A Formalization of JML in the Coq Proof System

Andreas Kägi, Hermann Lehner, and Peter Müller

Technical Report

Chair of Programming Methodology
Department of Computer Science
ETH Zurich

http://www.pm.inf.ethz.ch/

July 2009
Abstract

JML is a complex specification language for Java. Its large scale and manifold features make it hard to precisely define its semantics in a reference manual. It is thus desirable to formally specify the syntax and semantics of JML.

There are many good reasons for a formalized semantics of JML in a theorem prover: It can be used to develop a sound verification condition generator for JML constructs. By formally defining the semantics in a theorem prover, we can detect and eliminate ambiguities in the language. When using the semantics with an operational semantics for Java source code, we can define a runtime assertion checker and prove it’s soundness with respect to the semantics in Coq.

We divide the problem of defining JML in Coq into several steps. Firstly, we define a basic JML subset that has the full expressiveness of JML, but without syntactic sugar. We define the semantics for this subset in Coq. We introduce an extended (full) JML Syntax and a syntactic rewriting function from the extended syntax into the basic syntax. Finally, we built a translation frontend that transforms a JML-annotated Java program into it’s equivalent in Coq.

We managed to define the full JML and Java syntax in Coq, minus some very rare and not clearly described concepts and minus everything related to floating point numbers. We implemented a lightweight translation frontend in Java. We defined a large set of rewritings that simplify the syntax of JML without loosing any precision. We then defined the semantics of the desugared JML, using Bicolano as a basis for the semantic domain. Finally, we conducted a case study evaluating the feasibility of proving on top of the formalisation.
4.3 Extended Syntax

4.3.1 Extended Syntax Interface

4.3.2 Omitted- and Multiple Method Specification Clauses

4.3.3 Nested Specification Cases

4.3.4 Rewritings

4.3.5 Notations

4.3.6 Implementation Remarks

4.4 Rewriting Details

4.4.1 Loop Annotations

4.4.2 Quantified Expressions with Multiple Variables

4.4.3 Implicit Invariants

4.4.4 Method Specification Cases

4.5 Proving Case Study

4.5.1 Proof Details

4.5.2 Results

4.5.3 Discussion

4.5.4 Second Proof

4.6 Java Translation Frontend

4.6.1 Design

4.6.2 Translation Overview

4.6.3 Translation Details

4.6.4 Implementation

4.6.5 Type Level Rewritings

4.6.6 Implementation Remarks

4.6.7 Limitations

5 Conclusions
Chapter 1

Introduction

JML is a widely accepted specification language for Java. It is a complex language offering manifold features. This results in a Reference Manual that counts more than 150 pages. Defining the semantics of a language in free textual form is a big challenge: it is easy to forget special cases or to give ambiguous or even contradictory definitions.

To circumvent these problems, it is desirable to formally specify the syntax and semantics of JML. Specification within a theorem prover has the further advantage that semantic properties may be formulated and proven or the formalization may be used as a verification environment for JML-annotated programs.

This report deals with a formalization of JML and Java in the Coq Proof Assistant. It presents four contributions: First, the definition of a basic syntax and semantics for JML. Second, the definition of an extended (full) syntax of JML and a transformation of the extended syntax superset into the basic syntax set, realized within the Coq Proof Assistant itself. Third, a frontend that translates JML-annotated Java source programs into Coq source files that represent the programs in the formalized syntax. Finally, the report conducts a case study that investigates the feasibility of formulating and discharging proofs on top of the existing formalization.

The goal of this approach is to define the largest possible part of JML in the Coq Proof Assistant and to have a frontend that is as simple as possible. Following this approach, even the simpler desugarings of constructs from the extended syntax superset are part of the formalization. Defining a syntax superset that is independent of the basic syntax used to formalize the semantics of JML has two major advantages: First, the definition of the semantics is not cluttered with mere syntactic sugar, while second, the syntax superset can still be built in a way that retains the readability of the original source code as much as possible.

The implementation of a frontend is still of value be it as part of a verification environment that requires programs to be represented in the formalization or to represent programs used in testing and evaluating the formalization.

The next chapter gives the necessary background information on JML and Coq. Chapter 3 gives an overview about the extensions to the formalization as done by this thesis. Chapter 4 gives the details of the individual components: basic syntax and semantics – extended syntax – rewritings – translation frontend – case study. The concludes the work in chapter 5.
Chapter 2

Background

2.1 JML

JML [10, 11], the Java Modeling Language, is a specification and modeling language for Java. It offers a broad range of features, including support for floating-point numbers and concurrency.

The most important features relevant to this report are type specifications and method specifications. JML also adds new statements and expressions to the Java language, most notably the logical connectives forall and exists of predicate logic that allow to quantify in an expression over arbitrary values of a given data type.

// In the following examples, we assume the existence of a function
// "boolean isPrime(int)" and an object "set1" of type java.util.Set.

\exists int i; 0 <= i &amp; i <= 10; isPrime(i);
\forall Object o; set1.contains(o); o != null;

Listing 2.1: Example predicates of quantified expressions

2.1.1 Type Specifications

JML supports the notion of invariants and history constraints. An invariant for an object o is a predicate that has to hold in all visible states of object o. A history constraint is a predicate that is a relationship that should hold for the combination of each visible state of object o and any visible state of o that occurs later in the program’s execution.

Further type specification clauses include the initially clause, a predicate that each constructor has to establish, and the represents clause that deals with model fields.

A model field mf is a field declared with modifier model. It is an abstract, specification-only field of an object, whose value is determined by the represents clause: either the value is given by the value of another field f (represents mf <- f) or the value is determined non-deterministically such that it satisfies a given predicate P: represents mf \such_that P.
Besides model fields, JML supports another kind of specification-only field, so called *ghost fields*. In contrast to model fields, ghost fields are concrete fields and their value is set via JML statement `set`.

The other type specification clauses are less relevant for this thesis.

### 2.1.2 Method Specifications

JML supports method specifications with a variety of clauses, of which the most important ones are:

- **requires** a predicate that has to hold in the pre-state of a method.
- **ensures** a predicate that has to hold in the post-state of a method, if it returns normally.
- **signals** a predicate that has to hold in the post-state of a method, if it returns exceptionally, i.e. throws an exception.
- **signals_only** a list of exception type, the method is allowed to throw. (runtime exceptions are always allowed.)
- **diverges** a predicate that specifies under which condition a method is allowed to diverge, i.e. not return.
- **assignable** a list of storage locations, the method is allowed to assign values to.

Storage locations include references to individual fields (e.g. ‘this.f’), all fields of an object (e.g. ‘o.*’), individual elements within an array (e.g. ‘this.a[1]’) or multiple elements within an array (e.g. ‘this.a[1..3]’ or ‘this.a[*]’).

JML supports *lightweight* specification cases that do not assume a full specification and generate reasonable default values for omitted clauses and *heavyweight* specification cases that assume a full method specification to be present (see section 4.4.4 for details).

JML also supports so called *nested* specification cases that allow to group multiple specification cases into groups, where a list of `forall`- and `old` variable declarations and `requires` clauses can be declared that apply to all cases in this group.

```java
normal_behavior
   requires x != null
   {/
     requires x.getValue() >= 0;
     ensures \result == x.getValue();
     also
     requires x.getValue() < 0;
     ensures \result >= 0;
   /
```

Listing 2.2: Example of a nested specification case

In the above example, requires clause ‘`requires x.getValue() >= 0`’ applies to both cases `/* 1 */` and `/* 2 */`. 
2.1.3 Data Groups

Data groups allow to conveniently group together sets of storage locations that can then be used within assignable clauses. A data group named \( f \) is automatically generated for every field declaration \( f \), initially containing only the storage location of field \( f \). Other storage locations can be added to a data group by adding another data group explicitly (static data group inclusion using \( \text{in} \)) or by mapping dynamically a set of storage locations into the data group (using clause maps \( \text{... into} \)). Example 2.3 illustrates the inclusion possibilities.

\[
\begin{align*}
\text{int } f = 0; & \quad \text{// Implicit creation of a data group } f = \text{["this.f"]} \\
\text{int } g = 1; & \quad \text{// Implicit creation of a data group } g = \text{["this.g"]} \\
& \quad \text{//@ in } f \quad \text{// add data group } g \text{ to data group } f: \ f = \text{["this.f", "this.g"]} \\
\text{Object[]} \ arr = \ldots; & \quad \text{//@ maps arr[*] \ into } f; \quad \text{// add storage location } arr[*] \text{ to data group } f \\
& \quad \text{//@ data group } f = \text{["this.f", "this.g", "this.a[*]"]}
\end{align*}
\]

Listing 2.3: Examples of data group inclusion

2.2 Coq Proof Assistant

Coq is a formal proof management system acting at the same time as an interactive theorem prover and as a full-fledged functional programming language\(^1\). Coq is based on a framework called ‘Calculus of Inductive Constructions’ which is turn is based on constructive logic. As such, according to the Curry-Howard correspondence, statements can be seen as types and proofs as programs. Following this idea, Coq makes no difference between statements and types or proofs and programs. Consequently, proofs are functional programs and checking whether a proof of a statement is correct is equivalent to checking whether a program satisfies a given type.

\[
\begin{align*}
\text{Section Program.} & \quad \text{(* Identity function on natural numbers *)} \\
& \quad \text{Definition id_nat : nat } \rightarrow \text{ nat := fun (n:nat) } \Rightarrow \text{ n.} \\
\text{End Program.}
\end{align*}
\]

\[
\begin{align*}
\text{Section Proof.} & \quad \text{Variables } P : \text{ Prop.} \\
& \quad \text{(* Proof of } P \Rightarrow P \text{ (written } P \rightarrow P \text{ in Coq),} \\
& \quad \text{i.e. the identity on proofs of proposition } P \text{ *)} \\
& \quad \text{Definition P_imp_P : } P \Rightarrow P = \text{ fun (p:P) } \Rightarrow \text{ p.} \\
\text{End Proof.}
\end{align*}
\]

Listing 2.4: Programs vs. Proofs

Coq offers a very strong type system with dependent types (called dependent products) that allows to express predicate logic formulae as well: As an example, \( \forall P:\text{Prop. } P \Rightarrow P \) states that \( P\_\text{imp}\_P \) holds for every proposition \( P \) and is at the same time the type of all programs proving this fact.

\(^1\)Its syntax is similar to those of languages from the ML family.
Using inductive definitions, the other logical connectives are embedded in this system as well:

\[
\text{Inductive and } (A \, B : \text{Prop}) : \text{Prop} := \text{conj} : A \rightarrow B \rightarrow A \land B.
\]

That is, a proof of \(A \land B\) (written \(A \land B\)) can be constructed from proofs \(a:A\) and \(b:B\) \((a\) is a proof of \(A\), \(b\) a proof of \(B\)) using constructor \(\text{conj}\): \(\text{conj} \, a \, b : (A \land B)\).

Coq offers various proof tactics and (semi-)automatic decision procedure algorithms for some (semi-)decidable logics. Furthermore it offers a standard library implementing the most common functional data structures (lists and maps) as well as formalizations of natural numbers, integer numbers, rational numbers and real numbers.

Coq is well-suited for programming language formalizations due to its close link between (functional) programming and proving as well as its flexibility in supporting additional syntactic notations. Further desirable features include the presence of a relatively small certification “kernel” to check proofs, as well as the ability to extract certified (i.e. verified) functional programs\(^2\) out of Coq program developments.

### 2.2.1 Module System

This section gives a short introduction to the module system of Coq. As the formalization makes extensive use of this mechanism, it is necessary to understand its basic concepts. It is described in detail in Coq’Art\(^1\) and the Coq Proof Assistant Reference Manual\(^6\).

The module system augments Coq with two concepts: *name spaces* and *signatures*.

**Name Spaces**

In Coq, every declaration is part of a module. Top-level declarations in a file \(F\) are part of an implicit module \(F\). Every module defines its own name space. Within the same name space \(M\), a declaration with name \(f\) can be accessed by using the unqualified name \(f\), but from outside of the module, the qualified name \(M.f\) has to be used. This is true unless module \(M\) is explicitly imported to also allow unqualified access to names declared in \(M\).

**Signatures**

A signature describes the *interface* of a module. A module \(M\) implements signature \(S\), denoted \(M : S\), if every declaration with name \(x\) and type \(t\) is also declared in \(M\) with the same name \(x\) and the same type \(t\). A client should only rely on the types and functions declared in the signature of a module.

Signatures are well-suited to specify abstract data types. Thereby a signature \(S\) declares an abstract type \(t\) and functions that operate on type \(t\). The desired behavior is then specified as a set of axioms that are also part of the signature. A module implementing signature \(S\) then not only provides an implementation of type \(t\) and its functions but can also proves the axioms stated in \(S\).

---

\(^2\)Objective Caml or Haskell programs
Parameter top : t → option nat.

Axiom push_new : ∀ n:nat, top (push new n) = Some n.
End STACK_TYPE.

Module STACK : STACK_TYPE.

Definition t := list nat.
Definition new : t := nil.
Definition push (stack:t) (n:nat) : t := cons n stack.
Definition top : t → option nat := head (A := nat).

Lemma push_new : ∀ n:nat, top (push new n) = Some n.
Proof.
  auto.
Qed.
End STACK.

Listing 2.5: Example of an abstract data type and its implementation.

Type option is used to specify partial functions, that is, undefined results (None).

A signature $S$ that depends on functionality declared in another signature $S'$ can declare a module $M'$ with signature $S'$ and access from $M'$ the functions declared in $S'$. A module $M$ implementing $S$ then has to define an existing module $M' : S'$.

A module $M$ that depends on functionality declared in another signature $S'$ can either use a module $M'$ with signature $S'$ directly or declare itself as a parametric module with parameter $M' : S'$. 
Chapter 3

Overview

3.1 Overview

The definition of the formalization of JML and Java in Coq divide into four parts.

1. Basic syntax and semantics
2. Extended syntax superset definition
3. Extended syntax rewriting
4. Translation frontend

The frontend translates a given JML-annotated Java program into the extended syntax embedding of JML and Java in Coq. This embedding is called a deep embedding since a program is represented as a syntax tree with newly defined data types for the different syntactical constructs.

As an example, a JML expression ‘\fresh v1, v2’ is translated by the frontend into the extended syntax embedding ‘Fresh [(var v1); (var v2)]’ where Fresh : list Expr → Expr is a constructor of type Expr, the deep embedding type for expressions. var : Var → Expr is another constructor of this type. In the given example, the extended syntax and the basic syntax data type do not differ, but in other cases the two data types may be distinct (for example in the case of method specifications).

An extended syntax rewriting could syntactically rewrite ‘\fresh v1, v2’ as ‘\fresh v1 && \fresh v2’. This would result in a basic syntax deep embedding ‘InfixOp ConditionalAnd (Fresh [var v1]) (Fresh [var v2])’, where InfixOp ConditionalAnd : Expr → Expr → Expr is the basic syntax pendant to the &&-connective.

The definition of the semantics of JML then evaluates this Expr value into a higher-order logic formula, a shallow embedding of JML in Coq.

The evaluation of the &&-connective is defined as the ∧-operator on Prop and the evaluation of the \fresh e construct is defined as

\[\text{Definition EvalFresh (e : Expression) (h : Heap.t) (fr : Frame.t) : Prop :=}
\]

\[\text{1 We assume that v1 and v2 are variable declarations of type Var.}\]
\[\text{2although it doesn’t...}\]
∀ loc , EvalRefExpression e h fr = Some loc → Heap.typeof (PreHeap fr) loc = None.

Listing 3.1: Definition of EvalFresh for only one fresh expression

‘EvalFresh (var v)’ is thus equal to ‘True’, if for location loc, that corresponds to the evaluation of expression ‘var v’, the PreHeap does not contain a value at this storage location loc.

3.1.1 The Different Components

As seen in the introductory example, the translation frontend translated a given JML-annotated Java program into the extended syntax embedding. The frontend thereby generates for each input class a corresponding Coq output file. The frontend is the only component implemented in Java.

The extended syntax superset defines additional syntactical constructs to the syntax, most notably for the different forms of specification cases and a type for modifiers.

The extended syntax rewritings consists of the rewriting of loop annotations, quantified expressions with multiple variables, implicit invariants due to non-null constraints and the desugaring of method specifications.

The basic syntax and semantics define a core part of JML that doesn’t contain pure syntactic sugar.

3.2 Implementation Overview

The following section presents the architecture of the JML and Java formalization from an implementation point of view.

Figure 3.1 illustrates the dependencies between the different Coq modules defined. The convention in this figure is that signatures are written in capital letters, whereas modules are written in normal letters.

PROGRAM This signature (file JMLProgram.v) declares the abstract data types of the basic syntax interface.

PROGRAM_PLUS This signature (file JMLProgramPlus.v) declares the abstract data types of the extended syntax interface (within subsignature METHODSPEC_PLUS). Signature PROGRAM_--PLUS also declares a module of type PROGRAM, the PROGRAM_PLUS signature thus contains a superset of the definitions of the PROGRAM signature.

Full2Basic This module (file JMLFull2Basic.v) declares several signatures (incl. METHODSPEC_RE-WRITINGS_TYPE) that declare the different rewriting functions.

Make This module (file JMLProgramPlusImpl.v) implements the PROGRAM signature and the METHODSPEC_PLUS signature.

MakePlus This module (file JMLProgramPlusImpl.v) implements the PROGRAM_PLUS signature (trivially by including module Make).

Full2BasicImpl This module (file JMLFull2BasicImpl.v) contains submodules that implement the signatures declared in the Full2Basic module. This file thus contains the definitions of the rewriting functions.
ExpressionNotations This module (file JMLExpressionNotations.v) define notation shorthands for expressions.

Notations This module (file JMLNotations.v) define the rest of the notation shorthands (for statements, method specification cases, etc.).

Syntax This module (file JMLSyntax.v) gathers together all syntax modules, and is included in the output files, generated by the translation frontend.

DOMAIN This signature (file JMLDomain.v) declares the data types and functions of the semantic domain (formalization of heap, etc.).

Semantics This module (file JMLSemantics.v) defines the semantic functions of JML.

Notice that modules Full2Basic, ExpressionNotations and Semantics are implementation independent, they only depend on the interface signatures PROGRAM or PROGRAM_PLUS. Modules Notations and Syntax in contrast is dependent on the signature implementation modules Make/MakePlus and Full2BasicImpl.
3 Overview
Chapter 4

Formalization Details

4.1 Basic Syntax

The part of the Coq formalization of Java and JML called ‘Basic Syntax’ contains a formalization of the syntax of Java and a part of the syntax of JML. The Basic Syntax formalizes all JML constructs that are not mere syntactic sugar.

The chapter doesn’t introduce the whole Basic Syntax in every detail, but explains the general idea, the interesting aspects and issues with the Bicolano model that we use as a basis for our work.

The chapter ends with some comments on the implementation of abstract data types declared in the basic syntax interface.

4.1.1 Basic Syntax Interface

Most syntactic constructs are formalized as abstract data types, declaring a type \( t \) and functions operating on type \( t \). As an example, signature \texttt{METHODSIGNATURE\_TYPE} declares functions \texttt{name}, \texttt{parameters} and \texttt{result} that operate on the declared type \texttt{METHODSIGNATURE\_t} and give access to the name, the list of parameters and the result type of the given method signature.

\begin{verbatim}
Module Type METHODSIGNATURE\_TYPE.
  Parameter t : Type.
  Parameter name : t \rightarrow ShortMethodName.
  Parameter parameters : t \rightarrow list Param.
  Parameter result : t \rightarrow option type.
End METHODSIGNATURE\_TYPE.
\end{verbatim}

Listing 4.1: Example abstract data type \texttt{METHODSIGNATURE\_TYPE}

Notable exceptions to this pattern of defining syntactic constructs as abstract data types are \textit{expressions} and \textit{statements}. For these constructs, the basic syntax declares \texttt{inductive} data types. This allows convenient pattern matching on values of these types, a mechanism that is not supported for abstract data types. Pattern matching is syntactically convenient as it allows to directly refer to children in the different cases. Furthermore, the type checker checks that a case analysis is exhaustive, meaning all possible cases are covered.
(* inductive data type approach ... *)

match (type stmt) with
  | Compound block ⇒ process1 block
  | WhileStmt test body ⇒ process2 test body
  | ...

(* ... vs. abstract data type approach *)

if isCompound stmt
  then process1 (block stmt)
else if isWhileStmt stmt
  then process2 (test stmt) (body stmt)
else ...

Listing 4.2: Illustration of the two approaches to statements: Definition as an inductive data type vs. definition as an abstract data type.

Types are formalized as an inductive data type type. A distinction is made between primitive types (BOOLEAN, BYTE, SHORT and INT) and reference types (array types and class/interface types). Type char is not supported.

| Inductive type : Set := |
| ReferenceType : refType → type |
| PrimitiveType : primitiveType → type |

with refType : Set := |
| ArrayType : type → utsModifier → refType |
| EntityType : EntityName → utsModifier → refType |

with primitiveType : Set := |
| BOOLEAN | BYTE | SHORT | INT. |

Listing 4.3: Formalization of types

The interface to the formalization of the basic syntax of JML and Java is given as a single signature PROGRAM. Figure 4.1 shows the most important components of signature PROGRAM.

JML

JML expressions (and Java expressions as well) are formalized as part of the inductive Expr data type. For instance \( a \iff b \) is represented by constructor InfixPredOp : Operator → Expr → Expr → Expr and operator Equivalence. Quantifiers, like the universal quantifier are formalized by constructor Quantification, where Quantification q v r e corresponds to a quantifier of type q (Forall, Exists, Max, Min, NumOf, Product or Sum) declaring variable v and having range r and quantifier expression e. Notice that the basic syntax only supports quantifications with a single variable.

JML statements are formalized as part of the inductive StatementType data type. For instance assume(_redundantly) is represented by constructor LocalAssumption, where LocalAssumption e l r corresponds to an assumption e with an optional label l, either redundant or non-redundant (r).
Figure 4.1: Components of signature PROGRAM
A JML method specification case is formalized in the basic syntax by abstract data type 
SPECIFICATION_CASE_TYPE.

Listing 4.5: Specification Case Type

Notice that type SPECIFICATION_CASE.t supports only a very basic kind of specification case: 
namely a specification case, where every kind of clause is present exactly once. For the support of 
more sophisticated specification cases, see section 4.3. visibility is equal to Private, Protected, 
Package or Public. Every kind of clause is represented by its own data type. As an example, type 
ENSURES.t represents an ensures clause. Selector function ENSURES.pred : t → optional Expr 
gives access to the expression of the ensures clause. The types of the other clause kinds and its 
selector functions are defined similarly.

Storage locations are formalized as additional constructors to the Expr inductive data type:

- Nothing : Expr (‘\nothing’), Everything : Expr (‘\everything’)
- field : FieldSignature → option Expr (* target *) → Expr (e.g. ‘this.f’)

The actual type of LocalAssumption is ∀ (S:Type) (B:Type), Expr → option Expr → bool → Statement-
Type S B. For the discussion of the formalization of statements and blocks, see section 4.1.3.
• fieldAll : Expr (* target *) → Expr (e.g. “this.*”)
• array : Expr (* target *) → Expr (* index *) → Expr (e.g. ‘a[1]’)
• arrayAll : Expr (* target *) → Expr (e.g. ‘a[*]’)
• arrayRange : Expr (* target *) → Expr (* from *) → Expr (* to *) → Expr (e.g. ‘a[1..3]’)

4.1.2 Issues with the Existing Bicolano Interface

The formalization of the syntax of Java source code is realized as an extension to the formalization of the syntax of Java byte code as part of the [5.1.1] project. Our goal has been to reuse the parts of the Coq formalization that are common to both Java source code and Java bytecode such that eventually, our changes could be reintegrated into Bicolano. Unfortunately, the Bicolano interface makes “incorrect” usage of the module system that leads to problems in the implementation of the basic syntax interface. The problem is illustrated by means of the following excerpt from the basic syntax:

```coq
Module Type PROGRAM. (* Problem 1 *)
  Parameter Param : Set. (* Problem 2 *)
Module Type PARAM_TYPE.
  Parameter isFinal : Param → bool.
  ...
End PARAM_TYPE.
End PROGRAM.
```

The first problem is that PROGRAM is incorrectly declared as a module type (signature). A module P implementing PROGRAM has to declare signature PARAM_TYPE, too. This is clearly undesired as it leads to code duplication and hence to maintenance problems. Furthermore, P is not obliged to implement PARAM_TYPE, which obviously was not the intent.

The second problem, is that abstract type Param is declared outside of the corresponding signature PARAM_TYPE. The problem is seen on a hypothetical implementation of P1:

```coq
Module P1 : PROGRAM.
  Module PARAM.
    Inductive t : Set := Build_t { isFinal : bool; ... }
  End PARAM.
  Definition Param := PARAM.t.
  Module Type PARAM_TYPE.
    Parameter isFinal : Param → bool.
    ...
  End PARAM_TYPE.
End P1.
```

Implementation P1 is correct in the sense that it provides an implementation of the PARAM_TYPE signature, but this fact is not checked! Due to problem 2, the declaration of Param outside of PARAM_TYPE, the order of the declarations PARAM module – Param type – PARAM_TYPE signature is fixed, and PARAM cannot be declared as PARAM : PARAM_TYPE. This second problem can be lessened by defining an additional module PARAM’ : PARAM_TYPE := PARAM. So in a roundabout way, checking the adherence of module PARAM to signature PARAM_TYPE is still possible. Nevertheless,
the nice solution would be to avoid problems problem 1 and problem 2 altogether and provide the interface and implementation as follows:

```ocaml
Module PROGRAM.
  Module Type PARAM_TYPE.
    Parameter t : Set.
    Parameter isFinal : t → bool.
  End PARAM_TYPE.
End PROGRAM.
Module P.
  Module PARAM : PARAM_TYPE.
    Inductive t : Set := Build_t {
      isFinal : bool;
      ...}
  End PARAM.
End P.
```

4.1.3 Interesting Aspects

Statements and Blocks

The statement and block constructs make up the trickiest part of the syntax formalization. The two constructs are mutually dependent: the statement construct depends on the block statement and vice versa, since a block is a kind of statement, and a block consists of a sequence of statements.

Two requirements constrain the possibilities to formalize these constructs: First, a statement must give access to its label (if present) and its program counter. Second, a block must give access to its local variable declarations and its ordered sequence of statements. Adhering to these constraints, we define in a first try two abstract data types for statements and blocks:

```ocaml
Module Type STATEMENT_TYPE.
  Parameter label : Statement → Label.
  Parameter pc : Statement → PC.
End STATEMENT_TYPE.

Module Type BLOCK_TYPE.
  Parameter localVariables : Block → list Var.
  Parameter first : Block → PC
  Parameter last : Block → PC
  Parameter statementAt : Block → PC → Statement.
  Parameter next : Block → PC → option PC.
End BLOCK_TYPE.
```

As discussed in the basic syntax interface introduction, for convenient use, the different kind of statements are best described as an inductive type. Thus we try to declare an inductive statement type like this:
As such, this inductive type does not enable to implement the functions \texttt{label} and \texttt{pc}. We identified two possible solutions to integrate an inductive statement type with the required functions from the abstract \texttt{STATEMENT\_TYPE}.

1. Addition of a field \texttt{info} to each constructor of the inductive \texttt{StatementType} type. An implementation can then store the \texttt{label} and \texttt{pc} information in this field. This makes it straightforward to implement ADT \texttt{STATEMENT\_TYPE} by realizing the abstract type \texttt{Statement} as \texttt{StatementType}.

2. Addition of a parameter \texttt{kind} to the ADT enabling access to the statement kind. An implementation can then realize the abstract type \texttt{Statement} by defining a new record type \texttt{Statement} that stores the \texttt{label}, \texttt{pc} and \texttt{kind} information.

(*) "info" solution... *)
\begin{verbatim}
Inductive StatementType (Info:Type) : Type :=
  | Compound (info:Info) (block:Block)
  | WhileLoop (test:Expr) (body:Statement)
  | ...

Parameter info : ∀ Info:Type, StatementType Info → Info.
\end{verbatim}

(* ... vs. "kind" solution *)
\begin{verbatim}
Module Type STATEMENT\_TYPE.
  Parameter label : Statement → Label.
  Parameter pc : Statement → PC.
  Parameter kind : Statement → StatementType.
End STATEMENT\_TYPE.
\end{verbatim}

Listing 4.6: Two possible statement formalizations

We decided for the second solution in favor of the first one, mainly because of the additional \texttt{info} field. This \texttt{info} field should not be visible to clients. It is part of the solution domain (implementation) and does not correspond to a Java construct (problem domain).

The dependency graph (see figure 4.2) of \texttt{StatementType}, \texttt{STATEMENT\_TYPE} and \texttt{BLOCK\_TYPE} make some problems apparent. It nicely shows the mutual dependence between the types in the form of cycles. This makes it impossible to define the types separately. The most natural way to break these cycles is to make \texttt{StatementType} type parametric in terms of \texttt{Statement} and \texttt{Block} types. Hence the \texttt{StatementType} type can be declared without knowledge of the other types. Only when the type is used, the parameters have to be set.

However, one cycle still remains: the mutual dependencies between \texttt{STATEMENT\_TYPE} and \texttt{BLOCK\_TYPE}. \texttt{STATEMENT\_TYPE} depends on \texttt{BLOCK\_TYPE} as the \texttt{Block} type parameter of field \texttt{kind} has to be instantiated. This problem is solved by declaring the \texttt{Block} type as an additional abstract type parameter within \texttt{STATEMENT\_TYPE}.

The final interface to the three types is depicted in the following listing:
Figure 4.2: Dependencies between statements and blocks

(a) Original dependencies
(b) Resolved dependencies

Listing 4.7: Final form of statement and block formalization

Block Type Specification

The functions declared in signature BLOCK_TYPE are non-trivial: they do not merely return fields of a record as it is the case for signature METHODSIGNATURE_TYPE. Thus it is safer to specify their meaning formally. For this reason, we added several axioms to signature BLOCK_TYPE, that have to be proven as lemmas in an implementation. This greatly increases the trustworthiness of an implementation. For our list-based implementation, some details are given in section 4.1.5.

Function statementAt is specified by axiom

\[ \text{statementAt}_{\text{def}} : \forall t \, pc \, s, \text{statementAt} t \, pc = \text{Some} \, s \rightarrow \text{STATEMENT}.pc \, s = \, pc. \]

The program counter of the statement returned by statementAt t pc must be equal to the given
pc value.

For the specification of the other functions, a useful helper function is introduced:

\[ \text{elem} : t \rightarrow \text{PC} \rightarrow \text{bool} \]

were \( \text{elem} t \ p c \) states for a given block \( t \) and program counter \( p c \) whether \( p c \) corresponds to a program counter of a statement contained in block \( t \). This is specified formally with the help of \( \text{statementAt} \):

\[ \text{elem_def} : \forall t \ p c, \]
\[ \text{elem} t \ p c = \text{true} \rightarrow \exists s, \text{statementAt} t \ p c = \text{Some} s \land \text{STATEMENT}.p c s = p c. \]

Next can now easily be specified:

\[ \text{nextElem} : \forall t \ p c1 \ p c2, \text{next} t \ p c1 = \text{Some} p c2 \rightarrow \text{elem} t \ p c2 = \text{true}. \]

That is, for every block \( t \) and program counter \( p c1 \), if the result of next is a program counter \( p c2 \), \( p c2 \) must be the program counter of a statement contained in \( t \) (i.e. \( \text{elem} t \ p c2 = \text{true} \)).

The specification of \( \text{first} \) is also straightforward to understand:

\[ \text{firstElem} : \forall t \ p c, \text{first} t = \text{Some} p c \rightarrow \text{elem} t \ p c = \text{true}. \]

The same must be true for last:

\[ \text{lastElem} : \forall t \ p c, \text{last} t = \text{Some} p c \rightarrow \text{elem} t \ p c = \text{true}. \]

Using the current interface, it can unfortunately not be stated that the program counter found in \( \text{first} t \) corresponds to the first statement of \( t \). But it can be stated that the program counter found in \( \text{last} t \) indeed corresponds to the last statement:

\[ \text{lastDef} : \forall b \ p c, \text{last} b = \text{Some} p c \rightarrow \text{next} b \ p c = \text{None}. \]

That is, \( \text{next} t \) gives no further program counter, if the program counter found in \( \text{last} t \) is given as argument.

\[
\begin{align*}
\text{Axiom\ statementAt_def} : & \forall t \ p c s, \\
& \text{statementAt} t \ p c = \text{Some} s \rightarrow \text{STATEMENT}.p c s = p c. \\
\text{Axiom\ elem_def} : & \forall t \ p c, \\
& \text{elem} t \ p c = \text{true} \rightarrow \exists s, \text{statementAt} t \ p c = \text{Some} s \land \text{STATEMENT}.p c s = p c. \\
\text{Axiom\ next_elem} : & \forall t \ p c1 \ p c2, \text{next} t \ p c1 = \text{Some} p c2 \rightarrow \text{elem} t \ p c2 = \text{true}. \\
\text{Axiom\ first_elem} : & \forall t \ p c, \text{first} t = \text{Some} p c \rightarrow \text{elem} t \ p c = \text{true}. \\
\text{Axiom\ last_elem} : & \forall t \ p c, \text{last} t = \text{Some} p c \rightarrow \text{elem} t \ p c = \text{true}. \\
\text{Axiom\ last_def} : & \forall b \ p c, \text{last} b = \text{Some} p c \rightarrow \text{next} b \ p c = \text{None}. \\
\end{align*}
\]

Listing 4.8: Axiomatization of signature BLOCK_TYPE

**Loop Annotations**

In the initial design, loop annotations (loop invariants and loop variants) were realized as particular kind of statements, independent of the loop they precede. We dropped this approach in favor of loop statements that have an additional loop annotation field. The new loop annotation type is depicted in listing 4.9. Function \( \text{expression} \) gives access to a single loop invariant expression that is a rewrite of all given invariants and variants.

We think that this new approach is superior for the following reasons:

1. It follows more closely the syntax of (annotated) loops described in the JML Reference Manual, section 12.2.
2. From the viewpoint of a loop statement, it gives direct access to the annotations of this loop.
3. It conveniently allows the extended syntax rewriting of loop annotations to be formulated *locally*, that is only in terms of the loop annotation type.
With the old approach, to achieve both 2. and 3., implementations would have had to consider all statements within the enclosing block.

---

**Module Type** LOOP_ANNOTATION_TYPE.

- **Parameter** t : Type.
- **Parameter** expression : t → optional Expr.
- **Parameter** invariants : t → list Expr.
- **Parameter** variants : t → list Expr.
- **Parameter** expressionRedundantly : t → optional Expr.
- **Parameter** invariantsRedundantly : t → list Expr.
- **Parameter** variantsRedundantly : t → list Expr.

End LOOP_ANNOTATION_TYPE.

Listing 4.9: Loop annotation formalization

---

**Data Groups**

The basic syntax interface contains a signature DATA_GROUP_TYPE that formalizes data group inclusion clauses.

**Module Type** DATA_GROUP_TYPE.

- **Parameter** t : Type.
- **Parameter** isDynamic : t → bool.
- **Parameter** expression : t → Expr. (**only valid if isDynamic == true**)  
- **Parameter** dataGroups : t → list FieldSignature.
- **Parameter** isRedundant : t → bool.

End DATA_GROUP_TYPE.

Listing 4.10: Basic syntax data type for data groups

Values of implementation type DATA_GROUP.t are linked to (model-) fields as follows: Signatures FIELD_TYPE and MODELFIELD_TYPE declare a function dataGroupInclusions : Field → list DATA_GROUP.t.

This is illustrated on example 2.3:

```plaintext
int f = 0; // implicit creation of a data group f = ["this.f"]
int g = 1; // implicit creation of a data group g = ["this.g"]
//@ in f // add data group g to data group f: f = ["this.f", "this.g"] (dg1)
Object[] arr = ...;//
//@ \maps arr[*] \into f; // add storage location arr[*] to data group f (dg2)
// data group f = ["this.f", "this.g", "this.a[*]"]
```

Assuming the existence of constructor

DATA_GROUP.Build_t

: bool (* isDynamic *) → Expr → list FieldSignature
→ bool (* isRedundant *) → DATA_GROUP.t,
the above example translates to:

Definition dg1 :=
DATA_GROUP.Build_t
  false (* isDynamic *)
  null%jml (* irrelevant *)
  f_sig (* field signature of f *)
false. (* isRedundant *)

Definition dg2 :=
DATA_GROUP.Build_t
  true (* isDynamic *)
  "Tr[this.arr[*]]" (* maps expression *)
  f_sig (* field signature of f *)
false. (* isRedundant *)

dataGroupInclusions f_dcl would then contain DATA_GROUP.t values dg1 and dg2.

Type Specifications

To support type specifications, we add a reference to a ‘type spec’ data type to the class and
interface data types of Bicolano. The ‘type spec’ data type is merely a conglomerate that holds
references to the declared clauses. We added this additional level of indirection for two reasons: To
avoid cluttering the interface of the class and interface data types and to have a similar design for
type specification clauses and for method specification clauses. The SPECIFICATION_CASE_TYPE
data type of the latter corresponds to the TYPESPEC_TYPE type of the former.

Module Type TYPESPEC_TYPE.
  Parameter t : Type.
  Parameter invariant : t → list INVARIANT.t.
  Parameter constraint : t → list CONSTRAINT.t.
  Parameter represents : t → list REPRESENTS.t.
  Parameter initially : t → list INITIALLY.t.
  Parameter axiom : t → list AXIOM.t.
  Parameter readable_if : t → list READABLE_IF.t.
  Parameter writable_if : t → list WRITABLE_IF.t.
  Parameter monitors_for : t → list MONITORS_FOR.t.
End TYPESPEC_TYPE.

Module Type INVARIANT_TYPE.
  Parameter t : Type.
  Parameter pred : t → Expr.
  Parameter visibility : t → Visibility.
  Parameter isStatic : t → bool.
  Parameter isRedundant : t → bool.
End INVARIANT_TYPE.

Listing 4.11: Formalization of type specification clauses and invariants

The other kinds of type specification clauses are formalized similarly.
4.1.4 Notations

It has not yet been showed how a program or parts of a program are represented in the data types declared in the basic syntax. This section begins by showing, how expressions and statements are represented. The representation of other parts of a program is shown in section 4.3.3.

The data types defined for statements and blocks have already been presented in section 4.1.3. In the following, the existence of a module STATEMENT implementing abstract type STATEMENT_TYPE and a module BLOCK implementing BLOCK_TYPE is assumed, as well as constructors

- BLOCK.Build_t : list STATEMENT.t → BLOCK.t
- STATEMENT.Build_t : PC → option Label → StatementType t b → STATEMENT.t

that create values of the corresponding types.

The type defined for expressions is a simple inductive type (only parts used in the following are shown):

```latex
Inductive Expr : Type :=
| ... |
| literal (l : Literal) |
| var (var : Var) |
| AssignOp (op : Operator) (left : Expr) (right : Expr) |
| InfixOp (op : Operator) (left : Expr) (right : Expr) |
| InfixPredOp (op : Operator) (left : Expr) (right : Expr) |
| ... |
```

The representation of expressions and statements is shown by means of the following example:

```latex
expr1 = i%2 == 0;
stmt1 = sumEven += i;
stmt2 = sumOdd += i;

if (i%2 == 0) {
    sumEven += i;
} else {
    sumOdd += i;
}
```

It turns out that with the previously presented functions and types the definition of these examples is very cumbersome:

```latex
(* In the following, we assume that variables x, I, sumEven and sumOdd are previously defined and the type parameters for the StatementType Constructors already fixed. *)

expr1 := InfixOp Equal (InfixOp Mod (var x) (literal (IntLiteral 2))) (literal (IntLiteral 0)).

stmt1 :=
    STATEMENT.Build_t 6%Z None (ExprStmt (AssignOp Add (var sumEven) (var i))).

stmt2 :=
    STATEMENT.Build_t 7%Z None (ExprStmt (AssignOp Add (var sumOdd) (var i))).
```
STATEMENT.Build_t 5%Z None
  (IfStmt expr1
   (STATEMENT.Build_t 5%Z None (Compound (BLOCK.Build_t [] [stmt1])))
   (Some (STATEMENT.Build_t 7%Z None (Compound (BLOCK.Build_t [] [stmt2]))) ) )
).

Fortunately, Coq supports what it calls notations. By defining a notation, one can extend the parser of Coq to recognize a new syntactic construct and to replace it by user-defined code. Notations are very flexible and allow to specify their precedence level and associativity. Notations can even be defined for constructs with recursive patterns.

This embedding of Java and JML in Coq makes heavy use of notations. It defines notations or other shorthands for most syntactic constructs. The goal has been to make a piece of source code expressed in the embedding resemble as much as possible the original source code. Here is how the above examples are expressed using notations:

\[
\begin{align*}
\text{expr1} & := (\text{var } i) \text{ mod } (\text{int } 2) = (\text{int } 0) \\
\text{stmt1} & := \text{stmt } ((\text{var sumEven}) += (\text{var } i)) \\
\text{stmt2} & := \text{stmt } ((\text{var sumOdd}) += (\text{var } i)) \\
5\% & :> \text{ife } (\text{expr1}) \{ : 5 :> \\
& \quad [6\%N :> \text{stmt1}] \\
& \quad : \} \text{else} . \{ : 7 :> \\
& \quad [7\%N :> \text{stmt2}] \\
& \quad : \}
\end{align*}
\]

Expression Notations

Special notations have been defined for all expression constructs. Most notations define operators that are identical to the original Java/JML operators. For example, an expression ‘\(e1 \leftrightarrow e2\)’ is written as ‘\(e1 \leftrightarrow e2\)’ in the embedding, too. This is defined as a notation:

\[\text{Notation "x \leftrightarrow y" := (InfixPredOp Equivalence x y) (at level 96) : jml_scope.}\]

Some operators conflicted with existing Coq notations. In these cases, the operator in the embedding is suffixed with a prime character (‘). For example, an expression ‘\(e1 \&\& e2\)’ is written as ‘\(e1 \&\& e2\)’ in the embedding.

Statement Notations

As seen in the notation examples above, statements are built using the `STATEMENT.Build_t` constructor that is given a pc, an optional label and the statement type as arguments. As statements are so frequent and can be nested, it is nice to have a notation that replaces the use of the constructor. The notation defined for this case is: ‘\(\text{pc} :> \text{stmt} \)’ For example, ‘\(30\%N :> \text{nop}\)’ defines a new no operation (skip) statement at program counter 30.

There are also notations and shorthand functions that simplify the use of the different `StatementType` constructors. A switch `StatementType` is defined with the help of notation ‘\(\text{switch expr \{ \{ cases default dflt \} \}}\)’. All other statement types (except Compound) are created with simple shorthand functions. The following two tricks were used to beautify their syntax:

\[\%N\] opens the scope of natural numbers; \(30\) is thus interpreted as a natural number.
**Dummy arguments** Dummy arguments are added to functions to mimic Java syntax. The `ife` (if-then-else) function expects an argument of type `Else_` whose constructor `else_` is merely used to make a call `ife (test) stmt1 else_ stmt2` look more like the Java equivalent `if (test) stmt1 else stmt2`. (functions `ife, do_, try`)

**Dummy functions** Similar to dummy arguments. As an example, function `catch v s := (v,s)` is merely used to make a list of catch clauses look more like the Java equivalent: `[(v1, stmt1), (v2, stmt2)]` vs. `['catch v1 stmt1; catch v2 stmt2']` (functions `cases, catch`)

Block statements (Compound statement type) are treated differently. Once could define a notation like `{ ... }` for the Compound statement type and then create a block statement using `pc :> { ... }`. However, this is ugly since block statements often need to be passed to other statement creation functions such as `ife`. Then, `pc :> { ... }` would have to be surrounded by additional parenthesis to allow correct parsing. Instead, a new notation `{: pc :>> statements :}` is defined that directly creates a new block statement (instead of a `StatementType` value).

Listing 4.1 gives an overview of `StatementType` notations that differ from the Java notations: JML statements like `assert` or `assume` have notations identical to normal JML syntax.

**Loop Annotation Notations**

As table 4.1 already indicated, loop notations require a loop annotation value as argument. Section 4.1.3 has already presented the `LOOP_ANNOTATION_TYPE` signature. The implementation also provides notations to create values of type `LOOP_ANNOTATION.t` that implements signature `LOOP_ANNOTATION_TYPE`. This is best seen with an example:

```java
/*@ maintaining i <= 10;
   @ decreasing 10-i;
   @*/
for (int i = 0; i < 10; i++) ;
```

The loop annotations ‘maintaining i <= 10’ and ‘decreasing 10-i’ are created using the `maintaining` / `decreasing` notations and summarized into a single `LOOP_ANNOTATION.t` value using helper function `loop_annotation`:

```java
loop_annotation loop-label
    [maintaining (var i) <= (int 10);
     decreasing 10 - (var i)].
```

Declaration function `loop_annotation` and the need for a `loop-label` are explained in section 4.4.1

The whole for-loop translates to:

```java
(* We assume that i c.t. the variable declaration of i
   and i_name to the name declaration of i. *)
(\loop-label, pc_{for}) :> for_
```
### 4.1 Basic Syntax

<table>
<thead>
<tr>
<th>Statement kind</th>
<th>StatementType value/STATEMENT.t value</th>
</tr>
</thead>
<tbody>
<tr>
<td>Empty Statement “;”</td>
<td>nop</td>
</tr>
<tr>
<td>Labelled statement “label; statement”</td>
<td>(pc, label) -&gt; statement</td>
</tr>
<tr>
<td>Expression statement “e;”</td>
<td>stmt e</td>
</tr>
<tr>
<td>Block: “{ block }”</td>
<td>(: pcblock :&gt;&gt; block :)</td>
</tr>
<tr>
<td>Labelled block “label: { block }”</td>
<td>(: (pcblock, label) :&gt;&gt; block :)</td>
</tr>
<tr>
<td>“for (init; test; step) body”</td>
<td>for_ (init test step body)</td>
</tr>
<tr>
<td>“while (test) body”</td>
<td>while (test) body</td>
</tr>
<tr>
<td>“do body while (test)”</td>
<td>do_ body while_ test</td>
</tr>
<tr>
<td>“if (test) then-part”</td>
<td>if_ (test) then-part</td>
</tr>
<tr>
<td>“if (test) then-part else-part”</td>
<td>ife (test) then-part else-part</td>
</tr>
</tbody>
</table>
| “switch (test) {” |  switch (test) {
  case L1: case L2: body1
...   default: bodyn
} |
| “try {” |  tryF {
  try-block
} catch P1 {
  block1
} ...
} finally {
  finally-block
} |
| “break”/“break label” |  break/breakL label |
| “continue”/“continue label” |  continue/continueL label |
| “return”/“return expr” |  return/returnE expr |
| Java assertion “assert expr” |  jassert expr |
| Variable declaration “modifiers type name” |  |

*These notations directly create STATEMENT.t values instead of StatementType values.
*Loop annotations are discussed separately.
*If default case is missing, just omit default bodyn in translation.
*If finally-block is missing, use try instead of tryF and omit part finally {
  ... |
*Variable declaration statements are discussed in section 4.3.5.

Table 4.1: StatementType notations
Loop annotation loop-label

maintaining (var i) <= (int 10);

decreasing 10 - (var i)

[22 :> var_decl_stmt int_t iname (Some (int 0))]

((var i) < (int 10))

[(var i)++]

nop

4.1.5 Implementation Remarks

The implementation of the abstract data types defined in the basic syntax interface is based on lists. That is, all collections of values are implemented as lists. It would be very well possible to use more sophisticated data structures like maps in the implementation. We think that this is not necessary since the formalization will most often be used for proving. In this context, runtime efficiency of the operations given in the abstract data types is irrelevant. In case the formalization is used to evaluate concrete Java programs, we assume that classes and methods have reasonable size.

Most abstract data type implementations are straightforward: they are implemented by means of a newly defined record with appropriate fields.

Listing 4.12: Example abstract data type implementation of the signature for parameters. Compared to the stack example from the background section 2.5, this implementation is based on a record type and exploits the fact that Coq automatically generates selector functions for every field within a record.

Block Type Implementation

Recall the BLOCK_TYPE signature:

Module Type BLOCK_TYPE.

Declare Module STATEMENT : STATEMENT_TYPE.

Parameter t : Type.

Parameter localVariables : t -> list Var.

Parameter first : t -> PC

Parameter last : t -> PC

End BLOCK_TYPE.
Parameter statementAt : t → PC → STATEMENT.t.
Parameter next : t → PC → option PC.
Parameter elem : t → PC → bool.

Axiom statementAt_def : ∀ t pc s,  
    statementAt t pc = Some s → STATEMENT.pc s = pc.
Axiom elem_def : ∀ t pc,  
    elem t pc = true → ∃ s, statementAt t pc = Some s ∧ STATEMENT.pc s = pc.
Axiom next_elem : ∀ t pc1 pc2, next t pc1 = Some pc2 → elem t pc2 = true.
Axiom first_elem : ∀ t pc, first t = Some pc → elem t pc = true.
Axiom last_elem : ∀ t pc, last t = Some pc → elem t pc = true.
Axiom last_def : ∀ b pc, last b = Some pc → next b pc = None.

End BLOCK_TYPE.

Listing 4.13: BLOCK_TYPE signature

Signature BLOCK_TYPE is realized as a record of two lists: a list of local variables and a list of statements. Function localVariables is thus trivially defined. For convenience, the implementation also defines the other selector function statements : BLOCK.t → list Statement. Based on function statements the other functions are defined. The implementation uses so called strong specifications.

The idea is that the result of a strongly-specified function not only contains the final value of the desired computation, but also a proof that the computed result is correct. As an example, a strongly-specified square root function might look as follows:

```
sqrt : ∀ x:nat, x ≥ 0 → {y:nat | y*y = x}
```

That is, function sqrt takes as arguments a natural number x, a proof that x is larger or equal to zero and results in a natural number y that satisfied the equality $y^2 = x$.

`{y:nat | y*y = x}` is syntactic sugar for `sig(fun y:nat ⇒ y*y = x)`, an instance of type sig:

```
Inductive sig (A : Type) (P : A → Prop) : Type :=
    exist : ∀ y : A, P y → sig P.
```

Listing 4.14: Signature type sig

A value of type `sig nat (fun y:nat ⇒ y*y = x)` is given using constructor exist: `exist y P`, where in the example, y would correspond to the computed square root number and P to a proof of `$y^2 = x$`.

The advantage of using strongly-specified functions is that the specifications are always at hand (no separate lemmas necessary) and that strongly-specified functions can better be composed into larger strongly-specified functions. As such, it allows for a modular creation of large programs and proofs.

Continuing the square root example, it may be desirable to also allow negative numbers as input arguments. In this case, the result cannot be expressed as `{y:nat | y*y = x}` since a proof `$y^2 = x$ does not exist for negative numbers x. In this case, the result can be built using another helper type:

```
sqrt2 : ∀ x:nat, {y:nat | y*y = x}+(x < 0)
```

This is read as: Either the result is a value y together with the proof that y is the square root of
The whole implementation of the `BLOCK_TYPE` signature makes heavy use of the above concepts by defining strongly-specified counterparts to the functions declared in the signature and defining the signature versions in terms of the strongly-specified versions. This section now focuses on two particularly interesting lemmas, `last_elem` and `last_def`, that demonstrate the use of strong specifications but also the need to define new helper lemmas and even a new axiom.

\[
\text{last\_elem} : \forall t \ pc, \ \text{last} t = \text{Some} pc \rightarrow \text{elem} t \ pc = \text{true}
\]

is the simpler of the two lemmas. Its proof is based on a strongly-specified variant of `last`, `lastS : lastS (l:list A) : {x:A | Last x l} + {l=nil}` (last is defined in terms of `lastS`). This allows to derive from \(\text{last} t = \text{Some} \ pc\) that there exists a statement \(s\) satisfying the predicate \(\text{Last } s \ (\text{statements } t')\) (\(s\) is the last statement of list \(\text{statements } t\)) and \(\text{STATEMENT.pc } s = \pc\).

From a second lemma
\[
\text{elem\_def\_inv} : \forall t \ s0, \ \text{In } s0 \ (\text{statement } st) \rightarrow \text{elem} t \ (\text{STATEMENT.pc } s0) = \text{true},
\]

and knowing that element \(x\) of \(\text{Last } x \ (\text{statements } s)\)' is part of \(\text{statements } s\) (Lemma \text{Last\_In}), it follows that \(\text{elem } t \ pc\) is indeed equal to \text{true}.

\[
\text{Listing 4.15: \text{sum} type and definition of \text{Last}}
\]

The proof of `last_def` : \(\forall b \ pc, \ \text{last } b = \text{Some} pc \rightarrow \text{next } b \ pc = \text{None}\) is more involved. Again, as in the proof of `lastElem` it follows that there exists a statement \(s\), satisfying \(\text{Last } s \ (\text{statements } t')\) and \(\text{STATEMENT.pc } s = \pc\). Then the fact that next is defined in terms of a helper function `suffixS` is exploited:

\[
\text{suffixS} : \forall \ (\text{stmts:} \text{list } \text{STATEMENT.t}) \ (\text{pc:PC}), \ {l:\text{list } \text{STATEMENT.t} &\{s \mid \text{Suffix } l \ \text{stmts} \land \text{head } l = \text{Some } s \land \text{STATEMENT.pc } s = \pc\}} + \{\text{AllS (neq_PC_Stmt pc) stmts}\}
\]

In plain English: for all lists of statements \(\text{stmts}\) and program counters \(\text{pc}\), the result of `suffixS` \(\text{stmts} \ \text{pc}\) is

- either a suffix list \(l\) of list \(\text{stmts}\) where the statement in front of list \(l\) has a program counter equal to \(\text{pc}\) or
- all statements in list \(\text{stmts}\) have a program counter different from \(\text{pc}\).

Knowing that statement \(s\) is in the list and its program counter is equal to \(\text{pc}\), only the first case applies and it follows that there exists such a suffix list \(x0\). Now comes the tricky part: It is now known that the program counter of \(s\) is \(\text{pc}\) and that the program counter of the first element of list \(x0\) is also \(\text{pc}\). As no two distinct statements can have the same program counter (to be discussed),

\[\{y:\text{nat} \mid y*y = x\} + (x < 0)\] is syntactic sugar for \(\sum \{y:\text{nat} \mid y*y = x\} \ (x < 0)\) the widespread functional sum type.
it follows that the first statements of list \( x_0 \) is indeed \( s \). Using some arguments, it also follows that \( s \) is the last element of \( x_0 \). (From \( \text{lastS} \) already follows ‘\( \text{Last s (statements t)} \)’, and since \( x_0 \) is a suffix of ‘\( \text{statements t} \)’, ‘\( \text{Last s x0} \)’ is true as well. A last lemma (\( \text{Last_head_singleton} \)) ensures that the list \( x_0 \) is a singleton list (since both the first and the last element of \( x_0 \) is \( x \) and no two distinct statements may have identical program counters, this must indeed be the case). From this the desired result ‘\( \text{next b pc = None} \)’ follows, since the implementation of \( \text{next} \) selects the second element of suffix list \( x_0 \) (that does not exist) and thus selects \( \text{None} \).

(* The result of \( \text{next} \) is the pc of the second element of suffix list \( x_0 = \text{suffixS (statements block) pc0} \). *)

**Definition next** (block: t) (pc0:PC) : option PC :=

match suffixS (statements block) pc0 with
| inleft (existT l _) ⇒
  match tail l with
  | n::_ ⇒ Some (STATEMENT.pc n)
  | nil ⇒ None
  end
| inright _ ⇒ None
end.

Listing 4.16: Implementation of \( \text{next} \)

An important part of the proof is that two statements with the same program counters can be unified. This is ensured by lemma \( \text{PC_unique_impl} \) which in turn is not possible to be proven without a new axiom \( \text{Stmts_unique} \). \( \text{Stmts_unique} \) states that every program counter of a statement in \( b \) is unique.

**Lemma** \( \text{PC_unique_impl} \) : \( \forall \ x \ y \ b, \)

\[ \text{In } x \ (\text{statements } b) \rightarrow \]
\[ \text{In } y \ (\text{statements } b) \rightarrow \]
\[ \text{pc } x = \text{pc } y \rightarrow \]
\[ x = y. \]

(* For every suffix list \( x::s \) it holds that the program counter of list head \( x \) is different from the program counter of any element in tail \( s \). *)

**Axiom** \( \text{Stmts_unique} \) : \( \forall \ b \ x \ ss, \)

\[ \text{Suffix } (x::ss) \ (\text{statements } b) \rightarrow \]
\[ \text{AllS (neq\_PC\_Stmt (pc } x)) ss \]

Listing 4.17: A lemma and an axiom for the uniqueness of program counters.

4.2 Semantics for Basic JML Subset

4.2.1 Semantic Domain

This section presents the semantic domain of JML programs. The domain includes the definition of data types for all constructs that the Java semantics relies on. That is, we define the Java types booleans, integers, and references. We also need a heap model for instance and static fields, ghost fields, and arrays. We also provide a model for local variables and parameters, subtype relations between all types and exception handling.
Our domain describes the environment for a Java source program. However we base our work on the semantic domain of Bicolano (see section 5.1.1), which describes the Java runtime environment. The change consists in removing all bytecode specific concepts and replace them by their source code counterpart.

**Basic Data Types**

**Booleans** One main difference between Java bytecode and source code is the presence of the primitive type boolean. We consider two choices to represent boolean values.

- We can use the type bool that is defined in the Coq standard library as follows:
  \[
  \text{Inductive bool : Set := true : bool | false : bool.}
  \]
  We can express terms in propositional logic, but no first order logic, as quantifiers are not defined for this type. It would be necessary to implement another data type on top of bool to express first order logic terms.

- We can use the sort Prop for boolean values, which is the sort of propositions in the Coq proof system. By this, we directly have a shallow embedding of boolean expressions of JML into propositions in Coq. A boolean value evaluates to true iff we can give a prove for the corresponding proposition in Coq, using a classical setting.

Our choice is to use the sort Prop for boolean values. So far, this has proven to be the right choice to simplify proofs on the semantics. However, it’s sometimes necessary to use the type bool which is the reason for the conversion function Prop2bool which is defined as follows.

```
Parameter Prop2bool : Prop → bool.
Axiom Prop2bool_true: ∀ (P : Prop) , P → Prop2bool P = true.
Axiom Prop2bool_false: ∀ (P : Prop) , P → Prop2bool P = false.
```

In addition, we define the conversion from bool to Prop as coercion:

```
Coercion Is_true : bool → Sortclass..
```

**Numbers** In Java, we have the possibility to use integer and floating point numbers of different sizes. Floating point numbers are not defined in Bicolano and we also don’t introduce them. They don’t play a big role for most specification constructs.

In Bicolano, numbers are defined using the Coq type Z, which is a binary representation of arbitrary large integer numbers. For machine integers, Bicolano defines the overflow behavior with the definition smod, which produces the correct value with respect to the size of the numeric type, power.

```
Open Local Scope Z_scope.

Module Type NUMERIC.
  Parameter t : Set.
  Parameter toZ : t → Z.
  Parameter power : Z.

  Definition half_base := 2 ^ power.
```

4.2 Semantics for Basic JML Subset

Definition base := 2 * half_base.

Definition smod (x : Z) : Z :=
    let z := x mod base in
    match Zcompare z half_base with
    | Lt ⇒ z
    | _ ⇒ z - base
    end.

... |

In the definition of addition for machine numbers below, add_def, we can see the application of smod to produce the correct result.

... |

Parameter add : t → t → t.
Axiom add_def : ∀ i1 i2,
    toZ (add i1 i2) = smod (toZ i1 + toZ i2).

... |

The Coq module system provides a simple way to generate a type for each size of numbers in Java. We declare a new module of type NUMERIC and fix the value for the parameter power to the appropriate number.

Declare Module Short : NUMERIC with Definition power := 15.
Declare Module Int : NUMERIC with Definition power := 31.

Having defined the different number formats for Java, which differ just in the size, Bicolano also adds some conversion functions between those types in the usual way. We show here the example of the narrowing conversion from int to short.

Parameter i2s : Int.t → Short.t.
Axiom i2s_def1 : ∀ i,
    (Int.toZ i) mod 2^16 < 2^15 →
    (Int.toZ i) mod 2^16 = Short.toZ (i2s i).
Axiom i2s_def2 : ∀ i,
    2^15 <= (Int.toZ i) mod 2^16 →
    (Int.toZ i) mod 2^16 - 2^15 = Short.toZ (i2s i).

Finally, Bicolano introduces an inductive type num that brings together all the integer types of different sizes.

Inductive num : Set :=
    | I : Int.t → num
    | B : Byte.t → num
    | Sh : Short.t → num.
References

With a reference type, we point to an address in the heap. For this, Bicolano defines a data type \texttt{Location}. The parameter \texttt{Location\_dec} states that it’s always decidable whether two locations are the same or not. This information is, amongst others, relevant to store locations in a set.

\begin{verbatim}
Parameter Location : Set.
Parameter Location\_dec : \forall l1 l2 : Location, \{l1=l2\} + \{\neg l1=l2\}.
\end{verbatim}

Values

A value can either be of primitive type (\texttt{boolean, byte, short, int}), of a reference type, or null.

\begin{verbatim}
Inductive value : Type :=
 | Bool : Prop \rightarrow value
 | Num : num \rightarrow value
 | Ref : Location \rightarrow value
 | Null : value.
\end{verbatim}

Besides the three kind of values \texttt{Num, Ref, and Null} that are defined in Bicolano, we add a constructor for boolean values to the inductive type \texttt{value}.

Another approach to define \texttt{null} is to introduce it as a special location, e.g.

\begin{verbatim}
Parameter null\_loc : Location.
\end{verbatim}

But as most subsequent definitions need to distinguish between references that point to an address in the heap and \texttt{null} anyway, it’s easier to keep \texttt{Null} as a constructor in the inductive \texttt{value} data type.

Local Variables and Parameters

One difference between source code and bytecode is the treatment of local variables and parameters. In bytecode, parameters and local variables are stored in an array of known size at compile time and are addressed by index. This is not suited for Java source code and JML.

We need to distinguish between parameters and local variables. Parameters are visible in method level specifications, whereas local variables can only appear in annotation statements within the method body. We also want to preserve the name of parameters and local variables.

What we need for local variables and parameters is a dictionary (a map) to store key/value pairs. We need two separate dictionaries, as the Coq types of the keys are not identical for local variables and parameters. For this reason, and because it can also be used on other parts of the formalization, we introduce a general purpose, but lightweight dictionary in Coq. The dictionary is parametrized by \texttt{A and B}.

It provides the three functions: \texttt{empty}, which is the initial (empty) dictionary, \texttt{get}, which accesses the element with key \texttt{A} in dictionary \texttt{t} and yields \texttt{Some B} if the key is in the dictionary or \texttt{None} otherwise, and \texttt{update}, which updates, respectively stores, the element with key \texttt{A} and value \texttt{B} in dictionary \texttt{t} and yields the updated version of the dictionary.

The dictionary is not implemented, but axiomatized. We provide three axioms that define the behavior described above. The first axiom states that the empty dictionary always returns no value for any given key. The second axiom states that you get the value \texttt{v} for an element that you just updated with that same value \texttt{v}. The third axiom states that updating an element in
the dictionary doesn’t change anything else in the dictionary. These three axioms are sufficient in proofs.

Module Type DICT.

(* The dictionary type *)
Parameter t : Type.

(* Parameterization *)
Parameter A : Type.
Parameter B : Type.

(* Functionality *)
Parameter empty : t.
Parameter get : t → A → option B.
Parameter update : t → A → B → t.

(* Axiomatization *)
Axiom get_empty : ∀ k, get empty k = None.
Axiom get_update_new : ∀ l x v, get (update l x v) x = Some v.
Axiom get_update_old : ∀ l x y v, x <> y → get (update l x v) y = get l y.

End DICT.

The dictionary is declared as module type. An implementation or declaration of a module of this module type needs to define the parameters A and B. Using the DICT module type, we define two dictionaries, one for local variables and one for parameters as below, where Var is the type of local variables, and Param is the type of parameters.

Declare Module VarDict : DICT
    with Definition A := Var
    with Definition B := value.

Declare Module ParamDict : DICT
    with Definition A := Param
    with Definition B := value.

Heap

We reuse the heap model of Bicolano with minor adaptations.

One part of the heap model is the definition of the addressing mode as an inductive type. In Bicolano, instance fields are called DynamicField. We add static and instance model fields to the definition of AdressingMode, although they’re never really stored in a heap and they’re not mentioned in the rest of the heap model. However, this simplifies the uniform handling of model fields and fields in the rest of the formalization.

Inductive AdressingMode : Set :=
    | StaticField : FieldSignature → AdressingMode
    | DynamicField : Location → FieldSignature → AdressingMode
Bicolano defines the dynamic type of a location in the inductive definition \texttt{LocationType} for Objects and Arrays. We add universe type modifiers\footnote{Inductive \texttt{utsModifier} : Set := peer | rep | any.} to the definition of dynamic types. The definition also includes multidimensional arrays if we choose the constructor \texttt{ArrayType} for \texttt{type}. In this case, the uts modifier is \texttt{peer} for all inner dimensions as it’s not possible to define different uts modifiers for the different array dimensions in JML.

\begin{verbatim}
Inductive LocationType : Set :=
  | LocationObject : ClassName → utsModifier → LocationType
  | LocationArray : Int.t → type → utsModifier → LocationType.
\end{verbatim}

With this two definitions at hand, we can now define the basic heap operations. \texttt{get} and \texttt{update} are defined similarly to the dictionary that we introduced in the last section. The type of the key is \texttt{AdressingMode}. The function \texttt{typeof} yields the dynamic type of the given location in heap \texttt{t} if it is alive, or \texttt{None} otherwise. With this behavior of \texttt{typeof}, we can use it to check whether a location is alive in a heap. \texttt{new} allocates a new object of the provided type and yields a tuple of the location where the object has been allocated and the updated heap if the object was allocated successfully. Otherwise, \texttt{new} can also yield \texttt{None}. However, in the current formalization of Bicolano, this feature is not used, we always assume that allocation of the new object succeeds. The heap model is axiomatized as in \cite{14}.

\begin{verbatim}
Parameter get : t → AdressingMode → option value.
Parameter update : t → AdressingMode → value → t.
Parameter typeof : t → Location → option LocationType.
Parameter new : t → Program → LocationType → option (Location * t).
\end{verbatim}

Return Values

We reuse the Bicolano formalization of return values. A method either returns normally, in case of a function with a return value, or it throws an exception and returns the location of the exception object on the heap.

\begin{verbatim}
Inductive ReturnVal : Type :=
  | Normal : option value → ReturnVal
  | Exception : Location → ReturnVal.
\end{verbatim}

Method Frame and Program State

The method frame model differs significantly from Bicolano. We define a module \texttt{Frame} that contains a parameter \texttt{OldStates} which we define later to prevent circular dependencies, and a record \texttt{t} that represents the actual frame type. It contains dictionaries for parameters and local values, the current \texttt{pc}, the return value which is initially set to \texttt{Normal None}, a dictionary of old

\footnote{Inductive \texttt{utsModifier} : Set := peer | rep | any.}
states (defined below), a dictionary of variables bound by JML quantifiers, and finally two sets for assignable and accessible heap addresses and two sets for the addresses that have actually been assigned and accessed, respectively.

```coq
Module Frame.
  Parameter OldStates : Type.
  Record t : Type := Build_t { 
    params : ParamDict.t;
    vars : VarDict.t;
    pc : PC;
    ret : ReturnVal;
    old_states : OldStates;
    quant : VarDict.t;
    assigned : set Heap.AdressingMode;
    accessed : set Heap.AdressingMode;
    assignable : set Heap.AdressingMode;
    accessible : set Heap.AdressingMode 
  }.

A heap and a frame form a program state. Now we can define the parameter OldStates that we left unspecified before, as a dictionary from Label to State. As it's possible to refer to an old value at a given label l of the current method, we need to save a copy of the whole state into the current method frame when execution reaches this label. Of course it's not really possible (and necessary) to save multiple copies of the whole state into the method frame. However, it's the most faithful interpretation of what the semantics of JML says in our eyes.

```coq
Definition State := (Heap.t * Frame.t)
Declare Module StateDict : DICT 
  with Definition A := Label 
  with Definition B := State 
  with Definition t := Frame.OldStates.

In order to work with the frame model, we provide a large set of access and update functions for the individual parts of the frame. The two most important ones are easy access to the prestate and the preheap from the frame. The function PreState looks up the state at the special label L_beginBody from the OldStates dictionary. The function PreHeap calls PreState and yields the first part of the tuple, which is the heap:

```coq
Definition PreState (f : Frame.t) : State := 
  OldState f L_beginBody.
Definition PreHeap (f : Frame.t) : Heap.t := 
  fst (PreState f).

Updating the frame is a bit more tricky, as a record data type in coq can't be modified. It's necessary to build a complete new frame, using most of the old frame's values plus the intended change. In the example below, we see two functions, one to update the dictionary of quantified variables in the frame, and one to update one single quantified variable in the frame. You can see
that all information, except for the quantified variables dictionary is accessed in the current frame and just copied into the new frame.

```coq
Definition UpdateQuants (f : Frame.t) (qs : VarDict.t) : Frame.t :=
  Frame.Build_t (Frame.params f) (Frame.vars f) (Frame.pc f)
  (Frame.ret f) (Frame.old_states f) qs
  (Frame.assigned f) (Frame.accessed f)
  (Frame.assignable f) (Frame.accessible f).

Definition UpdateQuant (f : Frame.t) (q : Var) (v : value) : Frame.t :=
  let qs := VarDict.update (Frame.quant f) q v in
  UpdateQuants f qs.
```

Bicolano distinguishes between normal and exceptional frames and states. We don’t do this, there is only one kind of state (and frame) that can express an arbitrary program state. Our frame also contains a dictionary of old states instead of having several states (e.g. pre-state and the current state) in a judgment, we only need one state, the current one, and it’s frame contains all old states that are of interest.

Our frame model also allows us to clearly define the semantics of assignable clauses. Initially, on method invocation, we evaluate the set of addresses that are assignable and store it in the current’s frame assignable set. On every field or array location update, we add that address to the current’s frame assigned set. The semantics of the assignable clause defines that the assigned set is a subset of the assignable set, at any point during the method execution. The accessible clauses are defined analogous.

### 4.2.2 Semantics for JML Expressions and Clauses

This section presents the semantics of JML expressions and specification clauses. In our formalization, we cover all JML 0 constructs except for annotation statements, that will be part of the operational semantics.

We define the JML semantics in terms of evaluation functions for all constructs in Coq. The evaluation defines the construct with respect to the underlying semantic domain. Functions in Coq are always total, that is, we need to define the behavior of the evaluation even for a non-wellformed part of the domain of the function. That is why we return an undefined value of the right type in cases that we don’t want to describe.

#### Dealing with Partial Functions in Coq

Let’s give an example for a little helper-function that we use in the formalization: Assume that we have a `value` at hand and we know from the current context, that the value is always a number. We want to define a function that yields the number for the given value. As we need to define a total function, we also have to consider the cases that the value might not be a number, but a reference, a boolean, or null. That’s why we define a function from value to number as follows:

```coq
Parameter UndefinedNum : num.

Definition v2n (v : option value) : num :=
  match v with
```

```coq
```
In the definition above, we do a syntactic matching on the type of \( v \). If it matches \( \text{Some (Num } n \text{)} \), we yield \( n \). In all other cases, depicted by the wildcard \( _{} \), we return \( \text{UndefinedNum} \), which is a value of type \( \text{num} \) that we intentionally don’t specify any further. When we do proofs that require to unfold such definitions, we either get a case split that hopefully leads to a contradiction in the hypotheses for the second case, or we prevent the case split a priori by adding (and proving) a hypothesis that there can only be a \( \text{num} \) in the \( \text{value} \).

An alternative approach to write partial function is to use \( \text{option value} \rightarrow \text{option num} \) as the type of the function:

\[
\text{Definition } v2n \ (v : \text{option value}) : \text{option num} :=
\begin{align*}
  & \text{match } v \text{ with} \\
  & | \text{Some (Num } n \text{)} \Rightarrow \text{Some } n \\
  & | _{} \Rightarrow \text{None} \\
  & \text{end.}
\end{align*}
\]

This looks nice in theory, but leads to yet another (unnecessary) indirection. Using the inductive type \( \text{option} \) as result means that we get \( \text{Some num} \) instead of a \( \text{num} \) back, and we need to extract the number from the result again by matching, even if we can enforce the first case of \( v2n \).

A third alternative to write partial functions is not to define the function at all, but just axiomatize it:

\[
\text{Parameter } v2n : \text{option value} \rightarrow \text{num}.
\]
\[
\text{Axiom } v2n\_\text{def} : \forall \ n \ , \ v = \text{Some (Num } n \text{)} \rightarrow v2n \ v = n.
\]

This is the closest to partial functions, as we only specifies the case in which we actually have a number in the value. Being sufficient to define the semantics, the function still needs to be defined if we ever want to use an executable version of the semantics in Coq. The first definition is actually a correct implementation for the axiomatized version of \( v2n \).

We decide to use definitions in the spirit of the first example whenever reasonable. This reduces the number of indirections in proofs and definitions and gives us defined (potentially executable) functions instead of an axiomatization.

**Implicit arguments to the evaluation functions**

In all evaluation functions, we implicitly make use of three variables that define the program, the class, and the method in which we evaluate a JML construct:

\[
\text{Variable } p : \text{Program}.
\]
\[
\text{Variable } c : \text{Class}.
\]
\[
\text{Variable } m : \text{Method}.
\]
The Semantics of JML Expressions

We define three mutually recursive definitions to evaluate JML expressions. One for boolean expressions, one for expressions that evaluate to a reference value, and one for numeric expressions. An alternative is to define just one function to evaluate expressions, but this results in more difficult proofs, as we lose more information about the type of an expression.

The definitions perform a syntactic matching of the expression $e$ and forward the evaluation to a more specialized ‘helper’ function for each construct. Some of those helper functions do need to evaluate sub-expressions again. That’s why we sometimes have to hand over one or more of the evaluation functions as parameter to the helper function. This should not be confused with evaluating a sub-expressions within a parameter.

For instance, in $EvalUnaryBoolOp \ op \ (EvalBoolExpr \ e' \ h \ fr)$ we first evaluate sub-expression $e'$ and pass the boolean value afterwards to $EvalUnaryBoolOp$, whereas in $EvalFresh \ EvalRefExpr \ f\ list \ h \ fr$, we pass the function $EvalRefExpr$ as argument, together with the list of fresh expressions, the heap and the frame. $EvalFresh$ can then use the passed function to evaluate each expression in the list to a reference.

We introduce a new notation $\text{"f \ % \ x" := (f \ x)}$ (at level 99) for the Coq parser to change the precedence of function application. That is, $v21 \ % \ \text{ParamDict.get (Frame.params fr) paramThis}$ is a notation for $v21 \ (\text{ParamDict.get (Frame.params fr) paramThis})$. We use the notation for small conversion functions to increase readability.

```coq
Fixpoint EvalBoolExpr (e : Expr) (h: Heap.t) (fr : Frame.t) {struct e} : Prop :=
match e with
| var _ | quantifier _ | param _ | field _ _ | modelField _ _ | method _ _ | \result
⇒ v2b \ % \ EvalExpr EvalBoolExpr EvalRefExpr e h fr
| literal (BoolLiteral b)
⇒ b
| UnaryBoolExpr op e'
⇒ EvalUnaryBoolOp op (EvalBoolExpr e' h fr)
| BinaryBoolExpr op e1 e2
⇒ EvalBinaryBoolOp op (EvalBoolExpr e1 h fr) (EvalBoolExpr e2 h fr)
| RelationalExpr op e1 e2
⇒ EvalRelationalOp op (EvalNumExpr e1 h fr) (EvalNumExpr e2 h fr)
| JMLBinaryBoolExpr op e1 e2
⇒ EvalJMLBoolOp op (EvalBoolExpr e1 h fr) (EvalBoolExpr e2 h fr)
| Conditional cond t e'
⇒ EvalConditional EvalBoolExpr cond t e' h fr
| e1 <: e2
⇒ types_compatible p (EvalType EvalRefExpr e1 h fr) (EvalType EvalRefExpr e2 h fr)
| \old e'
⇒ let (h' , fr') := PreState fr in EvalBoolExpr e' h fr
| \oldl e' at l
⇒ let (h' , fr') := OldState fr l in EvalBoolExpr e' h fr'
| \fresh flist
⇒ EvalFresh EvalRefExpr flist h fr
| \nonnull elements e'
⇒ EvalNonNullElements EvalRefExpr e' h fr
| forall1 qvar ; r ; e'
⇒ \forall v , EvalQuantifier EvalBoolExpr qvar r e' v h fr
| exists1 qvar ; r ; e'
⇒ \exists v , EvalQuantifier EvalBoolExpr qvar r e' v h fr
| _
⇒ UndefinedProp
end
with EvalRefExpr (e : Expr) (h: Heap.t) (fr : Frame.t) {struct e} : option Location :=
match e with
| var _ | quantifier _ | param _ | field _ _ | modelField _ _ | method _ _ | \result
⇒ v2b \ % \ EvalExpr EvalBoolExpr EvalRefExpr e h fr
| null
⇒ None
| this
⇒ v2l (\text{ParamDict.get (Frame.params fr) paramThis})
| Cast t e'
⇒ EvalRefExpr e' h fr
| \old e'
⇒ let (h' , fr') := PreState fr in EvalRefExpr e' h fr'
| \oldl e' at l
⇒ let (h' , fr') := OldState fr l in EvalRefExpr e' h fr'
| _
⇒ Some UndefinedLocation
end
with EvalNumExpr (e : Expr) (h: Heap.t) (fr : Frame.t) {struct e} : num :=
```
4.2 Semantics for Basic JML Subset

match e with
| var _ | quantifier _ | param _ | field _ | modelField _ | method _ | \result
⇒ v2n % EvalExpr EvalBoolExpr EvalRefExpr e h fr
| literal (IntLiteral n)
⇒ I (Int.const n)
| UnaryIntExpr op e’
⇒ EvalUnaryNumOp op (EvalNumExpr e’ h fr)
| BinaryIntExpr op e1 e2
⇒ EvalBinaryNumOp op (EvalNumExpr e1 h fr) (EvalNumExpr e2 h fr)
| Cast t e’
⇒ EvalNumExpr e’ h fr
| \old e’
⇒ let (h’, fr’) := PreState fr in EvalNumExpr e’ h’ fr’
| \oldl e’ at l
⇒ let (h’, fr’) := OldState fr l in EvalNumExpr e’ h’ fr’
| Quantification q v r e’
⇒ I (EvalGeneralizedQuantifier EvalBoolExpr EvalNumExpr q v r e’ h fr)
| _
⇒ UndefinedNum
end.

There are several JML expression constructs that are evaluated independently of the return type. For these constructs, we define the function EvalExpr that yields a value. For each of the evaluation functions above, we now of which type the yielded value needs to be and apply the conversion function as described earlier.

Definition EvalExpr (e : Expr) (h : Heap.t) (fr : Frame.t) : option value :=
match e with
| var l
⇒ VarDict.get (Frame.vars fr) l
| quantifier q
⇒ VarDict.get (Frame.quant fr) q
| param par
⇒ ParamDict.get (Frame.params fr) par
| field fsig target
⇒ EvalFieldAccess fsig target h fr
| modelField fsig target
⇒ EvalModelFieldAccess fsig target h fr
| method msig target params
⇒ EvalMethodInvocation msig target params h fr
| \result
⇒ EvalResult fr
| _
⇒ None
end.

Helper Evaluation Functions  All the helper functions are defined in a section called EvalHelpers. The section has three variables of the type of the three evaluation functions. By this, we make the evaluation functions implicit and we don’t get huge signatures in the helper functions. Within the section, we don’t see the variables in the signatures, however, outside the section, e.g. in the evaluation function for expressions above, we have to pass all used variables as arguments. This massively improves readability of the formalization.

Section EvalHelpers.

Variable EvalBoolExpr : Expr → Heap.t → Frame.t → Prop.
Variable EvalRefExpr : Expr → Heap.t → Frame.t → option Location.
Variable EvalNumExpr : Expr → Heap.t → Frame.t → num.

...

We now have a closer look at an interesting selection of helper functions for JML expressions.

Binary Operations on Numerical Expressions  For binary operations on numerical values, we need to consider the type of operator on the one hand, and the type of number on the other.
EvalBinaryIntOp matches the operator with the different constructors and delegates the computation of the result to the Int module that we’ve defined in the semantic domain. EvalBinaryNumOp does a matching on both numerical input values according to the type (int, byte, short). It converts both numbers to integers, applies EvalBinaryIntOp to get the result and converts the number back to the right numerical type if necessary and packs it into a num type.

```
Definition EvalBinaryIntOp (op:BinaryIntOp) (i1:Int.t) (i2:Int.t) : Int.t :=
match op with
| Addition ⇒ Int.add i1 i2
| Subtraction ⇒ Int.sub i1 i2
| ...  
| BitwiseOr ⇒ Int.or i1 i2
| BitwiseXor ⇒ Int.xor i1 i2
end.
```

```
Definition EvalBinaryNumOp (op : BinaryIntOp) (n1 : num) (n2 : num) : num :=
match n1 , n2 with
| I i1 , I i2 ⇒ I (EvalBinaryIntOp op i1 i2)
| I i1 , B b2 ⇒ I (EvalBinaryIntOp op i1 (b2i b2))
| ...  
| Sh s1, B b2 ⇒ B (i2b % EvalBinaryIntOp op (s2i s1) (b2i b2))
| Sh s1, Sh s2 ⇒ Sh(i2s % EvalBinaryIntOp op (s2i s1) (s2i s2))
end.
```

**JML Fresh Operator** We choose the JML fresh operator as an example for the evaluation of JML operators. It asserts that the locations denoted by the list of expressions passed as arguments are fresh, i.e. not alive in the preheap. The definition in Coq is straightforward and expresses exactly this. We map the evaluation function for references to all expressions in the list, which yields a list of locations. We then quantify over all possible locations and state that if the location is in the fresh list, it was not alive in the preheap. Remember that the typeof function in the Heap module yields None if the location is not alive.

```
Definition EvalFresh (flist : list Expr) (h : Heap.t) (fr : Frame.t) : Prop :=
∀ loc ,
In loc
(map
(fun e ⇒
match EvalRefExpr e h fr with
| Some loc ⇒ loc
| _ ⇒ UndefinedLocation
end)
)

Heap.typeof (PreHeap fr) loc = None.
```

**Static and Instance Field Access** We define one evaluation function for static and instance field access. A FieldSignature is defined as the tuple ClassName * ShortFieldSignature. There are two possibilities for the function to find out if the field access is static or instance. Either
we perform a query on the class data type to find the field that corresponds to the signature and
then check in the field data type if its static flag is set. The simpler version is to look at the
**target** parameter, which is set to None if it’s a static field access and to Some target’ otherwise.
In case of an instance field access, we evaluate the target’ expression to a location and can then
perform the lookup of the field’s value in the heap. In case of a static field access, we can directly
ask the heap for the value of the static field denoted by fsig.

```plaintext
Definition EvalFieldAccess (fsig : FieldSignature) (target : option Expr)
(h : Heap.t) (fr : Frame.t) : option value :=
match target with
| Some target’ ⇒
  match EvalRefExpr target’ h fr with
  | None ⇒ None
  | Some loc ⇒ Heap.get h (Heap.DynamicField loc fsig)
  end
| None ⇒ Heap.get h (Heap.StaticField fsig)
end.
```

**Method Calls in JML Specifications** Methods in specifications are pure and can be con-
sidered as mathematical functions. The evaluation of the method itself is not very challenging,
we query the program data type to get the method for the given signature with findMethod and
apply the function EvalPureMethod which is not defined a priori, but needs to be axiomatized
for each method. Preparing a new method frame involves a bit more work. For the parameters,
EvalParams evaluate the list of expressions to a list of values. Like for the field access, we evaluate
the target to a location if it’s an instance method call.

```plaintext
Definition EvalMethodInvocation (msig : MethodSignature) (target : option Expr)
(actuals : list Expr) (h : Heap.t) (fr : Frame.t)
: option value :=
let params’:=EvalParams (METHODSIGNATURE.parameters(snd msig)) actuals h fr in
match findMethod p msig , target with
| Some m , Some target’ ⇒
  let target’’ := Some (l2v % EvalRefExpr target’ h fr) in
  EvalPureMethod p m h (NewFrameInstanceMethodCall h m (target’’::params’))
| Some m , None ⇒ EvalPureMethod p m h
  (NewFrameStaticMethodCall h m params’)
| _ , _ ⇒ None
end.
```

To build a new frame for an instance method call we do the following: Add this as first parameter
to the list of parameters, fill the parameter dictionary and get the first PC of the method. All
other parts of the frame stay empty.

```plaintext
Definition NewFrameMethodCall (h : Heap.t) (callee : Method) (args : list Param)
(params : list (option value)) : Frame.t :=
let p := FillParamDict args params ParamDict.empty in
match METHOD.body callee with
| Some body ⇒
  let pc’ := STATEMENT.pc (METHODBODY.compound body) in
  let fr’ := Frame.Build_t p VarDict.empty pc’ (Normal None) StateDict.empty
```
Definition NewFrameInstanceMethodCall (h : Heap.t) (callee : Method)
  (params : list (option value)) : Frame.t :=
  let args := paramThis::METHODSIGNATURE.parameters (METHOD.signature callee) in
  NewFrameMethodCall h callee args params.

**Generalized Quantifiers** The evaluation of generalized quantifiers is tricky and currently only defined for int numbers. We use set operations on the set QuantSet of all integer numbers. In a first step, we throw away all elements in the set for which the QuantFilter yields False: For all elements (numbers) in the set, QuantFilter stores the value in the dictionary of quantified variables and evaluates the range expression with the updated frame. In a second step, we apply the quantified expression to all elements that are still in the set in the same way. QuantOp yields a binary operation on integers for a given quantifier and QuantIdentityElement yields the right starting element to fold the set of integers, which is the last step in the evaluation of the generalized quantifier expression.

Parameter QuantSet : set Int.t.

Parameter QuantOp : Quantifier → Int.t → Int.t → Int.t.

Definition QuantFilter (v:Var) (e:Expr) (h:Heap.t) (fr:Frame.t) (i:Int.t) :
  bool :=
  let fr’ := UpdateQuant fr v (Num (I i)) in
  Prop2bool % EvalBoolExpr e h fr’.

Definition QuantExpr (v : Var) (e : Expr) (h : Heap.t) (fr : Frame.t) (i : Int.t)
  : Int.t :=
  let fr’ := UpdateQuant fr v (Num (I i)) in
  match EvalNumExpr e h fr’ with
  | I n ⇒ n
  | B n ⇒ b2i n
  | Sh n ⇒ s2i n
  end.

Definition EvalGeneralizedQuantifier (q : Quantifier) (v : Var) (r : Expr)
  (e:Expr) (h:Heap.t) (fr:Frame.t) : Int.t :=
fold_right
  ( QuantOp q)
  ( QuantIdentityElement q)
  ( map
    (QuantExpr v e h fr)
    ( filter
      (QuantFilter v r h fr)
      QuantSet
    )
  )
  .
The Semantics of Method Level JML Clauses

We show the formalization of method level specification clauses on the example of the requires clause and the signals_only clause.

**Requires clause** We evaluate a requires clause for a given specification case by looking up the requires clause in the specification case and the predicate in the requires clause. If the predicate is `not_specified`, we yield `NotSpecifiedRequires` which is not defined. However, this function can be implemented by the client to define default values for not specified specification clauses that make sense for their purpose. If the requires clause is specified in the specification case, we evaluate the predicate using `EvalBoolExpr` that we discussed earlier.

\[
\text{Definition} \quad \text{EvalPredicate} \ (e \ h \ fr) \ := \ \text{EvalBoolExpr} \ (e \ h \ fr).
\]
\[
\text{Parameter} \quad \text{NotSpecifiedRequires} \ (h \ h) \ := \ \text{NotSpecifiedRequires} \ (h \ h).
\]
\[
\text{Definition} \quad \text{EvalRequires} \ (sc \ h \ fr) \ := \ \begin{cases} \text{EvalPredicate} \ (e) \ (h \ h \ fr) & \text{if } (e) \ \text{is specified} \\ \text{NotSpecifiedRequires} \ (h \ fr) & \text{otherwise} \end{cases}
\]

**Signals_only Clause** To evaluate a signals_only clause that we get from the given specification case, we check that there exists a type in the signals_only clause which is a supertype of the type of the exception object.

\[
\text{Definition} \quad \text{EvalSignalsOnly} \ (sc \ h \ fr) \ := \ \exists \ t \ : \ \text{In} \ t \ (\text{SIGNALS_ONLY.types} \ s) \ \land \ \assign_{\text{compatible}} \ (p \ h \ (\text{Ref} \ loc) \ t).
\]

The Semantics of Type Level Specification Clauses

We show the formalization of type level specification clauses on the example of the invariant clause.

**Invariant Clause** If we are currently in the context of a helper method, we don’t check invariants and always yield `True`. Otherwise, we yield the evaluation of the invariant for all invariants that are defined in `c`. That is, either the invariant is declared in `c` or it is declared in one of the super types of `c`.

\[
\text{Inductive} \ \text{DefinedInvariant} \ (c \ inv) \ (inv \ inv) \ := \begin{cases} \text{DefinedInvariant} \ (c \ inv) \ (inv \ inv) & \text{if } (c) \ \text{is defined} \\ \text{DefinedInvariant} \ (c \ inv) \ (inv \ inv) & \text{otherwise} \end{cases}
\]
SuperTypeSpec c ts →
In inv (TYPESPEC.invariant ts) →
DefinedInvariant c inv.

**Definition** EvalInvariant (h : Heap.t) (fr : Frame.t) : Prop :=
if METHOD.isHelper m then
   True
else
   ∀ inv ,
   DefinedInvariant c inv →
   EvalPredicate (INVARIANT.pred inv) h fr.

### 4.2.3 An Annotation Table for JML Specifications

We ultimately want to define the following annotation table to access JML specifications for a method (or constructor). This allows an external tool to use the semantics as a source for (first order logic) pre- and postconditions, as well as local assertions and loop invariants. These terms are of sort Prop and don’t contain any JML specific constructs any more.

**Module** Type ANNOTATION_TABLE.
**Parameter** Precondition : Program → Class → Method → State → Prop.
**Parameter** Postcondition : Program → Class → Method → State → Prop.
**Parameter** AssertionAt : Program → Class → Method → PC → State → Prop.
**Parameter** LoopInvariant : Program → Class → Method → PC → State → Prop.
**Module** ANNOTATION_TABLE.

In the following, we give an implementation for all four parameters in the **ANNOTATION_TABLE**. For Precondition and Postcondition we only show the implementation for methods.

First, we present the definition of DefinedCase which says that a specification case is defined for a method if the case is declared in that method or one of the overridden methods. Such definitions are purely syntactical and therefore defined in the module **PROGRAM**.

**Inductive** DefinedCase (c:ENTITY.t) (m:Method) (sc:SpecificationCase) : Prop :=
DefinedCase_def : ∀ m' c',
   subtype c c' →
   ENTITY.method c' (METHOD.signature m) = Some m' →
   In sc (METHOD.specs m') →
   DefinedCase c m sc.

We can now define what a precondition for a method should be. The precondition is the conjunction of all object invariants and at least one requires clause from a specification case that is defined in that method. This means that the method can be executed if all Objects are in a consistent state and there is at least one specification case that matches the current environment.

**Definition** Precondition (p:Program) (c:Class) (m:Method) (st:State) : Prop :=
(∀ c',
   EvalInvariant p c' m (fst st) (snd st)
) ∧
The postcondition of a method is the conjunction of all object invariants and one of the following:

- the requires clause (executed in the prestate) implies the ensures clause for all specification cases if the method returns normally.
- the requires clause (executed in the prestate) implies the signals and signals_only clause of all specification cases if the method throws an exception.

If there is a JML assertion or assumption on the given pc, we evaluate it and yield the proposition. If there is no such JML constructs at the given program point, we yield True. If we cannot evaluate the statement at the given program point, for instance because there is no such program point, we yield False.

We deal with loop invariants very similarly to local assertions. We extract the annotations from the syntax if there is a loop at the given program point, and evaluate it to a Coq proposition. If
the loop invariant is not specified, we yield True. Alternatively, we could also yield some undefined coq proposition.

Definition LoopInvariant (p: Program) (c: Class) (m: Method) (pc: PC) (st: State) : Prop :=
  match statement_at m pc with
  | Some stmt ⇒
    match STATEMENT.type stmt with
    | WhileLoop anno _ _
    | DoLoop anno _ _
    | ForLoop anno _ _ _ _ ⇒
      match LOOP_ANNOTATION.expression anno with
      | (:e:) ⇒ EvalPredicate p e (fst st) (snd st)
      | \not_specified ⇒ True
      end
    | _ ⇒ False
    end
  | _ ⇒ False
  end.


4.3 Extended Syntax

The part of the Coq formalization of Java and JML that we call ‘Extended Syntax’ contains a formalization of all syntactic constructs of Java and JML that are rewritable in terms of basic syntax formalization constructs. For Java, this amounts to the addition of declaration functions that generate default values omitted in the declaration. For JML, the extended syntax adds support for specification cases in all its forms. Furthermore it provides rewritings that desugar specification cases, loop annotations and non_null modifiers.

4.3.1 Extended Syntax Interface

First and foremost, the extended syntax defines new data types to support method specification cases in all its sugared forms:

- support for omitted and multiple method specification clauses per specification case
- support for nested specification cases
- support for lightweight-, normal behaviour- and exceptional behaviour specification cases

A (full) specification case is one that may use the above named specification case features, whereas it is called a basic specification case when these features are not allowed. In the formalization, a basic specification case is a value of type SPECIFICATION_CASE.t. The type for full specification cases will be defined below.

4.3.2 Omitted- and Multiple Method Specification Clauses

In a basic specification case every method specification clause has to be declared exactly one. This syntax is thus first extended to support omitted clauses and multiple clauses. The existing
SPECIFICATION\_CASE\_TYPE signature could have been adapted to provide a (possibly empty) list of clauses for every kind of clause but the extended syntax takes another approach: It provides one algebraic data type MethodSpecClause for all clause kinds. This gives more flexibility in omitting clauses and writing them in arbitrary order by having a single list of MethodSpecClause values per specification case.

\[
\text{Inductive MethodSpecClause : Type :=}
\]

| ensuresC (redundant : bool) (pred : optional Expr) |
| signalsC (redundant : bool) (pair : Var * optional Expr) |
| signalsOnlyC (redundant : bool) (types : list type) |
| divergesC (redundant : bool) (pred : optional Expr) |
| whenC (redundant : bool) (pred : optional Expr) |
| assignableC (redundant : bool) (storeRefs : optional (list Expr)) |
| accessibleC (redundant : bool) (storeRefs : optional (list Expr)) |
| callableC (redundant : bool) (storeRefs : optional CallableList) |
| measuredByC (redundant : bool) (pair : optional Expr * option Expr) |
| capturesC (redundant : bool) (storeRefs : optional (list Expr)) |
| workingSpaceC (redundant : bool) (pair : optional Expr * option Expr) |
| durationC (redundant : bool) (pair : optional Expr * option Expr). |

Listing 4.18: Extended syntax data type for method specification clauses

4.3.3 Nested Specification Cases

JML allows specification cases to be nested. Section 9.4 of the JML Reference Manual [10] defines the syntax of a nested specification case (generic-spec-case):

\[
generic-spec-case ::= \\
[ spec-var-decls ] spec-header [ generic-spec-body ] \\
| [ spec-var-decls ] generic-spec-body \\

generic-spec-body ::= \\
simple-spec-body \\
| \{ generic-spec-case-seq \}

generic-spec-case-seq ::= generic-spec-case [ also generic-spec-case ] ... \\
spec-header ::= requires-clause [ requires-clause ] ... \\
simple-spec-body ::= simple-spec-body-clause [ simple-spec-body-clause ] ... 
\]

Listing 4.19: Nested specification case grammar

A nested specification case contains a possibly empty list of variable declarations, an optional specification header (one or more requires clauses) and an optional specification body that is either one or more body clauses (all clauses except requires) or one or more nested specification cases. It is not allowed that a specification case has neither header nor body.

Grammar non-terminal generic-spec-case is formalized by means of the new signature GENERIC\_SPEC\_CASE\_TYPE:
Module Type GENERIC_SPEC_CASE_TYPE.
    Parameter t : Type.
    Parameter forallVarDecl : t → list FORALL_VAR_DECL.t.
    Parameter oldVarDecl : t → list OLD_VAR_DECL.t.
    Parameter specHeader : t → list SpecHeader.
    Parameter genericBody : t → (list MethodSpecClause) + (list t).
End GENERIC_SPEC_CASE_TYPE.

Listing 4.20: Nested specification case type

That is, a value of type GENERIC_SPEC_CASE.t implementing signature GENERIC_SPEC_CASE_TYPE gives access to the list of variable declarations, the list of specification headers and a body that is either a list of clauses or a list of nested GENERIC_SPEC_CASE.t values. Notice that this abstract data type does not enforce the above consistency criterion that requires the existence of either a header or a body. This criterion has to be enforced by a well-formedness predicate that guarantees that all values have the desired form. Furthermore an empty body could be represented as both ‘inl nil’ or ‘inr nil’, but this does not bother too much. The abstract data type in the present form has been chosen to be as simple as possible in order to allow short and clear definitions of the rewriting functions.

Types FORALL_VARDECL.t and OLD_VARDECL.t are part of the basic syntax interface:

Module Type FORALL_VARDECL_TYPE.
    Parameter t : Type.
    Parameter vars : t → list Var.
End FORALL_VARDECL_TYPE.

Module Type OLD_VARDECL_TYPE.
    Parameter t : Type.
    Parameter varDecls : t → list (Var * Expr).
End OLD_VARDECL_TYPE.

Listing 4.21: Basic Syntax signatures for ∀- and old variable declarations.

Type SpecHeader is defined to be a “pendant” to the spec-header grammar non-terminal and allows for a convenient definition of the requires/requires_redundantly notations as explained later.

Inductive SpecHeader : Type :=
  | requiresSH (redundant : bool) (pred : optionalSame Expr).

Listing 4.22: Extended syntax data type for a spec-header.

Lightweight-, Normal Behaviour- and Exceptional Behaviour Specification Cases

The different specification cases only differ with respect to the allowed clauses (normal- and exceptional behaviour), disallowed visibility (lightweight) and the semantics of omitted clauses.

6sum type $A+B$ has already been presented in section [4.1.5]

7Normal behaviour cases do not allow signals clauses, whereas exceptional behaviour cases do not allow ensures clauses.

8A lightweight case inherits the visibility of the enclosing method.
But the (possibly nested) specification clauses are described by `generic-spec-case` for all kind of specification cases. For simplicity, we therefore define one single data type for all kinds of cases. Again, a well-formedness predicate ensures that for every kind of specification case only allowed clauses are given. The semantics of omitted clauses is discussed in section 4.4.4.

```ocaml
Module Type FULL_SPEC_CASE_TYPE.
  Parameter t : Type.
  Parameter specCaseType : t \rightarrow SpecCaseType.
  Parameter visibility : t \rightarrow option Visibility.
  Parameter isRedundant : t \rightarrow bool.
  Parameter isCodeContract : t \rightarrow bool.
  Parameter genericSpecCase : t \rightarrow GENERIC_SPEC_CASE.t.
End FULL_SPEC_CASE_TYPE.
```

Listing 4.23: Extended syntax data type for the different kind of specification cases.

The specification case `type` (`\rightarrow` lightweight, behaviour, normal_behaviour or exceptional_behaviour) is used to distinguish between the different case types.

### Declarations with Omitted Modifiers

The basic syntax does not provide a type for modifiers. Instead, every modifier is realized as a separate boolean selector on the corresponding type. As an example, type `PARAM.t` provides a selector `isFinal : Param \rightarrow bool` that corresponds to the final modifier.

```ocaml
Module Type PARAM_TYPE.
  Parameter signature : Param \rightarrow ParamSignature.
  Parameter isFinal : Param \rightarrow bool.
  Parameter isNullable : Param \rightarrow bool.
End PARAM_TYPE.
```

Thus, the basic syntax requires an entity declaration to provide explicit values for all modifiers. As an example, a parameter declaration ‘paramDecl [final] int_t x name’ would translate to ‘PARAM.Build_t (PARAMSIGNATURE.Build_t x_name int_t) true false’. (true for `isFinal` and `false` for `isNullable`.) The full syntax changes these cumbersome declarations into forms that resemble more closely the original syntax: ‘paramDecl [final] int_t x_name’, where `int_t` corresponds to the integer type and `x_name` defines the translation of the name of `x`. As seen in the example, the new declaration function can be given a list of modifiers, defined through the new `Modifier` type.

```ocaml
Inductive Modifier : Set :=
  | public | protected | private
  | abstract | static | final
  | native
  | spec_public | spec_protected
  | model | ghost
  | pure
  | instance
  | helper
  | non_null | nullable
```
| monitored |
| uninitialized |
| code |
| implicit_constructor |

Listing 4.24: Extended syntax data type for modifiers.

Declaration function `paramDecl : list Modifier → type → ParamName → PARAM.t` not only supports omitted modifiers (in the example, modifier `nullable` was omitted, and is thus implicitly `non_null`) but also flattens the nested application of constructor `PARAMSIGNATURE.Build_t`: its arguments are given to `paramDecl` directly.

The full syntax provides declaration functions in the style of `paramDecl` for all kinds of declarations. The values for the different kind of modifiers are taken from the JML Reference Manual, Appendix B [10].

4.3.4 Rewritings

The rewritings of extended syntax constructs in terms of basic syntax constructs are initiated in the implementations of the declaration functions. For better understanding, we differentiate between structure-transforming and structure-preserving and between local and non-local rewritings.

A structure-transforming rewriting is one that transforms a full syntax data type into a basic syntax data type, whereas a structure-preserving rewriting operates on a basic syntax type only.

A local rewriting is one that has no information about the program other then the data type it is rewriting. A non-local rewriting in contrast has information about other data types from the program as well.

The declaration function for loop annotations, `loopAnnotationDecl : Label → list (LoopAnnotationTag * Expr) → LOOP_ANNOTATION.t`, is a structure-transforming local rewriting. It is structure-transforming since it transforms a list of loop invariants and loop variants into a single loop invariant value (`LOOP_ANNOTATION.expression`).

A second structure-transforming local rewriting is `rewriteFullQuantifier`: `rewriteFullQuantifier : FullQuantifier → Expr. Type FullQuantifier provides a single constructor FullQuantification : Quantifier → list Var → optional Expr → Expr → FullQuantifier`, the “pendent” to the `Quantification` constructor of type `Expr. rewriteFullQuantifier (FullQuantification q vs r e)` thereby rewrites a quantified expression with possibly multiple variables `vs` in terms of a simple basic syntax expression.

Another rewriting is `rewriteInvariants`: `rewriteInvariants : list INVARIANT.t → ENTITY.t → list INVARIANT.t. rewriteInvariant invs e results in a list of invariants, where for every `non_null` field `f` of reference type a new invariant ‘`f != null`’ is appended to `invs`. This rewriting is structure-preserving and non-local (it requires information about fields of type `e`).

The most complex of the extended syntax rewritings is the desugaring of method specification cases:

5modifier `implicit_constructor` is unofficial and private to the formalization. It tags the implicit zero-argument default constructor that needs special treatment in method specification desugarings. (see section 4.4.4)
4.3 Extended Syntax

\[ \text{rewriteFullSpecification} \colon \text{list FULL\_SPEC\_CASE.t \to Method \to ENTITY.t \to list SpecificationCase}. \]

\text{rewriteFullSpecification scs m e} desugars the list of full specification cases \text{scs} of method \text{m} declared in type \text{t} into a list of basic specification cases. \text{rewriteFullSpecification} is a non-local, structure-transforming rewriting (transformation of list \text{FULL\_SPEC\_CASE.t} into list \text{SpecificationCase}).

Fortunately, both non-local rewritings \text{rewriteInvariants} and \text{rewriteFullSpecification} only require parent node values as additional information: In the case of \text{rewriteInvariants} type \text{e} is the direct parent of the list of invariants \text{invs}. In the case of \text{rewriteFullSpecification}, type \text{e} is the parent of method \text{m}, and \text{m} is the parent of the list of specification cases \text{scs}. Thus, these rewritings can be done with a simple traversal of the syntax tree of the corresponding type declaration \text{e}. This AST traversal is initiated in the common implementation \text{typeDecl} of the corresponding type declaration functions \text{classDecl} and \text{interfaceDecl}.

4.3.5 Notations

In section [4.1.4] on basic syntax notations, the notation \text{loop\_annotation} for loop annotations has not yet fully been explained. The reason being that this notation is actually just a synonym for \text{loopAnnotationDecl}.

The use of quantified expressions using either constructor \text{Quantification} of \text{Expr} or \text{rewriteFullQuantifier}/\text{FullQuantification} is very cumbersome and ugly to read. The formalization thus defines notations that make translated quantified expressions appear very much like the original forms:

\[ \text{Notation } \forall \text{ vs } ; r \text{ ; e} := \]
\[ (\text{rewriteFullQuantifier} \text{ (FullQuantification Forall vs r e)}) \]
\[ \text{at level 0, e at level 200)} : \text{jml\_scope}. \]

Thus, a quantified expression \‘forall vs; r; e\‘ is in fact just a shorthand for the more cumbersome \‘\text{rewriteFullQuantifier} \text{ (FullQuantification Forall vs r e)}\‘.

A variable declaration statement, such as \‘final int i = 0\‘ is represented in the formalization using constructor \text{varDeclStmt} : \text{VAR.t \to Expr \to StatementType}. Two notations \text{var\_decl\_stmt} and \text{var\_decl\_stmtM} \cite{11} are defined for this case to hide the use of the \text{VAR.t} and \text{VARSIGNATURE.t} constructors as has been discussed for parameters in section [4.3.3]. The notation uses the declaration function \text{varDecl} : \text{list Modifier \to type \to VarName \to VAR.t} to make the declaration appear like the original form. The example \‘final int i = 0\‘ translates to \‘\text{var\_decl\_stmtM [final] int_t i_name (int 0)}\‘.

The full syntax also defines notations to make the creation of specification cases more natural. Using notations, a specification case

\text{normal\_behaviour}
\text{requires} x \neq null
\{\}
\text{requires} x.getValue() \geq 0;
\text{ensures} \ \backslash result == x.getValue();
\text{also}
\text{requires} x.getValue() < 0;
\text{ensures} \ \backslash result \geq 0;\]

\cite{10}Definition \text{loop\_annotation} := \text{loopAnnotationDecl}.
\cite{11}\text{var\_decl\_stmtM} is the variant with modifier list argument, \text{var\_decl\_stmt} the variant without modifier list argument.
can be represented as

```plaintext
spec_case [public] normal_behaviour (
    nested_case nil nil
    [requires (: (var x_decl) != null :)]
    |
    simple_case nil nil
    [requires (: (callT (var x_decl) getValue []) >= (int 0) :)]
    [ensures (: \result == (callT (var x_decl) getValue []) :)]
    ;
    simple_case nil nil
    [requires (: (callT (var x_decl) getValue []) < (int 0) :)]
    [ensures (: \result >= (int 0) :)]
)|%jml_nb
```

Listing 4.26: Example of a `normal_behaviour` specification case. (Extended syntax notation)

Notation `spec_case` is a synonym to declaration function `methodSpecDecl false : list Modifier
→ SpecCaseType → GENERIC_SPEC_CASE.t → FULL_SPEC_CASE.t`. Notation `simple_case` just hides an application of `GENERIC_SPEC_CASE.Build_t`, were the `genericBody` argument is fixed to be a simple body argument:

**Definition** simple_case fvd ovd sh (simple:list MethodSpecClause) :=
    GENERIC_SPEC_CASE.Build_t fvd ovd sh (inl _ simple).

Notation `nested_case` is defined analogously with argument (nested:list GENERIC_SPEC_CASE.t) instead of argument `simple` and the use of `inr` instead of `inl`.

Notation `{| generic-spec-case1; ...; generic-spec-case_n |}%jml` makes the declaration appear very JML-like but is in fact just a synonym to the normal list notation `{generic-spec-case1; ...; generic-spec-case_n}`.

### 4.3.6 Implementation Remarks

This section discusses a technical problem that emerge as a consequence to the formalization of method specification clauses as a single inductive data type:

```plaintext
Inductive MethodSpecClause : Type :=
| ensuresC (redundant : bool) (pred : optional Expr)
| signalsC (redundant : bool) (pair : Var * optional Expr)
| signalsOnlyC (redundant : bool) (types : list type)
| ...
```

A list of such clauses can be extracted from a simple specification case using selector function: `GENERIC_SPEC_CASE.genericBody`:

```plaintext
GENERIC_SPEC_CASE.t → (list MethodSpecClause) + (list t).```

However, most rewriting function implementations are only interested in a list of clauses of *one* kind, say ensures clauses.
In such a case, it would be easiest to define a function of the form
\[ \text{extractEnsuresClauses} : \text{list MethodSpecClause} \rightarrow \text{list (optional Expr)} \] (for ensures clauses) to extract the desired clauses. Unfortunately, such extraction functions would have to be defined for every clause kind separately. Even more, functions that operate on an arbitrary, but single clause kind would have to make a case analysis on the clause kind and apply the appropriate extraction function.

An example of such a function is \text{addDefault} that adds the default clause of a certain kind to a list of clauses, if no clause of the given kind is present yet. This function would first have to find all clauses of the given kind which is not possible with the discussed approach of extraction functions.

The solution to this problem is based on the idea of assigning \textit{tags} to data: Every clause kind is associated a unique tag:

\begin{verbatim}
Inductive tag : Set :=
| ensuresT
| signalsT
| signalsOnlyT
| ...
\end{verbatim}

Listing 4.27: \textit{tag} data type for method specification clauses.

Function \text{addDefault} is in this way given the signature
\[ \text{addDefault} : \text{tag} \rightarrow \text{list MethodSpecClause} \rightarrow \text{list MethodSpecClause}. \]
‘addDefault t l’ thus adds a default clause of tag type t (e.g. for an \textit{ensures} clause: tag \textit{ensuresT}) to clause list l if no such clause is present in l. Then, a generalised extraction function should be built on the basis of this tag type. The result of this extraction function must certainly depend on the tag type, it is thus a \textit{dependent type}: \text{mapType} : \text{tag} \rightarrow \text{Type}. In the case of method specification clauses this type is defined as

\begin{verbatim}
Definition mapType (t:tag) : Type :=
match t with
| ensuresT => optional Expr
| signalsT => (Param * optional Expr)
| signalsOnlyT => list type
| ...
\end{verbatim}

Listing 4.28: Dependent type \text{mapType}; result type of the generalised extraction function.

Then the extraction function could be defined as
\[ \text{extract} : \forall t:tag, \text{list MethodSpecClause} \rightarrow \text{list mapType t}, \] but we took a more general approach. First the extraction function is generalised to work on an arbitrary \textit{data} type (here: \textit{MethodSpecClause}). Then the extraction function is split into two separate functions:
\[ \text{filterTag} (t:tag) (l:list data) : \text{list} \{d:data | t=\text{tagOf d}\} \] and
\[ \text{mapData2Type} : \forall t:tag, \text{list} \{d:data | t=\text{tagOf d}\} \rightarrow \text{list} \{\text{mapType t}\}. \]
‘filterTag t l’ thereby extracts all clauses of tag type t resulting in a list l not of type ‘mapType t’ but type ‘\{d:data | t=\text{tagOf d}\}’. Function \text{tagOf}: data \rightarrow tag simply associates the corresponding tag to a data item (= clause kind). Result ‘\text{list} \{d:data | t=\text{tagOf d}\}’ thus contains all data items d whose tag ‘tagOf d’ is equal to the desired tag t and a proof of this fact. Applying function \text{mapData2Type} to the result yields a result of the desired type
The definition of a generalised function mapData2Type is only possible with the help of a function mapF : \forall t:tag, \{d:data \mid t=tagOf d\} \rightarrow \text{mapType } t. This function is indeed implemented for method specification clauses. As an example, for arguments ensuresT and data item ‘ensuresC false expr’, it merely results in value expr.

Why is this split of extract into filterTag and mapData2Type useful? It is useful because type ‘list \{d:data \mid t=tagOf d\}’ is useful. The implementation of the rewritings, for instance, use variants of the familiar map and fold operations defined for this special list type ‘list \{d:data \mid t=tagOf d\}’.

For more details on this idea of tagged lists, see the implementation (module TaggedList.v and module Full2BasicImpl.v for its application for method specification clauses).

### 4.4 Rewriting Details

#### 4.4.1 Loop Annotations

The rewriting of loop annotations, initiated by |loopAnnotationDecl| label annotations

|loopAnnotationDecl|

: Label \rightarrow list (LoopAnnotationTag * Expr) \rightarrow LOOP_ANNOTATION.t), rewrites the variants and invariants given in the list of annotations as a single invariant. Each variant expression e is thereby rewritten as an invariant ‘(0 <= e) \&\& (e < \text{old}(e, label))’, i.e. it is always the case that variant e is non-negative and that the value of e is strictly smaller than the old value of e at label, where label corresponds to the label of the loop body statement.

```jml
definition rewriteVariant (lbl:Label) (e:Expr) : Expr :=
  let lit0 := literal (IntLiteral 0%Z) in
  ( lit0 <= e ) \&\& ( e <= \text{oldl} e at lbl ) )%jml.
```

Listing 4.29: Rewriting of loop annotations.

The single invariant (LOOP_ANNOTATION.expression) consists of a conjunction of all invariants and all variants rewritten by rewriteVariant. The same procedure is applied to all redundant variants and invariants and stored in LOOP_ANNOTATION.expression_redundantly.

#### 4.4.2 Quantified Expressions with Multiple Variables

The rewriting of quantified expressions with multiple variables, initiated by ‘rewriteFullQuantifier q’ (rewriteFullQuantifier : FullQuantifier \rightarrow Expr) rewrites quantified expression q in terms of a nested expression of simple (single variable) quantified expressions. The rewriting is defined by:

```jml
definition rewriteFullQuantifier (fq:FullQuantifier) : Expr :=
  let (q,l,range,expr) := fq in rewriteFullQuantifier_rec q l range expr.
```

|12| extract : \forall t:tag, list data \rightarrow list mapType t could thus be defined as

```jml
definition extract t l := mapData2Type t (filterTag t l).
```


4.4 Rewriting Details

Fixpoint rewriteFullQuantifier_rec
(q:Quantifier) (l:list Var) (range:option Expr) (expr:Expr) {struct l} : Expr :=
match l with
| nil ⇒ false%jml (* error *)
| (v::nil) ⇒ Quantification q v range expr
| (v::vs) ⇒ Quantification q v None (rewriteFullQuantifier_rec q vs range expr)
end.

Listing 4.30: Rewriting of quantified expressions with multiple variables.

That is, rewriteFullQuantifier unpacks the FullQuantifier argument record and delegates the rewriting to rewriteFullQuantifier_rec a recursively defined function (structurally decreasing in argument expr). Note that case nil of the case analysis on l never matches l since type-checking prevents quantified expressions without a declaration of at least one variable.

Example ‘\forall int i,j; 0 <= i \&\& i < j \&\& j < 10; a[i] < a[j]’ from the JML Reference Manual, section 11.4.24.1 [10] is translated into quantified expression

\begin{verbatim}
(* We assume that a c.t. the declaration of array a, 
   i and j to the variable declarations of i and j, 
   and i\_name and j\_name to the corresponding name declarations. *)

(* with use of notations: *)
forall
  [var\_decl int_t i\_name; var\_decl int_t j\_name];
  0 <= i \&\& i < j \&\& j < 10;
  (array a i) < (array a j)

(* without use of notations: *)
rewriteFullQuantifier Forall
  FullQuantification
    [var\_decl int_t i\_name; var\_decl int_t j\_name]
    (0 <= i \&\& i < j \&\& j < 10)
    ((array a i) < (array a j))

\end{verbatim}

The rewriting of this quantified expression results in

‘\forall int i; ; (forall int j; 0 <= i \&\& i < j \&\& j < 10; a[i] < a[j])’ which is represented as

Quantification (var\_decl int_t i\_name) None
Quantification (var\_decl int_t j\_name)
  (0 <= i \&\& i < j \&\& j < 10)
  ((array a i) < (array a j))

in the formalization.

This rewriting is done for all kinds of quantified expressions: universal and existential quantifiers \forall and \exists, generalised quantifiers \max and \min, \product and \sum and numerical quantifier \num\_of.
4.4.3 Implicit Invariants

Every non_null field (or model field) \( f \) (of reference type) of a type \( e \) defines an implicit invariant ‘\( f \neq \text{null} \)’. `rewriteInvariants invs e` adds these invariants to the list of invariants `invs` of type \( e \).

```
Definition rewriteInvariants (invs:list INVARIANT.t) (e:ENTITY.t) : list INVARIANT.t :=
let fieldInvs :=
  fieldNonNullInvariants (ENTITY.name e) (ENTITY.fields e) in
let modelFieldInvs :=
  modelFieldNonNullInvariants (ENTITY .name e) (ENTITY.modelFields e) in
invs +++ fieldInvs +++ modelFieldInvs.
```

Listing 4.31: Rewriting of implicit invariants.

```
fieldNonNullInvariants : EntityName → list Field → INVARIANT.t gathers together all additional invariants generated by non_null fields:

Definition fieldNonNullInvariant (eName:ClassName) (f:Field) : option INVARIANT.t :=
if FIELD.isNullable f
  || negb (isReferenceType (FIELDSIGNATURE.type (FIELD.signature f)))
then None
else let target := if FIELD.isStatic f then None else Some (this%jml) in
  let fExpr := field (eName, (FIELD.signature f)) target in
  Some (INVARIANT.Build_t
    (fExpr != null)%jml
    (FIELD.visibility f)
    (FIELD.isStatic f)
    false
  ).
```

Listing 4.32: fieldNonNullInvariant

'fieldNonNullInvariant (ENTITY.name e) f' results in `None` if field \( f \) is nullable or not of reference type, otherwise in an invariant ‘\( f \neq \text{null} \)’. Invariant ‘\( f \neq \text{null} \)’ has the same visibility as \( f \) and is static iff \( f \) is static.

4.4.4 Method Specification Cases

The desugaring of JML method specification cases in discussed in depth in technical report #00-03e by Raghavan and Leavens [16]. The desugarings that are part of this formalization closely follow report #00-03e. To facilitate comparison, nomenclature is mostly taken from the report as well.

The author’s divide the desugaring into several steps:

3.1 Desugaring Non_Null for Arguments

3.2 Desugaring Non_Null Results
The desugarings as done by `rewriteFullSpecification : list FULL_SPEC_CASE.t → Method → ENTITY.t → list SpecificationCase` include desugarings 3.1 – 3.6, with the exception of not combining heavyweight specification cases of the same visibility (part of desugaring 3.6), and desugarings 3.8 and 3.9. Desugarings 3.7, 3.10 and 3.11 are not part of the rewrite from full specification cases to basic specification cases but are treated within the definition of the semantics of JML (module `JMLSemantics.v`). The individual desugaring steps are designed as transformation functions that operate on a list of `FULL_SPEC_CASE.t` values, that is, each rewriting is a function `list FULL_SPEC_CASE.t → ... → list FULL_SPEC_CASE.t`.

In the nomenclature of section 4.4, the rewritings are structure-preserving and mostly non-local, needing information from the parent class and/or parent type declaration. The advantage of this design is that the individual transformation functions could easily be applied in different orders or some transformation could be disabled for debugging purposes. More importantly, each desugaring is an independent unit, can be tested separately and could be specified and verified modularly.

The following subsections discuss the individual desugarings (transformation functions). The section ends with the discussion of the consolidation function `rewriteFullSpecification`.

The following listing remains the formalization of full method specification cases.

---

```lean
Module Type FULL_SPEC_CASE_TYPE.
  Parameter t : Type.
  Parameter specCaseType : t → SpecCaseType.
  Parameter visibility : t → option Visibility.
  Parameter isRedundant : t → bool.
  Parameter isCodeContract : t → bool.
  Parameter genericSpecCase : t → GENERIC_SPEC_CASE.t.
End FULL_SPEC_CASE_TYPE.

Module Type GENERIC_SPEC_CASE_TYPE.
  Parameter t : Type.
  Parameter forallVarDecl : t → list FORALL_VAR_DECL.t.
  Parameter oldVarDecl : t → list OLD_VAR_DECL.t.
  Parameter specHeader : t → list SpecHeader.
  Parameter genericBody : t → (list MethodSpecClause) + (list t).
End GENERIC_SPEC_CASE_TYPE.
```
Desugaring Non_Null for Arguments

This desugaring adds an additional requires clause ‘\texttt{p \neq null}’ for every non\_null parameter \texttt{p} of reference type to every specification case. If no specification case is given, these additional requires clauses are added as a lightweight specification case.

\begin{quote}
\textbf{Definition} \texttt{desugarNonNullArguments (specs:list FULL\_SPEC\_CASE.t) (m:METHOD.t)} : list FULL\_SPEC\_CASE.t :=
\end{quote}

\begin{quote}
\texttt{let reqs := paramsNonNullRequires (METHODSIGNATURE.parameters (METHOD.signature m)) in}
\end{quote}

\begin{quote}
\texttt{let defaultCase := GENERIC\_SPEC\_CASE.Build_t nil nil reqs (inl \_ nil) in}
\end{quote}

\begin{quote}
\texttt{let defaultSpecs := FULL\_SPEC\_CASE.Build_t lightweight None false false defaultCase in}
\end{quote}

\begin{quote}
\texttt{match reqs, specs with}
\end{quote}

\begin{quote}
\texttt{| nil, _ => specs (* 1 *)}
\end{quote}

\begin{quote}
\texttt{| _, nil => defaultSpecs :: nil (* 2 *)}
\end{quote}

\begin{quote}
\texttt{| _, _ => map (fun s => addHeaders s reqs) specs (* 3 *)}
\end{quote}

\begin{quote}
\texttt{end.}
\end{quote}

Listing 4.33: Desugaring Non\_Null for Arguments

\begin{quote}
\texttt{paramsNonNullRequires : list Param \rightarrow list SpecHeader results in a list of specification headers of ‘\texttt{requires p \neq null}’ clauses for a given list of parameter declarations. It is defined analogously to fieldNonNullInvariants discussed in section 4.4.3 on implicit invariants.}
\end{quote}

Case analysis on ‘\texttt{reqs, specs}’ distinguished three cases:

1. If no additional requires clauses are generated (there exist no null\_null parameter of reference type), the result is equivalent to the given \texttt{specs} argument.
2. If no specification case is given (‘\texttt{specs=\_\_}’), the result is a lightweight specification case with the additional requires clauses.
3. In all other cases, the result is equivalent to the given \texttt{specs} where the additional requires clauses are added to the header of each specification case (\texttt{addHeaders}).

\begin{quote}
\textbf{Definition} \texttt{addHeaders (specs:FULL\_SPEC\_CASE.t) (reqs:list SpecHeader) : FULL\_SPEC\_CASE.t :=}
\end{quote}

\begin{quote}
\texttt{let (sct,v,r,cc,case) := specs in}
\end{quote}

\begin{quote}
\texttt{FULL\_SPEC\_CASE.Build_t sct v r cc (GENERIC\_SPEC\_CASE.Build_t nil nil reqs (inr \_ (case::nil))).}
\end{quote}

Listing 4.34: Definition of the \texttt{addHeaders} helper function.

Desugaring Non_Null Results

This desugaring adds an additional ensures clause ‘\texttt{\_result \neq null}’ to every simple body of every (non-exceptional) specification case, if the method is declared \texttt{non\_null}.\footnote{Exceptional specification cases cannot have ensures clauses.} If no specification case
is given, the additional ensures clause is added as a lightweight specification case with a simple body.

**Definition** desugarNonNullResult (specs:list FULL_SPEC_CASE.t) (m:METHOD.t) : list FULL_SPEC_CASE.t :=

let ens := resultNonNullEnsures m in
let defaultCase ens’ :=
    GENERIC_SPEC_CASE.Build_t nil nil nil (inl _ (ens’::nil)) in
let defaultSpecs ens’ :=
    FULL_SPEC_CASE.Build_t lightweight None false false (defaultCase ens’) in

match ens, specs with
| None, _ ⇒ specs (* 1 *)
| Some ens’, nil ⇒ defaultSpecs ens’ :: nil (* 2 *)
| Some ens’, _ ⇒ map (fun s ⇒ addBody s ens’) specs (* 3 *)
end.

Listing 4.35: Desugaring Non_Null Results

resultNonNullEnsures : Method → option MethodSpecClause results in ‘\result != null’ if the method is declared non_null, otherwise it results in None.

Again, the function is defined with a case analysis: ‘match ens, specs’ distinguishes three cases:

1. If the method is declared nullable (‘resultNonNullEnsures m = None’), the result is equivalent to the given specs.
2. If the method is declared non_null, but no specification case is given (‘specs=nil’), the result is a lightweight specification case with a simple body containing an ensures clause ‘\result != null’.
3. In all other cases, the additional ensures clause is added to every simple body of every (non-exceptional) specification case (addBody).

Function addBody enables adding additional simple body clauses to a full specification case. If the given clause is an ensures clause and the case is exceptional or if the given clause is a signals or signals_only clause and the case is a normalBehaviour case, addBody has no effect.

**Definition** addBody (specs0:FULL_SPEC_CASE.t) (clause:MethodSpecClause)

let (sct,v,r,cc,case) := specs0 in
let addClause (clauses:list MethodSpecClause) := (clause :: clauses) in
let specs1 := FULL_SPEC_CASE.Build_t sct v r cc (clausesMap addClause case) in

match sct with
| lightweight ⇒ specs1
| behaviour ⇒ specs1
| normalBehaviour ⇒
    if isSignalsClause clause then specs0 else specs1
| exceptionalBehaviour ⇒
    if isEnsuresClause clause then specs0 else specs1
end.
Desugaring Pure

This desugaring adds additional simple body clauses to every specification case, if the method is declared pure. The additional clauses are ‘diverges false’ and, in case of a constructor ‘assignable this.*’ and in case of a normal method ‘assignable \nothing’. Again, if no specification case is given, the clauses are added as a lightweight specification case with a simple body.

Definition desugarPure (specs:list FULL_SPEC_CASE.t) (m:METHOD.t) : list FULL_SPEC_CASE.t :=
let addBody2 clause1 clause2 spec := addBody (addBody spec clause1) clause2 in
let div := divergesC false (: false' :)%jml in
let ass :=
  match METHOD.kind m with
  | Constructor ⇒ assignableC false (: [fieldAll this] :)%jml
  | _ ⇒ assignableC false (: \nothing :)%jml
end in

let defaultCase :=
  GENERIC_SPEC_CASE.Build_t nil nil nil (inl _ (div :: ass :: nil)) in
let defaultSpecs :=
  FULL_SPEC_CASE.Build_t lightweight None false false defaultCase in

match METHOD.isPure m, specs with
| false, _ ⇒ specs (* 1 *)
| true, nil ⇒ defaultSpecs :: nil (* 2 *)
| true, _ ⇒ map (addBody2 ass div) specs (* 3 *)
end.

Desugaring Empty Specifications

This desugaring adds a default specification to the given method if no (non-redundant) cases were explicitly declared with the method and no cases implicitly created as part of desugarings 3.1 – 3.3. The default specification amounts to a lightweight specification case ‘requires \not_specified’ for any non-override method or ‘also requires false’ for an override method.

As discussed in section 3.4 of TR #00-03e [16], the implicit zero-argument default constructor is treated specially in this desugaring: its default specification amounts to a lightweight specification case with an assignable clause that is “a copy of the superclass’s default constructor’s assignable clause”. The assignable clause of the superclass’s default constructor is equivalent to a union of all assignable clauses of the different specification cases. This is described in desugaring 3.10.
Definition desugarEmptySpecification (specs:list FULL_SPEC_CASE.t) (m:METHOD.t) (enc:ENTITY.t) : list FULL_SPEC_CASE.t :=
let req :=
match METHOD.override m with
| None  ⇒ requiresSH false \not_specified\jml
| Some _ ⇒ requiresSH false (: false’)\jml
end in

let defaultCase := GENERIC_SPEC_CASE.Build_t nil nil (req::nil) (inl _ nil) in
let defaultSpecs :=
FULL_SPEC_CASE.Build_t lightweight None false false defaultCase in

match METHOD.isImplicitDefaultConstructor m, isEmptySpecification specs with
| true, _ ⇒ (implicitCtorDefault Specs enc) :: nil (* 1 *)
| false, true ⇒ defaultSpecs :: nil (* 2 *)
| false, false ⇒ specs (* 3 *)
end.

Listing 4.38: Desugaring Empty Specification

Again, the function is defined with a case analysis:
‘METHOD.isImplicitDefaultConstructor m, isEmptySpecification specs’ distinguishes three cases:

1. If the method is the implicit zero-argument default constructor (modifier implicit_constructor set as argument to methodDecl), the default specification is given by implicitCtor-DefaultSpecs.

2. If the method has no specification, the result is equivalent to the default specification ‘requires \not_specified’ or ‘also requires false’.

3. In all other cases, the result is equivalent to the given argument specs.

The default specification for the implicit default constructor, implicitCtorDefaultSpecs, is defined as:

Definition implicitCtorDefaultSpecs (enc:ENTITY.t) : FULL_SPEC_CASE.t :=
let parentCtor :=
match ENTITY.superClass_ enc with
| Some sup ⇒ zeroArgCtor sup
| None ⇒ None
end in

let parentSpecs :=
match parentCtor with
| Some ctor ⇒ METHOD.fullSpecs ctor
| None ⇒ nil
end in

let clauses :=
match assignableUnion parentSpecs with
| Some ole ⇒ (assignableC false ole) :: nil
| None ⇒ nil
end

let case := GENERIC_SPEC_CASE.Build_t nil nil nil (inl _ clauses) in
FULL_SPEC_CASE.Build_t lightweight None false false case.

Listing 4.39: Default specification for the implicit constructor

assignableUnion : list FULL_SPEC_CASE.t → option (optional (list Expr)) thereby cre-
creates the union of all assignable clauses (type of an assignable list: 'optional (list Expr)') of
the given specification case list.

Two assignable clauses \(a_1\) and \(a_2\) are merged as follows: If \(a_1\) or \(a_2\) is NotSpecified, the result
is the other clause. Otherwise the two clauses are equivalent to two lists \(l_1\) and \(l_2\) of locations.
These are merged by appLocations:

\[
definition\text{appLocations} (l_1 l_2:list\ \text{Expr}) : list\ \text{Expr} :=
\text{match} \ l_1, l_2 \text{ with}
\text{| Nothing::nil, } _ \ ⇒ l_2
\text{| Everything::nil, } _ \ ⇒ l_1
\text{| }, Nothing::nil \ ⇒ l_1
\text{| }, Everything::nil \ ⇒ l_2
\text{| }, _ \ ⇒ \text{List.app} l_1 l_2
\end
\]

Desugaring Nested Specifications

This desugaring flattens all nested specification cases. A specification case \(sc\) is called nested if
its generic-body is nested:
\[
\text{GENERIC_SPEC_CASE.genericBody(FULL_SPEC_CASE.genericSpecCase sc)} = \text{inr } _ \text{nestedBody}
\]
A specification case is called flat if its generic-body is a simple-body:
\[
\text{GENERIC_SPEC_CASE.genericBody(FULL_SPEC_CASE.genericSpecCase sc)} = \text{inl } _ \text{simpleBody}
\]
The desugaring is described in the technical report [16] as well as in the JML Reference Manual,
section 9.6.5 [10] (Semantics of nested behavior specification cases), by means of a semi-formal
example:

\[
\text{spec-var-decls \ spec-header \ } \{\ \}
\text{GenSpecCase1 \ spec-var-decls \ spec-header \ also \ GenSpecCase1}
\text{... \ -- desugars to -- \ ...
\text{... \ also \ GenSpecCaseN \ spec-var-decls \ spec-header \ also \ GenSpecCaseN
\text{|}\}
\]

\[1^{4}\] Inductive optional (A:Type) := Specified A → optional A | NotSpecified : optional A.
Listing 4.40: Semi-formal description of the desugaring of nested specifications.

In the formalization this amounts to:

\[
\text{Definition \ desugarNested (specs:list FSC.t) : list FSC.t :=}
\]
\[
\text{let \ desugar1 (spec:FSC.t) : list FSC.t :=}
\]
\[
\text{let \ (sct,v,r,cc,case) := spec \ in}
\]
\[
\text{map (FSC.Build_t sct v r cc) (flatten case) in}
\]
\[
\text{flat_map desugar1 specs.}
\]

\[
\text{Fixpoint \ flatten (case:GSC.t) : list GSC.t :=}
\]
\[
\text{let \ addDecls (flattenedCase:GSC.t) :=}
\]
\[
\text{let \ (fvd, ovd, sh, genBody) := case \ in}
\]
\[
\text{GENERIC_SPEC_CASE.Build_t}
\]
\[
\text{(List.app fvd fvd1)}
\]
\[
\text{(List.app ovd ovd1)}
\]
\[
\text{simpleBody1 in}
\]
\[
\text{let \ fix flattenNested (l:list GSC.t)}
\]
\[
\text{match \ l \ with}
\]
\[
| \ \text{nil} \ \Rightarrow \ \text{nil}
\]
\[
| \ (nb::nbs) \ \Rightarrow \ \text{List.app (flatten nb) (flattenNested nbs)}
\]
\[
\text{end \ in}
\]
\[
\text{match \ genBody \ with}
\]
\[
| \ \text{inl simpleBody} \ \Rightarrow \ \text{case :: nil}
\]
\[
| \ \text{inr nested} \ \Rightarrow \ \text{map addDecls (flattenNested nested)}
\]
\[
\text{end.}
\]

Listing 4.41: Desugaring Nested Specifications

That is, the flattening function \texttt{flatten} is applied to the generic-spec-case of every specification case. \texttt{flatten} recursively flattens its nested generic-spec-cases and adds the variable declarations \((fvd, ovd)\) and requires clauses \((sh)\) to every recursively flattened case \((\texttt{addDecls})\). Finally, a full specification case is built out of every generic-spec-case in the result of \texttt{flatten}.

Desugaring Lightweight, Normal, and Exceptional Specifications

This desugaring transforms lightweight, normal- and exceptional behaviour specification cases into behaviour specification cases. For \texttt{normal\_behaviour} cases this amounts to adding a clause `\texttt{signals (Exception) false}` to each simple body. For \texttt{exceptional\_behaviour} cases, a clause `\texttt{ensures false}` is added to every simple body. For lightweight specification cases, the visibility is set to the visibility of the enclosing method and a default clause is is added for every clause kind that is missing in this case. Notice that this desugaring is required to be applied \textit{after} the flattening process such that every specification case has a simple body only.
Definition desugarBehaviour (specs:list FULL_SPEC_CASE.t) (m:METHOD.t)
: list FULL_SPEC_CASE.t :=
let desugar1 (spec0:FULL_SPEC_CASE.t) : FULL_SPEC_CASE.t :=
let vis :=
match FULL_SPEC_CASE.visibility spec0, FULL_SPEC_CASE.specCaseType spec0 with
| Some v', _ ⇒ v' (* case only possible for heavyweight case *)
| None, lightweight ⇒ METHOD.visibility m
| None, _ ⇒ Package (* default visibility for heavyweight case *)
end in
match FULL_SPEC_CASE.specCaseType spec0 with
| lightweight ⇒
  let spec1 := setSpecCaseType spec0 behaviour in
  let spec2 := setVisibility spec1 (Some vis) in
  let spec3 := addDefaults spec2 m (lightweightDefaults spec2 m) in
  spec3
| normal_behaviour ⇒
  let spec1 := setSpecCaseType spec0 behaviour in
  let spec2 := setVisibility spec1 (Some vis) in
  let spec3 :=
    addBody spec2 (signalsC false (Exception_e, (: false' :)%jml)) in
  spec3
| exceptional_behaviour ⇒
  let spec1 := setSpecCaseType spec0 behaviour in
  let spec2 := setVisibility spec1 (Some vis) in
  let spec3 := addBody spec2 (ensuresC false (: false' :)%jml) in
  spec3
| behaviour ⇒
  setVisibility spec0 (Some vis)
end in
map desugar1 specs.

The adding of default clauses (e.g. a default ensures clause is added to a lightweight specification case, if no ensures clause is present), is formalized by function addDefaults : FULL_SPEC_CASE.t -> METHOD.t -> FullSpecCaseDefaults -> FULL_SPEC_CASE.t. Function addDefaults draws default clauses from argument defaults : FullSpecCaseDefaults. defaults provides a function bodyDefault that returns the default clause for any given kind of clause (see section 4.3.6 for an explanation of type ClauseTag).

Record FullSpecCaseDefaults : Type := Build_FullSpecCaseDefaults {
  headerDefault : SpecHeader;
  bodyDefault : ClauseTag -> MethodSpecClause
}

The formalization provides two FullSpecCaseDefaults values: lightweightDefaults and heavyweightDefaults, both of which have signature FULL_SPEC_CASE.t -> Method -> FullSpecCaseDefaults. The use of the two arguments is explained below. The default values for the
4.4 Rewriting Details

<table>
<thead>
<tr>
<th>Clause Kind</th>
<th>Default for lightweight case</th>
<th>Default for heavyweight case</th>
</tr>
</thead>
<tbody>
<tr>
<td>requires</td>
<td>\not_specified (9.9.2)</td>
<td>(: true') (9.9.2)</td>
</tr>
<tr>
<td>ensures</td>
<td>\not_specified (9.9.3)</td>
<td>(: true') (9.9.3)</td>
</tr>
<tr>
<td>signals</td>
<td>(Exception_e, \not_specified) (9.9.4)</td>
<td>(Exception_e, (: true') (9.9.4)</td>
</tr>
<tr>
<td>signals_only</td>
<td>see below (9.9.5)</td>
<td>see below (9.9.5)</td>
</tr>
<tr>
<td>diverges</td>
<td>(: false') (9.9.7)</td>
<td>(: false') (9.9.7)</td>
</tr>
<tr>
<td>when</td>
<td>\not_specified (9.9.8)</td>
<td>(: true') (9.9.8)</td>
</tr>
<tr>
<td>assignable</td>
<td>\not_specified (9.9.9)</td>
<td>(: [everything] :) (9.9.9)</td>
</tr>
<tr>
<td>accessible</td>
<td>\not_specified (9.4)</td>
<td>(: [everything] :)</td>
</tr>
<tr>
<td>callable</td>
<td>\not_specified (9.4)</td>
<td>\not_specified / (: EveryMethod :)</td>
</tr>
<tr>
<td>measured_by</td>
<td>(\not_specified, None) (9.9.12)</td>
<td>(\not_specified, None) (9.9.12)</td>
</tr>
<tr>
<td>captures</td>
<td>\not_specified (9.4)</td>
<td>(: [everything] :) (9.9.13)</td>
</tr>
<tr>
<td>working_space</td>
<td>(\not_specified, None) (9.9.14)</td>
<td>(\not_specified, None) (9.9.14)</td>
</tr>
<tr>
<td>duration</td>
<td>(\not_specified, None) (9.9.15)</td>
<td>(\not_specified, None) (9.9.15)</td>
</tr>
</tbody>
</table>

\*within a non-code contract specification case
\*within a code contract specification case

Table 4.2: Default values for method specification clauses

Different clause kinds are given in the table above.

For signals_only, the default clause is the list of checked exceptions declared in the methods throws clause. Unchecked exceptions listed in the throws clause do not enter this default clause. In the formalization this is expressed through the use of function filterCheckedExceptions: ‘filterCheckedExceptions (METHOD.throws m)’. For accessible and callable clauses, it is not entirely clear from the Reference Manual if these clauses can be used in non-code contract specification cases. In any case, sections 9.9.10 and 9.9.11 only define a default value for code contract specification cases, which is \everything for both accessible and callable clauses. We decided to not explicitly disallow these clauses to be part of non-code contract specification cases. We use a default clause of \not_specified for both lightweight and heavyweight non-code contract specification cases. Considering these default values it becomes clear why both functions lightweightDefaults and heavyweightDefaults require arguments \(sc : \text{FULL\_SPEC\_CASE\_t}\) and \(m : \text{Method}\): Argument \(sc\) is required to determine whether a specification case is a code contract (\(\text{FULL\_SPEC\_CASE\_isCodeContract}\)) and argument \(m\) to extract the list of checked exceptions.

Standardising Signals Clauses

This desugaring standardises every signals clause ‘\(\text{signals} \ (\text{ET} \ n) \ P\)’ declaring an exception variable \(n\) of type ET into a signals clause ‘\(\text{signals} \ (\text{Exception} \ e) \ (e \ \text{instanceof} \ \text{ET}) \Rightarrow [(\text{ET})e/n] \ P\)’ declaring a signals variable \(e\) of type Exception. ‘\([(\text{ET}) \ e/n] \ P\’ denotes expression \(P\), where every occurrence of \(n\) is substituted by ‘\(\text{ET}) \ e\’. If exception type ET is equivalent to Exception, the normalisation is equivalent to ‘\(\text{signals} \ (\text{Exception} \ e) \ [e/n] \ P\)’, i.e. the normalisation ensures the use of always the same signals variable \(e\).

\(\text{desugarSignals} : \text{list} \ \text{FULL\_SPEC\_CASE\_t} \rightarrow \text{list} \ \text{FULL\_SPEC\_CASE\_t}\)\(^{15}\) applies a normalisation to every signals clause found in the given list of specification cases:

\(^{15}\)The numbers in parenthesis give the section from the Reference Manual that specify the corresponding default value.
\(^{16}\)The implementation of \text{desugarSignals} is not shown as it used some advanced features of Coq that does not help in understanding its meaning.
Definition normaliseSignals (data:{d:MethodSpecClause | signalsT = TAG.tagOf d}) : MethodSpecClause :=
let (par, oexpr0) := TAG.mapF signalsT data in
let paramType := PARAMSIGNATURE.type (PARAM.signature par) in
let oexpr1 :=
  match oexpr0 with
  | NotSpecified ⇒ NotSpecified
  | Specified expr0 ⇒
    if isExceptionType paramType
    then
      let expr1 := SIGNALS_SUBST.subst par (param Exception_e) expr0 in
      Specified expr1
    else
      let expr1 :=
        SIGNALS_SUBST.subst par (Cast paramType (param Exception_e)) expr0 in
      let expr2 :=
        (((param Exception_e) instanceof paramType) => expr1)%jml in
      Specified expr2
  end in
signalsC false (Exception_e, oexpr1).

Listing 4.44: Normalisation function for signals clauses

‘normaliseSignals d’ exactly performs the standardisation explained above. The substitution function SIGNALS_SUBST.subst is based on a more general substitution function expressionSubstitute : Expr → (Expr → bool) → Expr → Expr → Expr, where ‘expressionSubstitute x eq_x t e’ substitutes term x in expression e by term t. Function eq_x is used to determine if a given expression is equal to substitution term x.

Module SIGNALS_SUBST.
(* bool equality between a parameter and an expression: *)
(* Param_Expr_eqb p e = true iff e = (param q) and PARAM.eq_t p q *)
Definition Param_Expr_eqb p e : bool :=
machine e with
| param q ⇒ PARAM.eq_t p q
| _ ⇒ false
end.

(* subst p t e *)
(* Substitute expression /t/ for parameter p in e. *)
Definition subst p t e :=
  expressionSubstitute (param p) (Param_Expr_eqb p) t e.
End SIGNALS_SUBST.

Listing 4.45: Substitution function for the signals clause normalisation
Desugaring Multiple Clauses of the Same Kind

This desugaring merges multiple clauses of the same kind within a single body into a single clause. For instance, two requires clauses `requires p1; requires p2` are merged into a single requires clause `requires p1 \&\& p2`. The following section describes the individual mergings in Coq code. The mergings presented here are simplified to the point of only merging two clauses into one clause. The implementation generalises this to lists of clauses. Section 4.4.4 contains an example.

Forall Variable Declarations and Old Variable Declarations

Definition mergeForallVarDecls (c1 c2 : FORALL_VAR_DECL.t) : FORALL_VAR_DECL.t :=
let l1 = FORALL_VAR_DECL.vars c1 in
let l2 = FORALL_VAR_DECL.vars c2 in
FORALL_VAR_DECL.Build_t (List.app l1 l2).

Listing 4.46: Merging of forall variable declarations

Merging is analogous for old variable declarations.

Requires Clauses

Note that a requires clause may be a specified expression, `\not_specified` or `\same`. This is formalized by type `optionalSame`.

Inductive optionalSame (A:Type) :=
| SpecifiedOS : A \rightarrow optionalSame A |
| NotSpecifiedOS : optionalSame A |
| Same : optionalSame A. |

Listing 4.47: Extended syntax data type `optionalSame`

Definition mergeRequires (c1 c2 : optionalSame Expr) : optionalSame Expr :=
match c1, c2 with
| SpecifiedOS e1, SecifiedOS e2 \Rightarrow SpecifiedOS (e1 \&\& e2)%jml (* 1 *)
| NotSpecifiedOS, _ \Rightarrow c2 (* 2 *)
| _, NotSpecifiedOS \Rightarrow c1 (* 2 *)
| Same, _ \Rightarrow Same (* 3 *)
| _, Same \Rightarrow Same (* 3 *)
end.

Listing 4.48: Merging of two requires clauses

Notice that a requires clause `\same` is always the only requires clause in a simple body. Thus (* 3 *) is correct.

Ensures Clauses, Diverges Clauses and When Clauses

An ensures clause may be a specified expression or `ensures \not_specified`. This is formalized by type `optional`:
Inductive optional (A:Type) :=
| Specified : A → optional A
| NotSpecified : optional A.

Listing 4.49: Extended syntax data type optional

Definition mergeEnsures (c1 c2 : optional Expr) : optional Expr :=
match c1, c2 with
| Specified e1, Specified e2 ⇒ Specified (e1 &&' e2)%jml
| NotSpecified, _ ⇒ c2
| _, NotSpecified ⇒ c1
end.

Listing 4.50: Merging of two ensures clauses

The merging of diverges clauses and when clauses is done identically.

Signals Clauses

Definition mergeSignals (c1 c2 : Param * optional Expr) : Param * optional Expr :=
match c1, c2 with
| (p1, Specified e1), (p2, Specified e2) ⇒ (p1, Specified (e1 &&' e2)%jml)
| (p1, NotSpecified), _ ⇒ c2
end.

Listing 4.51: Merging of two signals clauses

Note that the correctness of this desugaring depends on the normalisation of signals clauses as performed by desugaring “Standardising Signals Clauses”. In this case, the exception parameter is always equal to Exception e and it does not matter if the parameter of c1 or c2 is taken.

Assignable Clauses, Accessible Clauses and Captures Clauses

Assignable and accessible clauses may both be not_specified or specify a list of storage locations. The merging of storage location lists has already been presented in section “Desugaring Empty Specifications”:

Definition appLocations (l1 l2:list Expr) : list Expr :=
match l1, l2 with
| Nothing::nil, _ ⇒ 12
| Everything::nil, _ ⇒ 11
| _, Nothing::nil ⇒ 11
| _, Everything::nil ⇒ 12
| _, _ ⇒ List.app l1 l2
end.

Listing 4.52: Merging of two lists of storage locations
Storage location lists \( l_1 \) and \( l_2 \) are appended unless either list is equal to \texttt{nothing} or \texttt{everything}.

\[
\begin{align*}
\text{Definition } & \text{mergeAssignable } (c_1 \ c_2 : \text{optional (list Expr)}) \ \\
& : \text{optional (list Expr)} := \\
& \text{match } c_1, c_2 \text{ with} \\
& \text{| Specified e1, Specified e2 } \Rightarrow \text{Specified (appLocations e1 e2)} \\
& \text{| NotSpecified, } \_ \Rightarrow c_2 \\
& \text{| , NotSpecified } \Rightarrow c_1 \\
& \text{end.}
\end{align*}
\]

Listing 4.53: Merging of two assignable clauses

Accessible and captures clauses are merged in the same way.

\textbf{Callable Clauses}

A callable clause is \texttt{not_specified}, \texttt{everything} or a list of method signatures. This is formalized as type \texttt{optional CallableList}, where inductive type \texttt{CallableList} is defined as:

\[
\begin{align*}
\text{Inductive } & \text{CallableList : Type :=} \\
& \text{| EveryMethod : CallableList} \\
& \text{| These list MethodSignature } \rightarrow \text{CallableList.}
\end{align*}
\]

\[
\begin{align*}
\text{Definition } & \text{mergeCallable } (c_1 \ c_2 : \text{optional CallableList}) \ \\
& : \text{optional CallableList :=} \\
& \text{match } c_1, c_2 \text{ with} \\
& \text{| Specified cl_1, Specified cl_2 } \Rightarrow \\
& \text{| These l_1, These l_2 } \Rightarrow \text{Specified (These List.app (l_1 l_2))} \\
& \text{| EveryMethod, } \_ \Rightarrow \text{Specified EveryMethod} \\
& \text{| , EveryMethod } \Rightarrow \text{Specified EveryMethod} \\
& \text{| NotSpecified, } \_ \Rightarrow c_2 \\
& \text{end.}
\end{align*}
\]

Listing 4.54: Merging of two callable clauses

\textbf{Working Space Clauses and Duration Clauses}

According to TR #00-03e, section 3.9 [16], two specified working space clauses are merged as follows:

\[
\begin{align*}
\text{‘working_space } E_1 \text{ if } C_1; \text{ working_space } E_2 \text{ if } C_2’ \text{ desugars to} \\
\text{‘duration Long.min(} C_1 \text{ ? } E_1 : \text{Long.MAX VALUE, } C_2 \text{ ? } E_2 : \text{Long.MAX VALUE) if } C_1 || C_2’.}
\end{align*}
\]

\[
\begin{align*}
\text{Definition } & \text{mergeWorkingSpace } (c_1 \ c_2 : \text{optional Expr } * \text{ option Expr}) \ \\
& : \text{optional Expr } * \text{ option Expr :=} \\
& \text{let } (expr_1, cond_1) := c_1 \text{ in} \\
& \text{let } (expr_2, cond_2) := c_2 \text{ in}
\end{align*}
\]
let cond (ws:optional Expr * option Expr) : Expr :=
match snd ws with
| None ⇒ true%
| Some c ⇒ c
end in

let rewrite1 (ws:optional Expr * option Expr) : optional Expr :=
match fst ws with
| NotSpecified ⇒ NotSpecified
| Specified expr ⇒ Specified ((cond ws)
then' expr
else' field java.lang.Integer.F_MAX_VALUE None)%jml
end in

let merge2 (e1:Expr) (e2:Expr) : Expr :=
(method java.lang.Integer.M_min_5 None [e1; e2])%jml in

match rewrite1 c1, rewrite1 c2 with
| Specified e1, Specified e2 ⇒ (Specified (merge2 e1 e2),
((cond c1) ||' (cond2))%jml)
| NotSpecified, _ ⇒ c2
| _, NotSpecified ⇒ c1
end.

Listing 4.55: Merging of two working space clauses

Notice the use of field signature java.lang.Integer.F_MAX_VALUE and method signature java.lang.Integer.M_min_5. Both are defined in a prelude library. The merging of two duration clauses is done analogously, where Long.MAX_VALUE and Long.min are used instead of Integer.MAX_VALUE and Integer.min.

Signals_Only Clauses and Measured_By Clauses

This formalization does not allow the usage of more than one signals_only clause per flat specification case. Section 9.9.5 of the JML Reference Manual describes the merging of multiple signals_only clauses, but strongly suggests not to use it: “Since this may be confusing, only one signals_only clause should ever be used in a given specification case.”

Multiple measured_by clauses per flat specification case are not supported as well.

Generalisation to Lists of Clauses

The generalisation of the above presented merging functions to lists of clauses is straightforward. As an example, the following definition is the actual merging function for ensures clauses.

Definition mergeEnsures (l:list (optional Expr)) : option (optional Expr) :=
let optionalAnd := liftOptional2 (InfixOp ConditionalAnd) in
match l with
| nil ⇒ None
| _ ⇒ Some (fold_left1 optionalAnd l NotSpecified)
end.

Listing 4.56: Merging of a list of ensures clauses
4.4 Rewriting Details

liftOptional2 is used to abstract

match c1, c2 with
| Specified e1, Specified e2 ⇒ Specified (e1 && e2)%jml
| NotSpecified, _ ⇒ c2
| _, NotSpecified ⇒ c1
end

This case analysis is used in a similar form is almost all merging functions.

Definition liftOptional2 (A:Type) (op:A → A → A) (oa ob:optional A) : optional A :=
match oa, ob with
| Specified a, Specified b ⇒ Specified (op a b)
| NotSpecified, b ⇒ b
| a, NotSpecified ⇒ a
end.

Listing 4.57: Abstraction of merging function case analysis

Putting it All Together

The desugarings can easily be applied in sequence. Function desugarAll does exactly that:

Definition desugarAll (specs0:list FULL_SPEC_CASE.t) (m:METHOD.t) (enc:ENTITY.t) :
list FULL_SPEC_CASE.t :=
let specs1 := desugarNonNullArguments specs0 m in
let specs2 := desugarNonNullResult specs1 m in
let specs3 := desugarPure specs2 m in
let specs4 := desugarEmptySpecification specs3 m enc in
let specs5 := desugarNested specs4 in
let specs6 := desugarBehaviour specs5 m in
let specs8 := desugarSignals specs6 in
let specs9 := desugarMultiple specs8 in
specs9.

Listing 4.58: Desugaring consolidation function desugarAll

There is still a little gap between desugarAll and rewriteFullSpecification : list FULL_SPEC_CASE.t → METHOD.t → ENTITY.t → list SpecificationCase: The resulting list of full specification cases has to be transformed into a list of basic specification cases. This is rather straightforward as desugaring desugarNested has already flattened the specification cases and desugaring desugarMultiple ensures that at most one clause is declared per clause kind. What remains is to add default clauses to ensure that exactly one clause is declared per clause kind. This is only necessary for specification cases that were originally not lightweight, as for those the adding of default clauses is already done in desugaring desugarBehaviour. For the other specification cases it is done in the same fashion, with the help of addDefaults and heavyweightDefaults (see section ‘Desugaring Lightweight, Normal, and Exceptional Specifications’).

The final transformation of a FULL_SPEC_CASE.t value into a SpecificationCase value is semantically trivial but not shown here, as it again uses some advanced Coq features.
### 4.5 Proving Case Study

As part of this work, several small case studies were carried out to evaluate the feasibility of doing proofs on top of the presented Coq formalization of JML and Java. The proofs carried out were intentionally chosen to be small example proofs. The focus did not lay on proving interesting semantic properties but rather to build a basis for later, more interesting proofs.

The two proofs that one of the case studies looks at, treat a simple rewriting of invariants:

```coq
Definition mergeInvariants (invs:list INVARIANT.t) : INVARIANT.t :=
  let and := InfixOp ConditionalAnd in
  let lPred := map INVARIANT.pred invs in
  (* extract invariant exprs *)
  INVARIANT.Build_t (fold_right and true'%jml lPred) Public false false.
```

Listing 4.59: Merging of invariants

The rewriting `mergeInvariants` simply merges all invariants of a given type into one big conjunction consisting of the individual invariants. The case study ignores the visibility and static/instance property of invariants.

The first theorem to prove states that the merged invariant of a given type `c` is equivalent to the list of individual invariants of `c`. This can be formalized by using the semantic function `EvalInvariant`:

```coq
Definition EvalInvariant (p : Program) (c : Class) (m : Method) (h : Heap.t) (fr : Frame.t) : Prop :=
  if METHOD.isHelper m then True 
  else ∀ inv , DefinedInvariant p c inv → EvalPredicate p (INVARIANT.pred inv) h fr.
```

Listing 4.60: Semantic function `EvalInvariant`

‘`EvalInvariant p c m h fr`’ holds exactly if `m` is a helper method or all invariants defined for type `c` hold (`EvalPredicate : Program → Expr → Heap.t → Frame.t → Prop`).

For the first proof, `EvalInvariant` is simplified to only treat invariants declared in type `c`, i.e. premise ‘`DefinedInvariant p c inv`’ is replaced by ‘`In inv (TYPESPEC.invariant (ENTITY.type-Spec c))`’.

Then, the first theorem can be stated as

\[ ∀ p c c' m h fr, c' = rewriteEntity c → (EvalInvariant p c m h fr ↔ EvalInvariant p c' m h fr). \]

(‘`rewriteEntity c`’ thereby performs the rewriting `mergeInvariants`.) Since the rewriting merges two invariants `inv1` and `inv2` into a single invariant `inv1 && inv2`, it soon became clear that an important helper lemma is necessary to prove the above theorem:

\[ ∀ p h fr e1 e2, \]
that is, the evaluation of ‘e1 && e2’, ‘Eval [e1 && e2]’ is equivalent to ‘Eval [e1] ∧ Eval [e2]’.

Without any further assumptions, this lemma can unfortunately not be proven, since EvalPredicate is defined in terms of a more general function EvalExpression : Program → Expr → Heap.t → Frame.t → option value. Not having an assumption that the given expressions e1 and e2 are well-formed boolean expressions, it is not necessarily the case that the evaluation of these two expressions (EvalExpression) results in a boolean value. Thus, the definition of a well-formedness predicate for boolean expressions became necessary.

For simplicity, the case study restricts well-formed boolean expressions to constants true and false and the two connectives && and ||:

\[
\text{Inductive Wf_pred : Expr → Prop :=}
\begin{align*}
| \text{Wf_false} & : \text{Wf_pred false}' \\
| \text{Wf_true} & : \text{Wf_pred true}' \\
| \text{Wf_and} & : \forall e1 \ e2, \text{Wf_pred e1} → \text{Wf_pred e2} → \text{Wf_pred (e1 &&' e2)} \\
| \text{Wf_or} & : \forall e1 \ e2, \text{Wf_pred e1} → \text{Wf_pred e2} → \text{Wf_pred (e1 ||' e2)}.
\end{align*}
\]

Listing 4.61: Well-formedness predicate for boolean expressions

Using this well-formedness predicate, it can be proven that the evaluation of a well-formed boolean expression always results in a boolean value:

\[
\text{Lemma EvalExpression_pred : } \forall p \ h \ fr \ e, \ Wf_pred e \to \exists b, \text{EvalExpression p e h fr} = \text{Some (DOMAIN.Bool b)}.
\]

Listing 4.62: Boolean expressions evaluate to boolean values

Then, a modified version of the above “Eval [e1 && e2] ⇔ Eval [e1] ∧ Eval [e2]” can be proven as well:

\[
\text{Lemma EvalPredicate_and : } \forall p \ h \ fr \ e1 \ e2, \ Wf_pred e1 \to \ Wf_pred e2 \to \ (\text{EvalPredicate p (e1 &&' e2)} \ h \ fr \leftrightarrow (\text{EvalPredicate p e1 h fr} ∧ \text{EvalPredicate p e2 h fr})).
\]

Listing 4.63: Eval [e1 && e2] ⇔ Eval [e1] ∧ Eval [e2]

Using lemma EvalPredicate_and, another helper lemma can be proven that is already very close to the desired goal:

\[
\text{Lemma mergeInvariants_ok : } \forall p \ h \ fr \ l, \ (\forall \text{inv}, \text{In inv} l \to \text{Wf_pred (INVARINT.pred inv)}) \to \ ((\forall \text{inv}, \text{In inv} l → \text{EvalPredicate p (INVARINT.pred inv) h fr}) \leftrightarrow \text{EvalPredicate p (INVARINT.pred (mergeInvariants l)) h fr}).
\]

Listing 4.64: mergeInvariants “preserves” EvalPredicate
The proof makes use of the fact that the result of `mergeInvariants` is well-formed as well:

```
Lemma mergeInvariants_Wf : \forall l, 
    (\forall inv, In inv l \rightarrow Wf_inv inv) \rightarrow 
    Wf_inv (mergeInvariants l).
```

Listing 4.65: The result of mergeInvariants is well-formed

Finally, using lemma `mergeInvariants_ok`, the desired theorem can be proven:

```
Theorem merge_invariants_simple1 : \forall p c c' m h fr, 
    (\forall inv, In inv (TYPESPEC.invariant (ENTITY.typeSpec c)) \rightarrow Wf_inv inv) \rightarrow  
    c' = rewriteEntity c \rightarrow 
    (EvalInvariant p c m h fr \leftrightarrow EvalInvariant p c' m h fr).
```

Listing 4.66: Main theorem of Proof 1

Compared to the originally presented theorem, an additional well-formedness premise has been added.

### 4.5.1 Proof Details

To get an impression of the proof scripts involved, we look at an example proof script in this section. The example is direction “→” of lemma `mergeInvariants_ok`.

```
Lemma merge_invariants_ok_lr : \forall p h fr l, 
    (\forall inv, In inv l \rightarrow Wf_inv inv) \rightarrow 
    ((\forall inv, In inv l \rightarrow EvalPredicate p (INVARIANT.pred inv) h fr) \rightarrow 
    EvalPredicate p (INVARIANT.pred (mergeInvariants l)) h fr).

Proof.  
intros p h fr.  
induction l.  
(* base case l=nil *)  intros.  
simpl.  
intuition.  
compute.  
trivial.
```

Listing 4.67: Proof of direction “→” of lemma `mergeInvariants_ok` (1)

The proof is by induction. The base case ‘l = nil’ is easy to prove:

‘\forall inv, In inv l \rightarrow EvalPredicate p (INVARIANT.pred inv) h fr)’ simplifies to ‘\forall inv, False \rightarrow ...’ which simplifies to True.

‘EvalPredicate p (INVARIANT.pred (mergeInvariants l)) h fr)’ simplifies to ‘EvalPredicate p (true’%jml) h fr’ since ‘mergeInvariants nil’ is equals to an invariant true’%jml.

‘EvalPredicate p (true’%jml) h fr’ is proven by tactic compute that computes the result True.
In the step case, a helper lemma ‘\( \text{Wf\_pred (INTEGRANT\_pred inv')} \)’ is proven, i.e. that the result \( \text{inv'} \) of ‘\text{mergeInvariants (a::l)}’ is well-formed. Then ‘\text{EvalPredicate inv}’ is simplified (tactic \text{simpl}) into “\( \text{EvalPredicate (a && fold\_right (InfixOp ConditionalAnd) true)} \) (map INTEGRANT\_pred l)”. This is split by applying \text{EvalPredicate\_and\_lr} (direction “\( \rightarrow \)” of \text{EvalPredicate\_and}) into “\( \text{EvalPredicate a \land EvalPredicate (fold\_right \ldots)} \)”. First, the two well-formedness premises are proven. Then, “\text{EvalPredicate a}” is proven by applying hypothesis ‘\( \text{H : In inv (a::l) \rightarrow EvalPredicate inv'} \)’. Lastly, “\text{EvalPredicate (fold\_right \ldots)}” is proven by applying the induction hypothesis \text{IHl}. Both premises of \text{IHl} are easily proven by applying hypothesis \text{Hwf} and \text{H}.

### 4.5.2 Results

All above conjectured lemmas and theorems have been proven as part of this case study.

The following table indicates the length of the various proofs:

---

\(^{17}\) In the code comment and this discussion, for simplicity, we write “\text{EvalPredicate inv}” for ‘\text{EvalPredicate p (INTEGRANT\_pred inv) h fr}’.
### 4.5.3 Discussion

The proof of theorem `merge_invariants_simple1` has successfully shown that proofs on top of the JML formalization are indeed possible. However, the proof script is surprisingly long for such a simple proof. In our opinion this has two major reasons:

First, the proof script also contains the development of the well-formedness predicate and the associated proof of `EvalExpression_pred` (i.e. a well-formed boolean expression evaluates to a boolean value). The need for this lemma stems from missing typing information in the semantics.

Second, the proof script only makes use of rather basic tactics. By defining higher-level tactics and automating rewritings and unfoldings of definitions specially suited to our formalization, the proofs may be simplified considerably.

### 4.5.4 Second Proof

A second proof has been discharged as part of the case study: theorem `merge_invariants_simple2`. The difference to `merge_invariants_simple1` being that the simplification of the semantic function `EvalInvariant` has been undone: `EvalInvariant` in this case treats all invariants defined for a type `c` (instead of invariants declared in type `c`).

```plaintext
Definition EvalInvariant (p : Program) (c : Class) (m : Method) (h : Heap.t) (fr : Frame.t) : Prop :=
  if METHOD.isHelper m then
    True
  else
    ∀ inv ,
    DefinedInvariant p c inv →
    EvalPredicate p (INVARIANT.pred inv) h fr.
```

The proof of lemma `merge_invariants_simple2` could reuse large parts of the first proof script, in particular lemma `mergeInvariants_ok`. In addition, two lemmas lay the basis of the second proof:

```plaintext
Lemma invariant_defined_2 : ∀ (c : Class) (inv : INVARIANT.t),
  DefinedInvariant c inv ↔
  In inv (TYPESPEC.invariant (ENTITY.typeSpec c)) ∨
```
That is, for every class \( c \) and invariant \( \text{inv} \), invariant \( \text{inv} \) is defined for class \( c \) if and only if the invariant is declared in \( c \) or there exists a direct super type \( \text{super} \) of \( c \) and invariant \( \text{inv} \) is defined for type \( \text{super} \). This enables a proof by cases on \( \text{invariant\_defined\_2} \) that is very much like a proof by induction.

The second helper lemma is \( \text{rewriteEntity\_maintains\_direct\_supertypes} \) that states the fact that ‘\( \text{mergeInvariants} (\text{rewriteEntity}) \)’ on a type \( c \) maintains the direct supertypes of type \( c \).

**Definition** \( \text{maintains\_direct\_supertypes} \) (\( p: \text{Program} \)) (\( f: \text{ENTITY.t} \rightarrow \text{ENTITY.t} \)) :=
\[
\forall e \text{ super}, \\
\text{direct\_subtype e super} \leftrightarrow \text{direct\_subtype (f e) super}.
\]

\[
(** \text{rewriteEntity maintains the direct super types} *)
\]
**Lemma** \( \text{rewriteEntity\_maintains\_direct\_supertypes} \) :
\[
\forall p, \\
\text{maintains\_direct\_supertypes} p (\text{fun e => rewriteEntity e}).
\]

Although the second proof case study is more involved, the second proof script is still shorter than the first proof script (~ 330 lines of codes and comments vs. ~ 430 lines). This is due to the large amount of reuse of the first proof script.

### 4.6 Java Translation Frontend

The Java Translation Frontend translates a set of JML-annotated Java source files (Java version 1.4) into a set of Coq output files. These output files make up an embedding of the Java/JML source code in the full syntax definition of Java and JML in Coq.

Besides this pure syntactic translation, the frontend desugar nullable by default and pure modifiers on type level. These desugarings are discussed in subchapter 4.6.5.

In the following, when referring to a Java source file, we always mean a JML-annotated Java source file.

The translation of a Java source file into a set of Coq output files is done as is common practice in compiler design: the Java source file is first parsed into an abstract syntax tree (AST). A traversal of the tree is then used to build the translation.

#### 4.6.1 Design

We had two major design goals in mind for this translation frontend:

First, we wanted the output generation to be *composable*. That is, the output should be built bottom-up in the same way the tree is traversed: the output for an AST node is built from the outputs of its children nodes. As an example serves the AST of ‘\( 1 + 2 \)’.

[+]

Listing 4.69: Case split on \( \text{DefinedInvariant} \)

Listing 4.70: \text{rewriteEntity} maintains direct supertypes
The output "1+2" of node ‘+’ is built from the outputs "1" and "2" of its children and its own information.

Second, we wanted modifications on the output syntax to be performed easily (maintainability), and even more, the whole output backend to be replaced easily (adaptability).

These design goals lead to a split of the visitor into three parts: the output visitor, the outputter and the pretty printer. The connection between the three parts is the output document representing the translated Coq file. A visit method for a node X first visits the child nodes of X. Then, the output document corresponding to node X is created by a delegating call to the outputter, giving the output documents, resulting from the children’s visit methods, as arguments. The output document of the root node can then be pretty-printed into a Coq output file.

That is, the split into output visitor, outputter and pretty printer is a division of labour: the visitor traverses the tree, the outputter builds output documents and the pretty printer provides facilities to create and compose those documents.

The output visitor is part of a visitor pattern [8]. The pattern is implemented in the visitor-controlled variant, that is, the traversal of the tree is controlled within the visit methods and not within the accept methods, that solely call the appropriate visit methods. The visit methods have the form ‘Object visitT(T x, Object o)’, where T is one of the node classes. The result object of the visit/accept methods of the output visitor is always the output document of the corresponding subtree. The argument object o is partially used to give additional information to visit methods of child nodes.

The pretty printer provides methods to create, compose and pretty-print documents. The pretty printer we used is an imperative implementation of the functional prettier printer by Philip Wadler [17]. The transformation of this functional pretty printer into an imperative version and its implementation in Java has been carried out as part of this work. To keep the link to the original prettier printer and to simplify notation, we use in the following Coq syntax to describe its functionality. The pretty printer operates on an abstract type Doc. The following functions create or compose documents:

- nil : Doc
- text : String → Doc
- line : Doc
- concat : Doc → Doc → Doc
- nest : int → Doc → Doc

New documents are created using function nil, function text creates a new document representing a given string s and function line a new document representing a newline character. Function ‘concat d1 d2’ (written as ‘d1 <> d2’) creates a document that is equivalent to document d2 appended to document d1 and function ‘nest n d’ creates a document that is equivalent to document d where every new line (except the first one) is indented by n characters.

Table 4.3 gives example documents built using the different functions and their pretty print.

Documents are pretty-printed to an output String by function pretty : int → Doc → String[18].

[18]For an explanation of the first parameter, see the chapter 4.6.4.
In the imperative implementation, function `void prettyWriter(PrintWriter, int, Doc)` can be used to pretty print into any `PrintWriter`.

In the following, we give an example that illustrates the interaction between output visitor, outputter and pretty printer: the translation of the binary expression ‘this.f = 10’. Notice the division of labour: methods `Visitor.visitBinaryExpr` and `Visitor.visitLiteralExpr` only deal with visiting child nodes and calling the outputter. It is only the outputter’s methods `Outputter.literalExpression` and `Outputter.BinaryExpression` that deal with the syntax of the Coq formalization. UML diagram 4.3 gives an overview about the functions of the three classes.

```java
// translation of this.f = 10
1. Doc Visitor.visitBinaryExpr(BinaryExpr x, Object o)
   // x.op == TagConstants.ASSIGN
   // x.left == [x.f]
   // x.right == [10]
   Object leftArg = x.left.accept(this, VisitorArgument.NO_ARGUMENT); // 1.1
   Object rightArg = x.right.accept(this, VisitorArgument.NO_ARGUMENT); // 1.2
   // leftArg == "field Test.F_f (Some this)"
   // rightArg = "int 10"
```

In the imperative implementation, function `void prettyWriter(PrintWriter, int, Doc)` can be used to pretty print into any `PrintWriter`. In the following, we give an example that illustrates the interaction between output visitor, outputter and pretty printer: the translation of the binary expression ‘this.f = 10’. Notice the division of labour: methods `Visitor.visitBinaryExpr` and `Visitor.visitLiteralExpr` only deal with visiting child nodes and calling the outputter. It is only the outputter’s methods `Outputter.literalExpression` and `Outputter.BinaryExpression` that deal with the syntax of the Coq formalization. UML diagram 4.3 gives an overview about the functions of the three classes.

```java
// translation of this.f = 10
1. Doc Visitor.visitBinaryExpr(BinaryExpr x, Object o)
   // x.op == TagConstants.ASSIGN
   // x.left == [x.f]
   // x.right == [10]
   Object leftArg = x.left.accept(this, VisitorArgument.NO_ARGUMENT); // 1.1
   Object rightArg = x.right.accept(this, VisitorArgument.NO_ARGUMENT); // 1.2
   // leftArg == "field Test.F_f (Some this)"
   // rightArg = "int 10"
```
return out.binaryExpression(x.op, leftArg, rightArg); // 1.3
// \result == "field Test.F_f (Some this) ==’ int 10"

1.2. Doc Visitor.visitLiteralExpr(LiteralExpr x, Object o)
// x.tag == TagConstants.INTLIT, x.value == 10
switch (x.tag) {
  case TagConstants.INTLIT: {
    return out.literalExpression(x.tag, x.value); // 1.2.1
  }
}

1.2.1 Doc Outputter.literalExpression(int tag, Object value)
// tag == TagConstants.INTLIT, value == 10
switch (tag) {
  case TagConstants.INTLIT: {
    Integer i = (Integer) value;
    return pp.concatSpace(("int"),
                          pp.intValue(i))
  }
}

1.3. Doc Outputter.binaryExpression(int tag, Doc left, Doc right)
// tag == TagConstants.ASSIGN
// left == "field Test.F_f (Some this)"
// right == "int 10"
String tagString = exprTag2String(tag); // tagString == "=="
return pp.concatSpace("field Test.F_f (Some this) ==’ int 10",
left,
right);

Listing 4.71: Example translation

4.6.2 Translation Overview

Now, after having described the design of the translator and its three main components, we look
at the different steps involved in the translation process, beginning with the set of Java source
files as input.

Input: f1.java, ..., fn.java

1. Javefe parser, ESC/Java2 parser → compilation units cu1, ..., cu n
2. → split into compilation units cu1,m1, ..., cu n,m n
3. for every compilation unit cu ij in cu 1,m1, ..., cu n,m n:
   3.1 apply the rewritings (section 4.6.5)
   3.2 apply the output visitor to traverse the AST of the compilation unit
      (result: output document d ij)

\texttt{concatSpace} d1 d2 := d1 <> text " " <> d2; \texttt{concatSpace} d1 d2 d3 is defined analogously.
4.6 Java Translation Frontend

3.3 pretty print output document $d_{ij}$ to an output file $g_{ij}$

Coq output files $g_{1,m_1}.v$, ..., $g_{n,m_n}.v$

As previously described, an abstract syntax tree has to be built first for every input file. The translation frontend builds on top of ESC/Java2 (see section 5.1.3) that is used to parse JML and Java source code and to build the AST. ESC/Java2 creates a compilation unit (AST node) $cu_i$ for every input file $f_i$. In a second step, every compilation unit $cu_i$ containing $m_i$ type declarations is split into $m_i$ compilation units containing one type declaration only. The motivation behind this splitting is explained later. Every compilation unit $cu_{ij}$ is then first desugared by applying the type level rewritings. Then, the output visitor is applied to its accept method to create the output document $d_{ij}$. Finally, output document $d_{ij}$ is pretty-printed to a Coq output file $g_{ij}.v$.

The output file name $g_{ij}$ is built as follows: If compilation unit $cu_{ij}$ declares a class $X$ within package $A.B$, the output file name $g_{ij}$ is equivalent to $A_B_X.v$.

4.6.3 Translation Details

In the following we discuss some interesting details of the translation done in step 3.2, by means of example 4.72.

```
package test;

public class A {
    private int tally = 0;

    void inc(int n) {
        tally += n;
    }

    int tally() {
        inc();
        return tally;
    }
}
```

Listing 4.72: Example class A

The compilation unit of class $A$ is translated into the following Coq output file:

```
Tr [package test; public class A {...}] =
    Require Import JMLSyntax.
    Require Import java_lang_Object.

    (* Names and signature variables *)

    (* Import the names and signatures defined in the context of class test.A *)
    Import test.A.
    Definition Def’ := Tr [public class A {...}]
```

First, two files are imported, JMLSyntax that provides the formalization and java_lang_Object, the Coq output file defining the translation of base class Object of class $A$. Next comes a bulk
of name and signature variable definitions. These variables are used as shorthands to make the translation more readable. Finally, variable $\text{Def}'$ defines the translation of the declaration of class $A$.

The use of a name variable can be seen in the translation of the declaration of field $\text{tally}$:

\[
\text{Tr} \ [\text{int } \text{tally} = 0] = \\
\text{field_decl of_class } \textbf{[private]} \text{ int}_t \ F_{\text{tally}}_ \ (\text{Some } (\text{int } 0))
\]

Field name variable $F_{\text{tally}}_\text{('Definition } F_{\text{tally}}_ \ := 1002')$ is used instead of the real field name 1002 to make the translation more readable.

Name variables are generated for all identifiers appearing in the compilation unit to translate...

- the package name
- the class name
- field and routine names
- formal parameter names
- local variable names
- label names

... and for all referenced identifiers of other types

- package names
- class names
- field and routine names

The translation of expression ‘\text{return tally}’ is:

\[
\text{Tr} \ [\text{return tally}] = \\
\text{returnE} \ (\text{field test.A.F_tally} \ (\text{Some this}))
\]

In this translation, field signature variable $\text{test.A.F_tally}$ is used. Signature variables are used as a shorthand such that the signature does not have to be repeated over and over again. (‘Definition $F_{\text{tally}} := \text{field_signature_decl test.A_ int}_t \ F_{\text{tally}}_\text{'}’) Signature variables are generated for fields and routines of the type to translate and all referenced fields and routines of other types, as well as for formal parameters and local variables.

An example, illustrating the need for formal parameter signatures is the translation of the expression ‘$\text{tally} += \text{n}$’:

\[
\text{Tr} \ [\text{tally} += \text{n}] = \\
(\text{field test.jtf_only.A.F_tally} \ (\text{Some this})) += (\text{param P_n_1004})
\]

Here, parameter $\text{n}$ is referenced via its signature variable $\text{P_n_1004}$. 
4.6 Java Translation Frontend

Name- and Signature Variable Generation

The first idea regarding signature generation had been to only define name and signature variables for entities declared in type \( T \) within the Coq output file for type \( T \). Then, signatures for fields or methods of other types (that \( T \) depends on) would be imported by importing the corresponding output files. This turned out to be impossible because of mutual dependencies: classes \texttt{Object} and \texttt{String} for example have mutual dependencies ('String \texttt{.toString()}' references 'boolean \texttt{.equals(Object)}' and vice versa). In this case, output file \texttt{java-lang-Object} would have to import \texttt{java-lang-String} and vice versa, which is not possible in Coq.

The second idea had been to define all \(^{20}\) name and signature variables within the Coq output file for type \( T \) and only import the output file of \( T \)‘s base class \(^{21}\). In this approach, name and signature variables have to be organised in name spaces to prevent conflicts between variables of different types. Still, the idea had been to use one output file for a compilation unit with possibly multiple types. Two new problems emerged: “crosswise subtyping” between two compilation units and dependencies between signatures.

“Crosswise subtyping” occurs in the following situation:

<table>
<thead>
<tr>
<th>Compilation unit 1:</th>
<th>Compilation unit 2:</th>
</tr>
</thead>
<tbody>
<tr>
<td>\texttt{class PublicSuper {}}</td>
<td>\texttt{class PublicSub extends PublicSuper {}}</td>
</tr>
<tr>
<td>\texttt{class PackageSub extends PackageSuper {}}</td>
<td>\texttt{class PackageSuper {}}</td>
</tr>
</tbody>
</table>

Listing 4.73: Illustration of “crosswise subtyping” subtyping

The output file of compilation unit 1 imports the output file of compilation unit 2 (base class \texttt{PackageSuper}) and vice versa (base class \texttt{PublicSuper}). This problem is solved by allowing only one type definition per compilation unit as achieved by the splitting described in the translation overview.

The problem of dependencies between signatures is illustrated in the following scenario:

<table>
<thead>
<tr>
<th>package a</th>
<th>package b</th>
</tr>
</thead>
<tbody>
<tr>
<td>public class A {</td>
<td>public class B {</td>
</tr>
<tr>
<td>void testB(b.B b) {</td>
<td>void testA(a.A a) {</td>
</tr>
<tr>
<td>}</td>
<td>}</td>
</tr>
</tbody>
</table>

Listing 4.74: Illustration of signature dependencies

This example would lead to the following name and signature definitions in the output file of class \texttt{A}:

\[\text{Module a.}\\\text{Module A.}\\\text{Definition PKG_a_ := ...}\\\text{Definition C_A_ := ...}\\\text{Definition A_ := (PKG_a_, C_A_).}\\\text{Definition P_b_1004_ := ...}\]

\(^{20}\)Name and signatures of entities declared in type \( T \) and of referenced entities of other types.

\(^{21}\)Since the subtype graph is free of cycles, this would not pose any problems.
Table 4.4: “Uniqueness” scope of identifiers

<table>
<thead>
<tr>
<th>Identifier kind</th>
<th>Required “uniqueness” scope</th>
<th>Ensured “uniqueness” scope</th>
</tr>
</thead>
<tbody>
<tr>
<td>Labels</td>
<td>Label statement</td>
<td>Type</td>
</tr>
<tr>
<td>Local variables</td>
<td>Routine</td>
<td>Type</td>
</tr>
<tr>
<td>Formal parameters</td>
<td>Routine</td>
<td>Type</td>
</tr>
<tr>
<td>Fields</td>
<td>Global</td>
<td>Global</td>
</tr>
<tr>
<td>Routines</td>
<td>Global</td>
<td>Global</td>
</tr>
<tr>
<td>Types</td>
<td>Global</td>
<td>Global</td>
</tr>
<tr>
<td>Packages</td>
<td>Global</td>
<td>Global</td>
</tr>
</tbody>
</table>

In Coq, a definition can only refer to identifiers that are previously defined; either previously in the same file, or in an imported file. But in the definition of $M_{\text{testB}}$, class name $b.B.B_{\_\_}$ is not yet defined. (The same problem would occur with class name $a.A.A_{\_\_}$ if module $B$ were defined before module $A$.) Notice that the problem is restricted to references on other class names, since method/field signatures can only refer to other class names (as types of parameters). Thus the solution is to define all class names before any field/method signature definition.

A technical problem remains: Coq does not allow to use a name space again, once it is closed. But this would indeed be necessary in the proposed solution, first the name space would be used for the class name definition, then for the other name and signature definitions. This problem is overcome by enclosing the first name space in a dummy name space $N'$. This name space $N'$ is imported to create the illusion that the class name definitions are within the same name spaces as the rest of the definitions.

### Identifier Generation

The translation has to ensure that it generates unique identifiers. This problem is twofold: First, entity names, such as class or field names, represented as natural numbers in the formalization have to be unique up to a certain point. Field-, routine-, type- and package names have to be globally unique since they can be referenced from anywhere. Local variable- and formal parameter names have to be unique within the enclosing routine. Label names only have to be unique within the enclosing label statement.

The translation frontend not only ensures the aforementioned “uniqueness” scope, but ensures to
4.6 Java Translation Frontend

<table>
<thead>
<tr>
<th>Identifier Kind</th>
<th>Translated name</th>
<th>Declaration/signature identifier</th>
</tr>
</thead>
<tbody>
<tr>
<td>Label x</td>
<td>L_x_id</td>
<td></td>
</tr>
<tr>
<td>Local variable x</td>
<td>V_x_id</td>
<td>V_x_id</td>
</tr>
<tr>
<td>Formal routine parameter x</td>
<td>P_x_rid</td>
<td>P_x_rid</td>
</tr>
<tr>
<td>Other formal parameter x</td>
<td>P_x_id</td>
<td></td>
</tr>
<tr>
<td>Field x</td>
<td>F_x</td>
<td></td>
</tr>
<tr>
<td>Routine x</td>
<td>M_x_id</td>
<td>M_x_id</td>
</tr>
<tr>
<td>Type x</td>
<td>C_x</td>
<td></td>
</tr>
<tr>
<td>Package x</td>
<td>PKG_x</td>
<td></td>
</tr>
</tbody>
</table>

*x refers to the name of the original Java identifier.

Table 4.5: Identifier naming scheme

generate name numbers for labels, local variables and formal parameters that are unique within
the enclosing type. The reason for this widening becomes clear, when looking at the second form
of identifiers (besides name numbers) the translation frontend has to generate, namely the name-
and signature variable identifiers.

Name and signature variable definitions are used in the output file to make the translation shorter
and more readable. Recalling the example translation ‘Tr [package test; public class A
{...}]’, all name and signature variables are defined in one place, before the translation of the
class declaration, within name spaces corresponding to their enclosing type. Despite the use of
name spaces, name clashes could occur between different identifiers, if Java identifier names were
used directly as name and signature variable identifiers:

- two different kind of identifiers with the same name (e.g. a field tally and a method tally)
- two identifiers of the same kind within different scopes in Java but within the same enclosing
type (e.g. parameters n of two methods ‘void inc(int n)’ and ‘void dec(int n)’)
- overloaded methods (e.g. ‘void inc()’ and ‘void inc(int)’)

To prevent these name clashes, the scheme outlined in table 4.5 is used for name and signature
variable identifiers.

The translated name is used for the corresponding name definitions while the entry of the third
row is used for declarations of variables, parameters and types and for the signatures of fields
and routines. The natural number id used, corresponds to the entity name number. This is the
reason why uniqueness within the enclosing type is guaranteed for entity name numbers of labels,
variables and parameters. Otherwise, this simple naming scheme could not be used.

A distinction is made between formal routine parameters and other formal parameters (catch
clause parameter and signals clause parameter). For routine parameters, the scheme uses the
name number of its enclosing routine (rid) instead of the parameters number, to make an easy
association between routines and their parameters possible. Examples of signature (and name-)
identifiers from the full translation of example 4.72: Field signature id F_tally, routine signature
id M_inc_1004 and parameter declaration id P_n_1004 of routine inc.

4.6.4 Implementation

In the implementation, class NameRegistry is in charge of generating the entity name numbers.
UML diagram 4.4 shows the relevant classes. The name registry draws a distinction between local
entities (labels, variables and parameters) and global entities (fields, routines, types and packages). Local entities have to be registered with the name registry before the corresponding identifier can be retrieved. This goes hand-in-hand with the output visitor: A label, local variable or parameter is always declared before being used. Thus, for reasons of safety, the name registry also requires that a local entity is registered (NameRegistry.isRegistered) before the corresponding identifier (entity name number) can be retrieved.

// Example: Tr[void inc(int n) { tally += n; }]

1. Object OutputVisitor.visitMethodDecl(MethodDecl, Object) {
   ...
   for (int i = 0; i < nParams; i++) {
      formalParamArgs[i] = x.args.elementAt(i).accept(
         this,
         VisitorArgument.paramArgument(true) /* method param */
      );
   }
   ...

2. Object OutputVisitor.visitFormalParaDecl(FormalParaDecl x, Object o) {
   ...
   // register parameter x with registry → registry.isDeclared(x) == true
   registry.registerParameter(x, arg.getArgument());
   // query name result for param x from name registry
   NameRegistry.VariableIdentifier name = registry.getParameterIdentifier(x);
   ...
   }

Listing 4.75: Example of visitor and name registry interaction
With a global entity \( e \) in contrast, it is not possible in every case to process the declaration of \( e \) before every use of \( e \). The mutual dependency between classes \texttt{String} and \texttt{Object} again serve as an example: Whatever class is processed first by the output visitor, the name registry has to be queried for the entity name number of the other class. Thus, the name registry does not require the entities to be registered before queries can be done.

The name registry stores name numbers in hash tables. The hash values used are generated by class \texttt{SignatureGenerator} that generates unique strings for all types of entities, based on the package name, the name of the enclosing type, the name and eventual arguments (for routines) of the entity. Table 4.6 summaries the signature generation scheme.

The name registry uses a most simple scheme to ensure the uniqueness scopes presented in table 4.4: it uses a global counter that it queries and increments for each newly generated entity name number.

### Advanced Pretty Printer features

The pretty printer provides the possibility to generate alternative layouts and to choose the optimal layout based on a maximum line width. This possibility is not used in the current implementation of the outputter but it could easily be adapted to use this feature.

To generate alternative layouts, the pretty printer provides function \texttt{group : Doc \rightarrow Doc}.

\texttt{\textbf{group d} returns a new document representing document d (first layout) plus an alternative layout of d, where every newline character is replaced by a space character (second layout).}

Pretty print function \texttt{pretty : int \rightarrow Doc \rightarrow String} then chooses the best layout based on its first argument, the maximum line width: If the first line of both layouts fit the maximum line width, the layout with the longer line is chosen, otherwise the layout with the shorter first line is chosen.

This is best seen with an example: We either want a function call like ‘\texttt{main(arg1, arg2)}’ to be pretty printed as

\texttt{main(arg1, arg2)}

or as

\texttt{main(arg1, arg2)}

depending on whether the line width is sufficient for the first layout. The pretty printer code
(functional pseudo code) to output a document with those two layouts is the following:

```coq
(* The following code pretty prints a function 'name' (length of name 'n')
 with arguments 'args': *)

Definition ppFunction (name : Doc) (n : int) (args : list Doc) : Doc :=
text "(" <>
group (nest (n+1))
  (vsep (punctuate (text ",") args)) <>
text ")"
```

'punctuate punct docs' concatenates document punct to every document in docs. 'vsep docs' vertically separates docs, i.e. it concatenates a newline character to every document in docs.

The following listing illustrate ppFunction on 'main(arg1, arg2)'.

```coq
name = text "main"
args = [text "arg1", text "arg2"]
punctuate (text ",") args =
  [text "arg1" <> text ",", text "arg2"] ["arg1", "arg2"]
vsep (punctuate (text ",") args) =
  text "arg1" <> text "," <> line <> text "arg2" ["arg1,
arg2"]
```

Pretty print of first layout: `main(arg1,
␣␣␣␣arg2)`

Pretty print of second layout: `main(arg1,␣arg2)`

Thus depending on whether the first line of layout 1 (length: 10) fits the maximum line length, the first or the second layout is chosen.

For implementation details, in particular how to efficiently implement alternative layouts, see paper 'A prettier printer' by Philip Wadler [17].

We transformed this prettier printer library to imperative style and implemented it in Java. For details on this transformation of the original functional code into imperative code and in particular, how lazy evaluation is simulated, see the source code (`pp.PrettierPrinter, pp.impl.PrettierPrinterImpl`) and the Javadoc documentation.

4.6.5 Type Level Rewritings

Before translating the abstract syntax trees of the parsed compilation units, the Java translation frontend does two simple rewritings: the desugaring of the JML modifiers `nullable_by_default` and `pure` on the type level.

Modifier `nullable_by_default`, set on a top-level type declaration `T`, makes all declarations within `T` (and nested types of `T`) implicitly `nullable`. (JML Reference Manual, section 6.2.12 [10]).

Modifier `pure`, set on any type declaration `T` (T may be a nested or local type), makes all constructors and instance methods within `T` (not within nested types of `T`) implicitly `pure`. (JML Reference Manual, section 6.1.2).

We decided to perform these rewritings in the frontend and not within the Coq formalization, mainly because they are easier to write down in the frontend. Both rewritings require traversing
the AST of the affected types and, in case of `nullable_by_default`, the whole AST of method specification cases (to set forall- and old declarations `nullable`). These AST traversals would unnecessarily complicate the rewritings of the extended syntax. Furthermore these rewritings are straightforward: they replace one syntactic modifier keyword (`nullable_by_default/pure`) by other modifier keywords (`nullable/pure`). Also, the rewritings do not strongly affect the readability of the Coq syntax outputted by the frontend. In the end, the decision is a tradeoff between defining all JML semantics within the Coq formalization and having a short and comprehensible semantics. In this case, we decided for the latter.

The exact details of the two rewritings is described in the Javadoc documentation of the classes `NullableByDefaultRewriter` and `PureRewriter`.

### 4.6.6 Implementation Remarks

**Program Counter Handling**

For compatibility reasons with Bicolano (see section 5.1.1), every statement in our Java source code formalization has the program counter of the first corresponding bytecode instruction associated. In the current implementation, dummy program counters are used: a global counter is used as program counter, that is incremented with each visit of a statement node. Nevertheless, the implementation can easily be adapted to use the real program counters of the bytecode instructions. The only place in the source code of the translation frontend to change for this adaptation is method `void OutputVisitor.prepareOutput(ASTNode, Object)`.

```java
Object visitAssertStmt(AssertStmt x, Object o) {
    Object predArg = x.pred.accept(this, VisitorArgument.NO_ARGUMENT);
    Object labelArg = x.label == null ? null : x.label.accept(this, VisitorArgument.NO_ARGUMENT);
    prepareOut(x, o);
    return out.assertStatement(predArg, labelArg);
}
```

Listing 4.76: Example visit method showing the use of `prepareOutput`

With the current implementation, `prepareOutput` sets the program counter within the outputter and increments it. The outputter in turn uses the most recent program counter value set by `prepareOutput` as current program counter in its statement output functions.

**Using ESC/Java2 as Parsing Frontend**

The following section described how ESC/Java2 (see section 5.1.3) is used as parsing frontend. ESC/Java2 is thereby used as a library only, to reduce mixing of foreign code and our own code to a minimum. The easiest way to use ESC/Java2 is to let the application's main class extend `escjava.Main`. Then, the application merely has to call method `run` and do its processing in an override of method `postprocess`. It is best to also override method `makeOptions` to returns an `Options` object that has field `stages` set to “type-checking only”, unless the application needs the other functionalities of ESC/Java2 as well. Code snippet 4.77 gives an example.
public class Main extends escjava.Main {

public javafe.Options makeOptions() {
    escjava.Options options = new escjava.Options();
    // set type-checking only
    options.stages = 1;
    // set other default options...
    return options;
}

public static void main(String[] args) {
    new Main().run(args);
}

public void preprocess() {
    nUnitsToProcess = loaded.size();
    super.preprocess();
}

public void postprocess() {
    // do your processing here...
}
}

Listing 4.77: Example use of ESC/Java as parsing frontend

Note that ESC/Java2 only fully parses the explicitly given input files. Further compilation units, automatically loaded by ESC/Java2 during parsing are not fully parsed, unless argument '-Depend' is given to ESC/Java2. Unfortunately, this argument makes ESC/Java2 very slow. To know which compilation units were given explicitly and are thus fully parsed, it is easiest to remember the number of loaded units (this.loaded) at the time of preprocess. Example 4.77 also illustrates this issue.

Disabling the Annotation Handler

ESC/Java2 does a lot of JML desugarings during the parsing/type-checking phase that are unsuitable for tools desiring to obtain an abstract syntax tree of the sources as written by the programmer. More specifically, the following desugarings are done within the escjava.AnnotationHandler:

- it adds default specs if none are present (defaultSignalsOnly, defaultModifiers)
- it performs unnesting of nested specification cases (deNest)
- it desugars lightweight, normal- and exceptional behaviour specification cases (NestedPragmaParser)

The solution to disable these desugarings we found is to override the annotation handler and the nested pragma parser (escjava.AnnotationHandler.NestedPragmaParser):

- override all exported methods of the annotation handler (that are actually called) with an empty implementation:
  - void process(TypeDeclElement)
4.6 Java Translation Frontend

- void desugar(TypeDecl)
- void desugar(RoutineDecl)
- void parseAllRoutineSpecs(CompilationUnit)

- override parseSeq of the nested pragma parser to keep the behaviour-/normal Behaviour-/exceptional behaviour modifier pragmas. (see listing 4.78)

```java
void parseSeq(...) {
    // Save 'behavior' modifier in the 'result' ModifierPragmaVec if non-null
    // and the result is non-empty

    int retval = super.parseSeq(pos, pm, behaviorMode, behavior, result, rd);
    if (behavior != null && result.size() > 0) {
        result.insertElementAt(behavior, 0);
    }
    return retval;
}
```

Listing 4.78: Override of NestedPragmaParser.parseSeq

The two overridden classes (say they are called MyAnnotationHandler and MyNestedPragmaParser) are installed as follows:

- override javafe.tc.TypeCheck Main.makeTypeCheck() to return a TypeCheck object whose
  annotationHandler field is an instance of MyAnnotationHandler.
- set the static field specparser of class escjava.AnnotationHandler (type NestedPragmaParser) to an instance of MyNestedPragmaParser before ESC/Java2 begins its processing.

4.6.7 Limitations

The following section lists Java and JML features not supported by the Java Translation Frontend.

Concerning Java, the features not supported by the frontend (and not supported by the formalization at all) are

- concurrency features
- floating-point numbers
- string- and character literals
- nested classes

Concerning JML, the features not supported by the frontend (and not supported by the formalization at all) are

- model programs
- redundant examples
- set comprehensions
Features not supported by the frontend (but part of the formalization) because not supported by ESC/Java2:

- \not_assigned expression
- \not_modified expression (supported per se, but ESC/Java2 does not support the use of \nothing or \everything in the \not_modified clause)
- \only_accessed, \only_assigned, \only_called, \only_captured expressions
- \lockset (only keyword supported by ESC/Java2), \max expressions
- informal predicates (translated to true by ESC/Java2)
- \assume statement (no support for variant with a given label)
- \debug statement

Universe Type System \[7\] annotations are part of the formalization. However, their support in ESC/Java2 is very limited. Thus, the frontend makes no guarantees for correct translation of Universe Type System modifiers.
Chapter 5

Conclusions

We conclude this report by looking at related and future work, and finally by recapitulating the results from our work.

5.1 Related Work

5.1.1 Bicolano

Bicolano [13] (BYte COde LANguage in cOq) is a formalization of Java bytecode in the Coq Proof System. It is part of the Mobius project that concerns establishing a security architecture for Java on mobile platforms. Our formalization of JML and Java source code is based on Bicolano, in particular we reused parts of their program syntax-, semantic domain- and machine arithmetic definitions.

5.1.2 Desugaring JML Method Specifications

The work of Raghavan and Leavens [16] on the desugaring of method specifications builds the basis of the rewriting of our extended syntax. In particular the splitting of the desugaring into individual steps has shown to be very useful for our rewritings as it allows to express them as individual transformation functions.

5.1.3 ESC/Java2

ESC/Java2 [2, 3, 11], the Extended Static Checker for Java version 2, is a programming tool that attempts to find common run-time errors in JML-annotated Java programs by static analysis of the program code and its formal annotations. Similar to the method specification desugarings performed as part of the extended syntax rewriting, ESC/Java2 desugars JML method specifications in their frontend prior to static analysis. However, we prefer our approach of having the desugarings be part of the formalization instead of a parsing/type-checking frontend. We think that our approach leads to more transparent desugarings that can be reordered, disabled or even be replaced.
5.1.4 jmldoc

jmldoc is a documentation generation tool similar to Javadoc [12] but extended to support the JML language. It was initially designed by Raghavan [15]. The tool also performs method specification desugaring prior to output generation. The desugarings are performed in Java (class JmlTreeSurgery) and directly operate on the parsing AST. The comments on the ESC/Java2 method specification desugarings also apply here.

5.1.5 bico

bico [5] is a tool which transforms Java class files into the Bicolano (see section 5.1.1) embedding of Java bytecode in Coq. For each class file it generates three files:

- a type file which contains the class name definition,
- a signature file which contains the signatures of all the fields and all the methods, and
- a main file which contains a translation of the Java bytecode found in the file to Bicolano formalization and the proper definition of the class, with all its dependencies.

bico can be seen as the bytecode counterpart to our Java translation frontend. bico has a similar approach to generating identifiers as natural numbers. However it is somewhat different in that it generates multiple output files per class file and that it summarises type information. bico has an easier job to do than our frontend, as Java bytecode is much less complex than the Java source code language. Even more, bico does not support a specification language like BML.

5.2 Future Work

The case study has proven some simple properties, like the equivalence of signals_only clauses and their rewriting into signals clauses. In the future, it would be interesting to drive this idea even further and define a minimal syntax subset consisting of only those features of JML that are directly treated in the semantics. Rewritings of other syntactic constructs in terms of the minimal constructs could then be proven to conform to the defined semantics of JML.

In the same spirit, the presented method specification desugarings could be extended to also perform desugaring steps 3.7 (Desugaring Inheritance and Refinement), 3.10 (Make Assignable and Signals Only Clauses the Same) and 3.11 (Desugaring Also Combinations) [16]. As these desugarings are at present part of the semantics, the desugarings could be proven to conform to the semantics as well.

Concerning the translation frontend, it would be worth evaluating, if replacing ESC/Java2 as parsing frontend is an option, as ESC/Java2 does not fully support JML[1] an improvement of this situation in the near future does not seem likely.

The translation frontend itself could be improved to output even nicer code (make use of the pretty printer feature of alternative layouts, see section 4.6.4), or to support parsing in multiple steps[2] very much like C allows to compile multiple source files of the same program in multiple compiler “runs” through the use of object files.

A bit on the larger scale a future extension to our work is to define an operational semantics for Java source code. This opens the door for some interesting applications, most importantly the

---

1In particular, it does not fully support the Universe Type System.

2This would require to make the name registry permanent.
definition of a runtime assertion checker in our formalization and to prove its soundness w.r.t. the semantics.

Another goal is to define proof obligations for JML constructs that involve a minimal amount of proving effort, yet being strong enough to enforce the semantics. Having such proof obligations defined, we can start to use the formalization as a verification framework.

5.3 Conclusions

We presented the formalization of JML and Java in the Coq Proof Assistant. We defined a full JML and Java syntax within Coq that allows to embed JML-annotated Java classes in a form that closely resemble the original syntax. Furthermore it allows to represent the different forms of specification cases.

The Java translation frontend is deliberately kept rather small. Nevertheless is is designed and implemented in three parts – visitor, outputter and pretty printer – to keep the frontend maintainable and flexible for adaptations: being changes on the current output backend or even replacing the whole output backend.

The rewritings of the extended syntax constructs in terms of basic syntax constructs consists of the rewriting of loop annotations, quantified expressions with multiple variables, implicit invariants due to non-null constraints and the desugaring of method specifications. Concerning method specifications, the desugarings are designed and implemented as transformation functions in a way that conveniently allows to reorder or disable existing, or add further desugaring steps.

We defined the semantics as a set of evaluation functions for the basic syntax subset. This set contains all JML level 0 constructs except for floating point numbers and even some constructs from JML level 1 and up. We keep the definition of the semantics as readable as possible, to ensure that it can be used to understand the semantics of JML constructs.

The case study, as a last point, showed that proving on top of the formalization is indeed feasible, once the necessary basis is laid. The basis consisted of the definition of a well-formedness predicate for some data types that allow the embedding of not-welldefined JML constructs and several lemmas about subtyping and specification inheritance. Finally, the case study highlighted the need for higher-level tactics defined specifically for the needs of proofs within this formalization.
Bibliography


[8] Erich Gamma, Richard Helm, Ralph Johnson, and John Vlissides. Design Patterns: Elements of Reusable Object-Oriented Software. Addison-Wesley, 1995.


