ \Rightarrow (\tau \models \text{UML}_\text{OCL.is}_\text{OclAny} (\text{UML}_\text{OCL.get}_\text{OclAny}_\text{univ}) \tau) = \text{UML}_\text{OCL.is}_\text{OclAny}_\text{univ} \text{ univ} \tau \]
\[ \text{apply} (\text{unfold} \text{UML}_\text{OCL.is}_\text{OclAny}_\text{def} \text{UML}_\text{OCL.get}_\text{OclAny}_\text{def} \text{UML}_\text{OCL.mk}_\text{OclAny}_\text{def}) \]
\[ \text{apply} (\text{simp add: level1.is.OclAny.univ.implies.is.get}) \]
\[ \text{done} \]

**lemma** is.mk.OclAny:
\[ \tau \models \text{UML}_\text{OCL.is}_\text{OclAny} \text{ obj} \]
\[ \Rightarrow (\tau \models \text{UML}_\text{OCL.is}_\text{OclAny}_\text{univ} (\text{UML}_\text{OCL.mk}_\text{OclAny} \text{ obj})) \]
\[ \text{apply} (\text{unfold} \text{UML}_\text{OCL.is}_\text{OclAny}_\text{def} \text{UML}_\text{OCL.get}_\text{OclAny}_\text{def} \text{UML}_\text{OCL.mk}_\text{OclAny}_\text{def}) \]
\[ \text{apply} (\text{simp add: level1.is.mk.OclAny}) \]
\[ \text{done} \]

**setup** \[
\langle \langle \text{Theory.add_path /} \rangle \rangle \]

**Re-typing or Casting.**

The standard describes an operation that “casts” an object to one of its subtypes. Note that our OCL-encoder provides for each new class an own definition for this polymorphic scheme.

**consts**
\[ \text{OclAsType} :: (\tau, \alpha :: \text{bot}) \text{ VAL} \]
\[ (\tau, \beta \text{ Set}_0) \text{ VAL} \Rightarrow (\tau, \beta :: \text{bot}) \text{ VAL} \]
\[ (\_ \_ \rightarrow \text{OclAsType} (\_ \_)) \text{ VAL} \]

382
Checking Dynamic Types of “Kinds”

As a consequence of the difference between dynamic and static typing in object-oriented languages, a test for the “kind” of an object is provided. It checks for the dynamic type.

Note that this operator is in fact defined by our OCL encoder and uses the internally defined test functions.

\[
\text{OclIsKindOf} \quad \text{::} \quad \forall \tau, \alpha \quad \text{VAL} \Rightarrow \exists \beta \quad \text{VAL} \quad \Rightarrow \tau \quad \text{Boolean}
\]

\[
\text{OclIsKindOf self CharSet} \equiv \text{self} \in \text{CharSet}
\]

This simple fundamental definition is the fruit of our generalization of the OCL set theory.

\[
\text{OclIsTypeOf} \quad \text{::} \quad \forall \tau, \alpha \quad \text{VAL} \Rightarrow \exists \beta \quad \text{VAL} \quad \Rightarrow \tau \quad \text{Boolean}
\]

\[
\text{OclIsTypeOf self NarrowCharSet} \equiv \text{self} \in \text{NarrowCharSet}
\]

end

B.4.15. CharactersticSets

theory OCL_CharactersticSet imports $HOLOCL_HOME/src/library/basic/OCL_Real $HOLOCL_HOME/src/library/basic/OCL_String $HOLOCL_HOME/src/library/OclAny/OCL_OclAny begin

Embedding Characteristic Sets into OCL

lemma Integer_0_Defined [simp];
\[ \tau \models 0 \text{ Integer_0} \]
by (simp add: localValidDefined2sem Abs_Set_Mt_DEF Integer_0_def lift0_def)

lemma in_Integer_0;
\[ \tau \models \exists z \quad \Rightarrow \quad \tau \models (\text{z }\text{a }\text{Integer}) \in (\text{Integer_0};(\text{a }\text{Integer_0})\text{Set}) \]
apply (simp add: OCL_Set.OclIncludes_def Integer_0_def localValidDefined2sem
OclLocalValid_def OclIsDefined_def OclTrue_def
lift0_def lift1_def lift2_def strictify_def
Abs_Set_rangelift_DEF simplified DEF_def)
by (auto simp: DEF_def not_down_exists_lift)

lemma in_Integer_0R[simp]:

383
Appendix B. Isabelle Theories

\((\tau \models (x::'a Integer) \in (Integer_0::('a,Integer_0)Set)) = (\tau \models \partial x)\)

by (auto simp: in_Integer_0
     OCL_Logic_core.isDefined_if_valid
     [THEN OCL_Set.OCL_is_defopt_OclIncludes
      [THEN iffD1]])

The relevance of the latter type of rules become apparent if we look at the following example:

**Lemma test:**

\((\forall x \in Integer_0 . (x + (OclNegative x) \equiv 0)) = \top\)

oops

First of all, such an expression is syntactically illegal in OCL. In Standard syntax, this is presented as

\[
\text{Integer_0 -> forall } (x | x + (-x) = 0)
\]

\text{Integer_0} is considered a type name which is not part of the expression language; therefore, no quantification over them is possible in standard OCL. Since we interpret OCL types as sets in the sense of the OCL_Set theory, we can quantify and reason over it in HOL-OCL.

In particular, rule \textit{forallI} eliminates the quantifier and reduces this to

\[
\tau \models x \in Integer_0 \implies (x + (OclNegative x) \equiv 0)) = \top
\]  \hspace{1cm} (B.15)

which can be further simplified by in_Integer_0R to

\[
\tau \models \partial x \implies (x + (-x) \equiv 0)) = \top
\]  \hspace{1cm} (B.16)

This is a valid statement in OCL (although we omit the proof here). In contrast,

\[
(x + (-x) \equiv 0)) = \top
\]  \hspace{1cm} (B.17)

in itself is not valid since \(x\) may be undefined. Thus, quantifiers range over defined values only (as one would expect). These theorems also implicitly state that the characteristic sets of the basic types are non-empty.

**Lemma Real_DEFINED [simp]:**

\[
\tau \models \partial Real_0
\]

by (simp add: localValidDefined2sem Real_0_def lift0_def)

**Lemma in_Real_0:**

\[
\tau \models \partial x \implies (x::'a Real) \in (Real_0::('a,Real_0)Set)
\]

apply (simp add: OCL_Set.OclIncludes_def Real_0_def localValidDefined2sem
OclLocalValid_def OclIsDefined_def OclTrue_def
lift0_def lift1_def lift2_def strictify_def
Abs_Set_rangelift_DEF[simplified DEF_def])

384
B.4. Library

by (auto simp: DEF_def not_down_exists_lift)

lemma in_RealR_0 [simp]:
  \( \tau \vdash (x::'a Real) \in (\text{Real}_0::(\text{a,Real}_0)\text{Set}) = \)
  \( \tau \vdash \partial x \)
by (auto simp: in_Real_0
  isDefined_if_valid
  [THEN OCL_Set.OCL_is_defopt_OclIncludes
  [THEN iffD1]])

lemma Boolean_Defined [simp]:
  \( \tau \vdash \partial \text{Boolean}_0 \)
by (simp add: localValidDefined2sem Boolean_0_def lift0_def)

lemma in_Boolean:
  \( \tau \vdash \partial x \Rightarrow \)
  \( \tau \vdash (x::'a Boolean) \in (\text{Boolean}_0::(\text{a,Boolean}_0)\text{Set}) \)
apply (simp add: OCL_Set.OclIncludes_def Boolean_0_def localValidDefined2sem
  OclLocalValid_def OclIsDefined_def OclTrue_def
  lift0_def lift1_def lift2_def strictify_def
  Abs_Set_rangelift_DEF[simplified DEF_def])
by (auto simp: DEF_def not_down_exists_lift)

lemma in_BooleanR [simp]:
  \( \tau \vdash (x::'a Boolean) \in (\text{Boolean}_0::(\text{a,Boolean}_0)\text{Set}) = \)
  \( \tau \vdash \partial x \)
by (auto simp: in_Boolean
  isDefined_if_valid
  [THEN OCL_Set.OCL_is_defopt_OclIncludes
  [THEN iffD1]])

lemma String_0_Defined [simp]:
  \( \tau \vdash \partial \text{String}_0 \)
by (simp add: localValidDefined2sem String_0_def lift0_def)

lemma in_String_0:
  \( \tau \vdash \partial x \Rightarrow \)
  \( \tau \vdash (x::'a String) \in (\text{String}_0::(\text{a,String}_0)\text{Set}) \)
apply (simp add: OCL_Set.OclIncludes_def String_0_def localValidDefined2sem
  OclLocalValid_def OclIsDefined_def OclTrue_def
  lift0_def lift1_def lift2_def strictify_def
  Abs_Set_rangelift_DEF[simplified DEF_def])
by (auto simp: DEF_def not_down_exists_lift)

lemma in_StringR [simp]:
  \( \tau \vdash (x::'a String) \in (\text{String}_0::(\text{a,String}_0)\text{Set}) = \)
  \( \tau \vdash \partial x \)
Appendix B. Isabelle Theories

$$\tau \vdash \partial x$$
by \(\text{(auto simp: in(String) isDefined\_if\_valid)}\)

$$\Rightarrow \text{[THEN OCL\_Set.OCL\_is\_defopt\_OclIncludes [THEN iffD1]]}$$

**lemma** OclAny\_Defined [simp]:
\(\tau \vdash \partial\text{OclAny}\)
by (simp add: localValidDefined2sem OclAny\_def level1.OclAny\_def lift0\_def)

**lemma** in\_OclAny:
\(\tau \vdash \partial x \Rightarrow \tau \vdash (x::(\text{'}a,\text{'}b) \text{OclAny}) \in (\text{OclAny}::(\text{'}a,\text{'}b \text{OclAny}\_0)\text{Set})\)
apply (simp add: OCL\_Set.OclIncludes\_def OclAny\_def level1.OclAny\_def
localValidDefined2sem
OclLocalValid\_def OclUsDefined\_def OclTrue\_def
lift0\_def lift1\_def lift2\_def strictify\_def
Abs\_Set\_range\_lift\_DEF[simplified DEF\_def])
by (auto simp: DEF\_def not\_down\_exists\_lift)

**lemma** in\_OclAnyR[simp]:
\(\tau \vdash (x::(\text{'}a,\text{'}b) \text{OclAny}) \in (\text{OclAny}::(\text{'}a,\text{'}b \text{OclAny}\_0)\text{Set}) =\)
\((\tau \vdash \partial x)\)
by (auto simp: in\_OclAny
isDefined\_if\_valid
[THEN OCL\_Set.OCL\_is\_defopt\_OclIncludes [THEN iffD1]])

end

B.4.16. The OCL Library

theory OCL\_Library
imports
$\text{HOLOCL\_HOME}/src/library/collection/$\text{COLLECTION}/OCL\_CharacteristicSet
begin

Datatype Conversions

consts
\(\text{HolOclToReal} \ : 'a \text{Integer} => 'a \text{Real} \quad (_ \rightarrow \text{toReal}('') [66])\)
defs
\(\text{HolOclToReal\_def} : \text{lift1(strictify(λ x:Integer_0. _real (''x'')))}\)

end

386
B.5. State

B.5. OCL State

theory OCL_State
imports
$HOLOCL_HOME/src/library/OclAny/OCL_OclAny
begin

types
  ′a state = oid ⇒ ′a
  ′s St = ′s state × ′s state
  (′a,′t) V = (′a St,′t) VAL

translations
  a state <= (type) oid ⇒ a option
  a St <= (type) (a state) × (s state)
  (a,t) V <= (type) a St ⇒ t

To sum up, the HOL types assigned to the key OCL types of the OCL standard are a context (st) indexed family of types. Recall: the contexts are a state relation or a Kripke-Structure over states over the universe.

Provided that we have a representation of class types generated from an UML class diagram, we can therefore translate any OCL type into an HOL-type achieving type soundness on the HOL-level.

Checking Freshness of Object Instances in the State.

constdefs
  OclIsNew :: (′a U St,′b:bot) VAL ⇒ (′a U St) Boolean
  OclIsNew obj ≡ (λ(s,s′). level0.mk_OclAny ((obj) −> oclAsType(OclAny)) (s,s′)) ∈ ran s ∧ level0.mk_OclAny ((obj) −> oclAsType(OclAny)) (s,s′) ∈ ran s′

The OCL encoder must generate a coercion in order to make this operation applicable to any object of some class-type

Selecting the Objects living in a State.

consts
  OclAllInstances :: (′a U St,’b:bot) VAL ⇒ (′a U St,’b Set_0) VAL
  OclAllInstancesAtpre :: (′a U St,’b) VAL ⇒ (′a U St,’b Set_0) VAL

defs
  OclAllInstancesString_def[simp]:
  OclAllInstances (self:(′a U St,String_0) VAL) ≡ OclUndefined

  OclAllInstancesPreString_def[simp]:

end
Appendix B. Isabelle Theories

OclAllInstancesAtpre (self::('a U String_0) VAL) ≡ OclUndefined

defs
OclAllInstancesInteger_def[simp]:
OclAllInstances (self::('a U Integer_0) VAL) ≡ OclUndefined

OclAllInstancesPreInteger_def[simp]:
OclAllInstancesAtpre (self::('a U Integer_0) VAL) ≡ OclUndefined

defs
OclAllInstancesReal_def[simp]:
OclAllInstances (self::('a U Real_0) VAL) ≡ OclUndefined

OclAllInstancesPreReal_def[simp]:
OclAllInstancesAtpre (self::('a U Real_0) VAL) ≡ OclUndefined

defs
OclAllInstancesBoolean_def[simp]:
OclAllInstances (self::('a U Boolean_0) VAL) ≡ Boolean_0

OclAllInstancesPreBoolean_def[simp]:
OclAllInstancesAtpre (self::('a U Boolean_0) VAL) ≡ Boolean_0

And now the general case for all "objects", i.e. all instances of subclass OclAny:

defs
OclAllInstancesOclAny_def:
OclAllInstances (self::('a U 'a OclAny_0) VAL) ≡ λ(s,s'). Abs_Set_0 (level0.get_OclAny ' ran s')

OclAllInstancesPreOclAny_def:
OclAllInstancesAtpre (self::('a U 'a OclAny_0) VAL) ≡ λ(s,s'). Abs_Set_0 (level0.get_OclAny ' ran s')

The modifiedOnly() -clause and its Infrastructure

constdefs
OclOidOf :: 'a U state ⇒ 'a OclAny_0 ⇒ oid set
OclOidOf s X ≡ {x. (s x) = Some(OCL_OclAny.mk_OclAny X)}

The oid Set is a tribute to non-referential universes where there may be different object identifiers in the store for the same object. The restriction to 'a OclAny_0 assures that the projection is only applicable to objects.

constdefs
OclModifiedOnly :: ('a U St, 'a OclAny_0) Set ⇒ ('a U St) Boolean
OclModifiedOnly X ≡ (λ(s,s'). Abs_Set_0 (level0.get_OclAny ' ran s'))

∀ id ∈ -(∪ (OclOidOf s) "Rep_Set_0(X(s,s'))"). s id = s' id.

end
B.5.2. The OCL Calculi

theory OCL_Calculi
imports $HOLOCL_HOME/src/OCL_Library
begin

This theory develops some more experimental calculi for OCL such as the local judgement tableaux calculus (let!) or a particular congruence rewriting calculus. It is the basis of some more experimental proof procedures so far.

Some core tableaux calculus

The conceptual basis of this calculus is drawn from [28] where “labeled formulas” where introduced. These are:

- \( A^t \) (for \( A \models s = \top \ s \)),
- \( A^f \) (for \( A \models s = \bot \ s \)),
- \( A^u \) (for \( A \models s = s \))

A “labeled clause” is then a set of labelled formulas (considered as disjunctions). In their notation, the crucial rules for labeled clauses are:

\[
\begin{align*}
C, (A \land B)^t & \quad C, (A \land B)^f \quad C, (A \land B)^u \\
C, A^t & \quad C, A^f \quad C, A^u, B^u
\end{align*}
\]  

(B.18)

\[
\begin{align*}
C, (\neg A)^t & \quad C, (\neg A)^f \quad C, (\neg A)^u \\
C, A^t & \quad C, A^f
\end{align*}
\]  

(B.19)

Tableau-calculi were implemented in Isabelle not on the basis of clauses, but on Horn formula \( [A_1; \ldots; A_n] \Rightarrow B \) corresponding to \( \neg A_1, \ldots A_n, B \) in clause representation. The conclusion in the Horn-format is used with preference during proof search.

As a consequence, it is necessary to duplicate the set of rules into introduction- and elimination rules. Further, we use the validity normal form to represent negation; thus the handling of the not is different and has to be encoded into the rules (leading to further duplications of the rule sets), this is necessary in Isabelle’s representation of tableaux calculi anyway.

lemma and_TRUE_I:
\[
[ \tau \models A; \tau \models B ] \Rightarrow \tau \models (A \land B)
\]
by (erule subst_LJ_TRUE, erule subst_LJ_TRUE, auto)

lemma and_D1:
\[
[ \tau \models A \land B ] \Rightarrow \tau \models A
\]
apply (simp_all add: OclAnd_def OclLocalValid_def)
Appendix B. Isabelle Theories

DEF_def lift2_def split add: split_if
apply (case_tac A τ = down)
apply (simp_all add: OclTrue_def lift0_def)
apply (simp add: split_if_eq1)
apply (case_tac B τ = down)
apply (simp_all add: split_if_eq1 not_down_exists_lift)
apply (erule exE)
apply (erule exE)
apply (simp_all add: drop_lift)
done

lemma and_D2:
[[ τ ⊨ A ∧ B ]] ⇒ τ ⊨ B
by (rule and_D1, subst and_commute, assumption)

lemma and_TRUE_E:
[[ τ ⊨ (A ∧ B); [ τ ⊨ A; τ ⊨ B ]] ⇒ R ]] ⇒ R
by(frule and_D1, drule and_D2, auto)

lemma and_congr':
(τ ⊨ (A ∧ B)) = ((τ ⊨ A) ∧ (τ ⊨ B))
by(auto elim: and_TRUE_E and_D1 and_D2 intro: and_TRUE_I)

lemma not_and_undef_congr1[simp]:
(τ ⊨ ¬ (⊥ ∧ B)) = (τ ⊨ ¬ B)
apply (cut_tac τ = τ and A = B in non_quatrium_datur)
apply auto
apply (erule_tac subst_LJ_TRUE_fw, assumption, auto)
apply (erule_tac subst_LJ_FALSE, auto)
by (drule_tac subst_LJ_undef_fw, assumption, auto)

lemma or_undef_congr1[simp]:
(τ ⊨ ⊥ ∨ B) = (τ ⊨ B)
by (simp add: OclOr_def)

lemma not_and_undef_congr2[simp]:
(τ ⊨ ¬ (B ∧ ⊥)) = (τ ⊨ ¬ B)
by(subst and_commute, simp add: not_and_undef_congr1)

lemma or_undef_congr2[simp]:
(τ ⊨ B ∨ ⊥) = (τ ⊨ B)
by (simp add: OclOr_def)
lemma and_undef_congr3[simp]:
\neg (\tau \vdash \bot \land B)
  apply (cut_tac \tau = \tau \land A = B in non_quatrium_datur)
  apply auto
  apply (drule_tac P = \lambda . \bot \land B \land subst_LJ_TRUE-fw_rev)
  apply simp_all
  by (drule_tac subst_LJ_undef-fw, assumption, auto)

lemma and_undef_congr4[simp]:
\neg (\tau \vdash B \land \bot)
  by (subst and_commute, simp add: not_and_undef_congr1)

lemma and_FALSE_I_core:
[ (\tau \vdash \neg A) \lor (\tau \vdash \neg B) ] \Rightarrow (\tau \vdash \neg (A \land B))
  by (auto elim: subst_LJ_FALSE)

lemmas and_FALSE_I = disjCI [THEN and_FALSE_I_core, standard]

lemma and_FALSE_I1:
[ (\tau \vdash B) \lor (\tau \vdash \\neg B) ] \Rightarrow (\tau \vdash \neg A)
  by (rule and_FALSE_I, simp only: not_invalid)

lemma and_FALSE_E:
[ \neg (A \land B); \tau \vdash \neg A \Rightarrow R; \tau \vdash \neg B \Rightarrow R ] \Rightarrow R
  apply (cut_tac \tau = \tau \land A = A in non_quatrium_datur)
  apply auto
  apply (rotate_tac 1)
  apply (drule_tac subst_LJ_undef_fw)
  apply assumption
  prefer 3
  apply assumption
  apply auto
  done

lemma or_2_classic_disj:
[ \tau \vdash \theta X; \tau \vdash \theta Y ] \Rightarrow (\tau \vdash X \lor Y) = ((\tau \vdash X) \lor (\tau \vdash Y))
  apply auto
  defer 1 apply (erule_tac P = \lambda X St. (X \lor Y) \land subst_LJ_TRUE)
  prefer 3 apply (erule_tac P = \lambda Y St. (X \lor Y) \land subst_LJ_TRUE)
  apply (auto simp: not_valid)
  apply (rotate_tac 1)
  apply (drule_tac subst_LJ_FALSE-fw)
  apply auto
  done
Appendix B. Isabelle Theories

**lemma** isdef_and_congr:

\[
\begin{align*}
\tau \models (X \land Y) & \iff (\tau \models X \land \neg X) \lor \\
& \lor (\tau \models X \land \neg Y) \\
& \lor (\tau \models Y \land \neg Y)
\end{align*}
\]

apply (auto simp: isDefined_andD0 or_2_classic_disj or_2_classic_disj)
apply (simp_all add: or_2_classic_disj)
done

**lemma** solution0:

\[
\begin{align*}
\tau \models (X \land Y) & \lor \\
& \lor (X \land \neg X) \\
& \lor (Y \land \neg Y)
\end{align*}
\]

by (simp add: isdef_and_congr and_congr)

**lemma** solution1:

\[
\begin{align*}
\tau \models (X \land Y) & \iff (\tau \models X \land \neg X) \lor \\
& \lor (X \land \neg X)
\end{align*}
\]

apply (rule solution0)
apply (simp_all)
done

**lemma** solution2:

\[
\begin{align*}
\tau \models (X \land Y) & \iff (\tau \models X \land \neg X) \lor \\
& \lor (X \land \neg X)
\end{align*}
\]

apply (rule solution1)
apply (simp_all)
done

**lemma** solution3:

392
B.5. State

\[ \text{apply (rule isDefined_and_TRUE_I)} \]
\[ \text{apply (simp only: not_invalid)} \]
\[ \text{apply auto} \]
\[ \text{done} \]

\text{lemma isDefined_and_TRUE_E:} \\
\[ \text{apply (rule isDefined_and_TRUE_I)} \]
\[ \text{apply (simp only: not_invalid)} \]
\[ \text{apply auto} \]
\[ \text{done} \]

\text{The following rule follows the principal equality:} \\
\[ \tau \models \partial (\neg A \land \neg B) =\tau \models (\neg \partial \neg A \land \neg \partial \neg B) \] (B.20)

Via isDefined_andD0, de_morgan1, and de_morgan2 one gets:
\[ \tau \models \neg \partial (\neg (A \land B)) \] (B.21)

\text{lemma isDefined_and_FALSE_I:} \\
\[ \text{apply (rule isDefined_and_FALSE_I)} \]
\[ \text{apply (simp_all add: subst_LJ_undef subst_LJ_TRUE subst_LJ_FALSE)} \]
\[ \text{done} \]

\text{lemma isDefined_and_FALSE_I1:} \\
\[ \text{apply (rule isDefined_and_FALSE_I1)} \]
\[ \text{apply (simp_all add: subst_LJ_undef subst_LJ_TRUE subst_LJ_FALSE)} \]
\[ \text{done} \]

\text{lemma isDefined_and_FALSE_I2:} \\
\[ \text{apply (rule isDefined_and_FALSE_I2)} \]
\[ \text{apply (simp_all add: subst_LJ_undef subst_LJ_TRUE subst_LJ_FALSE)} \]
\[ \text{done} \]
Appendix B. Isabelle Theories

apply(auto simp del: isDefined_if_valid isDefined_if_invalid)
done

lemma isDefined_and_FALSE_I3:
assumes A: ((τ ⊥ ⊥ B) ⇒ (τ ⊥ ⊥ A)) ⇒ τ ⊥ ⊥ A
and B: ((τ ⊥ ⊥ B) ⇒ (τ ⊥ ⊥ A)) ⇒ ∼ (τ ⊥ ⊥ B)
shows τ ⊥ ⊥ (A ∧ B)
apply (rule isDefined_and_FALSE_I2)
apply (rule A)
apply (rule_tac [2] B)
apply auto
done

lemma isDefined_and_FALSE_I4:
[ τ ⊥ ⊥ (B) ⇒ (τ ⊥ ⊥ A);
τ ⊥ ⊥ (A) ⇒ (τ ⊥ ⊥ A);
τ ⊥ ⊥ (B) ⇒ (τ ⊥ ⊥ B) ]
⇒ τ ⊥ ⊥ (A ∧ B)
apply (rule isDefined_and_FALSE_I)
apply (rule and_FALSE_I)
prefer 2
apply (subst and_commute)
apply (rule_tac and_FALSE_I)
prefer 3
apply (subst and_commute)
apply (rule_tac and_FALSE_I)
apply (rule_tac auto simp del: isDefined_if_valid isDefined_if_invalid)
done

lemma isDefined_and_FALSE_I5:
[ (τ ⊥ ⊥ A) ∨ (τ ⊥ ⊥ B);
(τ ⊥ ⊥ A) ∨ (τ ⊥ ⊥ A);
(τ ⊥ ⊥ B) ∨ (τ ⊥ ⊥ B) ]
⇒ τ ⊥ ⊥ (A ∧ B)
apply(simp add: isDefined_andD0 de_morgan1 de_morgan2
 and_congr or_2_classic_disj)
apply(auto elim!: subst_LJ_undef subst_LJ_TRUE subst_LJ_FALSE
 simp del: isDefined_if_valid
 not_isUndefined_eq_isDefined)
done

lemma isDefined_and_FALSE_E:
[ τ ⊥ ⊥ (A ∧ B);
[ τ ⊥ ⊥ A; τ ⊥ ⊥ B] ⇒ R;
[ τ ⊥ ⊥ A; τ ⊥ ⊥ B ] ⇒ R;
[ τ ⊥ ⊥ A; τ ⊥ ⊥ B ] ⇒ R ]
\[ \Rightarrow R \]
by (auto elim: and_TRUE_E and_FALSE_E simp add: isDefined_andD0 OclOr_def)

lemmas isDefined_TRUE_E = non_ternium_datur_if_isDefined

lemma isDefined_TRUE_I:
\[ \llbracket \tau \models A \lor ( \tau \models \neg A ) \rrbracket \Rightarrow \tau \models \partial A \]
apply (erule disjE)
apply (drule isDefined_if_valid)
apply (drule_tac [2] isDefined_if_invalid)
apply auto
done

lemma isDefined_TRUE_I1:
\[ \llbracket ( \tau \models A ) \lor ( \tau \models ( \neg \partial ( A ))) = \Rightarrow \tau \models A \rrbracket = \Rightarrow \tau \models ( \partial A ) \]
apply (rule isDefined_TRUE_I)
apply (auto simp del: isDefined_if_valid isDefined_if_invalid)
apply (simp only: not_valid not_not isDefined_notD0)
done

lemma not_not_E:
\[ \llbracket \tau \models \neg (\neg A ) ; \tau \models A \Rightarrow R \rrbracket \Rightarrow R \]
by auto

lemma not_not_D:
\[ \tau \models \neg (\neg A ) \Rightarrow \tau \models A \]
by auto

lemma not_not_I:
\[ \llbracket \tau \models A \rrbracket \Rightarrow \tau \models \neg (\neg A ) \]
by auto

lemma isDefined_not_TRUE_E:
\[ \llbracket \tau \models \neg \partial (\neg A ) \rrbracket \Rightarrow \tau \models \neg \partial A \]
by auto

lemma isDefined_not_TRUE_I:
\[ \llbracket \tau \models \neg \partial A \rrbracket \Rightarrow \tau \models \neg \partial (\neg A ) \]
by auto

lemma isDefined_not_FALSE_E:
\[ \llbracket \tau \models \neg \partial (\neg A ) \rrbracket \Rightarrow \tau \models \neg \partial A \]
by auto

lemma isDefined_not_FALSE_I:
\[ \llbracket \tau \models \neg \partial A \rrbracket \Rightarrow \tau \models \neg \partial (\neg A ) \]
by auto

395
Appendix B. Isabelle Theories

Drived rules for the or

**lemma** or\_TRUE\_I\_core:

\[
(\tau \models A) \lor (\tau \models B) \implies \tau \models A \lor B
\]

apply (unfold OclOr\_def)
apply (rule and\_FALSE\_I\_core)
apply auto
done

**lemmas** or\_TRUE\_I = disjCI [THEN or\_TRUE\_I\_core, standard]

**lemma** or\_FALSE\_I:

\[
(\tau \models \neg A; \tau \models \neg B) \implies (A \lor B)
\]

apply (unfold OclOr\_def)
apply (rule not\_not\_I)
apply (rule and\_TRUE\_I)
apply auto
done

**lemma** or\_TRUE\_E:

\[
(\tau \models A \lor B; \tau \models A \implies R; \tau \models B \implies R) \implies R
\]

apply (unfold OclOr\_def)
apply (auto elim!: and\_TRUE\_E and\_FALSE\_E
simp add: isDefined\_andD0 OclOr\_def)
done

**lemma** or\_FALSE\_E:

\[
(\tau \models \neg (A \lor B); (\tau \models \neg A; \tau \models \neg B) \implies R) \implies R
\]

apply (unfold OclOr\_def)
apply (auto elim!: and\_TRUE\_E and\_FALSE\_E
simp add: isDefined\_andD0 OclOr\_def)
done

**lemma** isDefined\_or\_TRUE\_I:

\[
(\tau \models \not\emptyset B \lor (\tau \models \not\emptyset B); (\tau \models \not\emptyset A \lor (\tau \models \not\emptyset A)) \implies (\tau \models \not\emptyset A \land (\tau \models \not\emptyset B)) \implies (\tau \models \not\emptyset A \lor B)
\]

apply (unfold OclOr\_def)
apply (rule isDefined\_not\_TRUE\_I)
apply (rule isDefined\_and\_TRUE\_I)
B.5. State

apply auto
done

lemma isDefined_or_FALSE_I1:
\[
\begin{align*}
\tau \not\models \emptyset B & \Rightarrow \tau \not\models \emptyset A; \\
\tau \not\models \emptyset A & \Rightarrow \tau \models \neg A; \\
\tau \not\models \emptyset B & \Rightarrow \tau \models \neg B
\end{align*}
\Rightarrow (\tau \not\models \emptyset (A \lor B))
apply (unfold OclOr_def)
apply (rule isDefined_not_FALSE_I)
apply (rule isDefined_and_FALSE_I4)
apply auto
done

lemma isDefined_or_FALSE_I:
\[
\begin{align*}
(\tau \not\models A) \lor \neg (\tau \not\models \emptyset B) & \Rightarrow (\tau \not\models \emptyset A); \\
(\tau \not\models A) \lor \neg (\tau \not\models \emptyset B) & \Rightarrow \neg (\tau \not\models B)
\end{align*}
\Rightarrow (\tau \not\models \emptyset (A \lor B))
apply (unfold OclOr_def)
apply (rule isDefined_not_FALSE_I)
apply (rule isDefined_and_FALSE_I)
apply (rule and_FALSE_I)
apply (rule isDefined_not_FALSE_I)
apply (rule_tac [2] and_FALSE_I)
apply (rule_tac [3] and_FALSE_I)
apply (rule_tac [3] isDefined_not_FALSE_I)
apply auto
apply (simp_all only: not_valid)
apply (auto simp: isDefined_if_valid)
done

lemma isDefined_or_FALSE_E:
\[
\begin{align*}
\tau \not\models \emptyset (A \lor B); \\
\tau \not\models \emptyset A; \tau \not\models \emptyset B & \Rightarrow R; \\
\tau \not\models \emptyset A; \tau \not\models \neg B & \Rightarrow R; \\
\tau \models \neg A; \tau \not\models \emptyset B & \Rightarrow R
\end{align*}
\Rightarrow R
apply (unfold OclOr_def)
apply (drule isDefined_not_FALSE_E)
apply (auto elim!: and_TRUE_E and_FALSE_E isDefined_and_FALSE_E simp add: isDefined_andD0 OclOr_def)
done

lemma isDefined_or_TRUE_E:
\[
\tau \not\models \emptyset (A \lor B);
\]
Appendix B. Isabelle Theories

\[
\begin{align*}
[ \tau \vdash A; \tau \vdash B ] & \implies R; \\
\tau \vdash B & \implies R; \\
\tau \vdash A & \implies R \\
\implies R
\end{align*}
\]

apply (unfold OclOr_def)
apply (drule isDefined_not_TRUE_E)
apply (erule isDefined_and_TRUE_E)
apply auto
done

lemma strongEq_distrib_if1:
\[ \begin{align*}
\vdash \partial X & \implies \begin{align*}
(A \doteq ((if X then (Y::'a=>('b:bot)) else Z)\ doteq)) &= \\
((if X then A \doteq Y else A \doteq Z) \doteq A)
\end{align*}
\end{align*}
\]
by(rule if_distrib2, auto simp: OclValid_def OclLocalValid_def)

lemma strongEq_distrib_if2:
\[ \begin{align*}
\vdash \partial X & \implies \begin{align*}
((if X then (Y::'a=>('b:bot)) else Z) \doteq A) &= \\
((if X then Y \doteq A else Z \doteq A) \doteq A)
\end{align*}
\end{align*}
\]
by(rule if_distrib2, auto simp: OclValid_def OclLocalValid_def)

lemma cp_local_valid [simp,intro!]:
\[ cp P \implies cp (\lambda X \tau. (\tau \vdash P X)) \]
by(auto simp: OclValid_def OclLocalValid_def)

lemma and_congr:
\[ \begin{align*}
[ (\tau \vdash A) \lor (\tau \vdash \partial A) ] & \implies (\tau \vdash B) = (\tau \vdash B') \\
\implies (\tau \vdash A \land B) & = (\tau \vdash A \land B')
\end{align*}
\]
apply (cut_tac \tau = \tau and A = A in non_quatrium_datur)
apply (rule_tac X = A in cp_distinct_core_P)
apply (auto simp: OclLocalValid_def[symmetric])
done

lemma and_congr_a:
\[ \begin{align*}
[ \tau \vdash \partial A; \tau \vdash A ] & \implies (\tau \vdash B) = (\tau \vdash B') \\
\implies (\tau \vdash A \land B) & = (\tau \vdash A \land B')
\end{align*}
\]
apply (rule and_congr)
apply auto
done

lemma and_congr_b:

398
B.5. State

\[ \tau \models \neg A; \tau \models \not \partial A \Rightarrow (\tau \models B) = (\tau \models B') \]
\[ \Rightarrow (\tau \models A \land B) = (\tau \models A \land B') \]
apply (rule \text{and\_congr})
apply auto
done

lemma \text{not\_congr}:
\[ A \tau = A' \tau \Rightarrow (\tau \models \neg A) = (\tau \models \neg A') \]
apply (rule \text{sym})
apply (erule_tac P = \lambda A \tau. (\tau \models \neg A) in ocl\_cp\_subst)
apply auto
done

The relevance for automated deduction becomes apparent with the next rewrite rule, which boils down reductions in local congruences to check if the definedness is equivalent. This can be decomposed into two meta-implications; the advantage of this is presentation comes from the fact that such derivations are particularly easy and can be supported automatically by a forward-closure tactic for literals contained in valid definedness formulas. It has to be checked if the recomputation of these literals can be avoided.

lemma \text{not\_congr0}:
\[ (\tau \models \not \partial A) \Rightarrow (\tau \models \neg A) = (\tau \models \neg A') \]
apply (rule \text{not\_congr})
apply (subst equiv\_D\_local) back
apply auto
done

lemma \text{or\_congr}:
\[ A \tau = A' \tau; B \tau = B' \tau \Rightarrow (\tau \models A \lor B) = (\tau \models A' \lor B') \]
apply (rule_tac P = \lambda A \tau. (\tau \models A \lor B) in ocl\_cp\_subst)
apply (rule \text{sym}, \text{assumption})
apply (rule_tac P = \lambda B \tau. (\tau \models A' \lor B) in ocl\_cp\_subst)
apply (rule \text{sym}, \text{assumption})
apply auto
done

lemma \text{or\_congr0}:
\[ (\tau \models \not \partial A) \Rightarrow (\tau \models \neg A) = (\tau \models \neg A') \]
apply (rule \text{or\_congr})
Appendix B. Isabelle Theories

apply (subst equiv_D_local) back
prefer 2
apply (subst equiv_D_local) back
apply (auto simp: OclLocalValid_def)
apply (auto simp: OclLocalValid_def[symmetric])
done

lemma if_congr:
[ \( \tau \models C \implies (\tau \models A) = (\tau \models A') \); 
  \( \tau \models \neg C \implies (\tau \models B) = (\tau \models B') \) ]
\( \implies ((\tau \models if C then A else B endif) = 
  (\tau \models if C then A' else B' endif)) \)
apply (cut_tac \( \tau = \tau \) and \( A = A \) in non_quatrium_datur)
apply (rule_tac X = \( A \) in cp_distinct_core_P)
apply (auto simp: is_FALSE_charn_local OclLocalValid_def[symmetric])
done

lemma implies_undef_congr[simp]:
\( (\tau \models \bot \implies B) = (\tau \models B) \)
by (simp add: OclImplies_def)

lemma implies_congr:
[ \( (\tau \models A) \lor (\tau \models \emptyset \ A) \implies (\tau \models B) = (\tau \models B') \) ]
\( \implies ((\tau \models A \rightarrow B) = (\tau \models A \rightarrow B')) \)
apply (rule_tac X = \( A \) in cp_distinct_core_P)
apply (auto simp: isUndefined_charn_local OclLocalValid_def[symmetric])
done

lemmas LEC = and_congr not_congr or_congr if_congr
  not_and_undef_congr1
  not_and_undef_congr2
  or_undef_congr1
  or_undef_congr2
  and_undef_congr1
  and_undef_congr2
  and_undef_congr3
  and_undef_congr4
  implies_undef_congr

end

B.5.3. Encoding of OCL Operations

theory OCL_Operation
imports $HOLOCL_HOME/src/OCL_Calculi
begin

400
B.5. State

Standard and Strict Method Invocation, and Recursion

The core primitives *OCL_choose* and *OCL_invoke*, its strict variant *OCL_invokeS* are used to represent method.

**consts**

*OCL_choose* :: 

\[(\tau, b) \text{ VAL } \Rightarrow (\tau, b) \text{ VAL} \]

*OCL_choose* $F \equiv (\lambda \tau. \text{SOME } x. F (\lambda \tau. x) \tau = T) \tau$

**lemma**

$\text{cp } F \Rightarrow \text{cp(OCL_choose } F)\$

apply (simp add: cp_def OCL_choose_def)

apply safe

apply (rule exI, rule allI, rule allI)

apply (simp)

oops

**consts**

*OCL_invoke* :: 

\[(\text{e }\Rightarrow \text{'s}) \Rightarrow (\text{'s set }\rightarrow (\text{'a, 'b}) \text{ VAL } \Rightarrow 'c) \Rightarrow (\text{'a, 'b}) \text{ VAL } \Rightarrow 'c \]

*OCL_invokeS* :: 

\[(\text{'b }\Rightarrow \text{'s}) \Rightarrow (\text{'s set }\rightarrow (\text{'a, 'b}) \text{ VAL } \Rightarrow 'c) \Rightarrow (\text{'a, 'b}) \text{ VAL } \Rightarrow 'c \]

Instance for null-ary invocation with return value:

*OCL_invoke* $(C :: 'a \Rightarrow 'd)$

\[
\text{(tab :: 'd set }\rightarrow ('s \Rightarrow 'a) \Rightarrow ('s \Rightarrow 'b) \Rightarrow 's \Rightarrow 'c)
\]

\[
(\text{self :: 's }\Rightarrow 'a)(\text{result :: 's }\Rightarrow 'b)
\]

**defs**

*OCL_invoke0_def* :

\[
\lambda \text{ St. (case tab (LEAST } X. X \in \text{dom tab } \land C(\text{self }\text{ St}) \in X) of}
\]

None \Rightarrow arbitrary

| Some $f \Rightarrow f (\text{self :: 's }\Rightarrow 'a)$

(\text{result :: 's }\Rightarrow 'b) \text{ St})

**defs**

*OCL_invoke0S_def* :

\[
\text{(if } \partial(\text{self}) \text{ then OCL_invoke } C \text{ tab result else } \bot \text{ endif)}
\]

**lemma**

*OCL_invoke0S_undef0[simp] : *

\[
\text{OCL_invoke0S } C \text{ (tab :: 'c set }\rightarrow ('a \Rightarrow 'd : \text{bot}) \Rightarrow ('a \Rightarrow 'e : \text{bot}) \Rightarrow 'a \Rightarrow 'b : \text{bot}) \Rightarrow X \Rightarrow \bot
\]

by (simp add: OCL_invoke0S_def)

**lemma**

*OCL_invoke0S_is_def* :

\[
\Rightarrow \partial(\text{OCL_invoke0S } C
\]

\[
(\text{tab :: 'c set }\rightarrow ('a \Rightarrow 'd : \text{bot}) \Rightarrow ('a \Rightarrow 'e : \text{bot}) \Rightarrow 'a \Rightarrow 'b : \text{bot})
\]

401
Appendix B. Isabelle Theories

\[ \text{self result} \implies \]

\[ \models \neg(\text{self}) \]

by (simp add: OCL_invoke0S_def isDefined_ifD0 globalValidAnd2conj)

Instance for binary invocation:

OCL_invoke \((C :: 'a \Rightarrow 'e)\)

\[(\text{tab} :: 'e 	ext{set} \to ('s \Rightarrow 'a) \Rightarrow ('s \Rightarrow 'b) \Rightarrow ('s \Rightarrow 'c) \Rightarrow 'd)(\text{self} :: 's \Rightarrow 'a)\]

\[(a :: 's \Rightarrow 'b)(b :: 's \Rightarrow 'c)\]

defs OCL_invoke1_def :

\[ \text{OCL_invokeC tab self a result} \equiv \lambda \tau \cdot \begin{cases} \text{None} & \text{arbitrary} \\ \text{Some f} & f(\text{self} :: 'tau \Rightarrow 'a) \\ \text{result :: 'tau \Rightarrow 'c} & \tau \end{cases} \]

defs OCL_invoke1S_def :

\[ \text{OCL_invokeS C tab self a result} \equiv \begin{cases} \text{OCL_invoke C tab self (a :: 'tau \Rightarrow 'b \Rightarrow 'c \Rightarrow 'd \Rightarrow 'e \Rightarrow 'f)} & \text{if } \partial(\text{self}) \text{ and } \partial(\text{a}) \\ \bot & \text{else} \end{cases} \]

lemma OCL_invoke1S_undef[simp] :

\[ \text{OCL_invokeS C tab self a result} = \bot \]

by (simp add: OCL_invoke1S_def)

lemma OCL_invoke1S_undef[simp] :

\[ \text{OCL_invokeS C tab self a result} = \bot \]

by (simp add: OCL_invoke1S_def)

lemma OCL_invoke2S_is_def' :

\[ \models \partial(\text{OCL_invokeS C}) \]

\[ \text{OCL_invokeS C tab self a result} \equiv \begin{cases} \text{OCL_invokeC tab self a result} & \text{if } \partial(\text{self}) \text{ and } \partial(\text{X}) \\ \bot & \text{else} \end{cases} \]

by (simp add: OCL_invoke1S_def isDefined_ifD0 globalValidAnd2conj)

Instance for Binary Invocation:

defs OCL_invoke2_def :

\[ \]
OCL_invoke C tab self a b result
≡ λ St. (case tab (LEAST X. X ∈ dom tab ∧ C(self St) ∈ X) of
None ⇒ arbitrary
| Some f ⇒ f (self::′s⇒′a)(a::′s⇒′b)(b::′s⇒′c)(result::′s⇒′d) St )

defs
OCL_invoke2S_def:
OCL_invokeS C tab self a b result
≡ (if ∂(self) and ∂(a) and ∂(b)
then OCL_invoke C tab (self::′s⇒′a:bot)
(a::′s⇒′b:bot)(b::′s⇒′c:bot)
(result::′s⇒′d)
else ⊥
endif)

lemma OCL_invoke2S_undef0[simp] :
OCL_invoke2S C
(tab::′e set→(′s⇒′a:bot)⇒(′s⇒′b:bot)
⇒(′s⇒′c:bot)⇒(′s⇒′d:bot)⇒(′s⇒′e:bot))
⊥ X Y Z
=
⊥
by(simp add: OCL_invoke2S_def)

lemma OCL_invoke2S_undef1[simp] :
OCL_invoke2S C
(tab::′e set→(′s⇒′a:bot)⇒(′s⇒′b:bot)
⇒(′s⇒′c:bot)⇒(′s⇒′d:bot)⇒(′s⇒′e:bot))
X ⊥ Y Z
=
⊥
by(simp add: OCL_invoke2S_def)

lemma OCL_invoke2S_undef2[simp] :
OCL_invoke2S C
(tab::′e set→(′s⇒′a:bot)⇒(′s⇒′b:bot)
⇒(′s⇒′c:bot)⇒(′s⇒′d:bot)⇒(′s⇒′e:bot))
X Y ⊥ Z
=
⊥
by(simp add: OCL_invoke2S_def)

lemma OCL_invoke2S_is_def:
≡ ∂(OCL_invokeS C
(tab::′e set→(′s⇒′a:bot)⇒(′s⇒′b:bot)
⇒(′s⇒′c:bot)⇒(′s⇒′d:bot)⇒(′s⇒′e:bot))
self X Y result) ⊢
≡ ∂(self) and ∂(X) and ∂(Y)
by(simp add: OCL_invoke2S_def isDefined_ifD0 globalValidAnd2conj)
Appendix B. Isabelle Theories

B.5.4. The Object Constraint Language: OCL

theory UML_OCL
imports 
  "$HOLOCL_HOME/src/OCL_State"
  "$HOLOCL_HOME/src/OCL_Operation"
begin

The Theory-Morpher

In the theory file, this section contains an SML program implementing a theory morphism from HOL to HOL-OCL. It filters all combinator-based definitions in HOL-OCL, constructs a one-to-one constant symbol based signature morphism between the HOL and the HOL-OCL theory (modulo argument permutation), uses this signature morphism to translate HOL theorems to HOL-OCL proof goals and attempts to prove them with a generic case split tactic. For conceptual details, see [13].

end

B.6. Requirements

In this section, we present the requirements from the normative part of the OCL standard [41, Chapter 11] and prove (or disprove) that our embedding is faithfull to these requirements.

At time of writing, this is work in progress and not all requirements are analyzed. This section will be improved in one of the next releases of HOL-OCL.

B.6.1. Requirements for OCL Primitive Types

theory OCL_Basic_requirements
imports
  "$HOLOCL_HOME/src/OCL_Library"
begin

This theory contains the requirements for the primitive types defined in the OCL standard library (Integer, Real, String, and Boolean) [41] Chapter 11.5].

Real

+ (r : Real) : Real \hfill (11.5.1-a)
   The value of the addition of self and r.

404
### B.6. Requirements

<table>
<thead>
<tr>
<th>Function</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>( - (r : \text{Real}) : \text{Real} )</td>
<td>The value of the subtraction of ( r ) from self. (11.5.1-b)</td>
</tr>
<tr>
<td>( * (r : \text{Real}) : \text{Real} )</td>
<td>The value of the multiplication of self and ( r ). (11.5.1-c)</td>
</tr>
<tr>
<td>( - : \text{Real} )</td>
<td>The negative value of self. (11.5.1-d)</td>
</tr>
<tr>
<td>( / (r : \text{Real}) : \text{Real} )</td>
<td>The value of self divided by ( r ). (11.5.1-e)</td>
</tr>
<tr>
<td>( \text{abs}() : \text{Real} )</td>
<td>The absolute value of self. (11.5.1-f)</td>
</tr>
<tr>
<td>( \text{floor}() : \text{Integer} )</td>
<td>The largest integer which is less than or equal to self. (11.5.1-g)</td>
</tr>
<tr>
<td>( \text{round}() : \text{Integer} )</td>
<td>The integer which is closest to self. When there are two such integers, the largest one. (11.5.1-h)</td>
</tr>
</tbody>
</table>

#### Lemma

- **req_11_5_1-f:**
  \[
  \neg\ (\text{self} - \text{abs}()) = (\text{if} (\text{self} < 0) \text{OclNegative self} \text{else self endif})
  \]

- **req_11_5_1-g:**
  \[
  \neg((\text{self} - \text{floor}()) - \text{toReal}() \leq \text{self}) \land ((\text{self} - \text{floor}() + 1) - \text{toReal()} > \text{self})
  \]

- **req_11_5_1-h:**
  \[
  \neg((\text{self} - \text{result}).\text{abs}() < 0.5) \\
  \text{or} ((\text{self} - \text{result}).\text{abs}() = 0.5 \text{ and } (\text{result} > \text{self}))
  \]

#### Ouch

- **post** if self < 0 then result = - self else result = self endif

- **post** (result <= self) and (result + 1 > self)

- **post** ((self - result).abs() < 0.5) or ((self - result).abs() = 0.5 and (result > self))

- **post** true
Appendix B. Isabelle Theories

\textbf{max}(r : \text{Real}) : \text{Real} \quad \text{(11.5.1-i)}

The maximum of self and r.

\textbf{post}: \text{if self} \geq r \text{ then result} = \text{self}
   \text{else result} = r
   \text{endif}

\textbf{min}(r : \text{Real}) : \text{Real} \quad \text{(11.5.1-j)}

The minimum of self and r.

\textbf{post}: \text{if self} \leq r \text{ then result} = \text{self}
   \text{else result} = r
   \text{endif}

\textbf{<} (r : \text{Real}) : \text{Boolean} \quad \text{(11.5.1-k)}

True if self is less than r.

\textbf{>} (r : \text{Real}) : \text{Boolean} \quad \text{(11.5.1-l)}

True if self is greater than r.

\textbf{post}: \text{result} = \text{not (self} \leq r)

\textbf{\leq} (r : \text{Real}) : \text{Boolean} \quad \text{(11.5.1-m)}

True if self is less than or equal to r.

\textbf{post}: \text{result} = (\text{self} = r \text{ or (self} < r))

\textbf{\geq} (r : \text{Real}) : \text{Boolean} \quad \text{(11.5.1-n)}

True if self is greater than or equal to r.

\textbf{post}: \text{result} = (\text{self} = r \text{ or (self} > r))

\textbf{Integer}

\textbf{-} : \text{Integer} \quad \text{(11.5.2-a)}

The negative value of self.
### B.6. Requirements

<table>
<thead>
<tr>
<th>Operator</th>
<th>Type</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>+ (i : Integer) : Integer</code></td>
<td><code>(11.5.2-b)</code></td>
<td>The value of the addition of self and i.</td>
</tr>
<tr>
<td><code>- (i : Integer) : Integer</code></td>
<td><code>(11.5.2-c)</code></td>
<td>The value of the subtraction of i from self.</td>
</tr>
<tr>
<td><code>* (i : Integer) : Integer</code></td>
<td><code>(11.5.2-d)</code></td>
<td>The value of the multiplication of self and i.</td>
</tr>
<tr>
<td><code>/ (i : Integer) : Real</code></td>
<td><code>(11.5.2-e)</code></td>
<td>The value of self divided by i.</td>
</tr>
<tr>
<td><code>abs() : Integer</code></td>
<td><code>(11.5.2-f)</code></td>
<td>The absolute value of self.</td>
</tr>
<tr>
<td><code>div( i : Integer) : Integer</code></td>
<td><code>(11.5.2-g)</code></td>
<td>The number of times that i fits completely within self.</td>
</tr>
<tr>
<td><code>mod( i : Integer) : Integer</code></td>
<td><code>(11.5.2-h)</code></td>
<td>The result is self modulo i.</td>
</tr>
<tr>
<td><code>max(i : Integer) : Integer</code></td>
<td><code>(11.5.2-i)</code></td>
<td>The maximum of self an i.</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Post Condition for <code>abs()</code></th>
</tr>
</thead>
<tbody>
<tr>
<td><code>post: if self &lt; 0 then result = - self</code></td>
</tr>
<tr>
<td><code>else result = self</code></td>
</tr>
<tr>
<td><code>endif</code></td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Pre Condition and Post Condition for <code>div()</code></th>
</tr>
</thead>
<tbody>
<tr>
<td><code>pre: i &lt;&gt; 0</code></td>
</tr>
<tr>
<td><code>post: if self / i &gt;= 0 then result = (self / i).floor()</code></td>
</tr>
<tr>
<td><code>else result = -((-self/i).floor())</code></td>
</tr>
<tr>
<td><code>endif</code></td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Post Condition for <code>mod()</code></th>
</tr>
</thead>
<tbody>
<tr>
<td><code>post: result = self - (self.div(i) * i)</code></td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Post Condition for <code>max()</code></th>
</tr>
</thead>
<tbody>
<tr>
<td><code>post: if self &gt;= i then result = self</code></td>
</tr>
<tr>
<td><code>else result = i</code></td>
</tr>
<tr>
<td><code>endif</code></td>
</tr>
</tbody>
</table>
Appendix B. Isabelle Theories

\textbf{min(i : Integer) : Integer} \hfill (11.5.2-j)

The minimum of self an i.

\textbf{post:} if self <= i then result = self 
else result = i 
endif

\textbf{String}

\textbf{size(): Integer} \hfill (11.5.3-a)

The number of characters in self.

\textbf{concat(s : String) : String} \hfill (11.5.3-b)

The concatenation of self and s.

\textbf{post:} result.size() = self.size() + string.size()
\textbf{post:} result.substring(1, self.size()) = self
\textbf{post:} result.substring(self.size() + 1, result.size()) = s

\textbf{substring(lower : Integer, upper : Integer) : String} \hfill (11.5.3-c)

The sub-string of self starting at character number lower, up to and including character number upper. Character numbers run from 1 to self.size().

\textbf{pre:} 1 <= lower 
\textbf{pre:} lower <= upper 
\textbf{pre:} upper <= self.size()

\textbf{toInteger()} : Integer \hfill (11.5.3-d)

Converts self to an Integer value.

\textbf{toReal()} : Real \hfill (11.5.3-e)

Converts self to a Real value.

\textbf{Boolean}
B.6. Requirements

or (b : Boolean) : Boolean  \(\text{(11.5.4-a)}\)
True if either self or b is true.

xor (b : Boolean) : Boolean  \(\text{(11.5.4-b)}\)
True if either self or b is true, but not both.

\[\text{post: (self or b) and not (self = b)}\]

and (b : Boolean) : Boolean  \(\text{(11.5.4-c)}\)
True if both self and b are true.

not : Boolean  \(\text{(11.5.4-d)}\)
True if self is false.

\[\text{post: if self then result = false}\]
\[\quad \text{else result = true}\]
\[\quad \text{endif}\]

implies (b : Boolean) : Boolean  \(\text{(11.5.4-e)}\)
True if self is false, or if self is true and b is true.

\[\text{post: (not self) or (self and b)}\]

end

B.6.2. OCL Collection requirements

theory OCL_Collection_requirements
imports
$HOLOCL_HOME/src/OCL_Library
begin
This theory contains the requirements for the collection types defined in the OCL standard library (Set, OrderedSet, Bag, and Sequence) [H] Chapter 11.7.

Collection
Appendix B. Isabelle Theories

size() : Integer  
(11.7.1-a)
The number of elements in the collection self.

**post**: result = self->iterate(elem; acc : Integer = 0 | acc + 1)

Set

**OrderedSet**

**Bag**

**lemma** req_bag_11_7_1-a:

\[ ||(\text{self}:('a,'b::bot)\text{Bag})|| = \text{OclIterate self (λ (\text{elem}:('a,'b::bot)\text{VAL}) (\text{acc}:'a Integer). acc + 1) 0} \]

**by** (simp add: OCL_Bag.size_by_iterate)

**Sequence**

**lemma** req_seq_11_7_1-a:

\[ ||(\text{self}:('a,'b::bot)\text{Sequence})|| = \text{OclIterate self (λ (\text{elem}:('a,'b::bot)\text{VAL}) (\text{acc}:'a Integer). acc + 1) 0} \]

**by** (simp add: OCL_Sequence.size_by_iterate)

includes(object : T) : Boolean  
(11.7.1-b)
True if object is an element of self, false otherwise.

**post**: result = (self->count(object) > 0)

Set

**OrderedSet**

**Bag**

**lemma** req_bag_11_7_1-b:

\[ ((\text{obj}:('a,'b)\text{VAL}) \in (\text{self}:('a,'b::bot)\text{Bag})) = ((\text{self}->\text{count} \text{obj}) > 0) \]

**by** (simp only: OCL_Bag.includes_by_count_UC)

**Sequence**

**lemma** req_seq_11_7_1-b:

\[ ((\text{obj}:('a,'b)\text{VAL}) \in (\text{self}:('a,'b::bot)\text{Sequence})) = ((\text{self}->\text{count} \text{obj}) > 0) \]

**by** (simp only: OCL_Sequence.includes_by_count_UC)

410
excludes(object : T) : Boolean

True if object is not an element of self, false otherwise.

post: result = (self->count(object) = 0)

Set

OrderedSet

Bag

lemma req_bag_11_7_1-c:

((obj:('a,'b)VAL) $\notin$ (self:('a,'b::bot)Bag) ) = ((self->count obj) = 0)

oops

Sequence

lemma req_seq_11_7_1-c:

((obj:('a,'b)VAL) $\notin$ (self:('a,'b::bot)Sequence) ) = ((self->count obj) = 0)

by(simp)
Appendix B. Isabelle Theories

count\( (object : T) : Integer \) \(11.7.1\)-d
\(\) The number of times that object occurs in the collection self.
\[\text{post: result = } \text{self} \to \text{iterate( elem; acc : Integer = 0 | if elem = object then acc + 1 else acc endif)}\]

Set

OrderedSet

Bag

\textit{lemma req\_bag\_11\_7\_1-d:}
\[\begin{align*}
\llbracket \tau \models \partial(\text{obj}::(\text{'}a,\text{'}b)\text{VAL}) \rrbracket & \implies \\
((\text{self}::(\text{'}a,\text{'}b)::\text{bot}\text{Bag}) \to \text{count obj }) \tau & = \\
(\text{self} \to \text{iterate}(x;y=0 | (if (x \neq \text{obj}) \text{ then (y + 1) else yendif) )) \tau & \\
\text{by}(\text{simp add: OCL\_Bag.count\lBy\_iterate OCL\_Bag.OclIterate_def OclIf_def localValid2sem ss_lifting'})
\end{align*}\]

Sequence

\textit{lemma req\_seq\_11\_7\_1-d:}
\[\begin{align*}
\llbracket \tau \models \partial(\text{obj}::(\text{'}a,\text{'}b)\text{VAL}) \rrbracket & \implies \\
((\text{self}::(\text{'}a,\text{'}b)::\text{bot}\text{Sequence}) \to \text{count obj }) \tau & = \\
(\text{self} \to \text{iterate}(x;y=0 | (if (x \neq \text{obj}) \text{ then (y + 1) else yendif) )) \tau & \\
\text{by}(\text{simp add: OCL\_Sequence.count\lBy\_iterate OCL\_Sequence.OclIterate_def OclIf_def localValid2sem ss_lifting'})
\end{align*}\]

412
B.6. Requirements

`includesAll(c2 : Collection(T)) : Boolean`

Does self contain all the elements of c2?

post: result = c2->forall(elem | self->includes(elem))

**Set**

If `self = ⊥` and `(obj = ∅)` then the left-hand side of the equation evaluates to `⊥` whereas the right-hand side evaluates to `T`.

**Wrong definition of the correspondence of includesAll and iterate**

```
lemma req_set_11_7_1-e:
  (((self:(`τ`,α::bot):Set) -> includesAll(obj))
   = (obj ->forall(elem | (self ->includes((elem:(`τ`,α::bot):VAL))))) = False
apply(simp add: OclIncludes_def OclIncludesAll_def OclForAll_def
       lift0_def lift1_def lift2_def strictify_def DEF_def)
```

**oops**

**OrderedSet**

**Bag**

```
lemma req_bag_11_7_1-e:
  [ `τ` |= δ(self:(`a`,`b`):bot):Bag ] ==>
  (OclIncludesAll self obj) `τ` = (∀ elem ∈ obj . ((elem:(`a`,`b`):VAL) ∈ self)) `τ`
```

**oops**

**Sequence**

```
lemma req_seq_11_7_1-e:
  [ `τ` |= δ(self:(`a`,`b`):bot):Sequence ] ==>
  (OclIncludesAll self obj) `τ` = (∀ elem ∈ obj . ((elem:(`a`,`b`):VAL) ∈ self)) `τ`
apply(rule_tac X=self in Sequence_sem_cases)
apply(rule_tac X=obj in Sequence_sem_cases)
apply(simp_all add: localValidDefined2sem DEF_def)
apply(auto simp: OCL_Sequence.OclForAll_def
OCL_Sequence.OclIncludesAll_def OclStrongEq_def
OCL_Sequence.OclIncludes_def ss_lifting LOCALValid2sem)
done
```

413
Appendix B. Isabelle Theories

excludesAll(c2 :: Collection(T)) : Boolean

Does self contain none of the elements of c2?

\[ \text{post: result} = \text{c2} \rightarrow \forall \text{elem} \mid \text{self} \rightarrow \text{excludes(elem)} \]

Set

If self = ⊥ and obj = ∅ then the left-hand side of the equation evaluates to ⊥ whereas the right-hand side evaluates to T.

\[ \text{lemma req_set_11_7_1-f:} \]
\[ ((\text{self}::(\tau,\alpha::\bot)\text{Set}) \rightarrow \text{excludesAll(obj)}) \]
\[ = (\text{obj} \rightarrow \forall \text{elem} \mid (\text{self} \rightarrow \text{excludes((\text{elem}::(\tau,\alpha::\bot)\text{VAL}))})) \]
\[ \text{apply(rule ext)} \]
\[ \text{apply(simp add: OclExcludes_def OclExcludesAll_def OclForAll_def lift0_def lift1_def lift2_def strictify_def DEF_def)} \]
\[ \text{apply(auto)} \]
\[ \text{oops} \]

OrderedSet

Bag

\[ \text{lemma req_bag_11_7_1-f:} \]
\[ [\tau \models \partial (\text{self}::(a,b::\bot)\text{Bag})] \implies \]
\[ \tau \models (\text{OclExcludesAll self obj}) \triangleq (\forall \text{elem} \in \text{obj} \cdot ((\text{elem}::(a,b)\text{VAL}) \not\in \text{self}) \]
\[ \text{oops} \]

Sequence

\[ \text{lemma req_seq_11_7_1-f:} \]
\[ [\tau \models \partial (\text{self}::(a,b::\bot)\text{Sequence})] \implies \]
\[ \tau \models (\text{OclExcludesAll self obj}) \triangleq (\forall \text{elem} \in \text{obj} \cdot ((\text{elem}::(a,b)\text{VAL}) \not\in \text{self}) \]
\[ \text{apply(rule_tac X=self in Sequence_sem_cases)} \]
\[ \text{apply(rule_tac X=obj in Sequence_sem_cases)} \]
\[ \text{apply(simp_all add: localValidDefined2sem DEF_def)} \]
\[ \text{apply(auto simp: OCL_Sequence.OclForAll_def OCL_Sequence.OclExcludesAll_def OclStrongEq_def)} \]
\[ \text{apply(auto simp: OCL_Sequence.OclIncludes_def ss_lifting' localValid2sem OclNot_def)} \]
\[ \text{done} \]
**B.6. Requirements**

**isEmpty() : Boolean**

Is self the empty collection?

| post: | result = ( self->size() = 0 ) |

**Set**

Due to our extension to infinite sets, the usual connection between `isEmpty` and `size` holds only under a precondition

**lemma** REQ11_7_1_7:

\[ \vdash \partial(\text{self} \rightarrow \text{size}()) \Rightarrow \text{self} \rightarrow \text{isEmpty}() = (\text{self}:(\tau,\alpha::\text{bot})\text{Set}) \rightarrow \text{size}() \equiv 0 \]

**apply** (rule ext)

**apply** (drule_tac \( \tau = x \) in valid_elim)

**apply** (subgoal_tac finite (Lifting.drop (Rep_Set_0 (self x))))

**prefer** 2

**apply** (erule finite_Sets)

**apply** (simp add: OclIsEmpty_def OclSize_def OclStrictEq_def OCL_Integer.Zero_ocl_int_def)

**apply** (simp only: lift0_def lift1_def lift2_def strictify_def o_def not_def)

**apply** (simp_all add: OCL_Bag.OclSize_def OclMtBag_def OclStrictEq_def Zero_ocl_int_def ss_lifting)

**done**

**OrderedSet**

**Bag**

**lemma** req_bag_11_7_1-g:

\[ (\cong \emptyset \text{self}) = \| (\text{self}:(\tau,\alpha::\text{bot})\text{Bag}) \| \equiv 0 \]

**apply** (rule Bag_sem_cases_ext, simp_all)

**apply** (simp_all add: OCL_Bag.OclSize_def OclMtBag_def OclStrictEq_def Zero_ocl_int_def ss_lifting)

**done**

**Sequence**

**lemma** req_seq_11_7_1-g:

\[ (\cong \emptyset \text{self}) = \| (\text{self}:(\tau,\alpha::\text{bot})\text{Sequence}) \| \equiv 0 \]

**by** simp

---

415
Appendix B. Isabelle Theories

**notEmpty() : Boolean**

Is self not the empty collection?

**post:** result = ( self->size() <> 0 )

**Set**

- **lemma** `req_set_11_7_1-h`:
  
  OclNotEmpty(X::('τ, 'α::bot) Set) = ¬ (OclIsEmpty(X))

- **apply** (rule `ext`)
  
  **apply** (simp add: OclNotEmpty_def OclIsEmpty_def OclNot_def)
  
  **apply** (simp only: lift1_def lift2_def strictify_def o_def not_def)
  
  **apply** (simp_all (no_asm_use) add: UU_fun_def DEF_def strictify_def)

- **done**

**OrderedSet**

**Bag**

- **lemma** `req_bag_11_7_1-h`:
  
  ( ≠ ∅ self) = (∥(self:('a,'b::bot)Bag))<>'0
  
  **by**(simp add: req_bag_11_7_1-g[symmetric])

**Sequence**

- **lemma** `req_seq_11_7_1-h`:
  
  ( ≠ ∅ self) = (∥(self:('a,'b::bot)Sequence))<>'0
  
  **by**(simp add: req_seq_11_7_1-g[symmetric])

416
sum() : T  \hspace{1cm} (11.7.1-i)

The addition of all elements in self. Elements must be of a type supporting the +
operation. The + operation must take one parameter of type T and be both associa-
tive: \((a+b)+c = a+(b+c)\), and commutative: \(a+b = b+a\). Integer and Real fulfill this
condition.

\textbf{post}: result = self->iterate( elem; acc : T = 0 | acc + elem )

\textbf{Set}

\textbf{OrderedSet}

\textbf{Bag}

\textit{lemma} req_bag_11_7_1-i:

\((\text{->sum()} S) = ((S::\{\text{bot, plus, zero}\})\text{Bag}) \rightarrow \text{iterate}(x;y=0 | (x + y)))\)

\textit{by}(\text{simp add: OCL\_Bag.OclSum\_def})

\textbf{Sequence}

\textit{lemma} req_seq_11_7_1-i:

\((\text{->sum()} S) = ((S::\{\text{a,Integer\_0}\}\text{Sequence}) \rightarrow \text{iterate}(x;y=0 | (x + y)))\)

\textit{by}(\text{simp add: OCL\_Sequence.OclSum\_def})

Note: That this requirement is not fulfilled completely as it currently
holds only for integers. But this will change in the future.
Appendix B. Isabelle Theories

\textbf{product}(c2: Collection(T2)) : Set(Tuple(first: T, second: T2))  \hspace{1cm} (11.7.1-j)

The cartesian product operation of self and c2.

\begin{verbatim}
post: result =
  self->iterate(e1; acc: Set(Tuple(first: T, second: T2)) = Set{} |
  c2->iterate(e2; acc2: Set(Tuple(first: T, second: T2)) = acc |
  acc2->including (Tuple(first = e1, second = e2))) )
\end{verbatim}

Set

OrderedSet

Bag

\textbf{lemma} req_bag_11_7_1-j:
(OclProduct (self:(a,b::bot)Bag) (c2:(a,c::bot)Bag)) =
(self->iterate(e1; acc=OclMtSet |
  (c2->iterate(e2; acc2=acc |
  (acc2->including (OclTuple e1 e2)))))

\textit{oops}

Sequence

\textbf{lemma} req_seq_11_7_1-j:
(OclProduct (self:(a,b::bot)Sequence) (c2:(a,c::bot)Sequence)) =
(self->iterate(e1; acc=OclMtSet |
  (c2->iterate(e2; acc2=acc |
  (acc2->including (OclTuple e1 e2)))))

\textit{oops}

Set

OrderedSet

Bag

= (bag : Bag(T)) : Boolean  \hspace{1cm} (11.7.7-b)

True if self and bag contain the same elements, the same number of times.

\begin{verbatim}
post: result=
  (self->forAll(elem | self->count(elem) = bag->count(elem)) and
  bag->forAll(elem | bag->count(elem) = self->count(elem)))
\end{verbatim}

418
B.6. Requirements

\[ \text{union(bag : Bag(T)) : Bag(T)} \] (11.7.7-b)

The union of self and bag.

\[ \text{post: result -> forall (elem | result -> count(elem) = self -> count(elem) + bag -> count(elem))} \]

\[ \text{post: self -> forall (elem | result -> count(elem) = self -> count(elem) + bag -> count(elem))} \]

\[ \text{post: bag -> forall (elem | result -> count(elem) = self -> count(elem) + bag -> count(elem))} \]

\[ \text{union(set : Set(T)) : Bag(T)} \] (11.7.7-b)

The union of self and set.

\[ \text{post: result -> forall (elem | result -> count(elem) = self -> count(elem) + set -> count(elem))} \]

\[ \text{post: self -> forall (elem | result -> count(elem) = self -> count(elem) + set -> count(elem))} \]

\[ \text{post: set -> forall (elem | result -> count(elem) = self -> count(elem) + set -> count(elem))} \]

\[ \text{intersection(bag : Bag(T)) : Bag(T)} \] (11.7.7-b)

The intersection of self and bag.

\[ \text{post: result -> forall (elem | result -> count(elem) = self -> count(elem). min(bag -> count(elem)))} \]

\[ \text{post: self -> forall (elem | result -> count(elem) = self -> count(elem). min(bag -> count(elem)))} \]

\[ \text{post: bag -> forall (elem | result -> count(elem) = self -> count(elem). min(bag -> count(elem)))} \]

\[ \text{intersection(set : Set(T)) : Set(T)} \] (11.7.7-b)

The intersection of self and set.

\[ \text{post: result -> forall (elem | result -> count(elem) = self -> count(elem). min(set -> count(elem)))} \]

\[ \text{post: self -> forall (elem | result -> count(elem) = self -> count(elem). min(set -> count(elem)))} \]

\[ \text{post: set -> forall (elem | result -> count(elem) = self -> count(elem). min(set -> count(elem)))} \]
including(object : T) : Bag(T)  
The bag containing all elements of self plus object.

\[ \begin{align*} 
\text{post: } & \text{ result -> forall (elem | } \\
& \quad \text{ if elem = object then} \\
& \quad \quad \text{ result -> count (elem) = self -> count (elem) + 1} \\
& \quad \text{ else} \\
& \quad \quad \text{ result -> count (elem) = self -> count (elem)} \\
& \quad \text{ endif) } \\
\text{post: } & \text{ self -> forall (elem | } \\
& \quad \text{ if elem = object then} \\
& \quad \quad \text{ result -> count (elem) = self -> count (elem) + 1} \\
& \quad \text{ else} \\
& \quad \quad \text{ result -> count (elem) = self -> count (elem)} \\
& \quad \text{ endif) } 
\end{align*} \]

excluding(object : T) : Bag(T)  
The bag containing all elements of self apart from all occurrences of object.

\[ \begin{align*} 
\text{post: } & \text{ result -> forall (elem | } \\
& \quad \text{ if elem = object then} \\
& \quad \quad \text{ result -> count (elem) = 0} \\
& \quad \text{ else} \\
& \quad \quad \text{ result -> count (elem) = self -> count (elem)} \\
& \quad \text{ endif) } \\
\text{post: } & \text{ self -> forall (elem | } \\
& \quad \text{ if elem = object then} \\
& \quad \quad \text{ result -> count (elem) = 0} \\
& \quad \text{ else} \\
& \quad \quad \text{ result -> count (elem) = self -> count (elem)} \\
& \quad \text{ endif) } 
\end{align*} \]

count(object : T) : Integer  
The number of occurrences of object in self.
### flatten() : Bag(T2)

If the element type is not a collection type this result in the same bag. If the element type is a collection type, the r is the bag containing all the elements of all the elements of self.

**post:**

\[
\text{result } = \begin{cases} 
\text{if } \text{self.type.elementType.oclIsKindOf(CollectionType)} \\
\text{then} \\
\text{self} \rightarrow \text{iterate}(c; \text{acc : Bag() } = \text{Bag()} \mid \\
\text{acc} \rightarrow \text{union}(c \rightarrow \text{asBag()} ) ) \\
\text{else} \\
\text{self} \\
\text{endif}
\end{cases}
\]

### asBag() : Bag(T)

A Bag identical to self. This operation exists for convenience reasons.

**post:** result = self

### asSequence() : Sequence(T)

A Sequence that contains all the elements from self, in undefined order.

**post:**

\[
\text{result} \rightarrow \text{forAll} \ (\text{elem} \mid \text{self} \rightarrow \text{count}(\text{elem}) = \text{result} \rightarrow \text{count}(\text{elem}))
\]

**post:**

\[
\text{self} \rightarrow \text{forAll} \ (\text{elem} \mid \text{self} \rightarrow \text{count}(\text{elem}) = \text{result} \rightarrow \text{count}(\text{elem}))
\]

### asSet() : Set(T)

The Set containing all the elements from self, with duplicates removed.

**post:**

\[
\text{result} \rightarrow \text{forAll} \ (\text{elem} \mid \text{self} \rightarrow \text{includes}(\text{elem}))
\]

**post:**

\[
\text{self} \rightarrow \text{forAll} \ (\text{elem} \mid \text{result} \rightarrow \text{includes}(\text{elem}))
\]

### asOrderedSet() : OrderedSet(T)

An OrderedSet that contains all the elements from self, in undefined order, with duplicates removed.

**post:**

\[
\text{result} \rightarrow \text{forAll} \ (\text{elem} \mid \text{self} \rightarrow \text{includes}(\text{elem}))
\]

**post:**

\[
\text{self} \rightarrow \text{forAll} \ (\text{elem} \mid \text{result} \rightarrow \text{includes}(\text{elem}))
\]

**post:**

\[
\text{self} \rightarrow \text{forAll} \ (\text{elem} \mid \text{result} \rightarrow \text{count}(\text{elem}) = 1)
\]

![Image](https://via.placeholder.com/150)
Appendix B. Isabelle Theories

Sequence

\[ \text{count(object : } T) : \text{ Integer} \quad (11.7.5-a) \]
The number of occurrences of object in self.

\[ \equiv (s : \text{ Sequence}(T)) : \text{ Boolean} \quad (11.7.5-b) \]
True if self contains the same elements as s in the same order.

**post:**
\[
\text{result} = \text{Sequence}\{1..\text{self} \rightarrow \text{size}()\} \rightarrow \forall \text{index} : \text{ Integer} | \\
\text{self} \rightarrow \text{at}(\text{index}) = s \rightarrow \text{at}(\text{index}) \\
\text{and} \\
\text{self} \rightarrow \text{size}() = s \rightarrow \text{size}()
\]

**lemma req_11_7_5−b:**
\[ ((\forall i \in ((\text{OclCollectionRange 1 } || \text{self})::\text{\texttt{\char133}}\text{a,Integer_0})\text{Sequence}) \cdot ((\text{OclAt self S1 i}) \equiv (\text{OclAt S2 i})))) \land \\
((||\text{S1}|| \equiv ||\text{S2}||)) \]

union \( (s : \text{ Sequence}(T)) : \text{ Sequence}(T) \)  \( (11.7.5-c) \)
The sequence consisting of all elements in self, followed by all elements in s.

**post:**
\[ \text{result} \rightarrow \text{size}() = \text{self} \rightarrow \text{size}() + s \rightarrow \text{size}() \]

**post:**
\[ \text{Sequence}\{1..\text{self} \rightarrow \text{size}()\} \rightarrow \forall \text{index} : \text{ Integer} | \\
\text{self} \rightarrow \text{at}(\text{index}) = \text{result} \rightarrow \text{at}(\text{index}) \\
\text{post:**
\[ \text{Sequence}\{1..s \rightarrow \text{size}()\} \rightarrow \forall \text{index} : \text{ Integer} | \\
\text{result} \rightarrow \text{at}(\text{index}) = \text{self} \rightarrow \text{at}(\text{index}) + ||\text{self}||) \\
\text{oops}

**lemma req_11_7_5−c−1:**
\[ ((\text{self}::\text{\texttt{\char133}}\text{a,b::bot})\text{Sequence}) \equiv \text{S2}) = \\
(\forall i \in ((\text{OclCollectionRange 1 } || \text{self})::\text{\texttt{\char133}}\text{a,Integer_0})\text{Sequence}) \cdot ((\text{OclAt self S1 i}) \equiv (\text{OclAt S2 i})))) \land \\
((||\text{S1}|| \equiv ||\text{S2}||)) \]

**lemma req_11_7_5−c−2:**
\[ ((\text{self}::\text{\texttt{\char133}}\text{a,b::bot})\text{Sequence}) ] \equiv \\
\forall i \in ((\text{OclCollectionRange 1 } || \text{self})::\text{\texttt{\char133}}\text{a,Integer_0})\text{Sequence}) \cdot \\
(\text{OclAt (self } \cup s \text{ ) index} \equiv (\text{OclAt self index})) \]

apply(rule OCL_Sequence.ForAll)

apply(simp add: OCL_Sequence.OCL_is_defopt_OclCollectionRange)

apply(simp add: OCL_Sequence.includes_of_collectionRange)

**lemma req_11_7_5−c−3:**
\[ \exists \forall i \in ((\text{OclCollectionRange 1 } || \text{self})::\text{\texttt{\char133}}\text{a,Integer_0})\text{Sequence}) \cdot \\
(\text{OclAt s index}) \equiv (\text{OclAt (self } \cup s \text{ ) (index + ||\text{self}||)}) \\
\text{oops}

422
flatten() : Sequence(T2)  \[11.7.5-d\]

If the element type is not a collection type this result in the same self. If the element type is a collection type, the result is the sequence containing all the elements of all the elements of self. The order of the elements is partial.

**post:**

\[
\text{result} = \begin{cases} 
\text{if self.type.elementType.oclIsKindOf(CollectionType)} \text{ then} \\
\text{self->iterate(c; acc : Sequence() = Sequence{} | acc->union(c->asSequence() ) )} \\
\text{else} \\
\text{self} \\
\text{endif}
\end{cases}
\]

The postcondition in the standard is not typeable because it must have these two types at the same time which is not possible:

\[
(('a,'b::bot Sequence_0)Sequence) \Rightarrow ('a,'b)Sequence \\
('a,'b::bot)Sequence \Rightarrow ('a,'b)Sequence
\]

therefore we split the requirement into two parts:

1. characterises the case of a flatten of a sequence of sequences
2. characterises the case of a flatten of a sequence of a not collection type. This follows by definition of flatten on sequences of non collection types

**lemma** req_11_7_5_d-1:

\[
\text{((self::('a,'b::bot Sequence_0)Sequence)\::\::('a,'b)Sequence) =}
\]

\[
\text{(self->iterate(x:y=[] | (y U x )))}
\]

by simp add: OCL_Sequencen.flatten_by_iterate

---

423
Appendix B. Isabelle Theories

append (object:: T) : Sequence(T)

The sequence of elements, consisting of all elements of self, followed by object.

| post: result -> size() = self -> size() + 1 |
| post: result -> at(result -> size()) = object |
| post: Sequence(1..self -> size()) -> forAll(index : Integer | result -> at(index) = self -> at(index)) |

lemma req_11_7_5-e-1:
τ ⊨ (∂ (obj::('a, 'b::bot) VAL)) ⇒ (∥OclAppend (self::('a, 'b) Sequence) obj∥ τ) = (∥self∥ + 1) τ
by(simp)

lemma req_11_7_5-e-2:
τ ⊨ (∂ (self::('a, 'b::bot) Sequence)) ⇒ (OclAt (OclAppend self (obj::('a, 'b) VAL)) ∥OclAppend self obj∥ τ) = (obj τ)
apply(case_tac τ ⊭ ∂ obj)
apply(ocl_hypsubst, simp_all)
apply(rule OCL_Sequence.at_last_of_including)
apply(rule subst, rule sym)
apply(rule_tac A=∥self ->including obj ∥ in cp_charn)
apply(rule OCL_Sequence.size_of_including)
apply(simp_all)
oops

lemma req_11_7_5-e-3:
∀ index ∈ ((OclCollectionRange 1 ∥ (self::('a, 'b::bot) Sequence))) : ('a, Integer _0) Sequence
· (OclAt self index) = (OclAt (OclAppend self (obj::('a, 'b) VAL) ) index))
oops
B.6. Requirements

**prepend**(*object : T*) : **Sequence**(*T*)

The sequence consisting of object, followed by all elements in self.

post: result->size = self->size() + 1

post: result->at(1) = object

post: **Sequence**{1..self->size()}->forAll(index : Integer |
                  self->at(index) = result->at(index + 1))

**lemma** req_11_7_5−f−1:
\[ \tau \models (\partial (\text{obj}::(\text{a}, \text{b}::\text{bot})\text{VAL})) \implies (\parallel \text{OclPrepend (self::(\text{a}, \text{b})\text{Sequence}) \text{obj}}\parallel \tau) = (\parallel \text{self} \parallel + 1) \tau \]

apply(simp)

apply(rule trans)

apply(rule_tac A=\parallel ?S->including \text{obj}} \parallel \text{in cp_charn})

apply(rule OCL_Sequence.size_of_including)

apply(simp_all add: localValidDefined2sem)

apply(simp add: localValid2sem OCL_Sequence.OclSize_def OclStrictEq_def One_ocl_int_def Zero_ocl_int_def plus_def ss_lifting)

oops

**lemma** req_11_7_5−f−2:
\[ \tau \models (\partial (\text{self}::(\text{a}, \text{b}::\text{bot})\text{Sequence})) \implies (\text{OclAt (OclPrepend self (obj::(\text{a}, \text{b})\text{VAL})) 1} \tau) = (\text{obj} \tau) \]

by(simp only: at_first_of_prepend)

**lemma** req_11_7_5−f−3:
\[ \forall \text{index} \in ((\text{OclCollectionRange 1 ((self::(\text{a}, \text{b}::\text{bot})\text{Sequence})::(\text{a},\text{Integer}_0)\text{Sequence})})::(\text{a},\text{Integer}_0)\text{Sequence}) \cdot 
\]

\[ (\text{OclAt self index} \models (\text{OclAt (OclPrepend self (obj::(\text{a}, \text{b})\text{VAL})}) (\text{index}+1))) \]

oops

425
Appendix B. Isabelle Theories

\textbf{insertAt}(index : Integer, object : T) : \textit{Sequence}(T) \quad (11.7.5-g)

The sequence consisting of self with object inserted at position index.

\begin{verbatim}
post: result->size = self->size() + 1
post: result->at(index) = object
post: Sequence{1..(index - 1)}->forAll(i : Integer | self->at(i) = result->at(i))
post: Sequence{(index + 1)..self->size())->forAll(i : Integer | self->at(i) = result->at(i + 1))
\end{verbatim}

\textbf{lemma} \textit{req_11_7_5-g-1}:
\[ \models \tau \models \partial \langle \text{obj>::\langle \text{a}, \text{b}::\text{bot}\rangle\text{VAL} \rangle; \tau \models 1 \leq \text{index}; \tau \models \text{index} \leq (\|\text{self}\| + 1) \implies \langle \text{OclInsertAt self index obj} \rangle \tau = (\langle (\text{self}::\langle \text{a}, \text{b}::\text{bot}\rangle\text{Sequence}) \rangle + 1) \tau \]
\textit{by}(simp add: size_of_insertAt)

\textbf{lemma} \textit{req_11_7_5-g-2}:
\[ \models \tau \models 1 \leq \text{index}; \tau \models \text{index} \leq (\|\text{self}\| + 1) \implies \langle \text{OclAt (OclInsertAt (self::\langle \text{a}, \text{b}::\text{bot}\rangle\text{Sequence}) index obj)} \rangle \tau = (\langle \text{obj}::\langle \text{a}, \text{b}::\text{bot}\rangle\text{VAL} \rangle) \tau \]
\textit{by}(simp add: at_of_insertAt)

\textbf{lemma} \textit{req_11_7_5-g-3}:
\[ \models \tau \models \partial \langle \text{obj} \rangle; \tau \models 1 \leq i; \tau \models \text{index} \leq (\|\langle \text{self}::\langle \text{a}, \text{b}::\text{bot}\rangle\text{Sequence}\rangle \| + 1) \implies \tau \models \forall i \in (\langle \text{OclCollectionRange 1 (index - 1)} \rangle::\langle \text{a}, \text{Integer}_0\rangle\text{Sequence}) \cdot \langle \text{OclAt self i} \rangle = (\langle \text{OclAt (OclInsertAt self index (obj::\langle \text{a}, \text{b}::\text{bot}\rangle\text{VAL}) i)} \rangle \tau) \]
\textit{oops}

\textbf{lemma} \textit{req_11_7_5-g-4}:
\[ \models \tau \models \partial \langle \text{obj} \rangle; \tau \models 1 \leq i; \tau \models \text{index} \leq (\|\langle \text{self}::\langle \text{a}, \text{b}::\text{bot}\rangle\text{Sequence}\rangle \| + 1) \implies \tau \models \forall i \in (\langle \text{OclCollectionRange index}\rangle::\langle \text{a}, \text{Integer}_0\rangle\text{Sequence}) \cdot \langle \text{OclAt self i} \rangle = (\langle \text{OclAt (OclInsertAt self index (obj::\langle \text{a}, \text{b}::\text{bot}\rangle\text{VAL} \rangle (i+1))} \rangle) \]
\textit{oops}

\textbf{oops}
### B.6. Requirements

**subSequence(lower : Integer, upper : Integer) : Sequence(T)**  
(11.7.5-h)

The sub-sequence of self starting at number lower, up to and including element number upper.

<table>
<thead>
<tr>
<th>pre</th>
<th>1 &lt;= lower</th>
</tr>
</thead>
<tbody>
<tr>
<td>pre</td>
<td>lower &lt;= upper</td>
</tr>
<tr>
<td>pre</td>
<td>upper &lt;= self-&gt;size()</td>
</tr>
<tr>
<td>post</td>
<td>result-&gt;size() = upper - lower + 1</td>
</tr>
<tr>
<td>post</td>
<td>Sequence{ lower .. upper }-&gt;forAll( index</td>
</tr>
</tbody>
</table>

**lemma** req_11_7_5_h-2:

\[
\begin{align*}
\forall \tau : (1 \leq \text{lower}; \tau \models \text{lower} \leq \text{upper} - 1; \tau \models \text{upper} \leq \|\text{self}\|) \implies \\
\|\text{OclSubSequence} (\text{self} : (a,b::\text{bot}Sequence) \text{lower} \text{upper})\| \tau = (\text{upper} - \text{lower} + 1)
\end{align*}
\]

by (simp add: size_of_subsequence)

**lemma** req_11_7_5_h-2:

\[
\begin{align*}
\forall \tau : (1 \leq \text{lower}; \tau \models \text{lower} \leq \text{upper} - 1; \tau \models \text{upper} \leq \|\text{self}\|) \implies \\
\tau \models \forall \text{index} \in ((\text{OclCollectionRange lower upper}) : (a,\text{Integer}_0)\text{Sequence}) . \\
(\text{OclAt} (\text{OclSubSequence} \text{self lower upper}) (\text{index} - \text{lower} + 1) = \\
\text{OclAt} (\text{self} : (a,b::\text{bot}Sequence) \text{index})
\end{align*}
\]

**oops**

**at(i : Integer) : T**  
(11.7.5-i)

The i-th element of sequence.

| pre | i >= 1 and i <= self->size() |

**lemma** req_11_7_5_i:

\[
\begin{align*}
\tau \models 1 \leq i; \tau \models i \leq \|S\| \implies \\
(\text{OclAt} (S : (a,b::\text{bot}Sequence) i) : (a,b)\text{VAL})
\end{align*}
\]

apply (rotate_tac 1, frule isDefined_if_valid)

apply (simp only: OCL_is_defopt_OclLe OCL_Sequence.OCL_is_defopt_OclSize OCL_is_defopt_OclAt)

**oops**

427
Appendix B. Isabelle Theories

indexOf(obj : T) : Integer
(11.7.5-j)
The index of object obj in the sequence.

<table>
<thead>
<tr>
<th>pre</th>
<th>self-&gt;includes(obj)</th>
</tr>
</thead>
<tbody>
<tr>
<td>post</td>
<td>self-&gt;at(i) = obj</td>
</tr>
</tbody>
</table>

**lemma** req_11_7_5-j:
\[ \tau \models (\text{obj} :: (\tau a, \tau b) \text{VAL}) \in (\text{self} :: (\tau a, \tau b :: \text{bot}) \text{Sequence}) \implies \text{OclAt self (OclIndexOf self obj)} \tau = \text{obj} \tau \\
by (simp add: at_indexOf) |

first() : T
(11.7.5-k)
The first element in self.

| post | result = self->at(1) |

**lemma** req_11_7_5-k:
\[ \text{OclFirst (self :: (\tau a, \tau b :: \text{bot}) \text{Sequence}) = ((\text{OclAt self 1}) :: (\tau a, \tau b) \text{VAL})} \\
by (rule first_at_UC) |

last() : T
(11.7.5-l)
The last element in self.

| post | result = self->at(self->size() ) |

**lemma** req_11_7_5-l:
\[ \text{OclLast (self :: (\tau a, \tau b :: \text{bot}) \text{Sequence}) = ((\text{OclAt self \text{||self\text{||}}}) :: (\tau a, \tau b) \text{VAL})} \\
by (rule last_at_UC) |

including(object : T) : Sequence(T)
(11.7.5-m)
The sequence containing all elements of self plus object added as the last element.

| post | result = self . append (object) |

**lemma** req_11_7_5-m:
\[ \text{self ->including (obj :: (\tau a, \tau b) \text{VAL}) = OclAppend (self :: (\tau a, \tau b :: \text{bot}) \text{Sequence}) obj} \\
by (rule append_including_UC[symmetric]) |
excluding(object : T) : Sequence(T)  

The sequence containing all elements of self apart from all occurrences of object. The order of the remaining elements is not changed.

\[
\text{post: result -> includes(object) = false}
\]

\[
\text{post: result -> size() = self -> size() - self -> count(object)}
\]

\[
\text{post: result = self -> iterate(elem; acc : Sequence(T) = Sequence{} | if elem = object then acc else acc -> append(elem) endif)}
\]

\[\text{lemma req_11_7_5-n-1:}\]

\[
[\tau \models \partial(\text{obj:('a,'b)VAL}); \tau \models \partial(\text{self:('a,'b:bot)Sequence}) ] \implies
\]

\[
(\text{obj} \in (\text{self ->excluding obj})) \tau = \neg \top
\]

\[\text{by(simp add: OCL_Sequence.includes_of_excluding)}\]

\[\text{lemma req_11_7_5-n-2:}\]

\[
[(\text{self:('a,'b:bot)Sequence}) \rightarrow \text{excluding (obj:('a,'b)VAL)}] =
\]

\[
(\text{self} - \text{OclCount self obj}
\]

\[\text{by(simp add: OCL_Sequence.size_of_excluding)}\]

\[\text{lemma req_11_7_5-n-3:}\]

\[
[\tau \models \partial(\text{obj:('a,'b)VAL}) ] \implies
\]

\[
((\text{self ->excluding (obj:('a,'b)VAL)}) \tau =
\]

\[
(\text{self ->iterate(elem; acc=[] | (if (elem = obj) then acc else (OclAppend acc elem endif)))}) \tau
\]

\[\text{by(simp add: OCL_Sequence.excluding_by_iterate localValid2sem}
\]

\[\text{OCL_Sequence.OclIterate_def OclIf_def ss_lifting)}\]

\[\text{asBag() : Bag(T)}\]

The Bag containing all the elements from self, including duplicates.

\[
\text{post: result -> forAll(elem | self -> count(elem)=result -> count(elem))}
\]

\[
\text{post: self -> forAll(elem | self -> count(elem)=result -> count(elem))}
\]

\[\text{lemma req_11_7_5-o-1:}\]

\[
\exists \forall \text{elem} \in ((\rightarrow \text{asBag}()) \text{self:('a,'b:bot)Sequence}):(\rightarrow \text{asBag}()) \text{self:('a,'b:bot)Bag} .
\]

\[
(\text{self ->count elem} \models (((\rightarrow \text{asBag}()) \text{self:('a,'b:bot)Bag} \rightarrow \text{count elem}))
\]

\[\text{oops}\]

\[\text{lemma req_11_7_5-o-2:}\]

\[
\exists \forall \text{elem} \in (\text{self:('a,'b:bot)Sequence}) .
\]

\[
(\text{self ->count elem} \models (((\rightarrow \text{asBag}()) \text{self:('a,'b:bot)Bag} \rightarrow \text{count elem}))
\]

\[\text{oops}\]

429
Appendix B. Isabelle Theories

asSequence() : Sequence(T)  (11.7.5-p)
The Sequence identical to the object itself. This operation exists for convenience reasons.

post: result = self

lemma req_11_7_5-p:
(\mapsto asSequence() (self::(\alpha,\beta::bot)Sequence))::(\alpha,\beta::bot)Sequence = self
by(simp)

asSet() : Set(T)  (11.7.5-q)
The Set containing all the elements from self, with duplicated removed.

post: result ->forall (elem | self ->includes (elem))
post: self ->forall (elem | result ->includes (elem))

lemma req_11_7_5-q-1:
\equiv \forall elem \in (\mapsto asSet() (self::(\alpha,\beta::bot)Sequence))::(\alpha,\beta Set_0)VAL .
((elem::(\alpha,\beta)VAL) \in self)

oops

lemma req_11_7_5-q-2:
\equiv \forall elem \in (self::(\alpha,\beta::bot)Sequence) . ((elem::(\alpha,\beta)VAL) \in (\mapsto asSet() self)::(\alpha,\beta Set_0)VAL))

oops

asOrderedSet() : OrderedSet(T)  (11.7.5-r)
An OrderedSet that contains all the elements from self, in the same order, with duplicates removed.

post: result ->forall (elem | self ->includes (elem))
post: self ->forall (elem | result ->includes (elem))
post: self ->forall (elem1, elem2 | result ->count (elem) = 1)
post: self ->forall (elem1, elem2 |
self ->indexOf (elem1) < self ->indexOf (elem2)
implies result ->indexOf (elem1) < result ->indexOf (elem2) )

lemma req_11_7_5-r-1:
\equiv \forall elem \in (\mapsto asOrderedSet() (self::(\alpha,\beta::bot)Sequence))::(\alpha,\beta OrderedSet_0)VAL .
((elem::(\alpha,\beta)VAL) \in self)

oops

lemma req_11_7_5-r-2:
\equiv \forall elem \in (self::(\alpha,\beta::bot)Sequence) . ((elem::(\alpha,\beta)VAL) \in (\mapsto asOrderedSet() self)::(\alpha,\beta::bot OrderedSet_0)VAL))

oops

end
B.6.3. **OCL Iterator requirements**

theory OCL_Iterator_requirements
imports $HOLOCL_HOME/src/OCL_Library
begin

This theory contains the requirements for iterators defined in the OCL standard library [41, Chapter 11.9].

### Collection

**exists**

Results in true if body evaluates to true for at least one element in the source collection.

\[
\text{source} \rightarrow \exists (\text{iterators} \mid \text{body}) = \\
\text{source} \rightarrow \text{iterate (iterators; result : Boolean = false} \\
\mid \text{result or body})
\]

### Set

**OrderedSet**

**Bag**

**lemma** req_bag_11_9_1-a:

\[
(\exists x \in (\text{source}::(\text{a}, \text{b}:\text{bot})\text{Bag}) \cdot (P::(\text{a}, \text{b}:\text{bot})\text{VAL} \Rightarrow \text{a Boolean}) x) = \\
(\text{source} \rightarrow \text{iterate}(x;y=F \mid (P x) \lor y))
\]

by (rule OCL_Bag.exists_by_iterate)

### Sequence

**lemma** req_seq_11_9_1-a:

\[
(\exists x \in (\text{source}::(\text{a}, \text{b}:\text{bot})\text{Sequence}) \cdot (P::(\text{a}, \text{b}:\text{bot})\text{VAL} \Rightarrow \text{a Boolean}) x) = \\
(\text{source} \rightarrow \text{iterate}(x;y=F \mid (P x) \lor y))
\]

by (rule OCL_Sequence.exists_by_iterate)
### Appendix B. Isabelle Theories

**forAll**

Results in true if the body expression evaluates to true for each element in the source collection; otherwise, result is false.

\[
\text{source -> forAll (iterators | body) =}
\]

\[
\text{source -> iterate (iterators; result : Boolean = true}
\]

\[
\quad | \text{result and body)}
\]

**Set**

**OrderedSet**

**Bag**

*lemma req_bag_11_9_1-b:*

\[
(\forall x \in (\text{source}::(\text{a}, \text{b};\text{bot})\text{Bag}) \cdot (\text{P}::(\text{a}, \text{b};\text{bot})\text{VAL} \Rightarrow \text{a Boolean}) x) =
\]

\[
(\text{source -> iterate (xy = T | (P x) \land y))}
\]

*by (rule OCL_Bag forall_by_iterate)*

**Sequence**

*lemma req_seq_11_9_1-b:*

\[
(\forall x \in (\text{source}::(\text{a}, \text{b};\text{bot})\text{Sequence}) \cdot (\text{P}::(\text{a}, \text{b};\text{bot})\text{VAL} \Rightarrow \text{a Boolean}) x) =
\]

\[
(\text{source -> iterate (x;y = T | (P x) \land y))}
\]

*by (rule OCL_Sequence forall_by_iterate)*

**isUnique**

Results in true if body evaluates to a different value for each element in the source collection; otherwise, result is false.

\[
\text{source -> isUnique (iterators | body) =}
\]

\[
\text{source -> collect (iterators}
\]

\[
\quad \rightarrow \text{forAll (x, y | (x.iter <> y.iter)
}\]

\[
\quad \quad \quad \text{implies (x.value <> y.value))}
\]

isUnique may have at most one iterator variable.

**Set**

**OrderedSet**

**Bag**

**Sequence**
B.6. Requirements

any

(11.9.1-d)

Returns any element in the source collection for which body evaluates to true. If there is more than one element for which body is true, one of them is returned. There must be at least one element fulfilling body, otherwise the result of this IteratorExp is OclUndefined.

\[
\text{source} \rightarrow \text{any (iterator | body)} = \text{source} \rightarrow \text{select (iterator | body)} \rightarrow \text{asSequence ()} \rightarrow \text{first ()}
\]

any may have at most one iterator variable.

Set

OrderedSet

Bag

\text{lemma req\_bag\_11\_9\_1=d}:

\begin{align*}
\text{OCL\_Collection.OclAny (Source::('a,'b::bot)Bag) (P::('a,'b::bot)VAL \Rightarrow 'a Boolean)} &= \\
\text{OclFirst ((}\text{asSequence()} (\{x : source | (P x)\}):(\text{'a,'b::bot}Sequence))}
\end{align*}

oops

Sequence

\text{lemma req\_seq\_11\_9\_1=d}:

\begin{align*}
\text{OCL\_Collection.OclAny (source::('a,'b::bot)Sequence) (P::('a,'b::bot)VAL \Rightarrow 'a Boolean)} &= \\
\text{OclFirst ((}\text{asSequence()} (\{x : source | (P x)\}):(\text{'a,'b::bot}Sequence))}
\end{align*}

\text{by (rule OCL\_Sequence.any\_by\_first\_of\_select)}
Appendix B. Isabelle Theories

one  (11.9.1-e)

Results in true if there is exactly one element in the source collection for which body is true.

| source -> one(iterator | body) = |
| source -> select(iterator | body)->size() = 1 |

one may have at most one iterator variable.

Set

OrderedSet

Bag

lemma req_bag_11_9_1-e:
(oclone (source::(a,b::bot)Bag) (p::(a,b::bot)val ⇒ a Boolean)) =
(|{|x : source | (p x)|}| = 1)
by (simp add: OCL_Bag.OclOne_def)

Sequence

lemma req_seq_11_9_1-e:
(oclone (source::(a,b::bot)Sequence) (p::(a,b::bot)val ⇒ a Boolean)) =
(|{|x : source | (p x)|}| = 1)
by (rule OCL_Sequence.one_by_size_of_select)
The Collection of elements which results from applying body to every member of the source set. The result is flattened. Notice that this is based on collectNested, which can be of different type depending on the type of source. collectNested is defined individually for each subclass of CollectionType.

\[
\text{source} \rightarrow \text{collect (iterators | body)} = \text{source} \rightarrow \text{collectNested (iterators | body)} \rightarrow \text{flatten()}
\]

collect may have at most one iterator variable.

Set

OrderedSet

Bag

\[
\text{lemma req_bag_11_7_4-f:}\quad \text{OclCollect S P} \equiv \llceil (\text{OclCollectNested S P}) \rrceil
\]

Bag

Sequence

\[
\text{lemma req_seq_11_9_1-f:}\quad \text{OclCollect S P} \equiv \llceil (\text{OclCollectNested S P}) \rrceil
\]

Set

Bag

Sequence

\[
\text{select(expression : OclExpression) : Sequence(T)}
\]

The subsequence of the source sequence for which body is true.

\[
\text{source} \rightarrow \text{select (iterator | body) =}
\]

\[
\text{source} \rightarrow \text{iterate (iterator; result : Sequence(T)=Sequence{}} | \begin{align*}
\text{if body then result} & \rightarrow \text{including (iterator)} \\
\text{else result} & \end{align*}
\]

select may have at most one iterator variable. \textit{lemma req_11_9_4-a:}

\[
(\forall x : (\text{source}:(a,b :: \text{bot})\text{Sequence}) \mid ((P:((a,b :: \text{bot})\text{VAL} \Rightarrow a \text{ Boolean}) x)) = (\text{source} \rightarrow \text{iterate} (\text{iterator}; \text{result} = [] | \\
\text{if (P \text{ iterator}) then} (\text{OclIncluding result iterator) else result \ text{ endif})}
\]

\textit{by} (rule OCL.Sequence.select_by_iterate)
Appendix B. Isabelle Theories

**reject**

The subsequence of the source sequence for which body is false.

\[
\text{source -> reject(iterator | body)} = \\
\text{source -> select(iterator | not body)}
\]

reject may have at most one iterator variable. **lemma** req_11_9_4-b:

\[\text{OclReject source P = } (\{x : (source::(\'a,\'b::bot)Sequence) | (\neg (P::(\'a,\'b::bot)VAL \Rightarrow \'a Boolean)) x\})\]

\[\text{by (rule OCL_Sequence.reject_by_select)}\]

**collectNested**

The Sequence of elements which results from applying body to every member of the source sequence.

\[
\text{source -> collect(iterators | body)} = \\
\text{source -> iterate(iterators; result : Sequence(body.type)} \\
\text{= Sequence() | result -> append(body) )}
\]

collectNested may have at most one iterator variable. **lemma** req_11_9_4-c:

\[\text{OclCollectNested (source::(\'a,\'b::bot)Sequence) (P::(\'a,\'b)VAL \Rightarrow (\'a,\'c)VAL))::(\'a,\'c::bot)Sequence} = \]

\[\text{(source -> iterate(iterators; result=[] | result -> including (P iterators))} \]

\[\text{by (rule OCL_Sequence.collectNested_by_iterate)}\]

**sortedBy**

Results in the Sequence containing all elements of the source collection. The element for which body has the lowest value comes first, and so on. The type of the body expression must have the < operation defined. The < operation must return a Boolean value and must be transitive i.e. if a < b and b < c then a < c.

\[
\text{source -> sortedBy(iterator | body)} = \\
\text{iterate( iterator ; result : Sequence(T) : Sequence {} |} \\
\text{if result -> isEmpty() then} \\
\text{result.append(iterator) else} \\
\text{let position : Integer = result -> indexOf (result -> select (item | body (item) > body (iterator)) -> first() ) } \\
\text{in} \\
\text{result.insertAt(position, iterator) endif}
\]

sortedBy may have at most one iterator variable.

end
B.7. **OCL**

B.7.1. The Object Constraint Language: **OCL**

theory OCL
imports
$HOLOCL_HOME/src/UML_OCL
$HOLOCL_HOME/src/library/basic/OCL_Basic_requirements
$HOLOCL_HOME/src/library/collection/$COLLECTION/OCL_Collection_requirements
$HOLOCL_HOME/src/library/collection/$COLLECTION/OCL_Iterator_requirements
begin end
Appendix B. Isabelle Theories
Appendix C.

Encoding UML/OCL by Example

C.1. Encoding UML/OCL into HOL by Example

C.2. A Simple Example

In this section we present the details of our encoding by an tiny example. We use the class diagram presented in Figure C.1 together with the OCL specification shown in Listing C.1. Our Example consists only out of two classes with an inheritance relationship. Class A has one attribute $i$ of type Integer that is constraint by an invariant. Further, class A has a method $f(x:\text{Integer}):\text{Integer}$ which is described by a pre-condition and post-condition. Class B only has one attribute $j$ of type B. Note that this is, in contrast to the attribute $i$ of class A, an object type. In the following we describe which definition and theorems are generated during the import of this simple class diagram into HOL-OCL.

C.3. Importing UML/OCL models

UML/OCL models can be directly imported into HOL-OCL if they are available in XMI format, e.g.,

```
load_xmi encoding_example_ocl.xmi
```

```haskell
package encoding
  context A::i:Integer
    init: 42
  context A
    inv: i >= 0
  context A::f(x:Integer):Integer
    pre: x > 0
    post: result = x.div(i)
endpackage
```

Listing C.1: A simple Encoding Example: OCL specification
Appendix C. Encoding \texttt{UML/OCL} by Example

**Figure C.1.:** A simple Encoding Example: Data Model

imports the \texttt{UML/OCL} specification of our example. During the import, different definitions and theorems are generated into the current theory context. HOL-OCL organizes these automatically generated stuff into a hierarchy of namespaces, i.e., in our example:

- \texttt{UML\_OCL}: This is the standard namespace for internal definitions, i.e., definitions that normally should not be needed by the users. This namespace is divided as follows:
  - \texttt{univ.model}: contains the basic type definitions of classes and the type of the universe
  - \texttt{level0.model}: contains the level 0 definitions of the model.
  - \texttt{level1.model}: contains the level 1 definitions of the model.

- \texttt{model}: It contains all definitions a user would expect, namely all the level 2 stuff which ties the \texttt{UML} and \texttt{OCL} part together.

In all namespaces, “\texttt{model}” is replaced by the top-level \texttt{UML} package of the encoded model, i.e., for our example “\texttt{encoding}”. Thus, in our example all level 0 definitions are defined within “\texttt{UML\_OCL.level0.encoding}”.

**C.3.1. Encoding Level 0:**

**The Universe Type**

The first major step of encoding classes into the base \texttt{OCL} universe is to construct the universe type, but first we have to construct the core types for classes. For each class
we construct a “identifier” type which is crucial for our strong type discipline:

\[ \text{UML}_\text{OCL.univ.encoding.A} \_\text{key} = A \_\text{key} \]

the base type of the class, describing its locally defined attributes

\[ \text{UML}_\text{OCL.univ.encoding.A} \_\text{base} \_\text{type} = A \_\text{key} \times \text{Integer}\_0 \]

the type of the class, describing the inherited type together with the base type

\[ \text{UML}_\text{OCL.univ.encoding.A} \_\text{type} = (A \_\text{key} \times \text{Integer}\_0) \times \text{\'a up} \]

These types are analogous defined for the inherited class B:

\[ \text{UML}_\text{OCL.univ.encoding.B} \_\text{key} = B \_\text{key} \]
\[ \text{UML}_\text{OCL.univ.encoding.B} \_\text{base} \_\text{type} = B \_\text{key} \times \text{oid} \]
\[ \text{UML}_\text{OCL.univ.encoding.B} \_\text{type} = (A \_\text{key} \times \text{Integer}\_0) \times (B \_\text{key} \times \text{oid}) \times \text{\'a up} \]

Together with the pre-defined OCL universe type they build the basis for the model specific universe type:

\[ (\text{\'a, \'b, \'c}) \_\text{U\_encoding\_example} = U ((A \_\text{key} \times \text{Integer}\_0) \times ((B \_\text{key} \times \text{oid}) \times \text{\'a up} + \text{\'b} \text{ up} + \text{\'c}) \]

Now we can encode types for classes represented in the universe, e.g. for class A:

\[ (\text{\'a, \'b}) \_\text{UML\_OCL.univ.encoding.A} = \text{OclAny}\_0 ((A \_\text{key} \times \text{Integer}\_0) \times ((B \_\text{key} \times \text{oid}) \times \text{\'a up} + \text{\'b} \text{ up}) \]

and for class B:

\[ \text{\'a UML\_OCL.univ.encoding.B} = \text{OclAny}\_0 ((A \_\text{key} \times \text{Integer}\_0) \times ((B \_\text{key} \times \text{oid}) \times \text{\'a up}) \text{ up}) \]

**Defining the Basic Methods over Classes**

For each class we define two type tests for objects, one for testing if a object is of a specific type or subtype (i.e., this test is true, if the object is of the given kind):

\[ \text{UML\_OCL.level0.encoding.is\_A} :: \\
\text{OclAny\_0 ((A \_\text{key} \times \text{Integer}\_0) \times ((B \_\text{key} \times \text{oid}) \times \text{\'a up} + \text{\'b} \text{ up}) \Rightarrow bool} \]

\[ \text{level0.encoding.is\_A} \equiv \text{DEF} \circ \text{base} \]
and one for testing if a object is exactly the requested type

\[ UML\_OCL.level0.encoding.isType\_A ::
OclAny\_0 ((A\_key \times \text{Integer}\_0) \times ((B\_key \times \text{oid}) \times 'a up + 'b) up) \Rightarrow \text{bool}
\]
\[ \lambda obj. level0.encoding.is\_A obj \land \neg (\text{DEF} \circ \text{base} \circ \text{base}) obj
\]

Further, we define a type for universe instances:

\[ UML\_OCL.level0.encoding.is\_A\_univ ::
U (\text{OclAny}\_0 ((A\_key \times \text{Integer}\_0) \times ((B\_key \times \text{oid}) \times 'a up + 'b) up + 'c) \Rightarrow \text{bool}
\]
\[ level0.encoding.is\_A\_univ ?\text{univ} \equiv level0.is\_OclAny\_univ ?\text{univ} \land (\text{DEF} \circ \text{FromL} \circ \text{base} \circ \text{level0.get\_OclAny}) ?\text{univ}
\]

Analogous we define for the inherited class B:

\[ UML\_OCL.level0.encoding.is\_B ::
OclAny\_0 ((A\_key \times \text{Integer}\_0) \times ((B\_key \times \text{oid}) \times 'a up) up) \Rightarrow \text{bool}
\]
\[ level0.encoding.is\_B \equiv \text{DEF} \circ \text{base} \circ \text{base}
\]

\[ UML\_OCL.level0.encoding.isType\_B ::
OclAny\_0 ((A\_key \times \text{Integer}\_0) \times ((B\_key \times \text{oid}) \times 'a up) up) \Rightarrow \text{bool}
\]
\[ \lambda obj. level0.encoding.is\_B obj \land \neg (\text{DEF} \circ \text{base} \circ \text{base} \circ \text{base}) obj
\]

\[ UML\_OCL.level0.encoding.is\_B\_univ ::
U (\text{OclAny}\_0 ((A\_key \times \text{Integer}\_0) \times ((B\_key \times \text{oid}) \times 'a up + 'b) up + 'c) \Rightarrow \text{bool}
\]
\[ level0.encoding.is\_B\_univ ?\text{univ} \equiv level0.encoding.is\_A\_univ ?\text{univ} \land (\text{DEF} \circ \text{FromL} \circ \text{base} \circ \text{base} \circ \text{level0.encoding.get\_A}) ?\text{univ}
\]
We define a function for embedding a class into the corresponding universe

\[
UML\_OCL.\text{level}\_0.\text{encoding}.\text{mk}\_A ::
\]

\[
\text{OclAny}_0 (\text{A\_key} \times \text{Integer}_0) \times ((\text{B\_key} \times \text{oid}) \times \text{'}a\text{ up} + \text{'}b\text{ up})
\]

\[
\Rightarrow U ((\text{A\_key} \times \text{Integer}_0) \times ((\text{B\_key} \times \text{oid}) \times \text{'}\text{a up} + \text{'}\text{b up}) + \text{'}c)
\]

\[
\text{level0.encoding.mk}_A\ ?\text{obj} \equiv
\]

\[
\text{level0.\text{mk\_OclAny}} \; (\text{sup} \; \text{?obj}, (\text{lift} \; \circ \; \text{Inl} \; \circ \; \text{Lifting.drop} \; \circ \; \text{base}) \; ?\text{obj})
\]

and one for extracting an object out of a given universe:

\[
UML\_OCL.\text{level}\_0.\text{encoding}.\text{get}_A ::
\]

\[
U ((\text{A\_key} \times \text{Integer}_0) \times ((\text{B\_key} \times \text{oid}) \times \text{'}\text{a up} + \text{'}\text{b up}) + \text{'}c)
\]

\[
\Rightarrow \text{OclAny}_0 ((\text{A\_key} \times \text{Integer}_0) \times ((\text{B\_key} \times \text{oid}) \times \text{'}\text{a up} + \text{'}\text{b up}) + \text{'}c
\]

\[
\text{level0.\text{encoding}.\text{get}_A}\ ?\text{univ} \equiv
\]

\[
((\text{sup} \; \circ \; \text{level0.\text{get\_OclAny}}) \; ?\text{univ}, (\text{FromL} \; \circ \; \text{base} \; \circ \; \text{level0.\text{get\_OclAny}}) \; ?\text{univ})\]

These two functions can also be understood as conversion between object types and universe types.

Analogous we define for the inherited class B:

\[
UML\_OCL.\text{level}\_0.\text{encoding}.\text{mk}_B ::
\]

\[
\text{OclAny}_0 ((\text{A\_key} \times \text{Integer}_0) \times ((\text{B\_key} \times \text{oid}) \times \text{'}\text{a up} + \text{'}\text{b up}) \; \text{up})
\]

\[
\Rightarrow U ((\text{A\_key} \times \text{Integer}_0) \times ((\text{B\_key} \times \text{oid}) \times \text{'}\text{a up} + \text{'}\text{b up}) + \text{'}c
\]

\[
\text{level0.\text{encoding}.\text{mk}_B}\ ?\text{obj} \equiv
\]

\[
\text{level0.\text{encoding}.\text{mk}_A}
\]

\[
((\text{sup} \; ?\text{obj}, (\text{sup} \; \circ \; \text{base}) \; ?\text{obj})\]

\[
UML\_OCL.\text{level}\_0.\text{encoding}.\text{get}_B ::
\]

\[
U ((\text{A\_key} \times \text{Integer}_0) \times ((\text{B\_key} \times \text{oid}) \times \text{'}\text{a up} + \text{'}\text{b up}) \; \text{up})
\]

\[
\Rightarrow \text{OclAny}_0 ((\text{A\_key} \times \text{Integer}_0) \times ((\text{B\_key} \times \text{oid}) \times \text{'}\text{a up} \; \text{up}) + \text{'}c
\]

\[
\text{level0.\text{encoding}.\text{get}_B}\ ?\text{univ} \equiv
\]

\[
((\text{sup} \; \circ \; \text{level0.\text{encoding}.\text{get}_A}) \; ?\text{univ}, (\text{FromL} \; \circ \; \text{base} \; \circ \; \text{level0.\text{encoding}.\text{get}_A}) \; ?\text{univ})\]

\[
443
\]
Appendix C. Encoding UML/OCL by Example

For converting along type hierarchies (i.e. down-casting and up-casting) we provide for each class two functions for the conversion to the direct predecessor and direct successor in the class hierarchy:

\[
\text{UML\_OCL.level0.encoding.A}_2\text{OclAny} ::
\]

\[
\text{OclAny}_0 ((A\_key \times \text{Integer}_0) \times ((B\_key \times \text{oid}) \times 'a up + 'b) up)
\]

\[
\implies \text{OclAny}_0 ((A\_key \times \text{Integer}_0) \times ((B\_key \times \text{oid}) \times 'a up + 'b) up + 'c)
\]

\[
\text{level0.encoding.A}_2\text{OclAny} \equiv \text{level0.get\_OclAny} \circ \text{level0.encoding.mk\_A}
\]

Analogous we define for the inherited class B:

\[
\text{UML\_OCL.level0.encoding.OclAny}_2\text{A} ::
\]

\[
\text{OclAny}_0 ((A\_key \times \text{Integer}_0) \times ((B\_key \times \text{oid}) \times 'a up + 'b) up + 'c)
\]

\[
\implies \text{OclAny}_0 ((A\_key \times \text{Integer}_0) \times ((B\_key \times \text{oid}) \times 'a up + 'b) up)
\]

\[
\text{level0.encoding.OclAny}_2\text{A} \equiv \text{level0.encoding.get\_A} \circ \text{level0 mk\_OclAny}
\]

Finally, we provide accessor functions for the class attributes:

\[
\text{UML\_OCL.level0.encoding.A.i} ::
\]

\[
\text{OclAny}_0 ((A\_key \times \text{Integer}_0) \times ((B\_key \times \text{oid}) \times 'a up + 'b) up) \implies \text{Integer}_0
\]

\[
\text{level0.encoding.A.i} \equiv \text{snd} \circ \text{fst} \circ \text{Lifting.drop} \circ \text{base}
\]
C.3. Importing UML/OCL models

\( UML\_OCL\text{-}level0\_encoding.B.j :: \)

\( \text{OclAny}_0 ((A\_key \times \text{Integer}_0) \times ((B\_key \times \text{oid}) \times \text{'a up}) \text{ up }) \Rightarrow \text{oid} \)

\( \text{level0}\_encoding.B.j \equiv \text{snd} \circ \text{fst} \circ \text{Lifting.drop} \circ \text{base} \circ \text{base} \)

Proving Properties

Already on this level, we can prove important properties of our encoding. For example, the following theorem “UML\_OCL\_level0\_encoding\_is\_A\_univ\_implies\_OclAny\_univ” shows, that on the universe types, every object of class \( A \) is also an object of class \( \text{OclAny} \):

\( \text{level0}\_encoding\_is\_A\_univ \ ? \text{obj} \Rightarrow \text{level0}\_is\_OclAny\_univ \ ? \text{obj} \)

Using \( \text{get\_mk\_A\_id} \) and \( \text{get\_mk\_A\_id} \) we can convert “lossless” between our class type representation and universe representation, this is shown by “UML\_OCL\_level0\_encoding\_get\_mk\_A\_id”:

\( \text{level0}\_encoding\_is\_A \ ? \text{obj} \Rightarrow \)

\( \text{level0}\_encoding\_get\_A \ (\text{level0}\_encoding\_mk\_A \ ? \text{obj}) = \ ? \text{obj} \)

and “UML\_OCL\_level0\_encoding\_mk\_get\_A\_id”:

\( \text{level0}\_encoding\_is\_A\_univ \ ? \text{univ} \Rightarrow \)

\( \text{level0}\_encoding\_mk\_A \ (\text{level0}\_encoding\_get\_A \ ? \text{univ}) = \ ? \text{univ} \)

and “UML\_OCL\_level0\_encoding\_is\_A\_univ\_implies\_is\_get” shows that a successful type test on the universe types also implies a successful type test on the class types:

\( \text{level0}\_encoding\_is\_A\_univ \ ? \text{univ} \Rightarrow \)

\( \text{level0}\_encoding\_is\_A \ (\text{level0}\_encoding\_get\_A \ ? \text{univ}) \)

The theorem “UML\_OCL\_level0\_encoding\_is\_mk\_A” shows that the constructor creates an object that is of type \( A \):

\( \text{level0}\_encoding\_is\_A \ ? \text{obj} \Rightarrow \)

\( \text{level0}\_encoding\_is\_A\_univ \ (\text{level0}\_encoding\_mk\_A \ ? \text{obj}) \)
Appendix C. Encoding UML/OCL by Example

and also satisfies the test of its super-type, e.g., `OclAny`. This is shown by the following theorem “UML_OCL.level0.encoding.is_OclAny_mk_A”:

\[
\text{level0.encoding.is}\_A\ ?\text{obj} \implies \text{level0.is}\_\text{OclAny}\_\text{univ} (\text{level0.encoding.mk}\_A\ ?\text{obj})
\]

“UML_OCL.level0.encoding.cast_A_id” shows that we can successfully downcast objects:

\[
\text{level0.encoding.is}\_A\ ?\text{obj} \implies \\
\text{level0.encoding.OclAny}\_\text{2}_\text{A} (\text{level0.encoding.A}\_\text{2}_\text{OclAny}\ ?\text{obj}) = ?\text{obj}
\]

Analogous, these properties are also shown for class B:

UML_OCL.level0.encoding.is_B_univ_implies_A_univ:

\[
\text{level0.encoding.is}\_B\_\text{univ}\ ?\text{obj} \implies \text{level0.encoding.is}\_A\_\text{univ} \ ?\text{obj}
\]

UML_OCL.level0.encoding.is_B_univ_implies_is_get:

\[
\text{level0.encoding.is}\_B\_\text{univ}\ ?\text{univ} \implies \\
\text{level0.encoding.is}\_B (\text{level0.encoding.get}\_B\ ?\text{univ})
\]

UML_OCL.level0.encoding.get_mk_B_id:

\[
\text{level0.encoding.is}\_B\ ?\text{obj} \implies \\
\text{level0.encoding.get}\_B (\text{level0.encoding.mk}\_B\ ?\text{obj}) = ?\text{obj}
\]

UML_OCL.level0.encoding.is_mk_B:

\[
\text{level0.encoding.is}\_B\ ?\text{obj} \implies \\
\text{level0.encoding.is}\_B\_\text{univ} (\text{level0.encoding.mk}\_B\ ?\text{obj})
\]

UML_OCL.level0.encoding.is_A_mk_B:

\[
\text{level0.encoding.is}\_B\ ?\text{obj} \implies \\
\text{level0.encoding.is}\_A\_\text{univ} (\text{level0.encoding.mk}\_B\ ?\text{obj})
\]
C.3. Importing UML/OCL models

UML_OCL.level0.encoding.mk_get_B_id:

\[
\text{level0.encoding.is}_B\_univ\ ?\text{univ} \implies \\
\text{level0.encoding.mk}_B(\text{level0.encoding.get}_B\ ?\text{univ}) = \text{?univ}
\]

UML_OCL.level0.encoding.cast_B_id:

\[
\text{level0.encoding.is}_B\ ?\text{obj} \implies \\
\text{level0.encoding.A}_2\_B(\text{level0.encoding.B}_2\_A\ ?\text{obj}) = \text{?obj}
\]

C.3.2. Encoding Level 1:

The encoding on level 1 follows strictly the schema from level 0, i.e., we also start two type tests for objects, one for testing if a object is of a specific type or subtype (i.e., this test is true, if the object is of the given kind):

\[
\text{UML_OCL.level1.encoding.is}_A :: \\
\text{state } U \ ((A\_key \times \text{Integer}_0) \times ((B\_key \times \text{oid}) \times 'a \ up + 'b) \ up + 'c) \times \\
\text{state } U \ ((A\_key \times \text{Integer}_0) \times ((B\_key \times \text{oid}) \times 'a \ up + 'b) \ up + 'c) \\
\implies \text{OclAny}_0 \ ((A\_key \times \text{Integer}_0) \times ((B\_key \times \text{oid}) \times 'a \ up + 'b) \ up)) \\
\implies \text{state } U \ ((A\_key \times \text{Integer}_0) \times ((B\_key \times \text{oid}) \times 'a \ up + 'b) \ up + 'c) \times \\
\text{state } U \ ((A\_key \times \text{Integer}_0) \times ((B\_key \times \text{oid}) \times 'a \ up + 'b) \ up + 'c) \\
\implies \text{Boolean}_0
\]

\[
\text{level1.encoding.is}_A \equiv \text{lift}_1(\text{strictify}(\text{lift} \circ \text{level0.encoding.is}_A))
\]

and one for testing if a object is exactly the requested type

\[
\text{UML_OCL.level1.encoding.isType}_A :: \\
\text{state } U \ ((A\_key \times \text{Integer}_0) \times ((B\_key \times \text{oid}) \times 'a \ up + 'b) \ up + 'c) \times \\
\text{state } U \ ((A\_key \times \text{Integer}_0) \times ((B\_key \times \text{oid}) \times 'a \ up + 'b) \ up + 'c) \\
\implies \text{OclAny}_0 \ ((A\_key \times \text{Integer}_0) \times ((B\_key \times \text{oid}) \times 'a \ up + 'b) \ up)) \\
\implies \text{state } U \ ((A\_key \times \text{Integer}_0) \times ((B\_key \times \text{oid}) \times 'a \ up + 'b) \ up + 'c) \times \\
\text{state } U \ ((A\_key \times \text{Integer}_0) \times ((B\_key \times \text{oid}) \times 'a \ up + 'b) \ up + 'c) \\
\implies \text{Boolean}_0
\]

\[
\text{level1.encoding.isType}_A \equiv \text{lift}_1(\text{strictify}(\text{lift} \circ \text{level0.encoding.isType}_A))
\]
Appendix C. Encoding UML/OCL by Example

Further, we define a type for universe instances:

\[
\text{UML\_OCL.\text{level}\_1.\text{encoding}.\text{is}\_A\_univ}::
\]

\[
(\text{state } U ((A\_key \times \text{Integer}\_0) \times ((B\_key \times \text{oid}) \times \text{up} + \text{b} \times \text{up} + \text{c})) \times
\]

\[
\text{state } U ((A\_key \times \text{Integer}\_0) \times ((B\_key \times \text{oid}) \times \text{up} + \text{b} \times \text{up} + \text{c})
\]

\[
\Rightarrow U ((A\_key \times \text{Integer}\_0) \times ((B\_key \times \text{oid}) \times \text{up} + \text{b} \times \text{up} + \text{c}))
\]

\[
\Rightarrow \text{state } U ((A\_key \times \text{Integer}\_0) \times ((B\_key \times \text{oid}) \times \text{up} + \text{b} \times \text{up} + \text{c}) \times
\]

\[
\text{state } U ((A\_key \times \text{Integer}\_0) \times ((B\_key \times \text{oid}) \times \text{up} + \text{b} \times \text{up} + \text{c})
\]

\[
\Rightarrow \text{Boolean}\_0
\]

\[
\text{level}\_1.\text{encoding}.\text{is}\_A\_univ \equiv \text{lift}_1 (\text{lift} \circ \text{level}\_0.\text{encoding}.\text{is}\_A\_univ)
\]

Analogous we define for the inherited class B:

\[
\text{UML\_OCL.\text{level}\_1.\text{encoding}.\text{is}\_B}::
\]

\[
(\text{state } U ((A\_key \times \text{Integer}\_0) \times ((B\_key \times \text{oid}) \times \text{up} + \text{b} \times \text{up} + \text{c})) \times
\]

\[
\text{state } U ((A\_key \times \text{Integer}\_0) \times ((B\_key \times \text{oid}) \times \text{up} + \text{b} \times \text{up} + \text{c})
\]

\[
\Rightarrow \text{OclAny}\_0 ((A\_key \times \text{Integer}\_0) \times ((B\_key \times \text{oid}) \times \text{up}))
\]

\[
\Rightarrow \text{state } U ((A\_key \times \text{Integer}\_0) \times ((B\_key \times \text{oid}) \times \text{up} + \text{b} \times \text{up} + \text{c}) \times
\]

\[
\text{state } U ((A\_key \times \text{Integer}\_0) \times ((B\_key \times \text{oid}) \times \text{up} + \text{b} \times \text{up} + \text{c})
\]

\[
\Rightarrow \text{Boolean}\_0
\]

\[
\text{level}\_1.\text{encoding}.\text{is}\_B \equiv \text{lift}_1 (\text{strictify (lift} \circ \text{level}\_0.\text{encoding}.\text{is}\_B))
\]

\[
\text{UML\_OCL.\text{level}\_1.\text{encoding}.\text{isType}\_B}::
\]

\[
(\text{state } U ((A\_key \times \text{Integer}\_0) \times ((B\_key \times \text{oid}) \times \text{up} + \text{b} \times \text{up} + \text{c})) \times
\]

\[
\text{state } U ((A\_key \times \text{Integer}\_0) \times ((B\_key \times \text{oid}) \times \text{up} + \text{b} \times \text{up} + \text{c})
\]

\[
\Rightarrow \text{OclAny}\_0 ((A\_key \times \text{Integer}\_0) \times ((B\_key \times \text{oid}) \times \text{up}))
\]

\[
\Rightarrow \text{state } U ((A\_key \times \text{Integer}\_0) \times ((B\_key \times \text{oid}) \times \text{up} + \text{b} \times \text{up} + \text{c}) \times
\]

\[
\text{state } U ((A\_key \times \text{Integer}\_0) \times ((B\_key \times \text{oid}) \times \text{up} + \text{b} \times \text{up} + \text{c})
\]

\[
\Rightarrow \text{Boolean}\_0
\]

\[
\text{level}\_1.\text{encoding}.\text{isType}\_B \equiv \text{lift}_1 (\text{strictify (lift} \circ \text{level}\_0.\text{encoding}.\text{isType}\_B))
\]
We define a functions for embedding a class into the corresponding universe

\[
\text{UML\_OCL}\text{.level1}\text{.encoding.is}_B\text{._univ} ::
\]
\[
\text{state } U \ ((A\text{._key} \times \text{Integer}_0) \times ((B\text{._key} \times \text{oid}) \times \text{'}a\text{ up} + \text{'}b\text{ up} + \text{'}c) \times \]
\[
\Rightarrow U \ ((A\text{._key} \times \text{Integer}_0) \times ((B\text{._key} \times \text{oid}) \times \text{'}a\text{ up} + \text{'}b\text{ up} + \text{'}c)
\]
\[
\Rightarrow \text{OclAny}_0 \ ((A\text{._key} \times \text{Integer}_0) \times ((B\text{._key} \times \text{oid}) \times \text{'}a\text{ up} + \text{'}b\text{ up})
\]
\[
\Rightarrow \text{state } U \ ((A\text{._key} \times \text{Integer}_0) \times ((B\text{._key} \times \text{oid}) \times \text{'}a\text{ up} + \text{'}b\text{ up} + \text{'}c) \times \]
\[
\Rightarrow U \ ((A\text{._key} \times \text{Integer}_0) \times ((B\text{._key} \times \text{oid}) \times \text{'}a\text{ up} + \text{'}b\text{ up} + \text{'}c)
\]
\[
\text{level1}\text{.encoding.is}_B\text{._univ} \equiv \text{lift}_1 \ (\text{lift} \circ \text{level0}\text{.encoding.is}_B\text{._univ})
\]

and one for extracting an object out of a given universe:

\[
\text{UML\_OCL}\text{.level1}\text{.encoding.mk}_A ::
\]
\[
\text{state } U \ ((A\text{._key} \times \text{Integer}_0) \times ((B\text{._key} \times \text{oid}) \times \text{'}a\text{ up} + \text{'}b\text{ up} + \text{'}c) \times \]
\[
\Rightarrow \text{OclAny}_0 \ ((A\text{._key} \times \text{Integer}_0) \times ((B\text{._key} \times \text{oid}) \times \text{'}a\text{ up} + \text{'}b\text{ up})
\]
\[
\Rightarrow \text{state } U \ ((A\text{._key} \times \text{Integer}_0) \times ((B\text{._key} \times \text{oid}) \times \text{'}a\text{ up} + \text{'}b\text{ up} + \text{'}c) \times \]
\[
\Rightarrow U \ ((A\text{._key} \times \text{Integer}_0) \times ((B\text{._key} \times \text{oid}) \times \text{'}a\text{ up} + \text{'}b\text{ up} + \text{'}c)
\]
\[
\text{level1}\text{.encoding.mk}_A \equiv \text{lift}_1 \ \text{level0}\text{.encoding.mk}_A
\]

These two functions can also be understood as conversion between object types and universe types.
Appendix C. Encoding $\text{UML|OCL}$ by Example

Analogous we define for the inherited class $B$:

\[
\text{UML\_OCL.level1.encoding.mk\_B ::}
\]

\[
\text{(state } U \text{ ((A\_key } \times \text{ Integer}_0) \times (B\_key } \times \text{ oid)} \times 'a up + 'b) up + 'c) \times
\text{state } U \text{ ((A\_key } \times \text{ Integer}_0) \times (B\_key } \times \text{ oid)} \times 'a up + 'b) up + 'c)
\Rightarrow \text{OclAny\_0 ((A\_key } \times \text{ Integer}_0) \times ((B\_key } \times \text{ oid)} \times 'a up) up)
\Rightarrow \text{state } U \text{ ((A\_key } \times \text{ Integer}_0) \times ((B\_key } \times \text{ oid)} \times 'a up + 'b) up + 'c) \times
\text{state } U \text{ ((A\_key } \times \text{ Integer}_0) \times ((B\_key } \times \text{ oid)} \times 'a up + 'b) up + 'c)
\Rightarrow U ((A\_key } \times \text{ Integer}_0) \times ((B\_key } \times \text{ oid)} \times 'a up + 'b) up + 'c)
\text{level1.encoding.mk\_B } \equiv \text{lift}_1 \text{ level0.encoding.mk\_B}
\]

\[
\text{UML\_OCL.level1.encoding.get\_B ::}
\]

\[
\text{(state } U \text{ ((A\_key } \times \text{ Integer}_0) \times ((B\_key } \times \text{ oid)} \times 'a up + 'b) up + 'c) \times
\text{state } U \text{ ((A\_key } \times \text{ Integer}_0) \times ((B\_key } \times \text{ oid)} \times 'a up + 'b) up + 'c)
\Rightarrow U ((A\_key } \times \text{ Integer}_0) \times ((B\_key } \times \text{ oid)} \times 'a up + 'b) up + 'c)
\Rightarrow \text{state } U \text{ ((A\_key } \times \text{ Integer}_0) \times ((B\_key } \times \text{ oid)} \times 'a up + 'b) up + 'c) \times
\text{state } U \text{ ((A\_key } \times \text{ Integer}_0) \times ((B\_key } \times \text{ oid)} \times 'a up + 'b) up + 'c)
\Rightarrow \text{OclAny\_0 ((A\_key } \times \text{ Integer}_0) \times ((B\_key } \times \text{ oid)} \times 'a up) up)
\Rightarrow \text{state } U \text{ ((A\_key } \times \text{ Integer}_0) \times ((B\_key } \times \text{ oid)} \times 'a up + 'b) up + 'c) \times
\text{state } U \text{ ((A\_key } \times \text{ Integer}_0) \times ((B\_key } \times \text{ oid)} \times 'a up + 'b) up + 'c)
\Rightarrow \text{OclAny\_0 ((A\_key } \times \text{ Integer}_0) \times ((B\_key } \times \text{ oid)} \times 'a up + 'b) up + 'c)
\Rightarrow U ((A\_key } \times \text{ Integer}_0) \times ((B\_key } \times \text{ oid)} \times 'a up + 'b) up + 'c)
\text{level1.encoding.get\_B } \equiv \text{lift}_1 \text{ level0.encoding.get\_B}
\]

For converting along type hierarchies (i.e. down-casting and up-casting) we provide for each class two functions for the conversion to the direct predecessor and direct successor in the class hierarchy:

\[
\text{UML\_OCL.level1.encoding.A\_2\_OclAny ::}
\]

\[
\text{(state } U \text{ ((A\_key } \times \text{ Integer}_0) \times ((B\_key } \times \text{ oid)} \times 'a up + 'b) up + 'c) \times
\text{state } U \text{ ((A\_key } \times \text{ Integer}_0) \times ((B\_key } \times \text{ oid)} \times 'a up + 'b) up + 'c)
\Rightarrow \text{OclAny\_0 ((A\_key } \times \text{ Integer}_0) \times ((B\_key } \times \text{ oid)} \times 'a up + 'b) up)
\Rightarrow \text{state } U \text{ ((A\_key } \times \text{ Integer}_0) \times ((B\_key } \times \text{ oid)} \times 'a up + 'b) up + 'c) \times
\text{state } U \text{ ((A\_key } \times \text{ Integer}_0) \times ((B\_key } \times \text{ oid)} \times 'a up + 'b) up + 'c)
\Rightarrow \text{OclAny\_0 ((A\_key } \times \text{ Integer}_0) \times ((B\_key } \times \text{ oid)} \times 'a up + 'b) up + 'c)
\Rightarrow U ((A\_key } \times \text{ Integer}_0) \times ((B\_key } \times \text{ oid)} \times 'a up + 'b) up + 'c)
\text{level1.encoding.A\_2\_OclAny } \equiv \text{lift}_1 \text{ level0.encoding.A\_2\_OclAny}
\]
Analogous we define for the inherited class `UML`:

\[ UML_{\text{OCL}}.\text{level1.encoding.OclAny}_2.A \::= \]
\[ (\text{state } U ((A_{\text{key}} \times \text{Integer}_0) \times ((B_{\text{key}} \times \text{oid}) \times ('a \uparrow + 'b) \uparrow + 'c) \times \]
\[ \text{state } U ((A_{\text{key}} \times \text{Integer}_0) \times ((B_{\text{key}} \times \text{oid}) \times ('a \uparrow + 'b) \uparrow + 'c) \]
\[ \Rightarrow \text{OclAny}_0 ((A_{\text{key}} \times \text{Integer}_0) \times ((B_{\text{key}} \times \text{oid}) \times ('a \uparrow + 'b) \uparrow + 'c) \)
\[ \Rightarrow \text{state } U ((A_{\text{key}} \times \text{Integer}_0) \times ((B_{\text{key}} \times \text{oid}) \times ('a \uparrow + 'b) \uparrow + 'c) \)
\[ \Rightarrow \text{state } U ((A_{\text{key}} \times \text{Integer}_0) \times ((B_{\text{key}} \times \text{oid}) \times ('a \uparrow + 'b) \uparrow + 'c) \]
\[ \Rightarrow \text{OclAny}_0 ((A_{\text{key}} \times \text{Integer}_0) \times ((B_{\text{key}} \times \text{oid}) \times ('a \uparrow + 'b) \uparrow) \]
\[ \text{level1.encoding.OclAny}_2.A \equiv \text{lift}_1 \text{level0.encoding.OclAny}_2.A \]

C.3. Importing `UML/OCL` models

Analogous we define for the inherited class `B`:

\[ UML_{\text{OCL}}.\text{level1.encoding.B}_2.A \::= \]
\[ (\text{state } U ((A_{\text{key}} \times \text{Integer}_0) \times ((B_{\text{key}} \times \text{oid}) \times ('a \uparrow + 'b) \uparrow + 'c) \times \]
\[ \text{state } U ((A_{\text{key}} \times \text{Integer}_0) \times ((B_{\text{key}} \times \text{oid}) \times ('a \uparrow + 'b) \uparrow + 'c) \]
\[ \Rightarrow \text{OclAny}_0 ((A_{\text{key}} \times \text{Integer}_0) \times ((B_{\text{key}} \times \text{oid}) \times ('a \uparrow) \uparrow)) \]
\[ \Rightarrow \text{state } U ((A_{\text{key}} \times \text{Integer}_0) \times ((B_{\text{key}} \times \text{oid}) \times ('a \uparrow + 'b) \uparrow + 'c) \]
\[ \Rightarrow \text{state } U ((A_{\text{key}} \times \text{Integer}_0) \times ((B_{\text{key}} \times \text{oid}) \times ('a \uparrow + 'b) \uparrow + 'c) \]
\[ \Rightarrow \text{OclAny}_0 ((A_{\text{key}} \times \text{Integer}_0) \times ((B_{\text{key}} \times \text{oid}) \times ('a \uparrow + 'b) \uparrow) \]
\[ \text{level1.encoding.B}_2.A \equiv \text{lift}_1 \text{level0.encoding.B}_2.A \]

\[ UML_{\text{OCL}}.\text{level1.encoding.A}_2.B \::= \]
\[ (\text{state } U ((A_{\text{key}} \times \text{Integer}_0) \times ((B_{\text{key}} \times \text{oid}) \times ('a \uparrow + 'b) \uparrow + 'c) \times \]
\[ \text{state } U ((A_{\text{key}} \times \text{Integer}_0) \times ((B_{\text{key}} \times \text{oid}) \times ('a \uparrow + 'b) \uparrow + 'c) \]
\[ \Rightarrow \text{OclAny}_0 ((A_{\text{key}} \times \text{Integer}_0) \times ((B_{\text{key}} \times \text{oid}) \times ('a \uparrow + 'b) \uparrow)) \]
\[ \Rightarrow \text{state } U ((A_{\text{key}} \times \text{Integer}_0) \times ((B_{\text{key}} \times \text{oid}) \times ('a \uparrow + 'b) \uparrow + 'c) \]
\[ \Rightarrow \text{state } U ((A_{\text{key}} \times \text{Integer}_0) \times ((B_{\text{key}} \times \text{oid}) \times ('a \uparrow + 'b) \uparrow + 'c) \]
\[ \Rightarrow \text{OclAny}_0 ((A_{\text{key}} \times \text{Integer}_0) \times ((B_{\text{key}} \times \text{oid}) \times ('a \uparrow) \uparrow) \]
\[ \text{level1.encoding.A}_2.B \equiv \text{lift}_1 \text{level0.encoding.A}_2.B \]
Finally, we provide accessor functions for the class attributes:

\[\textit{UML\_OCL.level1.encoding.A.i} :: \]
\[
\text{(state } U ((A\_key \times \text{Integer}_0) \times ((B\_key \times \text{oid}) \times 'a up + 'b) up + 'c) \times \\
\text{state } U ((A\_key \times \text{Integer}_0) \times ((B\_key \times \text{oid}) \times 'a up + 'b) up + 'c) \Rightarrow \text{OclAny}_0 ((A\_key \times \text{Integer}_0) \times ((B\_key \times \text{oid}) \times 'a up + 'b) up)) \\
\Rightarrow \text{state } U ((A\_key \times \text{Integer}_0) \times ((B\_key \times \text{oid}) \times 'a up + 'b) up + 'c) \times \\
\text{state } U ((A\_key \times \text{Integer}_0) \times ((B\_key \times \text{oid}) \times 'a up + 'b) up + 'c) \Rightarrow \text{Integer}_0
\]

\[\text{level1.encoding.A.i} \equiv \text{lift}_1 (\text{strictify level0.encoding.A.i})\]

\[\textit{UML\_OCL.level1.encoding.B.j} :: \]
\[
\text{(state } U ((A\_key \times \text{Integer}_0) \times ((B\_key \times \text{oid}) \times 'a up + 'b) up + 'c) \times \\
\text{state } U ((A\_key \times \text{Integer}_0) \times ((B\_key \times \text{oid}) \times 'a up + 'b) up + 'c) \Rightarrow \text{OclAny}_0 ((A\_key \times \text{Integer}_0) \times ((B\_key \times \text{oid}) \times 'a up) up)) \\
\Rightarrow \text{state } U ((A\_key \times \text{Integer}_0) \times ((B\_key \times \text{oid}) \times 'a up + 'b) up + 'c) \times \\
\text{state } U ((A\_key \times \text{Integer}_0) \times ((B\_key \times \text{oid}) \times 'a up + 'b) up + 'c) \Rightarrow \text{OclAny}_0 ((A\_key \times \text{Integer}_0) \times ((B\_key \times \text{oid}) \times 'a up) up) \\
\text{level1.encoding.B.j} \equiv \lambda X \text{ St. case } \text{fst St} (\text{level0.encoding.B.j (X St)}) \text{ of None } \Rightarrow \bot \\
| \text{Some x } \Rightarrow \text{level0.encoding.get_B x}
\]

**Proving Properties**

Already on this level, we can prove important properties of our encoding. For example, the following theorem “\textit{UML\_OCL.level1.encoding.is\_A\_univ\_implies\_OclAny\_univ}” shows, that on the universe types, every object of class A is also an object of class OclAny:

\[\text{?St } \models \text{level1.encoding.is\_A\_univ ?obj } \Rightarrow \text{?St } \models \text{level1.is\_OclAny\_univ ?obj}\]
C.3. Importing UML/OCL models

Using get_mk_A_id and get_mk_A_id we can convert “lossless” between our class type representation and universe representation, this is shown by “UML_OCL.level1.encoding.get_mk_A_id”:

\[ \text{?St} \models \text{level1.encoding.is}_\text{A} \ ?\text{obj} \implies \text{level1.encoding.get}_\text{A} (\text{level1.encoding.mk}_\text{A} \ ?\text{obj}) \ ?\text{St} = \ ?\text{obj} \ ?\text{St} \]

and “UML_OCL.level1.encoding.mk_get_A_id”:

\[ \text{?St} \models \text{level1.encoding.is}_\text{A}_\text{univ} \ ?\text{univ} \implies \text{level1.encoding.mk}_\text{A} (\text{level1.encoding.get}_\text{A} \ ?\text{univ}) \ ?\text{St} = \ ?\text{univ} \ ?\text{St} \]

and “UML_OCL.level1.encoding.is_A_univ_implies_is_get” shows that a successful type test on the universe types also implies a successful type test on the class types:

\[ [\text{DEF} (\text{level1.encoding.get}_\text{A} \ ?\text{univ} \ ?\text{St}); \ ?\text{St} \models \text{level1.encoding.is}_\text{A}_\text{univ} \ ?\text{univ}] \implies \ ?\text{St} \models \text{level1.encoding.is}_\text{A} (\text{level1.encoding.get}_\text{A} \ ?\text{univ}) \]

The theorem “UML_OCL.level1.encoding.is_mk_A” shows that the constructor creates an object that is of type A:

\[ \text{?St} \models \text{level1.encoding.is}_\text{A} \ ?\text{obj} \implies \ ?\text{St} \models \text{level1.encoding.is}_\text{A}_\text{univ} (\text{level1.encoding.mk}_\text{A} \ ?\text{obj}) \]

and also satisfies the test of its super-type, e.g., OclAny. This is shown by the following theorem “UML_OCL.level1.encoding.is_OclAny_mk_A”:

\[ \text{?St} \models \text{level1.encoding.is}_\text{A} \ ?\text{obj} \implies \ ?\text{St} \models \text{level1.is}_\text{OclAny}_\text{univ} (\text{level1.encoding.mk}_\text{A} \ ?\text{obj}) \]

“UML_OCL.level1.encoding.cast_A_id” shows that we can successfully downcast objects:

\[ \text{?St} \models \text{level1.encoding.is}_\text{A} \ ?\text{obj} \implies \text{level1.encoding.OclAny}_2\text{A} (\text{level1.encoding.A}_2\text{OclAny} \ ?\text{obj}) \ ?\text{St} = \ ?\text{obj} \ ?\text{St} \]
Appendix C. Encoding UML/OCL by Example

Analogous, these properties are also shown for class B:

UML_OCL.level1.encoding.is_B_univ_implies_A_univ:

\[ \text{level1.encoding.is\textunderscore B\textunderscore univ}\implies\text{level1.encoding.is\textunderscore A\textunderscore univ} \]

UML_OCL.level1.encoding.is_B_univ_implies_is_get:

\[ [\text{DEF (level1.encoding.get\textunderscore B\textunderscore univ}\implies\text{St})}\implies\text{St}\implies\text{level1.encoding.is\textunderscore B\textunderscore univ}\implies\text{level1.encoding.get\textunderscore B\textunderscore univ} \]

UML_OCL.level1.encoding.get\_mk\_B\_id:

\[ \text{level1.encoding.get\textunderscore B}\implies\text{level1.encoding.is\textunderscore B\textunderscore univ}\implies\text{St} = \text{obj}\implies\text{St} \]

UML_OCL.level1.encoding.is\_mk\_B:

\[ \text{level1.encoding.is\textunderscore B}\implies\text{level1.encoding.is\textunderscore B\textunderscore univ}\implies\text{St} = \text{obj}\implies\text{St} \]

UML_OCL.level1.encoding.is\_A\_mk\_B:

\[ \text{level1.encoding.is\textunderscore A}\implies\text{level1.encoding.is\textunderscore A\textunderscore univ}\implies\text{St} = \text{obj}\implies\text{St} \]

UML_OCL.level1.encoding.mk\_get\_B\_id:

\[ \text{level1.encoding.mk\textunderscore B}\implies\text{level1.encoding.get\textunderscore B}\implies\text{St} = \text{obj}\implies\text{St} \]

UML_OCL.level1.encoding.cast\_B\_id:

\[ \text{level1.encoding.cast\textunderscore B}\implies\text{level1.encoding.A\textunderscore 2\textunderscore B}\implies\text{level1.encoding.A\textunderscore 2\textunderscore A}\implies\text{St} = \text{obj}\implies\text{St} \]
Further we show that our definitions are context passing:

UML\_OCL.level1.encoding.cp\_get\_A:

\[ cp \text{ level1.encoding.get}_A \]

UML\_OCL.level1.encoding.cp\_mk\_A:

\[ cp \text{ level1.encoding.mk}_A \]

UML\_OCL.level1.encoding.cp\_is\_A:

\[ cp \text{ level1.encoding.is}_A \]

UML\_OCL.level1.encoding.cp\_is\_univ\_A:

\[ cp \text{ level1.encoding.is}_A\_univ \]

UML\_OCL.level1.encoding.cp\_get\_B:

\[ cp \text{ level1.encoding.get}_B \]

UML\_OCL.level1.encoding.cp\_mk\_B:

\[ cp \text{ level1.encoding.mk}_B \]

UML\_OCL.level1.encoding.cp\_is\_B:

\[ cp \text{ level1.encoding.is}_B \]

UML\_OCL.level1.encoding.cp\_is\_univ\_B:

\[ cp \text{ level1.encoding.is}_B\_univ \]
we generate two invariants; The first one describes the initial value
\( \text{level} \geq 42 \) which
breaks cyclic dependencies and thus encode
later with the required definedness constraints. This two-staged process allows us to
encourage the definedness of the occurring path expressions, we extend these formulae

\[ \text{C.3.3. Encoding Level 2:} \]

\[ \text{Invariants: Part One} \]

Here we bring the data part (i.e., \text{UML}) together with the \text{OCL} specification. The encoding of \text{OCL} formulae is two-staged. First, the formulae are encoding without
requiring the definedness of the occurring path expressions, we extend these formulae
this two-staged process allows us to
break cyclic dependencies and thus encode \text{OCL} conservatively.

Thus, for class A we generate two invariants; The first one describes the initial value
of the attribute i

\[ \text{UML}_{-}\text{OCL.\_encoding.A.\_model.hol.\_invariant.init\_i} :: \]
\[ \text{state U} ((\text{A\_key} \times \text{Integer\_0}) \times ((\text{B\_key} \times \text{oid}) \times 'a up + 'b up + 'c) \times \]
\[ \text{state U} ((\text{A\_key} \times \text{Integer\_0}) \times ((\text{B\_key} \times \text{oid}) \times 'a up + 'b up + 'c) \]
\[ \Rightarrow \text{state U} ((\text{A\_key} \times \text{Integer\_0}) \times ((\text{B\_key} \times \text{oid}) \times 'a up + 'b up + 'c) \times \]
\[ \Rightarrow \text{OclAny}_0 ((\text{A\_key} \times \text{Integer\_0}) \times ((\text{B\_key} \times \text{oid}) \times 'a up + 'b up)) \text{ set} \]
\[ \Rightarrow \text{state U} ((\text{A\_key} \times \text{Integer\_0}) \times ((\text{B\_key} \times \text{oid}) \times 'a up + 'b up + 'c) \times \]
\[ \Rightarrow \text{OclAny}_0 ((\text{A\_key} \times \text{Integer\_0}) \times ((\text{B\_key} \times \text{oid}) \times 'a up + 'b up)) \]
\[ \Rightarrow \text{bool} \]

\[ \text{model.hol.\_invariant.init\_i} \equiv \lambda \tau C \text{ self}. \tau \models \text{OclIsNew} \text{ self} \rightarrow \tau \not\models \text{level1.\_encoding.A.\_i} \text{ self} \not\models 42 \]

and the second one describes the user defined invariant:

\[ \text{UML}_{-}\text{OCL.\_encoding.A.\_model.hol.\_invariant.inv\_1} :: \]
\[ \text{state U} ((\text{A\_key} \times \text{Integer\_0}) \times ((\text{B\_key} \times \text{oid}) \times 'a up + 'b up + 'c) \times \]
\[ \text{state U} ((\text{A\_key} \times \text{Integer\_0}) \times ((\text{B\_key} \times \text{oid}) \times 'a up + 'b up + 'c) \]
\[ \Rightarrow \text{state U} ((\text{A\_key} \times \text{Integer\_0}) \times ((\text{B\_key} \times \text{oid}) \times 'a up + 'b up + 'c) \times \]
\[ \Rightarrow \text{OclAny}_0 ((\text{A\_key} \times \text{Integer\_0}) \times ((\text{B\_key} \times \text{oid}) \times 'a up + 'b up)) \text{ set} \]
\[ \Rightarrow \text{state U} ((\text{A\_key} \times \text{Integer\_0}) \times ((\text{B\_key} \times \text{oid}) \times 'a up + 'b up + 'c) \times \]
\[ \Rightarrow \text{OclAny}_0 ((\text{A\_key} \times \text{Integer\_0}) \times ((\text{B\_key} \times \text{oid}) \times 'a up + 'b up)) \]
\[ \Rightarrow \text{bool} \]

\[ \text{model.hol.\_invariant.inv\_1} \equiv \lambda \tau C \text{ self}. \tau \not\models \text{level1.\_encoding.A.\_i} \text{ self} \geq 0 \]
The overall invariant for class $\Lambda$ is constructed by conjoining these invariants.

\[
\text{UML\_OCL.level1.encoding.A.model.hol\_invariant ::}
\]
\[
\text{state } U ((A\_key \times \text{Integer}_0) \times ((B\_key \times \text{oid}) \times 'a \ \text{up} + 'b) \ \text{up} + 'c) \times \text{state } U ((A\_key \times \text{Integer}_0) \times ((B\_key \times \text{oid}) \times 'a \ \text{up} + 'b) \ \text{up} + 'c) \\
\Rightarrow (\text{state } U ((A\_key \times \text{Integer}_0) \times ((B\_key \times \text{oid}) \times 'a \ \text{up} + 'b) \ \text{up} + 'c) \times \\
\text{state } U ((A\_key \times \text{Integer}_0) \times ((B\_key \times \text{oid}) \times 'a \ \text{up} + 'b) \ \text{up} + 'c) \\
\Rightarrow \text{OclAny}_0 ((A\_key \times \text{Integer}_0) \times ((B\_key \times \text{oid}) \times 'a \ \text{up} + 'b) \ \text{up}) \ \text{set} \\
\Rightarrow (\text{state } U ((A\_key \times \text{Integer}_0) \times ((B\_key \times \text{oid}) \times 'a \ \text{up} + 'b) \ \text{up} + 'c) \times \\
\text{state } U ((A\_key \times \text{Integer}_0) \times ((B\_key \times \text{oid}) \times 'a \ \text{up} + 'b) \ \text{up} + 'c) \\
\Rightarrow \text{OclAny}_0 ((A\_key \times \text{Integer}_0) \times ((B\_key \times \text{oid}) \times 'a \ \text{up} + 'b) \ \text{up}) \\
\Rightarrow \text{bool}
\]

\[
A\text{.model.hol\_invariant} \equiv \lambda \tau \ C \ \text{obj},
\]
\[
\text{model.hol\_invariant.init}_i \ \tau \ C \ \text{obj} \land \\
\text{model.hol\_invariant.inv}_1 \ \tau \ C \ \text{obj} \land \\
A\text{.model.hol\_invariant.check\_parent\_inv} \ \tau \ C \ \text{obj}
\]

Analogously we are encoding the pre-condition and post-condition of the operation $f$:

\[
\text{encoding.A.f\_Integer\_Integer.pre1.pre}_0 ::
\]
\[
(\text{state } U ((A\_key \times \text{Integer}_0) \times ((B\_key \times \text{oid}) \times 'a \ \text{up} + 'b) \ \text{up} + 'c) \times \\
\text{state } U ((A\_key \times \text{Integer}_0) \times ((B\_key \times \text{oid}) \times 'a \ \text{up} + 'b) \ \text{up} + 'c) \\
\Rightarrow \text{OclAny}_0 ((A\_key \times \text{Integer}_0) \times ((B\_key \times \text{oid}) \times 'a \ \text{up} + 'b) \ \text{up}) \\
\Rightarrow (\text{state } U ((A\_key \times \text{Integer}_0) \times ((B\_key \times \text{oid}) \times 'a \ \text{up} + 'b) \ \text{up} + 'c) \times \\
\text{state } U ((A\_key \times \text{Integer}_0) \times ((B\_key \times \text{oid}) \times 'a \ \text{up} + 'b) \ \text{up} + 'c) \\
\Rightarrow \text{Integer}_0 \\
\Rightarrow \text{state } U ((A\_key \times \text{Integer}_0) \times ((B\_key \times \text{oid}) \times 'a \ \text{up} + 'b) \ \text{up} + 'c) \times \\
\text{state } U ((A\_key \times \text{Integer}_0) \times ((B\_key \times \text{oid}) \times 'a \ \text{up} + 'b) \ \text{up} + 'c) \\
\Rightarrow \text{Boolean}_0
\]

\[
\text{pre}_0 \equiv \lambda \text{self } x. x > 0
\]
Appendix C. Encoding UML/OCL by Example

encoding.A.f_Integer_Integer.post1.post_0 ::
(state U ((A_key × Integer_0) × ((B_key × oid) × 'a up + 'b) up + 'c) × 
state U ((A_key × Integer_0) × ((B_key × oid) × 'a up + 'b) up + 'c)
⇒ OclAny_0 ((A_key × Integer_0) × ((B_key × oid) × 'a up + 'b) up))
⇒ (state U ((A_key × Integer_0) × ((B_key × oid) × 'a up + 'b) up + 'c) × 
state U ((A_key × Integer_0) × ((B_key × oid) × 'a up + 'b) up + 'c)
⇒ Integer_0)
⇒ (state U ((A_key × Integer_0) × ((B_key × oid) × 'a up + 'b) up + 'c) × 
state U ((A_key × Integer_0) × ((B_key × oid) × 'a up + 'b) up + 'c)
⇒ Integer_0)
⇒ state U ((A_key × Integer_0) × ((B_key × oid) × 'a up + 'b) up + 'c) × 
state U ((A_key × Integer_0) × ((B_key × oid) × 'a up + 'b) up + 'c)
⇒ Boolean_0

post_0 ≡ λself x result. result ∘ x ->div( level1.encoding.A.i self )

encoding.A.f_Integer_Integer.pre1 ::
(state U ((A_key × Integer_0) × ((B_key × oid) × 'a up + 'b) up + 'c) × 
state U ((A_key × Integer_0) × ((B_key × oid) × 'a up + 'b) up + 'c)
⇒ OclAny_0 ((A_key × Integer_0) × ((B_key × oid) × 'a up + 'b) up))
⇒ (state U ((A_key × Integer_0) × ((B_key × oid) × 'a up + 'b) up + 'c) × 
state U ((A_key × Integer_0) × ((B_key × oid) × 'a up + 'b) up + 'c)
⇒ Integer_0)
⇒ state U ((A_key × Integer_0) × ((B_key × oid) × 'a up + 'b) up + 'c) × 
state U ((A_key × Integer_0) × ((B_key × oid) × 'a up + 'b) up + 'c)
⇒ Boolean_0

pre1 ≡ pre_0
C.3. Importing UML/\textit{OCL} models

encoding.A.f\_Integer\_Integer.post1 ::
\[
\text{(state } U \text{ ((A\_key } \times \text{ Integer } \_0) \times ((B\_key } \times \text{ oid}) \times '{a\ up + 'b\ up + 'c}') \times
\text{state } U \text{ ((A\_key } \times \text{ Integer } \_0) \times ((B\_key } \times \text{ oid}) \times '{a\ up + 'b\ up + 'c})
\Rightarrow \text{OclAny}_0 \text{ ((A\_key } \times \text{ Integer } \_0) \times ((B\_key } \times \text{ oid}) \times '{a\ up + 'b\ up))
\Rightarrow \text{(state } U \text{ ((A\_key } \times \text{ Integer } \_0) \times ((B\_key } \times \text{ oid}) \times '{a\ up + 'b\ up + 'c}') \times
\text{state } U \text{ ((A\_key } \times \text{ Integer } \_0) \times ((B\_key } \times \text{ oid}) \times '{a\ up + 'b\ up + 'c})
\Rightarrow \text{Integer}_0
\Rightarrow \text{(state } U \text{ ((A\_key } \times \text{ Integer } \_0) \times ((B\_key } \times \text{ oid}) \times '{a\ up + 'b\ up + 'c}') \times
\text{state } U \text{ ((A\_key } \times \text{ Integer } \_0) \times ((B\_key } \times \text{ oid}) \times '{a\ up + 'b\ up + 'c})
\Rightarrow \text{Boolean}_0
\]
\[
\text{post1 } \equiv \text{ post}_0
\]

For class \textit{B} no initial values or invariants are given, thus we generate invariants that are just \texttt{true}:

encoding.B.inv.init\_j ::
\[
\text{(state } U \text{ ((A\_key } \times \text{ Integer } \_0) \times ((B\_key } \times \text{ oid}) \times '{a\ up + 'b\ up + 'c}') \times
\text{state } U \text{ ((A\_key } \times \text{ Integer } \_0) \times ((B\_key } \times \text{ oid}) \times '{a\ up + 'b\ up + 'c})
\Rightarrow \text{OclAny}_0 \text{ ((A\_key } \times \text{ Integer } \_0) \times ((B\_key } \times \text{ oid}) \times '{a\ up))
\Rightarrow \text{state } U \text{ ((A\_key } \times \text{ Integer } \_0) \times ((B\_key } \times \text{ oid}) \times '{a\ up + 'b\ up + 'c}') \times
\text{state } U \text{ ((A\_key } \times \text{ Integer } \_0) \times ((B\_key } \times \text{ oid}) \times '{a\ up + 'b\ up + 'c})
\Rightarrow \text{Boolean}_0
\]
\[
\text{inv.init\_j } \equiv \lambda \text{self. } \texttt{T}
\]

encoding.B.inv.init\_j ::
\[
\text{(state } U \text{ ((A\_key } \times \text{ Integer } \_0) \times ((B\_key } \times \text{ oid}) \times '{a\ up + 'b\ up + 'c}') \times
\text{state } U \text{ ((A\_key } \times \text{ Integer } \_0) \times ((B\_key } \times \text{ oid}) \times '{a\ up + 'b\ up + 'c})
\Rightarrow \text{OclAny}_0 \text{ ((A\_key } \times \text{ Integer } \_0) \times ((B\_key } \times \text{ oid}) \times '{a\ up))
\Rightarrow \text{state } U \text{ ((A\_key } \times \text{ Integer } \_0) \times ((B\_key } \times \text{ oid}) \times '{a\ up + 'b\ up + 'c}') \times
\text{state } U \text{ ((A\_key } \times \text{ Integer } \_0) \times ((B\_key } \times \text{ oid}) \times '{a\ up + 'b\ up + 'c})
\Rightarrow \text{Boolean}_0
\]
\[
\text{inv.init\_j } \equiv \lambda \text{self. } \texttt{T}
\]
Appendix C. Encoding UML/ocl by Example

The overall invariant for class $B$ is constructed by conjoining these invariants.

$$encoding.B.inv.inv :: (state U ((A\_key \times \text{Integer}_0) \times ((B\_key \times \text{oid}) \times 'a up + 'b) up + 'c) \times state U ((A\_key \times \text{Integer}_0) \times ((B\_key \times \text{oid}) \times 'a up + 'b) up + 'c) \Rightarrow \text{OclAny}_0 ((A\_key \times \text{Integer}_0) \times ((B\_key \times \text{oid}) \times 'a up) up)) \Rightarrow state U ((A\_key \times \text{Integer}_0) \times ((B\_key \times \text{oid}) \times 'a up + 'b) up + 'c) \times state U ((A\_key \times \text{Integer}_0) \times ((B\_key \times \text{oid}) \times 'a up) up + 'c) \Rightarrow \text{Boolean}_0 B.inv.inv \equiv inv.init_j$$

Encoding Of Operations

First we define a constant for the operation $f$ describing its core semantics specified by the pre-condition and post-condition:

$$encoding.A.f :: (state U ((A\_key \times \text{Integer}_0) \times ((B\_key \times \text{oid}) \times 'a up + 'b) up + 'c) \times state U ((A\_key \times \text{Integer}_0) \times ((B\_key \times \text{oid}) \times 'a up + 'b) up + 'c) \Rightarrow \text{OclAny}_0 'd) \Rightarrow (state U ((A\_key \times \text{Integer}_0) \times ((B\_key \times \text{oid}) \times 'a up + 'b) up + 'c) \times state U ((A\_key \times \text{Integer}_0) \times ((B\_key \times \text{oid}) \times 'a up + 'b) up + 'c) \Rightarrow \text{Integer}_0) \Rightarrow (state U ((A\_key \times \text{Integer}_0) \times ((B\_key \times \text{oid}) \times 'a up + 'b) up + 'c) \times state U ((A\_key \times \text{Integer}_0) \times ((B\_key \times \text{oid}) \times 'a up + 'b) up + 'c) \Rightarrow \text{Integer}_0) \Rightarrow state U ((A\_key \times \text{Integer}_0) \times ((B\_key \times \text{oid}) \times 'a up + 'b) up + 'c) \times state U ((A\_key \times \text{Integer}_0) \times ((B\_key \times \text{oid}) \times 'a up + 'b) up + 'c) \Rightarrow \text{Boolean}_0) f \equiv \lambda self x result. \text{pre1 self x} \land \text{post1 self x result}$$

460
C.3. Importing \texttt{UML/OCL} models

For each method, we create a table storing all variants, i.e., overloading.

\texttt{UML/OCL.ops.f\_Integer\_Integer.tab} ::

\begin{align*}
\text{\textquote[\prime]{a} set} & \mapsto \\
\text{(state } U \text{ (}(A\text{\_key} \times \text{Integer}\_0) \times ((B\text{\_key} \times \text{oid}) \times \text{\textquote[\prime]{b} up + \textquote[\prime]{c} up + \textquote[\prime]{d}) \times} & \Rightarrow \text{OclAny\_0 \textquote[\prime]{a}} \\
\text{state } U & \text{(}(A\text{\_key} \times \text{Integer}\_0) \times ((B\text{\_key} \times \text{oid}) \times \text{\textquote[\prime]{b} up + \textquote[\prime]{c} up + \textquote[\prime]{d})} \Rightarrow \text{Integer\_0} \\
\text{state } U & \text{(}(A\text{\_key} \times \text{Integer}\_0) \times ((B\text{\_key} \times \text{oid}) \times \text{\textquote[\prime]{b} up + \textquote[\prime]{c} up + \textquote[\prime]{d})} \Rightarrow \text{Integer\_0} \\
\text{state } U & \text{(}(A\text{\_key} \times \text{Integer}\_0) \times ((B\text{\_key} \times \text{oid}) \times \text{\textquote[\prime]{b} up + \textquote[\prime]{c} up + \textquote[\prime]{d})} \Rightarrow \text{Boolean\_0}
\end{align*}

The entry for \texttt{f} in this operation table is defined as follows:

\texttt{UML/OCL.ops.f\_Integer\_Integer.Op} ::

\begin{align*}
\text{(state } U \text{ (}(A\text{\_key} \times \text{Integer}\_0) \times ((B\text{\_key} \times \text{oid}) \times \text{\textquote[\prime]{a} up + \textquote[\prime]{b} up + \textquote[\prime]{c})} & \Rightarrow \text{OclAny\_0 \textquote[\prime]{d}} \\
\text{state } U & \text{(}(A\text{\_key} \times \text{Integer}\_0) \times ((B\text{\_key} \times \text{oid}) \times \text{\textquote[\prime]{a} up + \textquote[\prime]{b} up + \textquote[\prime]{c})} \Rightarrow \text{Integer\_0} \\
\text{state } U & \text{(}(A\text{\_key} \times \text{Integer}\_0) \times ((B\text{\_key} \times \text{oid}) \times \text{\textquote[\prime]{a} up + \textquote[\prime]{b} up + \textquote[\prime]{c})} \Rightarrow \text{Integer\_0} \\
\text{state } U & \text{(}(A\text{\_key} \times \text{Integer}\_0) \times ((B\text{\_key} \times \text{oid}) \times \text{\textquote[\prime]{a} up + \textquote[\prime]{b} up + \textquote[\prime]{c})} \Rightarrow \text{Integer\_0} \\
\text{Op} & \equiv \lambda \text{self } x. \text{OCL\_choose (OCL\_invokeS arbitrary tab self } x) \\
\text{tab (range encoding.mk } A) & \equiv \text{Some } f
\end{align*}

and as \texttt{UML/OCL.ops.f\_Integer\_Integer.overwrite\_A}:

\begin{align*}
\text{\texttt{tab (range encoding.mk } A) } & \equiv \text{Some } f
\end{align*}

(C.1)
Appendix C. Encoding UML/OCL by Example

Encoding the Basic UML part

The encoding of the data model on level 2 follows strictly the known schema from level 0 and 1, i.e., we also start two type tests for objects, one for testing if a object is of a specific type or subtype (i.e., this test is true, if the object is of the given kind):

\[ \text{encoding.is}_A : \]
\[
\begin{align*}
\text{state } U (\langle A \_key \times \text{Integer}_0 \rangle \times (\langle B \_key \times \text{oid} \rangle \times \langle 'a \text{ up} + 'b \text{ up} + 'c \rangle \times \\
\text{state } U (\langle A \_key \times \text{Integer}_0 \rangle \times (\langle B \_key \times \text{oid} \rangle \times \langle 'a \text{ up} + 'b \text{ up} + 'c \rangle \\
\Rightarrow \text{OclAny}_0 (\langle A \_key \times \text{Integer}_0 \rangle \times (\langle B \_key \times \text{oid} \rangle \times \langle 'a \text{ up} + 'b \text{ up} \rangle \\
\Rightarrow \text{state } U (\langle A \_key \times \text{Integer}_0 \rangle \times (\langle B \_key \times \text{oid} \rangle \times \langle 'a \text{ up} + 'b \text{ up} + 'c \rangle \\
\text{state } U (\langle A \_key \times \text{Integer}_0 \rangle \times (\langle B \_key \times \text{oid} \rangle \times \langle 'a \text{ up} + 'b \text{ up} + 'c \rangle \\
\Rightarrow \text{Boolean}_0
\end{align*}
\]

\[ \text{encoding.is}_A \equiv \lambda x. \text{if } A \text{.model.inv } x \text{ then level1.encoding.is }_A \text{ else } \text{Fendif} \]

and one for testing if a object is exactly the requested type

\[ \text{encoding.isType}_A : \]
\[
\begin{align*}
\text{state } U (\langle A \_key \times \text{Integer}_0 \rangle \times (\langle B \_key \times \text{oid} \rangle \times \langle 'a \text{ up} + 'b \text{ up} + 'c \rangle \times \\
\text{state } U (\langle A \_key \times \text{Integer}_0 \rangle \times (\langle B \_key \times \text{oid} \rangle \times \langle 'a \text{ up} + 'b \text{ up} + 'c \rangle \\
\Rightarrow \text{OclAny}_0 (\langle A \_key \times \text{Integer}_0 \rangle \times (\langle B \_key \times \text{oid} \rangle \times \langle 'a \text{ up} + 'b \text{ up} \rangle \\
\Rightarrow \text{state } U (\langle A \_key \times \text{Integer}_0 \rangle \times (\langle B \_key \times \text{oid} \rangle \times \langle 'a \text{ up} + 'b \text{ up} + 'c \rangle \\
\text{state } U (\langle A \_key \times \text{Integer}_0 \rangle \times (\langle B \_key \times \text{oid} \rangle \times \langle 'a \text{ up} + 'b \text{ up} + 'c \rangle \\
\Rightarrow \text{Boolean}_0
\end{align*}
\]

\[ \text{encoding.isType}_A \equiv \lambda x. \text{if } A \text{.model.inv } x \text{ then level1.encoding.isType }_A \text{ else } \text{Fendif} \]

Further, we define a type for universe instances:

\[ \text{encoding.is}_A \_\text{univ} : \]
\[
\begin{align*}
\text{state } U (\langle A \_key \times \text{Integer}_0 \rangle \times (\langle B \_key \times \text{oid} \rangle \times \langle 'a \text{ up} + 'b \text{ up} + 'c \rangle \times \\
\text{state } U (\langle A \_key \times \text{Integer}_0 \rangle \times (\langle B \_key \times \text{oid} \rangle \times \langle 'a \text{ up} + 'b \text{ up} + 'c \rangle \\
\Rightarrow U (\langle A \_key \times \text{Integer}_0 \rangle \times (\langle B \_key \times \text{oid} \rangle \times \langle 'a \text{ up} + 'b \text{ up} + 'c \rangle) \\
\Rightarrow \text{state } U (\langle A \_key \times \text{Integer}_0 \rangle \times (\langle B \_key \times \text{oid} \rangle \times \langle 'a \text{ up} + 'b \text{ up} + 'c \rangle \times \\
\text{state } U (\langle A \_key \times \text{Integer}_0 \rangle \times (\langle B \_key \times \text{oid} \rangle \times \langle 'a \text{ up} + 'b \text{ up} + 'c \rangle \\
\Rightarrow \text{Boolean}_0
\end{align*}
\]

\[ \text{encoding.is}_A \_\text{univ} \equiv \\
\lambda x. \text{if } (A \text{.model.inv } \circ \text{level1.encoding.get }_A ) x \text{ then level1.encoding.is }_A \_\text{univ} \text{ else } \text{Fendif} \]
C.3. Importing UML/OCL models

Analogous we define for the inherited class B:

encoding.is_B ::
(state U ((A_key × Integer_0) × ((B_key × oid) × 'a up + 'b) up + 'c) ×
  state U ((A_key × Integer_0) × ((B_key × oid) × 'a up + 'b) up + 'c)
  ⇒ OclAny_0 ((A_key × Integer_0) × ((B_key × oid) × 'a up) up))
⇒ state U ((A_key × Integer_0) × ((B_key × oid) × 'a up + 'b) up + 'c) ×
  state U ((A_key × Integer_0) × ((B_key × oid) × 'a up + 'b) up + 'c)
  ⇒ Boolean_0

encoding.is_B ≡ λx. if B.model.inv x then level1.encoding.is_B x else Fendif

encoding.isType_B ::
(state U ((A_key × Integer_0) × ((B_key × oid) × 'a up + 'b) up + 'c) ×
  state U ((A_key × Integer_0) × ((B_key × oid) × 'a up + 'b) up + 'c)
  ⇒ OclAny_0 ((A_key × Integer_0) × ((B_key × oid) × 'a up) up))
⇒ state U ((A_key × Integer_0) × ((B_key × oid) × 'a up + 'b) up + 'c) ×
  state U ((A_key × Integer_0) × ((B_key × oid) × 'a up + 'b) up + 'c)
  ⇒ Boolean_0

encoding.isType_B ≡ λx. if B.model.inv x then level1.encoding.isType_B x else Fendif

encoding.is_B_univ ::
(state U ((A_key × Integer_0) × ((B_key × oid) × 'a up + 'b) up + 'c) ×
  state U ((A_key × Integer_0) × ((B_key × oid) × 'a up + 'b) up + 'c)
  ⇒ U ((A_key × Integer_0) × ((B_key × oid) × 'a up + 'b) up + 'c))
⇒ state U ((A_key × Integer_0) × ((B_key × oid) × 'a up + 'b) up + 'c) ×
  state U ((A_key × Integer_0) × ((B_key × oid) × 'a up + 'b) up + 'c)
  ⇒ Boolean_0

encoding.is_B_univ ≡
λx. if (B.model.inv ∘ level1.encoding.get_B) x then level1.encoding.is_B_univ x else Fendif
Appendix C. Encoding UML/\textit{OCL} by Example

We define a functions for embedding a class into the corresponding universe

\[
\text{encoding.mk}_A :: \\
\text{encoding.mk}_A :: (\text{state } U \ (A\_key \times \text{Integer}_0) \times (B\_key \times \text{oid}) \times 'a up + 'b) up + 'c) \times \text{state } U \ ((A\_key \times \text{Integer}_0) \times (B\_key \times \text{oid}) \times 'a up + 'b) up + 'c) \Rightarrow \text{OclAny}_0 ((A\_key \times \text{Integer}_0) \times (B\_key \times \text{oid}) \times 'a up + 'b) up + 'c) \Rightarrow \text{state } U \ ((A\_key \times \text{Integer}_0) \times (B\_key \times \text{oid}) \times 'a up + 'b) up + 'c) \\
\text{encoding.mk}_A \equiv \lambda x. \text{if } A.model.inv x \text{ then level1.encoding.mk}_A x \text{ else level1.encoding.mk}_A x \text{ endif}
\]

and one for extracting an object out of a given universe:

\[
\text{encoding.get}_A :: \\
\text{encoding.get}_A :: (\text{state } U \ (A\_key \times \text{Integer}_0) \times (B\_key \times \text{oid}) \times 'a up + 'b) up + 'c) \times \text{state } U \ ((A\_key \times \text{Integer}_0) \times (B\_key \times \text{oid}) \times 'a up + 'b) up + 'c) \Rightarrow U ((A\_key \times \text{Integer}_0) \times (B\_key \times \text{oid}) \times 'a up + 'b) up + 'c) \\
\Rightarrow \text{state } U \ ((A\_key \times \text{Integer}_0) \times (B\_key \times \text{oid}) \times 'a up + 'b) up + 'c) \\
\text{encoding.get}_A \equiv \lambda x. \text{if } (A.model.inv \circ \text{level1.encoding.get}_A) x \text{ then level1.encoding.get}_A x \text{ else } \text{ endif}
\]

These two functions can also be understood as conversion between object types and universe types.

Analogous we define for the inherited class B:

\[
\text{encoding.mk}_B :: \\
\text{encoding.mk}_B :: (\text{state } U \ (A\_key \times \text{Integer}_0) \times (B\_key \times \text{oid}) \times 'a up + 'b) up + 'c) \times \text{state } U \ ((A\_key \times \text{Integer}_0) \times (B\_key \times \text{oid}) \times 'a up + 'b) up + 'c) \Rightarrow \text{OclAny}_0 ((A\_key \times \text{Integer}_0) \times (B\_key \times \text{oid}) \times 'a up + 'b) up + 'c) \\
\Rightarrow \text{state } U \ ((A\_key \times \text{Integer}_0) \times (B\_key \times \text{oid}) \times 'a up + 'b) up + 'c) \\
\text{encoding.mk}_B \equiv \lambda x. \text{if } B.model.inv x \text{ then level1.encoding.mk}_B x \text{ else level1.encoding.mk}_B x \text{ endif}
\]
C.3. Importing UML/OCL models

encoding.get_B ::
(state U ((A_key × Integer_0) × ((B_key × oid) × 'a up + 'b) up + 'c) × state U ((A_key × Integer_0) × ((B_key × oid) × 'a up + 'b) up + 'c)) ⇒ U ((A_key × Integer_0) × ((B_key × oid) × 'a up + 'b) up + 'c)
⇒ state U ((A_key × Integer_0) × ((B_key × oid) × 'a up + 'b) up + 'c) × state U ((A_key × Integer_0) × ((B_key × oid) × 'a up + 'b) up + 'c)) ⇒ OclAny_0 ((A_key × Integer_0) × ((B_key × oid) × 'a up) up)
encoding.get_B ≡ λx. if (B.model.inv ◦ level1.encoding.get_B) x then level1.encoding.get_B x else ⊥ endif

For converting along type hierarchies (i.e. down-casting and up-casting) we provide for each class two functions for the conversion to the direct predecessor and direct successor in the class hierarchy:

encoding.A_2_OclAny ::
(state U ((A_key × Integer_0) × ((B_key × oid) × 'a up + 'b) up + 'c) × state U ((A_key × Integer_0) × ((B_key × oid) × 'a up + 'b) up + 'c)) ⇒ OclAny_0 ((A_key × Integer_0) × ((B_key × oid) × 'a up + 'b) up))
⇒ state U ((A_key × Integer_0) × ((B_key × oid) × 'a up + 'b) up + 'c) × state U ((A_key × Integer_0) × ((B_key × oid) × 'a up + 'b) up + 'c)) ⇒ OclAny_0 ((A_key × Integer_0) × ((B_key × oid) × 'a up + 'b) up + 'c)
encoding.A_2_OclAny ≡ λx. if A.model.inv x then level1.encoding.A_2_OclAny x else ⊥ endif

encoding.OclAny_2_A ::
(state U ((A_key × Integer_0) × ((B_key × oid) × 'a up + 'b) up + 'c) × state U ((A_key × Integer_0) × ((B_key × oid) × 'a up + 'b) up + 'c)) ⇒ OclAny_0 ((A_key × Integer_0) × ((B_key × oid) × 'a up + 'b) up + 'c))
⇒ state U ((A_key × Integer_0) × ((B_key × oid) × 'a up + 'b) up + 'c) × state U ((A_key × Integer_0) × ((B_key × oid) × 'a up + 'b) up + 'c)) ⇒ OclAny_0 ((A_key × Integer_0) × ((B_key × oid) × 'a up + 'b) up)
encoding.OclAny_2_A ≡ λx. if ⊤ then level1.encoding.OclAny_2_A x else ⊥ endif

465
Appendix C. Encoding UML/OCL by Example

Analogous we define for the inherited class B:

\[
\text{encoding.B}_2.A :: \\
\text{(state } U (\text{A}_\text{key} \times \text{Integer}_0) \times (\text{B}_\text{key} \times \text{oid}) \times (\text{a up} + \text{b}) \uparrow + \text{c}) \times \text{encoding.A}_i \equiv \lambda x. \text{if } B\text{.model.inv } x \text{ then level1}\text{.encoding.B}_2.A \text{ else } _\text{endif}
\]

\[
\text{encoding.A}_2.B :: \\
\text{(state } U (\text{A}_\text{key} \times \text{Integer}_0) \times (\text{B}_\text{key} \times \text{oid}) \times (\text{a up} + \text{b}) \uparrow + \text{c}) \times \text{encoding.A}_i \equiv \lambda x. \text{if } A\text{.model.inv } x \text{ then level1}\text{.encoding.A}_2.B \text{ else } _\text{endif}
\]

Finally, we provide accessor functions for the class attributes:

\[
\text{encoding.A}_i :: \\
\text{(state } U (\text{A}_\text{key} \times \text{Integer}_0) \times (\text{B}_\text{key} \times \text{oid}) \times (\text{a up} + \text{b}) \uparrow + \text{c}) \times \text{encoding.A}_i \equiv \lambda x. \text{if } A\text{.model.inv } x \text{ then level1}\text{.encoding.A}_i \text{ else } _\text{endif}
\]
encoding.B.j ::

(state U ((A_key × Integer_0) × ((B_key × oid) × 'a up + 'b) up + 'c) × state U ((A_key × Integer_0) × ((B_key × oid) × 'a up + 'b) up + 'c) ⇒ OclAny_0 ((A_key × Integer_0) × ((B_key × oid) × 'a up) up))
⇒ state U ((A_key × Integer_0) × ((B_key × oid) × 'a up + 'b) up + 'c) × state U ((A_key × Integer_0) × ((B_key × oid) × 'a up + 'b) up + 'c) ⇒ OclAny_0 ((A_key × Integer_0) × ((B_key × oid) × 'a up) up)

encoding.B.j ≡ λx. if B.model.inv x then level1.encoding.B.j x else ⊥endif

**Defining oclAsType and allInstances**

The constants for oclAsType and allInstances must be defined for each class separately (using overloading). We provide for each class two definitions for oclAsType:
one for the actual state (“UML_OCL.univ.encoding.A.OclAllInstancesencoding_A“)

OclAllInstances ?self ≡ λ(s, s′). Abs_Set_0.level0.encoding.get_A ↑ ran s′

and one for the previous state (“UML_OCL.univ.encoding.A.OclAllInstancesencoding_AAtpre“):

OclAllInstancesAtpre ?self ≡ λ(s, s′). Abs_Set_0.level0.encoding.get_A ↑ ran s

Analogus for class B we define “UML_OCL.univ.encoding.B.OclAllInstancesencoding_B”:

OclAllInstances ?self ≡ λ(s, s′). Abs_Set_0.level0.encoding.get_B ↑ ran s′

and ”UML_OCL.univ.encoding.B.OclAllInstancesencoding_BAtpre“:

OclAllInstancesAtpre ?self ≡ λ(s, s′). Abs_Set_0.level0.encoding.get_B ↑ ran s

In a similar way, we have to overload the definition of oclAsType, based on the already defined conversion for adjacent classes. For class A the following cases must be considered along the inheritance hierarchy: The downcast from OclAny “UML_OCL.univ.encoding.A.from_OclAny”:

OclAsType ≡ λself cSet. level1.encoding.OclAny_2_A self

the upcast to OclAny “UML_OCL.univ.encoding.A.to_OclAny”:

OclAsType ≡ λself cSet. level1.encoding.A_2_OclAny self
Appendix C. Encoding UML/ OCL by Example

and the identity “UML_OCL.univ.encoding.A.to_A”:

\[ OcAsType \equiv \lambda self \ cSet. \ self \]

For class B more cases have to be considered, i.e., the downcast from A “UML_OCL.univ.encoding.B.from_A”:

\[ OcAsType \equiv \lambda self \ cSet. \ level1.encoding.A_2_B \ self \]

the downcast from OclAny “UML_OCL.univ.encoding.B.from_OclAny”:

\[ OcAsType \equiv \lambda self \ cSet. \ level1.encoding.A_2_OclAny \ self \]

the upcast to OclAny “UML_OCL.univ.encoding.B.to_OclAny”:

\[ OcAsType \equiv \lambda self \ cSet. \ level1.encoding.B_2_A \ self \]

and the identity “UML_OCL.univ.encoding.B.to_B”:

\[ OcAsType \equiv \lambda self \ cSet. \ self \]

Optional, the encoder generates all definitions that are orthogonal to the class hierarchy. On the one hand, they are defined to be OclUndefined according to the OCL standard, on the other hand, they violate a well-formed constraint.

Encoding of OCL Invariants: Part Two

Finally, we generate the the final invariant, e.g., for class A we generate:

\[
encoding.A.inv.init_i ::
\begin{align*}
(state \ U \ ((A\_key \times \ Integer\_0) \times ((B\_key \times \ oid) \times \ 'a\ up + 'b) \ up + 'c)) & \times \\
state \ U \ ((A\_key \times \ Integer\_0) \times ((B\_key \times \ oid) \times \ 'a\ up + 'b) \ up + 'c) & \Rightarrow OclAny \_0 ((A\_key \times \ Integer\_0) \times ((B\_key \times \ oid) \times \ 'a\ up + 'b) \ up)) \\
state \ U \ ((A\_key \times \ Integer\_0) \times ((B\_key \times \ oid) \times \ 'a\ up + 'b) \ up + 'c) & \Rightarrow state \ U \ ((A\_key \times \ Integer\_0) \times ((B\_key \times \ oid) \times \ 'a\ up + 'b) \ up + 'c) \\
& \Rightarrow Boolean\_0
\end{align*}
\]

\[ inv.init_i \equiv \lambda self. \ OclIsNew self \rightarrow encoding.A.i self \triangleq 42 \]
and the second one describes the user defined invariant:

\[
\text{encoding.A.inv.inv}_1 ::
\]
\[
\text{state } U \ ((A\_key \times \text{Integer}_{-}0) \times (B\_key \times \text{oid}) \times 'a up + 'b) up + 'c) \times \\
\text{state } U \ ((A\_key \times \text{Integer}_{-}0) \times (B\_key \times \text{oid}) \times 'a up + 'b) up + 'c) \\
\Rightarrow \text{OclAny}_{-}0 \ ((A\_key \times \text{Integer}_{-}0) \times (B\_key \times \text{oid}) \times 'a up + 'b) up) \\
\Rightarrow \text{state } U \ ((A\_key \times \text{Integer}_{-}0) \times (B\_key \times \text{oid}) \times 'a up + 'b) up + 'c) \times \\
\text{state } U \ ((A\_key \times \text{Integer}_{-}0) \times (B\_key \times \text{oid}) \times 'a up + 'b) up + 'c) \\
\Rightarrow \text{Boolean}_{-}0 \\
\]

\[
\text{inv.inv}_{-}1 \equiv \lambda \text{self}. \text{encoding.A.i self} \geq 0 \\
\]

The overall invariant for class A is constructed by conjoining these invariants.

\[
\text{encoding.A.inv.inv ::}
\]
\[
\text{state } U \ ((A\_key \times \text{Integer}_{-}0) \times (B\_key \times \text{oid}) \times 'a up + 'b) up + 'c) \times \\
\text{state } U \ ((A\_key \times \text{Integer}_{-}0) \times (B\_key \times \text{oid}) \times 'a up + 'b) up + 'c) \\
\Rightarrow \text{OclAny}_{-}0 \ ((A\_key \times \text{Integer}_{-}0) \times (B\_key \times \text{oid}) \times 'a up + 'b) up) \\
\Rightarrow \text{state } U \ ((A\_key \times \text{Integer}_{-}0) \times (B\_key \times \text{oid}) \times 'a up + 'b) up + 'c) \times \\
\text{state } U \ ((A\_key \times \text{Integer}_{-}0) \times (B\_key \times \text{oid}) \times 'a up + 'b) up + 'c) \\
\Rightarrow \text{Boolean}_{-}0 \\
\]

\[
\text{A.inv.inv} \equiv \lambda \text{obj}. \text{inv.init}_i \text{ obj} \land \text{inv.inv}_{-}1 \text{ obj} \\
\]

Analogous we are encoding the pre-condition and post-condition of the operation f:

\[
\text{encoding.A.f}\_\text{Integer}\_\text{Integer}.\text{pre}_{-}1.\text{pre}_{-}0 ::
\]
\[
\text{state } U \ ((A\_key \times \text{Integer}_{-}0) \times (B\_key \times \text{oid}) \times 'a up + 'b) up + 'c) \times \\
\text{state } U \ ((A\_key \times \text{Integer}_{-}0) \times (B\_key \times \text{oid}) \times 'a up + 'b) up + 'c) \\
\Rightarrow \text{OclAny}_{-}0 \ ((A\_key \times \text{Integer}_{-}0) \times (B\_key \times \text{oid}) \times 'a up + 'b) up) \\
\Rightarrow (\text{state } U \ ((A\_key \times \text{Integer}_{-}0) \times (B\_key \times \text{oid}) \times 'a up + 'b) up + 'c) \times \\
\text{state } U \ ((A\_key \times \text{Integer}_{-}0) \times (B\_key \times \text{oid}) \times 'a up + 'b) up + 'c) \\
\Rightarrow \text{Integer}_{-}0 \\
\Rightarrow \text{state } U \ ((A\_key \times \text{Integer}_{-}0) \times (B\_key \times \text{oid}) \times 'a up + 'b) up + 'c) \times \\
\text{state } U \ ((A\_key \times \text{Integer}_{-}0) \times (B\_key \times \text{oid}) \times 'a up + 'b) up + 'c) \\
\Rightarrow \text{Boolean}_{-}0 \\
\]

\[
\text{pre}_{-}0 \equiv \lambda \text{self } x . x > 0 \\
\]
Appendix C. Encoding UML/\texttt{OCL} by Example

\begin{align*}
\text{encoding.A.f\_Integer\_Integer.post0} &::
\begin{array}{c}
\text{state } U ((\text{A.key } \times \text{Integer}_0) \times ((\text{B.key } \times \text{oid}) \times 'a up + 'b) up + 'c) \times \\
\text{state } U ((\text{A.key } \times \text{Integer}_0) \times ((\text{B.key } \times \text{oid}) \times 'a up + 'b) up + 'c) \\
\Rightarrow \text{OclAny}_0 ((\text{A.key } \times \text{Integer}_0) \times ((\text{B.key } \times \text{oid}) \times 'a up + 'b) up)) \\
\Rightarrow (\text{state } U ((\text{A.key } \times \text{Integer}_0) \times ((\text{B.key } \times \text{oid}) \times 'a up + 'b) up + 'c) \times \\
\text{state } U ((\text{A.key } \times \text{Integer}_0) \times ((\text{B.key } \times \text{oid}) \times 'a up + 'b) up + 'c) \\
\Rightarrow \text{Integer}_0 \\
\Rightarrow (\text{state } U ((\text{A.key } \times \text{Integer}_0) \times ((\text{B.key } \times \text{oid}) \times 'a up + 'b) up + 'c) \times \\
\text{state } U ((\text{A.key } \times \text{Integer}_0) \times ((\text{B.key } \times \text{oid}) \times 'a up + 'b) up + 'c) \\
\Rightarrow \text{Boolean}_0
\end{array}
\end{align*}

\begin{align*}
\text{post0} &\equiv \lambda self \ x. \ \text{result} \triangleq x \rightarrow \text{div}(\text{level1.encoding.A.i self})
\end{align*}

\begin{align*}
\text{encoding.A.f\_Integer\_Integer.pre1} &::
\begin{array}{c}
\text{state } U ((\text{A.key } \times \text{Integer}_0) \times ((\text{B.key } \times \text{oid}) \times 'a up + 'b) up + 'c) \times \\
\text{state } U ((\text{A.key } \times \text{Integer}_0) \times ((\text{B.key } \times \text{oid}) \times 'a up + 'b) up + 'c) \\
\Rightarrow \text{OclAny}_0 ((\text{A.key } \times \text{Integer}_0) \times ((\text{B.key } \times \text{oid}) \times 'a up + 'b) up)) \\
\Rightarrow (\text{state } U ((\text{A.key } \times \text{Integer}_0) \times ((\text{B.key } \times \text{oid}) \times 'a up + 'b) up + 'c) \times \\
\text{state } U ((\text{A.key } \times \text{Integer}_0) \times ((\text{B.key } \times \text{oid}) \times 'a up + 'b) up + 'c) \\
\Rightarrow \text{Integer}_0 \\
\Rightarrow (\text{state } U ((\text{A.key } \times \text{Integer}_0) \times ((\text{B.key } \times \text{oid}) \times 'a up + 'b) up + 'c) \times \\
\text{state } U ((\text{A.key } \times \text{Integer}_0) \times ((\text{B.key } \times \text{oid}) \times 'a up + 'b) up + 'c) \\
\Rightarrow \text{Boolean}_0
\end{array}
\end{align*}

\begin{align*}
\text{pre1} &\equiv \text{pre0}
\end{align*}
C.3. Importing UML/OCL models

encoding.A.inv.init

\(\text{encoding.A.f \_Integer \_Integer.post1 ::}
\)
\(\begin{align*}
\text{state } U \left( (A \_key \times \text{Integer} \_0) \times (B \_key \times \text{oid}) \times \text{`a up `b up `c} \times \text{state } U \left( (A \_key \times \text{Integer} \_0) \times (B \_key \times \text{oid}) \times \text{`a up `b up `c} \\
\Rightarrow \text{OclAny} \_0 \left( (A \_key \times \text{Integer} \_0) \times (B \_key \times \text{oid}) \times \text{`a up `b up`)}
\end{align*}\)

\(\Rightarrow \text{state } U \left( (A \_key \times \text{Integer} \_0) \times (B \_key \times \text{oid}) \times \text{`a up `b up `c} \\
\Rightarrow \text{Integer} \_0
\)

\(\Rightarrow \text{state } U \left( (A \_key \times \text{Integer} \_0) \times (B \_key \times \text{oid}) \times \text{`a up `b up `c} \\
\Rightarrow \text{Integer} \_0
\)

\(\Rightarrow \text{state } U \left( (A \_key \times \text{Integer} \_0) \times (B \_key \times \text{oid}) \times \text{`a up `b up `c} \\
\Rightarrow \text{Integer} \_0
\)

\(\Rightarrow \text{state } U \left( (A \_key \times \text{Integer} \_0) \times (B \_key \times \text{oid}) \times \text{`a up `b up `c} \\
\Rightarrow \text{Integer} \_0
\)

\(\Rightarrow \text{state } U \left( (A \_key \times \text{Integer} \_0) \times (B \_key \times \text{oid}) \times \text{`a up `b up `c} \\
\Rightarrow \text{Integer} \_0
\)

\(\Rightarrow \text{state } U \left( (A \_key \times \text{Integer} \_0) \times (B \_key \times \text{oid}) \times \text{`a up `b up `c} \\
\Rightarrow \text{Boolean} \_0
\)

\text{post1} \equiv \text{post} \_0

For class B no initial values or invariants are given, thus we generate invariants that are just true:

encoding.B.inv.init \_j

\(\begin{align*}
\text{encoding.B.inv.init} \_j ::
\end{align*}\)

\(\begin{align*}
\text{encoding.B.inv.init} \_j ::
\end{align*}\)

\(\begin{align*}
\text{encoding.B.inv.init} \_j ::
\end{align*}\)

\(\begin{align*}
\text{encoding.B.inv.init} \_j ::
\end{align*}\)

\text{inv.init} \_j \equiv \lambda \text{self. } \text{T}

\text{inv.init} \_j \equiv \lambda \text{self. } \text{T}

\text{inv.init} \_j \equiv \lambda \text{self. } \text{T}

\text{inv.init} \_j \equiv \lambda \text{self. } \text{T}

471
The overall invariant for class B is constructed by conjoining these invariants.

\[
encoding.B.inv.inv ::
\begin{align*}
& (\text{state } U ((A\_key \times \text{Integer}_0) \times ((B\_key \times \text{oid}) \times 'a up + 'b) up + 'c) \times \\
& \quad \text{state } U ((A\_key \times \text{Integer}_0) \times ((B\_key \times \text{oid}) \times 'a up + 'b) up + 'c) \\
& \Rightarrow \text{OclAny}_0 ((A\_key \times \text{Integer}_0) \times ((B\_key \times \text{oid}) \times 'a up) up)
\end{align*}
\Rightarrow state U ((A\_key \times \text{Integer}_0) \times ((B\_key \times \text{oid}) \times 'a up + 'b) up + 'c) \times \\
\quad \text{state } U ((A\_key \times \text{Integer}_0) \times ((B\_key \times \text{oid}) \times 'a up + 'b) up + 'c) \\
& \Rightarrow \text{Boolean}_0
\]

\(B.inv.inv \equiv \text{inv.init}_j\)
Appendix D.

The GNU General Public Licence

The following is the text of the GNU General Public Licence, under the terms of which this software is distributed.

GNU GENERAL PUBLIC LICENSE
Version 2, June 1991
Copyright (C) 1989, 1991 Free Software Foundation, Inc.
675 Mass Ave, Cambridge, MA 02139, USA
Everyone is permitted to copy and distribute verbatim copies of this license document, but changing it is not allowed.

D.1. Preamble

The licenses for most software are designed to take away your freedom to share and change it. By contrast, the GNU General Public License is intended to guarantee your freedom to share and change free software—to make sure the software is free for all its users. This General Public License applies to most of the Free Software Foundation’s software and to any other program whose authors commit to using it. (Some other Free Software Foundation software is covered by the GNU Library General Public License instead.) You can apply it to your programs, too.

When we speak of free software, we are referring to freedom, not price. Our General Public Licenses are designed to make sure that you have the freedom to distribute copies of free software (and charge for this service if you wish), that you receive source code or can get it if you want it, that you can change the software or use pieces of it in new free programs; and that you know you can do these things.

To protect your rights, we need to make restrictions that forbid anyone to deny you these rights or to ask you to surrender the rights. These restrictions translate to certain responsibilities for you if you distribute copies of the software, or if you modify it.

For example, if you distribute copies of such a program, whether gratis or for a fee, you must give the recipients all the rights that you have. You must make sure that they, too, receive or can get the source code. And you must show them these terms so they know their rights.
Appendix D. The GNU General Public Licence

We protect your rights with two steps: (1) copyright the software, and (2) offer you this license which gives you legal permission to copy, distribute and/or modify the software.

Also, for each author’s protection and ours, we want to make certain that everyone understands that there is no warranty for this free software. If the software is modified by someone else and passed on, we want its recipients to know that what they have is not the original, so that any problems introduced by others will not reflect on the original authors’ reputations.

Finally, any free program is threatened constantly by software patents. We wish to avoid the danger that redistributors of a free program will individually obtain patent licenses, in effect making the program proprietary. To prevent this, we have made it clear that any patent must be licensed for everyone’s free use or not licensed at all.

The precise terms and conditions for copying, distribution and modification follow.

D.2. Terms and conditions for copying, distribution and modification

0. This License applies to any program or other work which contains a notice placed by the copyright holder saying it may be distributed under the terms of this General Public License. The “Program”, below, refers to any such program or work, and a “work based on the Program” means either the Program or any derivative work under copyright law: that is to say, a work containing the Program or a portion of it, either verbatim or with modifications and/or translated into another language. (Hereinafter, translation is included without limitation in the term “modification”.) Each licensee is addressed as “you”.

Activities other than copying, distribution and modification are not covered by this License; they are outside its scope. The act of running the Program is not restricted, and the output from the Program is covered only if its contents constitute a work based on the Program (independent of having been made by running the Program). Whether that is true depends on what the Program does.

1. You may copy and distribute verbatim copies of the Program’s source code as you receive it, in any medium, provided that you conspicuously and appropriately publish on each copy an appropriate copyright notice and disclaimer of warranty; keep intact all the notices that refer to this License and to the absence of any warranty; and give any other recipients of the Program a copy of this License along with the Program.

You may charge a fee for the physical act of transferring a copy, and you may at your option offer warranty protection in exchange for a fee.

2. You may modify your copy or copies of the Program or any portion of it, thus forming a work based on the Program, and copy and distribute such modifications or work under the terms of Section 1 above, provided that you also meet all of these conditions:
D.2. Terms and conditions for copying, distribution and modification

(a) You must cause the modified files to carry prominent notices stating that you changed the files and the date of any change.

(b) You must cause any work that you distribute or publish, that in whole or in part contains or is derived from the Program or any part thereof, to be licensed as a whole at no charge to all third parties under the terms of this License.

(c) If the modified program normally reads commands interactively when run, you must cause it, when started running for such interactive use in the most ordinary way, to print or display an announcement including an appropriate copyright notice and a notice that there is no warranty (or else, saying that you provide a warranty) and that users may redistribute the program under these conditions, and telling the user how to view a copy of this License. (Exception: if the Program itself is interactive but does not normally print such an announcement, your work based on the Program is not required to print an announcement.)

These requirements apply to the modified work as a whole. If identifiable sections of that work are not derived from the Program, and can be reasonably considered independent and separate works in themselves, then this License, and its terms, do not apply to those sections when you distribute them as separate works. But when you distribute the same sections as part of a whole which is a work based on the Program, the distribution of the whole must be on the terms of this License, whose permissions for other licensees extend to the entire whole, and thus to each and every part regardless of who wrote it.

Thus, it is not the intent of this section to claim rights or contest your rights to work written entirely by you; rather, the intent is to exercise the right to control the distribution of derivative or collective works based on the Program.

In addition, mere aggregation of another work not based on the Program with the Program (or with a work based on the Program) on a volume of a storage or distribution medium does not bring the other work under the scope of this License.

3. You may copy and distribute the Program (or a work based on it, under Section 2) in object code or executable form under the terms of Sections 1 and 2 above provided that you also do one of the following:

(a) Accompany it with the complete corresponding machine-readable source code, which must be distributed under the terms of Sections 1 and 2 above on a medium customarily used for software interchange; or,

(b) Accompany it with a written offer, valid for at least three years, to give any third party, for a charge no more than your cost of physically performing source distribution, a complete machine-readable copy of the corresponding
Appendix D. The GNU General Public Licence

source code, to be distributed under the terms of Sections 1 and 2 above on a medium customarily used for software interchange; or,

(c) Accompany it with the information you received as to the offer to distribute corresponding source code. (This alternative is allowed only for noncommercial distribution and only if you received the program in object code or executable form with such an offer, in accord with Subsection b above.)

The source code for a work means the preferred form of the work for making modifications to it. For an executable work, complete source code means all the source code for all modules it contains, plus any associated interface definition files, plus the scripts used to control compilation and installation of the executable. However, as a special exception, the source code distributed need not include anything that is normally distributed (in either source or binary form) with the major components (compiler, kernel, and so on) of the operating system on which the executable runs, unless that component itself accompanies the executable.

If distribution of executable or object code is made by offering access to copy from a designated place, then offering equivalent access to copy the source code from the same place counts as distribution of the source code, even though third parties are not compelled to copy the source along with the object code.

4. You may not copy, modify, sublicense, or distribute the Program except as expressly provided under this License. Any attempt otherwise to copy, modify, sublicense or distribute the Program is void, and will automatically terminate your rights under this License. However, parties who have received copies, or rights, from you under this License will not have their licenses terminated so long as such parties remain in full compliance.

5. You are not required to accept this License, since you have not signed it. However, nothing else grants you permission to modify or distribute the Program or its derivative works. These actions are prohibited by law if you do not accept this License. Therefore, by modifying or distributing the Program (or any work based on the Program), you indicate your acceptance of this License to do so, and all its terms and conditions for copying, distributing or modifying the Program or works based on it.

6. Each time you redistribute the Program (or any work based on the Program), the recipient automatically receives a license from the original licensor to copy, distribute or modify the Program subject to these terms and conditions. You may not impose any further restrictions on the recipients' exercise of the rights granted herein. You are not responsible for enforcing compliance by third parties to this License.

7. If, as a consequence of a court judgment or allegation of patent infringement or for any other reason (not limited to patent issues), conditions are imposed on you
D.2. Terms and conditions for copying, distribution and modification

(whether by court order, agreement or otherwise) that contradict the conditions of this License, they do not excuse you from the conditions of this License. If you cannot distribute so as to satisfy simultaneously your obligations under this License and any other pertinent obligations, then as a consequence you may not distribute the Program at all. For example, if a patent license would not permit royalty-free redistribution of the Program by all those who receive copies directly or indirectly through you, then the only way you could satisfy both it and this License would be to refrain entirely from distribution of the Program.

If any portion of this section is held invalid or unenforceable under any particular circumstance, the balance of the section is intended to apply and the section as a whole is intended to apply in other circumstances.

It is not the purpose of this section to induce you to infringe any patents or other property right claims or to contest validity of any such claims; this section has the sole purpose of protecting the integrity of the free software distribution system, which is implemented by public license practices. Many people have made generous contributions to the wide range of software distributed through that system in reliance on consistent application of that system; it is up to the author/donor to decide if he or she is willing to distribute software through any other system and a licensee cannot impose that choice.

This section is intended to make thoroughly clear what is believed to be a consequence of the rest of this License.

8. If the distribution and/or use of the Program is restricted in certain countries either by patents or by copyrighted interfaces, the original copyright holder who places the Program under this License may add an explicit geographical distribution limitation excluding those countries, so that distribution is permitted only in or among countries not thus excluded. In such case, this License incorporates the limitation as if written in the body of this License.

9. The Free Software Foundation may publish revised and/or new versions of the General Public License from time to time. Such new versions will be similar in spirit to the present version, but may differ in detail to address new problems or concerns.

Each version is given a distinguishing version number. If the Program specifies a version number of this License which applies to it and “any later version”, you have the option of following the terms and conditions either of that version or of any later version published by the Free Software Foundation. If the Program does not specify a version number of this License, you may choose any version ever published by the Free Software Foundation.

10. If you wish to incorporate parts of the Program into other free programs whose distribution conditions are different, write to the author to ask for permission. For software which is copyrighted by the Free Software Foundation, write to the Free Software Foundation; we sometimes make exceptions for this. Our decision
Appendix D. The GNU General Public Licence

will be guided by the two goals of preserving the free status of all derivatives of our free software and of promoting the sharing and reuse of software generally.

NO WARRANTY

11. Because the Program is licensed free of charge, there is no warranty for the Program, to the extent permitted by applicable law. except when otherwise stated in writing the copyright holders and/or other parties provide the program “as is” without warranty of any kind, either expressed or implied, including, but not limited to, the implied warranties of merchantability and fitness for a particular purpose. The entire risk as to the quality and performance of the Program is with you. Should the Program prove defective, you assume the cost of all necessary servicing, repair or correction.

12. In no event unless required by applicable law or agreed to in writing will any copyright holder, or any other party who may modify and/or redistribute the program as permitted above, be liable to you for damages, including any general, special, incidental or consequential damages arising out of the use or inability to use the program (including but not limited to loss of data or data being rendered inaccurate or losses sustained by you or third parties or a failure of the Program to operate with any other programs), even if such holder or other party has been advised of the possibility of such damages.

END OF TERMS AND CONDITIONS

D.3. Appendix: How to Apply These Terms to Your New Programs

If you develop a new program, and you want it to be of the greatest possible use to the public, the best way to achieve this is to make it free software which everyone can redistribute and change under these terms.

To do so, attach the following notices to the program. It is safest to attach them to the start of each source file to most effectively convey the exclusion of warranty; and each file should have at least the “copyright” line and a pointer to where the full notice is found.

<one line to give the program’s name and a brief idea of what it does.>
Copyright (C) 19yy <name of author>

This program is free software; you can redistribute it and/or modify it under the terms of the GNU General Public License as published by the Free Software Foundation; either version 2 of the License, or
D.3. Appendix: How to Apply These Terms to Your New Programs

(at your option) any later version.

This program is distributed in the hope that it will be useful, but WITHOUT ANY WARRANTY; without even the implied warranty of MERCHANTABILITY or FITNESS FOR A PARTICULAR PURPOSE. See the GNU General Public License for more details.

You should have received a copy of the GNU General Public License along with this program; if not, write to the Free Software Foundation, Inc., 675 Mass Ave, Cambridge, MA 02139, USA.

Also add information on how to contact you by electronic and paper mail.

If the program is interactive, make it output a short notice like this when it starts in an interactive mode:

Gnomovision version 69, Copyright (C) 19yy name of author
Gnomovision comes with ABSOLUTELY NO WARRANTY; for details type ‘show w’.
This is free software, and you are welcome to redistribute it under certain conditions; type ‘show c’ for details.

The hypothetical commands ‘show w’ and ‘show c’ should show the appropriate parts of the General Public License. Of course, the commands you use may be called something other than ‘show w’ and ‘show c’; they could even be mouse-clicks or menu items—whatever suits your program.

You should also get your employer (if you work as a programmer) or your school, if any, to sign a “copyright disclaimer” for the program, if necessary. Here is a sample; alter the names:

Yoyodyne, Inc., hereby disclaims all copyright interest in the program ‘Gnomovision’ (which makes passes at compilers) written by James Hacker.

<signature of Ty Coon>, 1 April 1989
Ty Coon, President of Vice

This General Public License does not permit incorporating your program into proprietary programs. If your program is a subroutine library, you may consider it more useful to permit linking proprietary applications with the library. If this is what you want to do, use the GNU Library General Public License instead of this License.
Appendix D. The GNU General Public Licence
Part V.

Bibliography and Index
Bibliography


[3] Isabelle. URL http://isabelle.in.tum.de (Cited on pages 12 and 91.)


Bibliography


Bibliography


486
Bibliography


## List of Acronyms

<table>
<thead>
<tr>
<th>Acronym</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>ACI</td>
<td>Associativity, commutativity, and idempotency</td>
</tr>
<tr>
<td>CASE</td>
<td>Computer Aided Software Engineering</td>
</tr>
<tr>
<td>CCS</td>
<td>A Calculus of Communicating Systems</td>
</tr>
<tr>
<td>CTL</td>
<td>Computational Tree Logic</td>
</tr>
<tr>
<td>CSP</td>
<td>Communicating Sequential Processes</td>
</tr>
<tr>
<td>DNF</td>
<td>disjunctive normal form</td>
</tr>
<tr>
<td>ETH</td>
<td>Swiss Federal Institute of Technology</td>
</tr>
<tr>
<td>EBNF</td>
<td>Extended Backus-Naur Form</td>
</tr>
<tr>
<td>FXP</td>
<td>Functional XML Parser</td>
</tr>
<tr>
<td>GNU</td>
<td>GNU is not Unix</td>
</tr>
<tr>
<td>GUI</td>
<td>Graphical User Interface</td>
</tr>
<tr>
<td>HOAS</td>
<td>higher-order abstract syntax</td>
</tr>
<tr>
<td>HOL</td>
<td>higher-order logic</td>
</tr>
<tr>
<td>Isar</td>
<td>Intelligible semi-automated reasoning</td>
</tr>
<tr>
<td>LCF</td>
<td>Logic for Computable Functions</td>
</tr>
<tr>
<td>LE</td>
<td>Local (Formula) Equivalence</td>
</tr>
<tr>
<td>LJE</td>
<td>Local Judgement Equivalence</td>
</tr>
<tr>
<td>LTL</td>
<td>Linear Temporal Logic</td>
</tr>
<tr>
<td>LTC</td>
<td>Local Judgement Tableaux Calculus</td>
</tr>
<tr>
<td>OCL</td>
<td>Object Constraint Language</td>
</tr>
<tr>
<td>OMG</td>
<td>Object Management Group</td>
</tr>
<tr>
<td>SKL</td>
<td>Strong Kleene Logic</td>
</tr>
<tr>
<td>SML</td>
<td>Standard Meta Language</td>
</tr>
<tr>
<td>Acronym</td>
<td>Description</td>
</tr>
<tr>
<td>---------</td>
<td>-------------</td>
</tr>
<tr>
<td>sml/NJ</td>
<td>SML of New Jersey</td>
</tr>
<tr>
<td>su4sml</td>
<td>SecureUML for SML</td>
</tr>
<tr>
<td>UC</td>
<td>Universal (Formula) Congruence</td>
</tr>
<tr>
<td>UJE</td>
<td>Universal Judgement Equivalence</td>
</tr>
<tr>
<td>UML</td>
<td>Unified Modelling Language</td>
</tr>
<tr>
<td>VDM</td>
<td>Vienna Definition Method</td>
</tr>
<tr>
<td>XMI</td>
<td>XML Metadata Interchange</td>
</tr>
<tr>
<td>XML</td>
<td>Extensible Markup Language</td>
</tr>
<tr>
<td>ZF</td>
<td>Zermelo-Fränkel</td>
</tr>
<tr>
<td>ZFC</td>
<td>Zermelo-Fränkel set theory with the axiom of choice.</td>
</tr>
</tbody>
</table>
List of Glitches

The formal semantics of implies does not satisfy its requirements. 42
Underspecification of Overload-Resolution 47
referential vs. non-referential universe construction 52
Underspecification of Recursion 68
self.OclIsNew() ill-defined 74
The truth for implies does not fulfill the normative requirement. 177
Definition for implies inappropriate 192
Typo in requirement 11.7.5.10, 291
No specification of the returned index of indexOf 291
sum, 367
Wrong definition of the correspondence of includesAll and iterate 413
Wrong definition of the correspondence of excludesAll and iterate 414
no preconditions about range of index. But they are surely needed because of the correspondence to at(), 426
the valid range must be from index to self->size() 426
Bibliography
List of Extensions

late-binding invocation, 47
Characteristic Type Sets, 64
infinite set, 64
typeSetOf(), 64
kindSetOf(), 64
late-binding invocation, 65
strong \(\triangleq\) and strict \(\equiv\) referential equality, 74
strong \(\triangleq\) and strict \(\dot\equiv\) shallow value equality, 74
->modifiedOnly(), 75
## List of Figures

2.1. Modeling a simple banking scenario with UML .......................... 21

4.1. An High-Level Overview of the HOL-OCL Embedding Architecture ... 37
4.2. Derivation of the OCL-library ........................................... 44
4.3. Extending Class Hierarchies and Universes with Holes .................. 50
4.4. Modelling Directed Graphs: Data Model ............................... 53
4.5. The Relation Between Type Casts and Injections and Projections ... 59

6.1. Overview of the HOL-OCL architecture ................................. 98

7.1. Overview of the High-level System Architecture of HOL-OCL ........... 107
7.2. A HOL-OCL session Using the Isar Interface of Isabelle ................ 111

9.1. The HOL-OCL Workflow .................................................. 118
9.2. Using ArgoUML for Data Modelling ..................................... 118
9.3. Using the Dresden OCL Parser and Type-checker ....................... 120

10.1. Modelling a Stack: Data Model ......................................... 125

B.1. Session Graph ............................................................ 136

C.1. A simple Encoding Example: Data Model ............................... 440
List of Figures
## List of Tables

<table>
<thead>
<tr>
<th>Table</th>
<th>Title</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>3.1</td>
<td>Comparing Logical Embeddings</td>
<td>32</td>
</tr>
<tr>
<td>4.1</td>
<td>Equalities in an Object-oriented Setting</td>
<td>74</td>
</tr>
<tr>
<td>4.2</td>
<td>Comparing Equalities (non-referential universe)</td>
<td>74</td>
</tr>
<tr>
<td>4.3</td>
<td>Comparing Equalities (referential universe)</td>
<td>75</td>
</tr>
<tr>
<td>5.1</td>
<td>The Context Passing Side-Calculus (Excerpt)</td>
<td>81</td>
</tr>
<tr>
<td>5.2</td>
<td>The Definedness Calculi</td>
<td>83</td>
</tr>
<tr>
<td>5.3</td>
<td>The UC Core-Calculus (“Propositional Calculus”)</td>
<td>86</td>
</tr>
<tr>
<td>5.4</td>
<td>The OCL to HOL Conversion Rules</td>
<td>91</td>
</tr>
<tr>
<td>5.5</td>
<td>The Core of LTC</td>
<td>92</td>
</tr>
<tr>
<td>A.1</td>
<td>Formal Grammar of OCL (fragment)</td>
<td>130</td>
</tr>
<tr>
<td>A.2</td>
<td>Comparison of different concrete syntax variants for OCL</td>
<td>131</td>
</tr>
<tr>
<td>B.1</td>
<td>Comparison of the formal syntax and the Isabelle syntax</td>
<td>138</td>
</tr>
</tbody>
</table>
List of Tables
List of Definitions

4.1. Attribute Types .................................................. 54
4.2. Tag Types .......................................................... 54
4.3. Base Class Types .................................................. 54
4.4. OclAny ............................................................... 55
4.5. Universe Types .................................................... 55
4.6. Referential Universe Types ....................................... 55
4.7. Non-Referential Universe Types ................................. 56
4.8. Class Type .......................................................... 56
4.9. Finite Constant Definition Family .............................. 67
4.10. Value Types ......................................................... 71
4.11. Values .............................................................. 71
4.12. Equivalence Relation ............................................. 71
4.13. Reference Equality ................................................. 71
4.14. Shallow Value Equality .......................................... 71
4.15. Deep Value Equality .............................................. 72
4.16. Equivalence Operators ............................................ 72
4.17. Strong Equality .................................................... 73
4.18. Strict Equality ..................................................... 73
List of Tables
## Listings

<table>
<thead>
<tr>
<th>Section</th>
<th>Title</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>6.1.</td>
<td>su4sml: Representing OCL types</td>
<td>98</td>
</tr>
<tr>
<td>6.2.</td>
<td>su4sml: Representing OCL expressions</td>
<td>99</td>
</tr>
<tr>
<td>6.3.</td>
<td>su4sml: Representing the UML core</td>
<td>100</td>
</tr>
<tr>
<td>6.4.</td>
<td>The Top-level Interface of the Repository Encoder</td>
<td>101</td>
</tr>
<tr>
<td>6.5.</td>
<td>Proving Cast and Re-Cast (simplified)</td>
<td>103</td>
</tr>
<tr>
<td>9.1.</td>
<td>A OCL Specification (Excerpt of the Royals and Loyals Model)</td>
<td>119</td>
</tr>
<tr>
<td>9.2.</td>
<td>A simple HOL-OCL Theory File</td>
<td>121</td>
</tr>
<tr>
<td>10.1.</td>
<td>Modelling a Stack: OCL specification</td>
<td>126</td>
</tr>
<tr>
<td>C.1.</td>
<td>A simple Encoding Example: OCL specification</td>
<td>439</td>
</tr>
</tbody>
</table>
Index

symbols
\( V_{\alpha} \), 38
\texttt{sml/NJ}, 107

A
abstraction, 17
activity charts, 12
activity diagram, 20
application, 17
association, 20
attribute, 20
attribute type, 54

B
base class type, 54

C
catch-all rule, 67
characteristic set, 87
\langle \texttt{clasimpmod} \rangle, 123
class, 20
class diagram, 20, 39, 43, 49, 51
class diagram extension, 50
class diagrams, 12
class type, 51, 56, 168
compliance, 39, 40, 45
comprehensions, 85
computational rules, 40
conformance, 49
congruence, 78
conservative theory extension, 18
conservative type definition, 46
consistent OCL specification, 48
constant definition, 18, 39, 40, 44, 46
constant specification, 18
context, 30, 35, 42
context lifting, 30, 38

D
datatype adaption, 36, 45
definedness, 78
derivation tree, 19

equality
\begin{itemize}
\item referential, 71
\item strict, 79
\item strong, 73
\end{itemize}
value
\begin{itemize}
\item deep, 72
\item shallow, 71
\end{itemize}
equality operators, 73
equivalence, 78
equivalence operator, 72
equivalence relation, 71

F
finalized, 67
formal methods, 11
formality, 11
functional adaption, 36, 46

G
global validity judgement, 77
goal, 19

H
Haskell, 17
higher-order logic, \textit{see} HOL
HOL, 17
INDEX

I
implication introduction, 19
inheritance, 20, 35, 49, 50
invariant, 36
Isabelle, 19, 107

K
Kleene Logic, 42
Kleene-Logic, 173

L
late-binding invocation, 46
layer, 36, 43
level, 36
lifting, 138
lifting over contexts, 38, 39, 48
Local (Formula) Equivalence, 79
Local Judgement Equivalence, 79
local judgement tableaux calculus, 80
local validity judgements, 77
logical judgement, 61, 73

M
Main (theory), 121
meta-implication, 19
meta-models, 12
meta-quantifier, 19
meta-variable, 19
method invocation, 47
methods, 35
model-based testing, 11
model-checking, 11
modularity, 30

N
natural deduction rules, 19
non-referential, 135
notation $\tau_L$, 38
notation $V_L(\alpha)$, 30

O
object, 51
instance, 51
object diagram, 20
object id, 49
object instances, 20
object structure, 35
object-orientation, 65
object-oriented, 20, 30
object-type system, 30
OCL (theory), 121
OclAny, 55
operation, 20, 29, 35, 36
operation invocation, 47
operation specification, 30, 35, 47, 64
overriding, 20

P
partial map, 66
Poly/ML, 107
postcondition, 29, 36, 65
precondition, 29, 36, 65
Proof General, 107
proof state, 19
proof-object, 19
propositional fragment, 85

R
reference types, 52
referential, 135

S
semantic combinator, 44, 46
semantic combinators, 43
sequence charts, 12
set constructors, 87
side-calculus, 30, 38
smashed, 95, 137
smashing, 138
SML, 17
specialization, 20
splintering, 95
splinters, 95
Standard ML, see SML
state, 29, 49
state chart, 20
state charts, 12
state diagram, 30
state transition, 29, 49
static types, 30
strict, 39

503
INDEX

strictification, 46
strictness, 37, 40, 139
subgoal, 19
subtyping, 20, 49

T
tactics, 19
tag type, 54
theorem proving, 11
theory, 135
theory file, 17
theory morphism, 43
totalized operation specification, 66
type class, 17
type constructor, 17, 31
type definition, 18
type specification, 18
type variables, 17
type-checking, 11
types, 46

U
UML_OCL (theory), 121
undefined, 36, 37, 39, 138
undefinedness, 78, 138
undefinedness lifting, 38, 39
Universal (Formula) Congruence, 79
Universal Judgement Equivalence, 79
universal validity judgement, 77
universe
  non-referential, 52
  referential, 52
universe type, 55
  non-referential, 56
  referential type, 55
universe types, 168
unsmashed, 137

V
validation, 11
value, 71
value type, 51, 71
value types, 168