Enhancing separation logic for object-orientation

Diss. ETH No. 21293

Enhancing Separation Logic for Object-Orientation

A dissertation submitted to
ETH ZURICH

for the degree of
Doctor of Sciences

presented by
Stephanus Johannes van Staden
Master of Science, International University in Germany

born
February 22nd, 1983

citizen of
South Africa

accepted on the recommendation of

Prof. Dr. Bertrand Meyer, examiner
Prof. Dr. Peter O'Hearn, co-examiner
Dr. Matthew Parkinson, co-examiner
Dr. Cristiano Calcagno, co-examiner

2013

Acknowledgements

I would like to thank Bertrand Meyer, my doctoral advisor, for his patience and support. The stimulating atmosphere at the Chair of Software Engineering at ETH Zurich made the ideas possible that eventually culminated in this thesis. Many thanks to the current and former group members who gave me feedback on draft papers: Jason (Yi) Wei, Sebastian Nanz, Martin Nordio, Carlo Furia and Scott West. My thanks also goes to Andreas Leitner, Ilinca Moser, Manuel Oriol, Michela Pedroni, Marco Piccioni, Chris Poskitt, Benjamin Morandi, Christian Estler, Julian Tschannen, Marco Trudel, Mischael Schill, Nadia Polikarpova and Claudia Günthart. My research was supported by ETH Research Grant ETH-15 10-1, for which I am very grateful. The organisation and infrastructure of ETH Zurich made it a real pleasure to do the work.

I am greatly indebted to Cristiano Calcagno for his advice and help. He spent many hours reading, correcting, advising and guiding my work, and just as many in explaining things and making helpful suggestions. Apart from our research collaboration, I am also thankful for his constant encouragement and support.

My thanks also goes to Peter O'Hearn and Matthew Parkinson, who made time in their busy schedules to co-examine this work. Matthew kindly explained details about jStar to me that I needed for implementing MultiStar. My internship at Microsoft Research in Cambridge and several discussions at conferences and workshops helped to shape my work.

Finally, I would like to thank my family and friends for their encouragement and support. Apart from my parents, brothers and sister, my thanks goes to Wha Suck, Dean, Alexandru, Antoine and Riaan. Last but not least, a very special thanks goes to Teresa, my wife, the sunshine in my life.

Contents

Abstract

Reasoning about programs can be tricky and error-prone. Formal verification facilitates the formulation and demonstration of rigorous arguments about program correctness. To be usable and useful, a program logic, i.e. a proof system for program verification, should accommodate the principal mechanisms of the programming language and crystallise the informal reasoning of programmers.

Recently, separation logic emerged as a promising tool for reasoning about shared mutable state, which pervades mainstream programming languages such as C, Java and Eiffel. Parkinson and others proposed a proof system for Object-Oriented (OO) programs that combines separation logic with mechanisms to reason about inheritance and dynamic dispatch. This system can verify a wide range of OO programs and design patterns in a concise way. The flexibility and simplicity of the system made it an attractive target for further improvement and broader application.

The contributions of this thesis address the following problems:

1. Specifying, verifying and using relationships between state abstractions. This is especially important when reasoning about OO programs that use multiple inheritance.

2. Reasoning about executable contracts. Executable contracts are often weak and may perform side-effects. Yet they capture useful design information, are programmer-friendly and assist in debugging.

3. Refinement and correctness by construction. Instead of first writing the code and then proving it correct, these techniques make it possible to write a correct program in the first place. The program and its correctness proof grow and evolve together.

State abstraction mechanisms, such as abstract predicates, are useful for reasoning about programs with modules that encapsulate state and hide information. Parkinson adapted abstract predicates to the OO setting, and

they play a central role in his proof system. Since OO code frequently relies on relationships between state abstractions of a class or a class hierarchy, this thesis enhances Parkinson's system with mechanisms for specifying, verifying and exploiting such relationships. The extension also makes it possible to establish the logical consistency of a class hierarchy without considering implementation details, and it facilitates reasoning about multiple inheritance.

Existing OO code often contains contracts in the form of executable preconditions, postconditions and class invariants. These specifications are typically weaker than separation logic assertions, but they are more lightweight and perhaps more likely to be written by programmers. Contracts also record valuable information about program design and are useful for testing and debugging. This thesis contributes a new technique for using the separation logic assertions to verify that executable contracts will always hold at runtime and that they will not perform unwanted side-effects. As a result, verified contracts need not be monitored at runtime, and they add confidence in the correctness of the code and the separation logic specification.

Correctness by construction is an important feature of a mature engineering discipline. In the context of software engineering, it is realised by a calculus for top-down program development that features refinement as a central technique. A refinement calculus helps to construct correct code from a given specification in a series of steps. This thesis proposes freefinement – an algorithm for obtaining a sound refinement calculus from a modular program logic. The resulting refinement calculus can interoperate closely with the program logic, and it is even possible to reuse and translate proofs between them.

Many aspects of the work also apply to other settings. None of the contributions rely on a specific flavour of separation logic. The work on multiple inheritance subsumes interface inheritance, and the reasoning techniques for executable contracts generalise to non-OO languages that use explicit memory management. Finally, freefinement applies to a great variety of formal systems, including program logics for other languages and type systems.

Zusammenfassung

Beweisführung über Programme kann knifflig und fehleranfällig sein. Formale Verifikation unterstützt das Formulieren und Beweisen von präzisen Behauptungen über die Korrektheit von Programmen. Um brauchbar und nützlich zu sein, sollte eine Programmlogik, d.h. ein Beweissystem für Programmverifikation, die Hauptmechanismen der Programmiersprache unterstützen und die informelle Beweisführung der Programmierer präzisieren.

In letzter Zeit hat sich Separation Logic als nützliches Werkzeug erwiesen, um Beweise über einen gemeinsam genutzten, veränderlichen Zustandsraum zu führen, der in populären Programmiersprachen wie z.B. C, Java und Eiffel allgegenwärtig ist. Parkinson et al. schlugen ein Beweissystem für objektorientierte (OO) Programme vor, das Separation Logic mit Mechanismen für die Beweisführung über Vererbung und dynamischer Bindung kombiniert. Dieses System kann ein breites Spektrum von OO-Programmen und Entwurfsmustern knapp und präzise verifizieren. Die Flexibilität und Einfachheit des Systems haben es zu einem attraktiven Zielobjekt für weitere Verbesserungen und Einsatzmöglichkeiten gemacht.

Die Beiträge dieser Dissertation befassen sich mit den folgenden Problemen:

1. Spezifikation, Verifikation und Verwendung von Beziehungen zwischen Zustandsabstraktionen. Dies ist speziell dann wichtig, wenn Beweise über OO-Programme geführt werden, die Mehrfachvererbung verwenden.

2. Beweisführung über ausführbare Spezifikationen. Ausführbare Spezifikationen sind oft schwach und können mit Nebeneffekten behaftet sein. Dennoch erfassen sie nützliche Designinformationen, sind programmiererfreundlich und helfen bei der Fehlersuche.

3. Verfeinerung und konstruktionsbedingte Korrektheit. Anstatt zuerst den Programmtext zu schreiben und dann zu verifizieren, erlauben diese Techniken es, das Programm von Anfang an korrekt zu schreiben. Das

Programm und der Beweis seiner Korrektheit wachsen und entwickeln sich gemeinsam.

Mechanismen für die Zustandsabstraktion, wie z.B. abstrakte Prädikate, sind nützlich für die Beweisführung über Programme mit Modulen, die Zustand und Informationen kapseln. Parkinson adaptierte abstrakte Prädikate an das OO-Umfeld und sie spielen eine zentrale Rolle in seinem Beweissystem. Da OO-Programme sich oft auf Beziehungen zwischen Zustandsabstraktionen einer Klasse oder Klassenhierarchie stützen, erweitert diese Dissertation Parkinsons System mit Mechanismen für die Spezifikation, Verifikation und Instrumentalisierung dieser Beziehungen. Die Erweiterung ermöglicht es auch, die logischen Konsistenz einer Klassenhierarchie zu begründen ohne Implementationsdetails zu berücksichtigen, und sie unterstützt die Beweisführung über Mehrfachvererbung.

Bereits existierende OO-Programme enthalten oft Spezifikationen in der Form von ausführbaren Vor-, Nachbedingungen und Klasseninvarianten. Diese Spezifikationen sind typischerweise schwächer als Zusicherungen in Separation Logic, aber sie sind auch schlanker und werden vielleicht eher von Programmierern geschrieben. Auch enthalten sie wertvolle Informationen über den Programmaufbau und sind nützlich für Tests und die Fehlerbehebung. Ein Beitrag dieser Dissertation ist eine neue Technik für die Verwendung von Zusicherungen in Separation Logic, die sicherstellt, dass ausführbare Spezifikationen zur Laufzeit immer eingehalten werden und keine ungewollten Nebeneffekte haben. Dadurch müssen ausführbare Spezifikationen zur Laufzeit nicht mehr überprüft werden und sie erhöhen das Vertrauen in die Korrektheit des Programms und der Spezifikationen in Separation Logic.

Konstruktionsbedingte Korrektheit (correctness-by-construction) ist eine wichtige Eigenschaft ausgereifter Ingenieursdisziplinen. Im Kontext von Software Engineering wird es durch einen Kalkül für Top-Down-Programmentwicklung realisiert, der Verfeinerung als zentrale Technik aufweist. Ein Verfeinerungskalkül hilft bei der Konstruktion korrekter Programme in einer Serie von Schritten, ausgehend von einer gegebenen Spezifikation. Diese Dissertation schlägt Freefinement vor – ein Algorithmus um einen korrekten Verfeinerungskalkül aus einer modularen Programmlogik zu erhalten. Der resultierende Verfeinerungskalkül ist in der Lage, mit der Programmlogik eng zu interagieren, und es ist sogar möglich, Beweise zwischen diesen beiden wiederzuverwenden und zu übersetzen.

Viele Aspekte dieser Arbeit sind auch auf andere Umgebungen anwendbar, keiner der Beiträge hängt von spezifischen Ausprägungen von Separation Logic ab. Die Arbeit an Mehrfachvererbungen umfasst auch Schnittstellenvererbung, und die Beweisführungstechnik für ausführbare Spezifikationen kann

auf Nicht-OO-Sprachen mit explizitem Speichermanagement verallgemeinert werden. Überdies betrifft Freefinement eine grosse Vielfalt von formalen Systemen, inklusive Programmlogiken für andere Sprachen und Typsysteme.

Chapter 1

Introduction

1.1 History and motivation

Object-oriented (OO) programming languages are widely used in software engineering. They follow the imperative paradigm, i.e. programs are commands that transform state. Perhaps the most distinguishing feature of OO languages is that they promote a data-centered approach to programming: objects encapsulate data and operations associated with that data. Since humans often think about a system or a problem domain in terms of its constituent objects, the object abstraction is a natural fit for modelling them and hence reflecting this undertanding in software.

Mainstream OO languages have many features, such as classes, inheritance, dynamic binding, dynamic allocation of objects and sharing of references to objects. Although most OO programs use these features, their effects can be subtle and hard to understand. Especially in more complex programs where several features interact, it often difficult to see that the code is correct, i.e. that it will do what it is supposed to do. What is needed is a formalism that facilitates simple reasoning about the correctness of OO programs.

A system for verifying the correctness of imperative programs that manipulate simple variables has long been known: Hoare logic [32]. The basic judgement of Hoare logic is the triple $\{P\}c\{Q\}$, where P is the precondition, c the command and Q the postcondition. Informally, it means: if c is executed in an initial state satisfying P, and it terminates, then the final state will satisfy Q. An example of a valid triple is $\{x = 7\}x := x + 1\{x = 8\}$.

Hoare logic is compositional, or modular, which is important for larger verification efforts. Both programs and specifications should compose nicely. For example, the rule for reasoning about a sequential composition involves

only the specifications of its two operands:

$$\frac{\{P\}c\{Q\} \qquad \{Q\}c'\{R\}}{\{P\}c\,;c'\{R\}}$$

The rule of constancy is a good example of how specifications compose. Provided c does not modify any of the free variables in R, we have:

$$\frac{\{P\}c\{Q\}}{\{P \wedge R\}c\{Q \wedge R\}}$$

The rule states that a command preserves all properties of variables it did not modify. Continuing the earlier example, we would be able to conclude $\{x = 7 \wedge y = 6\}x := x + 1\{x = 8 \wedge y = 6\}$. A simple syntactic check suffices for determining the variables that a command might modify. Everything else must stay the same.

The situation is rather straightforward for imperative languages that manipulate simple variables: the state is just a mapping from variable names to values, and the only way to manipulate a variable is to mention it explicitly in the program text. OO languages are more complicated. In addition to simple variables (in the stack), the state also has shared objects (in the heap) that are accessed indirectly through references. Because of reference aliasing, there is no simple syntactic check for determining whether an object remains unmodified by a piece of code. OO programs regularly use many objects and much sharing, so it is important to reason about them in a convenient way.

It turns out that there are several ways to extend Hoare logic to programs that manipulate shared mutable state such as objects. One family of approaches keep the rule of constancy. Doing so requires more complicated techniques to determine 1) the set of objects or parts of objects that a command may modify, and 2) whether the predicate R describes any of them. Because 1 is not a simple syntactic check, and 2 tends to dissect R and not limit the inspection to its free variables, we shall not consider such approaches in this thesis.

Separation logic [54, 60] is another way of generalising Hoare logic to programs with shared mutable state. It adapts the rule of constancy by replacing the $\wedge$ with $*$, a new logical connective called separating conjunction. A state satisfies the predicate $P * Q$ when 1) its simple variables satisfy the constraints imposed on them by P and Q, and 2) it is possible to partition the objects into two disjoint sets of storage locations (i.e. heap locations or object-field pairs) such that the one set satisfies the constraints of P and the other those of Q. The reasoning about simple variables is exactly the same as in Hoare logic. Separation logic recognises that the concept of separation

is important for reasoning about the new kind of storage: mutable locations that may be aliased by references.

The adapted rule is known as the *frame rule*:

$$\frac{\{P\}c\{Q\}}{\{P * R\}c\{Q * R\}}$$

It is applicable whenever c does not modify any of the free variables in R. This simple syntactic check needs no information about the definitions of predicates that appear in R, which is good for abstraction. The frame rule says that the separate objects described by R will not be modified. In order for this to work, it must be impossible for c to change the storage locations that R describes. The separation logic triple $\{P\}c\{Q\}$ therefore has a generalised meaning[1]: if c is executed in an initial state satisfying P, then it will not access storage locations except those described in P or allocated by c, and if it terminates, then the final state will satisfy Q. The frame rule is very useful for knowing how a command behaves in a larger context, i.e. when more objects are present. It is heavily used in many proofs of correctness, and it tends to make them simple and short.

Separation logic simplifies reasoning about shared mutable state, but OO programs also use other features that are historically difficult to reason about. Inheritance and the associated dynamic dispatch (or dynamic binding) of routine calls enable a single piece of code to have an incredible variety of runtime behaviours. Static reasoning therefore needs to impose some discipline on inheritance and the behaviour of (overridden) routines. Moreover, the OO paradigm encourages programmers to think about objects as black boxes with hidden implementation details. Since the purpose of formalism is to make intuitive arguments rigorous, it is desirable to express and exploit such abstraction patterns in formal reasoning.

In his PhD thesis [57] and work with others [56, 58], Parkinson proposed an elegant proof system that addresses these issues. The main ingredients of the system are as follows (Chapter 2 contains the details):

- It uses separation logic to reason about the fields of objects.
- Abstract predicate families (apfs) provide encapsulated data abstractions in the logic. An apf is a predicate with a designated 'target' argument. Each class C can define the apf predicate for the case when the target argument has dynamic type C. It is only possible to use this

[1] That c does not access (i.e. read or modify) these locations is a stronger requirement than what we actually need here, but it is widely used and useful for reasoning about concurrency.

definition within class C when the dynamic type of the target argument is C – the predicate must be treated abstractly in all other cases. Apfs can thus express the idea that objects (or parts of objects or object aggregates) are black boxes for outsiders, and that subclasses may change the representation. A specification of a routine typically says how the code transforms one or more of these black boxes into others.

- Each routine has a static and a dynamic specification, each consisting of a precondition and a postcondition. The static specification states in detail what the routine body does. It is used to reason about calls that use static dispatch (e.g. *precursor* or *super* calls). The dynamic specification is typically more abstract – it describes the idea behind the routine that must be respected in all subclasses. It imposes discipline on inheritance, and is used to reason about dynamically dispatched calls.
  The organisation of specifications is modular: a routine body is verified only against the static specification, which must in turn be consistent with the dynamic specification. Static specifications make it unnecessary to re-verify code in subclasses, so each routine body is verified exactly once.

These ingredients combine to yield a compact system that can reason about many uses and abuses of inheritance, including behaviour extension, restriction and modification, and change of representation in subclasses.

Because of all its desirable features, this system is an ideal starting point for further enhancements. This thesis considers the improvement of the system and its broader application in reasoning. It tackles the following problems:

1. The basic apf mechanism is not always flexible enough. A class might want to share some properties of its definitions with others. Also, some properties of definitions, or relationships between them, may be invariant in a whole class hierarchy, and client code might rely on this. For example, it could be a design decision that all subclasses must use the same definition and hence the same implementation of a logical abstraction. Or it might be intended that an implication between two different apf predicates will always hold. We need ways to express, verify and use such information.

2. OO languages like Eiffel use multiple inheritance and deferred (abstract) classes and routines. Reasoning about these constructs requires extensions to the original system.

3. Much existing OO code has been specified with executable contracts, such as preconditions, postconditions and class invariants. They capture expectations and intentions in a programmer-friendly way. There has been several attempts to use them as a basis for verification, but it tends to be complicated because they 1) are often too weak, 2) do not have a separating conjunction, and hence struggle with framing and aliasing, and 3) often contain side-effects. The separation logic system ignores these assertions, but it would be better to reason about them and exploit them in verification.

4. Instead of hacking away at code and verifying its correctness afterwards, it is often easier to develop code that is correct by construction. However, the separation logic system is geared for bottom-up verification instead of top-down development. It does not directly support techniques for stepwise transformation such as refinement. A major goal of this thesis is therefore to support refinement and correctness by construction in a way that complements the separation logic system.

1.2 Overview of contributions

Chapter 3 addresses the first problem by adding features to Parkinson's system. It shows with examples how a single class may employ multiple apfs to achieve its purpose, and that these are often related and combined in disciplined ways. Since the correctness of OO code frequently depends on the relationships between these abstractions, we need mechanisms to express, verify and use such information in a rigorous fashion. In particular, the work introduces two general specification mechanisms: *export clauses* for relating abstractions in individual classes, and *axiom clauses* for relating abstractions in a class and all its descendants, i.e. for an entire (sub-)hierarchy of classes. The information expressed in export and axiom clauses is first verified before it is assumed for the verification of code. Proof rules that avoid logical inconsistencies formalise the manipulation.

Chapter 3 also extends Parkinson's system to support multiple inheritance and deferred (abstract) classes and routines. This subsumes reasoning about interface inheritance, a form of multiple inheritance used in languages like Java. Multiple inheritance usually intertwines several logical abstractions from parent classes, and it turns out that export and axiom clauses are useful for specifying such information. Knowing more about ancestor classes through specifications means that fewer of their implementation details have to be considered, and verification is consequently more modular.

Chapter 4 attacks the third problem: reasoning about executable contracts and exploiting them in verification. The work shows how separation logic can be used to verify that executable contracts are satisfied, i.e. that they will always hold at runtime and not perform undesirable side-effects. Both the program and its executable assertions are verified with respect to separation logic specifications. A novel notion called *relative purity* embraces historically problematic side-effects in executable specifications, and verification boils down to proving *connecting implications*. Even model-based specifications, i.e. expressive executable contracts that use mathematical abstractions, can be verified. A failed verification attempt may indicate a discrepancy between the two kinds of specification. For example, the separation logic precondition of a routine may be too liberal for the particular application, and the executable precondition can help to uncover this problem. Or a failed verification may point to situations in which a runtime violation of an executable postcondition would occur. In any case, the separation logic and executable specifications complement each other in verification. Several examples illustrate the use and utility of the approach.

The above mechanisms are well-suited to automation. We implemented them in MultiStar, a fully automatic verification tool for verifying Eiffel programs. MultiStar uses separation logic specifications that are embedded in the source program. The verification proceeds in two steps. Firstly, the specifications and code are translated into an intermediate representation that is simpler to process. Secondly, a reasoning engine based on jStar [23] performs the actual verification. Section 3.2 describes the architecture of MultiStar and its treatment of export/axiom clauses and multiple inheritance. Section 4.7 explains how it supports reasoning about executable contracts. MultiStar successfully verified the examples in Chapters 3 and 4. Moreover, Section 3.3 describes a case study in which MultiStar was used to verify the core iterator hierarchy of a popular data structure library. MultiStar is freely available as part of the EVE (Eiffel Verification Environment) download [28].

Chapter 5 tackles refinement and correctness by construction. The approach is to obtain a refinement calculus for OO commands, and then to use it as part of a larger calculus that handles other OO features. This reflects the fact that developing a routine body involves different reasoning than, say, adding an attribute to a class or a dynamic specification to a routine.

It is moreover a cost-effective way to structure a solution, because the refinement calculus comes for free! Section 5.1 describes *freefinement*, an algorithm that constructs a sound refinement calculus from a verification system under certain conditions. In this work, a verification system is any formal system for establishing whether an inductively defined term, such as a command, satisfies a specification. Examples of verification systems include

Hoare logics and type systems. Freefinement first extends the term language to include specification terms, and builds a verification system for the extended language that is a sound and conservative extension of the original system. The extended system is then transformed into a sound refinement calculus. The resulting refinement calculus can interoperate closely with the verification system – it is even possible to reuse and translate proofs between them. Freefinement gives a semantics to refinement at an abstract level: it associates each term of the extended language with a set of terms from the original language, and refinement simply reduces this set. For illustration purposes, Section 5.2 applies freefinement to a simple type system for the lambda calculus and also to a Hoare logic.

It is straightforward to apply freefinement to the part of the separation logic proof system that reasons about OO commands. The resulting refinement calculus is plugged into a larger system that deals with other OO constructs, and this larger calculus is compatible with (and complements) the full separation logic system. In particular, all its transformation steps preserve correctness with respect to the system, and every verifiable OO program can be constructed from scratch in a stepwise manner.

Many of the results in this thesis are quite general and also apply to different settings. None of the contributions rely on a particular flavour of separation logic. The work on executable contracts can be used for non-OO languages with explicit memory management such as C. Finally, freefinement can give refinement calculi for a great variety of formal systems, including modular type systems such as System F.

Chapter 2

Background material

This chapter describes the OO programming language and its separation logic proof system that are used throughout the thesis. It summarises the published work of Parkinson [57] and his joint work with Bierman [58] with minor changes in the presentation.

2.1 Language syntax

The grammar of our kernel OO language with specifications is shown in Figure 2.1. It is deliberately minimal to keep the presentation simple, but it contains the essential language constructs of popular OO languages. The concrete notation is based on Eiffel [26].

A class is divided into top-level sections. The **inherit** section lists its parent class (or 'ANY'), the **define** section contains predicate definitions, and methods and fields are written in the **feature** section. Empty sections and '**inherit** ANY' will simply be omitted in examples.

A sequence of c's is denoted by $\overline{\mathrm{c}}$. The letters G and H are used for class names, m for method names, and f for field names. Variables are denoted by u, x, y and z. **Void** corresponds to 'null' in other languages. Two reserved program variables **Current** and **Result** denote the current object ('this') and the result of a function call respectively. **Current** is never **Void**. The term *feature* describes methods and fields, sometimes called the 'members' of a class. Feature overloading, including method overloading, is not allowed.

Separate namespaces exist for class names, p, m and f. We assume the absence of clashes when names are introduced. This precludes method overloading and field shadowing. Later, when we extend the language with multiple inheritance, it will guarantee that methods or fields with the same name in parent classes stem from common ancestors.

L ::= **class** G **inherit** H **define** $\overline{\text{D}}$ **feature** $\overline{\text{M}}$ $\overline{\text{F}}$ **end**	
D ::= x.$\text{p}_G(\overline{\text{t:y}})$ **as** P	*Define clause*
M ::= **introduce** m(Args) Rt Sd Ss B	*Method declaration*
| **override** m(Args) Rt Sd Ss B	
| **inherit** m(Args) Rt Sd Ss	
F ::= f: Type	*Field declaration*
Sd ::= **dynamic** Spec	*Dynamic specification*
Ss ::= **static** Spec	*Static specification*
Spec ::= {P}_{Q} | {P}_{Q} **also** Spec	*Specification*
B ::= **do** s **end**	*Method body*
s ::= **skip**	*Statement*
| x := e	*Assignment*
| s ; s	*Sequential compostition*
| **if** e **then** s **else** s **end**	*Conditional*
| **while** e **do** s **end**	*Loop*
| x: Type. s	*Local variable block*
| x := **new** G	*Object allocation*
| x := y.f	*Field lookup*
| x.f := e	*Field update*
| x := y.m($\overline{\text{e}}$) | y.m($\overline{\text{e}}$)	*Dynamically dispatched call*
| x := y.G::m($\overline{\text{e}}$) | y.G::m($\overline{\text{e}}$)	*Direct method call*
e ::= x | e + e | e = e | **Void** | 0 | 1 | 2 ...	*Expression*
Type ::= $\overline{\text{INT}}$ | $\overline{\text{BOOL}}$ | G	
Args ::= $\overline{\text{x: Type}}$	*Formal arguments*
Rt ::= ϵ | : Type	*Return type*

Figure 2.1: The kernel language grammar.

A constructor is simply an introduced method m where m is a class name. Except for the restriction that subclasses cannot inherit or override constructors, no special treatment is needed otherwise. In the examples, we will omit the target of a method call or field assignment if it is **Current**.

Methods have both **static** and **dynamic** specifications. A specification is written in pre-post form $\{P\}\_\{Q\}$, or $\{P_1\}\_\{Q_1\}$ **also** $\{P_2\}\_\{Q_2\}$ to indicate that both are satisfied (Section 2.3 shows how the **also** form abbreviates a single pre-post specification). A method's dynamic specification must be satisfied by all subclasses, and is used to verify dynamically-dispatched calls. A static specification describes properties about the particular method body, and is used to verify statically-dispatched calls, including **Precursor** ('super' or 'base') calls and direct calls $x.C::m(\overline{e})$ in C++ style.

To provide subclasses with the opportunity to respecify a method and to simplify the proof rules that follow later, we require a subclass to inherit or override explicitly all non-constructor methods present in its parent (in the examples of later chapters and the MultiStar tool, specification shorthands are employed to achieve this).

We assume the formal argument names of methods stay the same in subclasses. This simplifies the proof rules that follow, which would otherwise need additional substitutions.

Finally, to improve the readability of examples in later chapters, we frequently write statements in a list and omit the interspersed sequential composition operators.

2.2 Logic syntax and semantics

The predicates used in specifications and proofs have the following grammar.

$$
\begin{array}{lll}
P, Q, S, T, \Delta ::= \forall x{\cdot}P \mid P \Rightarrow Q \mid \mathrm{false} \mid e = e' \mid x : G \mid x <: G \mid x.f \hookrightarrow e \mid P * Q & & \\
\quad \mid x.p(\overline{t{:}\,e}) & & \textit{Apf predicate} \\
\quad \mid x.p_G(\overline{t{:}\,e}) & & \textit{Apf entry}
\end{array}
$$

The predicate $x : G$ means x references an object whose dynamic type is exactly G, and $x <: G$ means x references an object whose dynamic type is a subtype of G. In both cases $x \neq$ **Void**, and $x : G \Rightarrow x <: G$ holds. Within a context, if x is declared of type G then $x <: G$ whenever $x \neq$ **Void**.

Abstract predicate families [56, 58] provide logical abstractions of state for the OO setting. An abstract predicate family (abbreviated *apf*) provides an abstract predicate p for which each class C can define an entry p_C. The first argument of an apf predicate or entry is called the *root*. The root followed by a dot is written before the apf predicate or entry name. The whole prefix

is omitted when the root is **Current**. An apf predicate's root can never be **Void**. Since the meaning of an apf predicate depends on the dynamic type of the root object, it can be seen as mirroring dynamic dispatch of object-orientation in the logic. The other arguments of an apf predicate or entry are a set (i.e. order-independent collection) of tagged arguments, where tag names t provide useful hints about the purpose of tagged values. Apfs offer high levels of abstraction in specifications and proofs: apf predicates and entries are treated abstractly by the clients of a class.

Other predicates have the usual intuitionistic separation logic semantics. Informally, the predicate $\mathrm{x.f} \hookrightarrow \mathrm{e}$ means that the f field of object x has value e ($\mathrm{f} \hookrightarrow \mathrm{e}$ abbreviates $\mathbf{Current}.\mathrm{f} \hookrightarrow \mathrm{e}$), and $\mathrm{P} * \mathrm{Q}$ means that P and Q hold for disjoint portions of the heap. Readers are referred to [60, 54, 57] for a formal treatment of separation logic. Symbols such as $\Leftrightarrow$, $\neg$, true, $\vee$, $\wedge$ and $\exists$ are encoded in the standard way. Every occurrence of _ in a predicate denotes a fresh existentially quantified variable, where the quantifier is placed in the innermost position. $FV(\mathrm{P})$ denotes the free variables of P; every method precondition P must satisfy $\mathbf{Result} \notin FV(\mathrm{P})$.

In the rest of the formalisation, the symbols P, Q, S and T are used for assertions and predicates, and Δ for assumptions.

2.3 Specification refinement

A specification $\{\mathrm{P}_1\}\_\{\mathrm{Q}_1\}$ is refined by $\{\mathrm{P}_2\}\_\{\mathrm{Q}_2\}$ when any statement s that satisfies $\{\mathrm{P}_1\}\_\{\mathrm{Q}_1\}$ also satisfies $\{\mathrm{P}_2\}\_\{\mathrm{Q}_2\}$. Under the assumptions Δ, the specification $\{\mathrm{P}_1\}\_\{\mathrm{Q}_1\}$ is refined by $\{\mathrm{P}_2\}\_\{\mathrm{Q}_2\}$ if we can prove $\Delta \vdash \{\mathrm{P}_1\}\_\{\mathrm{Q}_1\} \Longrightarrow \{\mathrm{P}_2\}\_\{\mathrm{Q}_2\}$, i.e. provide a proof tree with leaves $\Delta \vdash \{\mathrm{P}_1\}\_\{\mathrm{Q}_1\}$ and root $\Delta \vdash \{\mathrm{P}_2\}\_\{\mathrm{Q}_2\}$ built with the structural rules of separation logic (Consequence, Frame, Auxiliary Variable Elimination, Disjunction, and others). In the context of method specification refinement, Δ contains the apf assumptions of the class, and the Consequence and Frame rules are given by:

$$\frac{\Delta \Rightarrow (\mathrm{P}' \Rightarrow \mathrm{P}) \qquad \Delta \vdash \{\mathrm{P}\}\_\{\mathrm{Q}\} \qquad \Delta \Rightarrow (\mathrm{Q} \Rightarrow \mathrm{Q}')}{\Delta \vdash \{\mathrm{P}'\}\_\{\mathrm{Q}'\}} \text{ Consequence}$$

$$\frac{\Delta \vdash \{\mathrm{P}\}\_\{\mathrm{Q}\}}{\Delta \vdash \{\mathrm{P} * \mathrm{T}\}\_\{\mathrm{Q} * \mathrm{T}\}} \text{ Frame}$$

The Frame rule is applicable whenever $\mathbf{Result} \notin FV(\mathrm{T})$, and expresses that disjoint portions of the heap stay unchanged.

Method specifications can be combined with **also** (Definition 1 in [58]):

$$\{P_1\}\_\{Q_1\} \textbf{ also } \{P_2\}\_\{Q_2\} \stackrel{\text{def}}{=} \{(P_1 \wedge x = 1) \vee (P_2 \wedge x \neq 1)\}\_\{(Q_1 \wedge x = 1) \vee (Q_2 \wedge x \neq 1)\}$$

where x denotes a fresh auxiliary variable. The specifications $\{P_1\}\_\{Q_1\}$ and $\{P_2\}\_\{Q_2\}$ are *equivalent w.r.t.* Δ iff both $\Delta \vdash \{P_1\}\_\{Q_1\} \Longrightarrow \{P_2\}\_\{Q_2\}$ and $\Delta \vdash \{P_2\}\_\{Q_2\} \Longrightarrow \{P_1\}\_\{Q_1\}$. Two specifications are *equivalent* iff they are equivalent w.r.t. all Δ. It can be shown that **also** is commutative, associative and idempotent modulo equivalence with identity $\{\text{false}\}\_\{\text{true}\}$. The notation $\textbf{also}_{i \in I}\ \{P_i\}\_\{Q_i\}$ denotes the specification $\{P_{e_1}\}\_\{Q_{e_1}\}$ **also** $\ldots$ **also** $\{P_{e_m}\}\_\{Q_{e_m}\}$, where $e_1 \ldots e_m$ are the elements of the finite index set I. Furthermore, when I is the empty set:

$$\textbf{also}_{i \in \emptyset}\ \{P_i\}\_\{Q_i\} \stackrel{\text{def}}{=} \{\text{false}\}\_\{\text{true}\}$$

It always holds that $\Delta \vdash \{P\}\_\{Q\} \Longrightarrow \{\text{false}\}\_\{\text{true}\}$. Other useful lemmas involving **also** are given in Section 2.8. Finally, we use the abbreviation

$$\Delta \vdash \{P_1\}\_\{Q_1\} \stackrel{\textbf{Current}:G}{\Longrightarrow} \{P_2\}\_\{Q_2\} \stackrel{\text{def}}{=} \Delta \vdash \{P_1\}\_\{Q_1\} \Longrightarrow \{P_2 * \textbf{Current} : G\}\_\{Q_2\}$$

in the formalisation of the Dynamic dispatch proof obligation for methods in Section 2.6.

2.4 The specification environment

Most of the proof rules that follow use an environment Γ, which maps method names to their specifications for all classes in a program:

$$\begin{array}{ll} \Gamma ::= G.m \mapsto (\overline{x},\{P\}\_\{Q\}) & \textit{Method dynamic specification} \\ \quad \mid G::m \mapsto (\overline{x},\{S\}\_\{T\}) & \textit{Method static specification} \\ \quad \mid \overline{\Gamma} & \end{array}$$

The $\overline{x}$ in a specification of m denote its formal argument names. Γ is guaranteed to be a partial function for well-formed programs, and we write $\Gamma(G.m) = (\overline{x},\{P\}\_\{Q\})$ for $G.m \mapsto (\overline{x},\{P\}\_\{Q\}) \in \Gamma$, and $\Gamma(G::m) = (\overline{x},\{S\}\_\{T\})$ for $G::m \mapsto (\overline{x},\{S\}\_\{T\}) \in \Gamma$.

2.5 Statement verification

The assumptions Δ used to verify statements contain apf information about the enclosing class. The rules for most statements are standard:

$$\frac{}{\Delta;\Gamma \vdash_s \{\mathrm{P}\}\mathbf{skip}\{\mathrm{P}\}}$$

$$\frac{}{\Delta;\Gamma \vdash_s \{\mathrm{P}[\mathrm{e}/\mathrm{x}]\}\mathrm{x} := \mathrm{e}\{\mathrm{P}\}}$$

$$\frac{\Delta;\Gamma \vdash_s \{\mathrm{P}\}\mathrm{s}\{\mathrm{Q}\} \qquad \Delta;\Gamma \vdash_s \{\mathrm{Q}\}\mathrm{s}'\{\mathrm{R}\}}{\Delta;\Gamma \vdash_s \{\mathrm{P}\}\mathrm{s};\mathrm{s}'\{\mathrm{R}\}}$$

$$\frac{\Delta;\Gamma \vdash_s \{\mathrm{P} * \mathrm{e}\}\mathrm{s}\{\mathrm{Q}\} \qquad \Delta;\Gamma \vdash_s \{\mathrm{P} * \neg\mathrm{e}\}\mathrm{s}'\{\mathrm{Q}\}}{\Delta;\Gamma \vdash_s \{\mathrm{P}\}\mathbf{if}\ \mathrm{e}\ \mathbf{then}\ \mathrm{s}\ \mathbf{else}\ \mathrm{s}'\ \mathbf{end}\{\mathrm{Q}\}}$$

$$\frac{\Delta;\Gamma \vdash_s \{\mathrm{P} * \mathrm{e}\}\mathrm{s}\{\mathrm{P}\}}{\Delta;\Gamma \vdash_s \{\mathrm{P}\}\mathbf{while}\ \mathrm{e}\ \mathbf{do}\ \mathrm{s}\ \mathbf{end}\{\mathrm{P} * \neg\mathrm{e}\}}$$

The rule for local variables has the proviso that $\mathrm{x} \notin FV(\mathrm{P}) \cup FV(\mathrm{Q})$:

$$\frac{\Delta;\Gamma \vdash_s \{\mathrm{P}\}\mathrm{s}\{\mathrm{Q}\}}{\Delta;\Gamma \vdash_s \{\mathrm{P}\}\mathrm{x}\colon \mathrm{Type}.\ \mathrm{s}\{\mathrm{Q}\}}$$

In the rule for object allocation, *allfields*(G) denotes the set of field names listed in G and all its ancestors:

$$\frac{\mathit{allfields}(\mathrm{G}) = \{\mathrm{f}_1,\mathrm{f}_2,\ldots,\mathrm{f}_\mathrm{n}\}}{\begin{array}{l}\Delta;\Gamma \vdash_s \{\mathrm{true}\} \\ \qquad \mathrm{x} := \mathbf{new}\ \mathrm{G} \\ \qquad \{\mathrm{x.f}_1 \hookrightarrow \_ * \mathrm{x.f}_2 \hookrightarrow \_ * \ldots * \mathrm{x.f}_\mathrm{n} \hookrightarrow \_ * \mathrm{x} : \mathrm{G}\}\end{array}}$$

For field lookup, when x is not free in e and not the same as y, we have:

$$\frac{}{\Delta;\Gamma \vdash_s \{\mathrm{y.f} \hookrightarrow \mathrm{e}\}\mathrm{x} := \mathrm{y.f}\{\mathrm{y.f} \hookrightarrow \mathrm{e} * \mathrm{x} = \mathrm{e}\}}$$

Field update is simple:

$$\frac{}{\Delta;\Gamma \vdash_s \{\mathrm{x.f} \hookrightarrow \_\}\mathrm{x.f} := \mathrm{e}\{\mathrm{x.f} \hookrightarrow \mathrm{e}\}}$$

Dynamically dispatched calls use the dynamic specs of methods in Γ, while direct calls use the static ones. Provided x is not y and x is not free in $\overline{\mathrm{e}}$, the rules for result-returning calls are:

$$\frac{\Gamma(\mathrm{G.m}) = (\overline{\mathrm{u}},\{\mathrm{P}\}\_\{\mathrm{Q}\})}{\begin{array}{l}\Delta;\Gamma \vdash_s \{\mathrm{P}[\mathrm{y},\overline{\mathrm{e}}/\mathbf{Current},\overline{\mathrm{u}}] * \mathrm{y} <: \mathrm{G}\} \\ \qquad \mathrm{x} := \mathrm{y.m}(\overline{\mathrm{e}}) \\ \qquad \{\mathrm{Q}[\mathrm{y},\overline{\mathrm{e}},\mathrm{x}/\mathbf{Current},\overline{\mathrm{u}},\mathbf{Result}]\}\end{array}}$$

$$\frac{\Gamma(\mathrm{G::m}) = (\overline{\mathrm{u}},\{\mathrm{S}\}\_\{\mathrm{T}\})}{\begin{array}{c}\Delta;\Gamma \vdash_s \{\mathrm{S}[\mathrm{y},\overline{\mathrm{e}}/\mathbf{Current},\overline{\mathrm{u}}] * \mathrm{y} \neq \mathbf{Void}\} \\ \mathrm{x} := \mathrm{y.G::m}(\overline{\mathrm{e}}) \\ \{\mathrm{T}[\mathrm{y},\overline{\mathrm{e}},\mathrm{x}/\mathbf{Current},\overline{\mathrm{u}},\mathbf{Result}]\}\end{array}}$$

The important structural rules for this thesis are Frame, Consequence, Auxiliary Variable Elimination, and Disjunction. The Frame rule is the key to local reasoning. Provided s modifies no variable in $FV(\mathrm{T})$:

$$\frac{\Delta;\Gamma \vdash_s \{\mathrm{P}\}\mathrm{s}\{\mathrm{Q}\}}{\Delta;\Gamma \vdash_s \{\mathrm{P} * \mathrm{T}\}\mathrm{s}\{\mathrm{Q} * \mathrm{T}\}}\ \text{Frame}$$

The rule of Consequence allows the use of assumptions Δ:

$$\frac{\Delta \Rightarrow (\mathrm{P}' \Rightarrow \mathrm{P}) \qquad \Delta;\Gamma \vdash_s \{\mathrm{P}\}\mathrm{s}\{\mathrm{Q}\} \qquad \Delta \Rightarrow (\mathrm{Q} \Rightarrow \mathrm{Q}')}{\Delta;\Gamma \vdash_s \{\mathrm{P}'\}\mathrm{s}\{\mathrm{Q}'\}}\ \text{Consequence}$$

In Auxiliary Variable Elimination, the variable x to be eliminated may not appear free in s.

$$\frac{\Delta;\Gamma \vdash_s \{\mathrm{P}\}\mathrm{s}\{\mathrm{Q}\}}{\Delta;\Gamma \vdash_s \{\exists \mathrm{x}.\mathrm{P}\}\mathrm{s}\{\exists \mathrm{x}.\mathrm{Q}\}}\ \text{AuxVarElim}$$

Finally, the Disjunction rule is standard:

$$\frac{\Delta;\Gamma \vdash_s \{\mathrm{P}\}\mathrm{s}\{\mathrm{Q}\} \qquad \Delta;\Gamma \vdash_s \{\mathrm{P}'\}\mathrm{s}\{\mathrm{Q}'\}}{\Delta;\Gamma \vdash_s \{\mathrm{P} \vee \mathrm{P}'\}\mathrm{s}\{\mathrm{Q} \vee \mathrm{Q}'\}}\ \text{Disjunction}$$

2.6 Method verification

As for statement verification, the assumptions Δ used to verify method definitions contain apf information about the method's enclosing class.

A newly introduced method's static and dynamic specifications must be consistent[1], and its body must satisfy the static specification. These two requirements are captured by the Dynamic dispatch [D.d.] and Body verification [B.v.] proof obligations respectively.

[1]We establish that the dynamic specification follows from the static one when the dynamic type of **Current** is G. If the dynamic type of x is G, then the body of m in G will be executed if x.m is called. The static and dynamic specifications must be consistent with each other in this case, since the dynamic specification is used for reasoning about the call statement, whereas the body was verified only w.r.t. the static specification.

$$\frac{\begin{array}{l} \mathrm{B} = \textbf{do}\ \mathrm{s}\ \textbf{end} \\ \mathrm{Sd} = \textbf{dynamic}\ \{\mathrm{P_G}\}\text{-}\{\mathrm{Q_G}\} \\ \mathrm{Ss} = \textbf{static}\ \{\mathrm{S_G}\}\text{-}\{\mathrm{T_G}\} \\ \Delta \vdash \{\mathrm{S_G}\}\text{-}\{\mathrm{T_G}\} \overset{\textbf{Current}:\mathrm{G}}{\Longrightarrow} \{\mathrm{P_G}\}\text{-}\{\mathrm{Q_G}\} \quad [\text{D.d.}] \\ \Delta;\Gamma \vdash_s \{\mathrm{S_G}\}\mathrm{s}\{\mathrm{T_G}\} \quad [\text{B.v.}] \end{array}}{\Delta;\Gamma \vdash_m \textbf{introduce}\ \mathrm{m(Args)\ Rt\ Sd\ Ss\ B\ in\ G\ parent\ H}}$$

The next rule is used whenever a method body gets redefined. Consistency must be proven between the new dynamic specification and those in the parent class; this is embodied in the Behavioural subtyping [B.s.] proof obligation. The other proof obligations are identical to those for method introduction above.

$$\frac{\begin{array}{l} \Gamma(\mathrm{H.m}) = (\overline{\mathrm{x}},\{\mathrm{P_H}\}\text{-}\{\mathrm{Q_H}\}) \\ \mathrm{B} = \textbf{do}\ \mathrm{s}\ \textbf{end} \\ \mathrm{Sd} = \textbf{dynamic}\ \{\mathrm{P_G}\}\text{-}\{\mathrm{Q_G}\} \\ \mathrm{Ss} = \textbf{static}\ \{\mathrm{S_G}\}\text{-}\{\mathrm{T_G}\} \\ \Delta \vdash \{\mathrm{P_G}\}\text{-}\{\mathrm{Q_G}\} \Longrightarrow \{\mathrm{P_H}\}\text{-}\{\mathrm{Q_H}\} \quad [\text{B.s.}] \\ \Delta \vdash \{\mathrm{S_G}\}\text{-}\{\mathrm{T_G}\} \overset{\textbf{Current}:\mathrm{G}}{\Longrightarrow} \{\mathrm{P_G}\}\text{-}\{\mathrm{Q_G}\} \quad [\text{D.d.}] \\ \Delta;\Gamma \vdash_s \{\mathrm{S_G}\}\mathrm{s}\{\mathrm{T_G}\} \quad [\text{B.v.}] \end{array}}{\Delta;\Gamma \vdash_m \textbf{override}\ \mathrm{m(Args)\ Rt\ Sd\ Ss\ B\ in\ G\ parent\ H}}$$

When a method is inherited, its static specification must follow from the one in the parent class. The Inheritance [Inh.] obligation ensures that this will be the case. The Behavioural subtyping and Dynamic dispatch obligations serve the same purposes as before.

$$\frac{\begin{array}{l} \Gamma(\mathrm{H.m}) = (\overline{\mathrm{x}},\{\mathrm{P_H}\}\text{-}\{\mathrm{Q_H}\}) \\ \Gamma(\mathrm{H::m}) = (\overline{\mathrm{x}},\{\mathrm{S_H}\}\text{-}\{\mathrm{T_H}\}) \\ \mathrm{Sd} = \textbf{dynamic}\ \{\mathrm{P_G}\}\text{-}\{\mathrm{Q_G}\} \\ \mathrm{Ss} = \textbf{static}\ \{\mathrm{S_G}\}\text{-}\{\mathrm{T_G}\} \\ \Delta \vdash \{\mathrm{P_G}\}\text{-}\{\mathrm{Q_G}\} \Longrightarrow \{\mathrm{P_H}\}\text{-}\{\mathrm{Q_H}\} \quad [\text{B.s.}] \\ \Delta \vdash \{\mathrm{S_H}\}\text{-}\{\mathrm{T_H}\} \Longrightarrow \{\mathrm{S_G}\}\text{-}\{\mathrm{T_G}\} \quad [\text{Inh.}] \\ \Delta \vdash \{\mathrm{S_G}\}\text{-}\{\mathrm{T_G}\} \overset{\textbf{Current}:\mathrm{G}}{\Longrightarrow} \{\mathrm{P_G}\}\text{-}\{\mathrm{Q_G}\} \quad [\text{D.d.}] \end{array}}{\Delta;\Gamma \vdash_m \textbf{inherit}\ \mathrm{m(Args)\ Rt\ Sd\ Ss\ in\ G\ parent\ H}}$$

2.7 Class and program verification

A class is verified by verifying all of its methods. Once again, the formula Δ contains apf information that is specific to the class.

$$\frac{\forall \mathrm{M_i} \in \overline{\mathrm{M}} \cdot \Delta;\Gamma \vdash_m \mathrm{M_i} \text{ in G parent H}}{\Delta;\Gamma \vdash_c \textbf{class G inherit H define } \overline{\mathrm{D}} \textbf{ feature } \overline{\mathrm{M}}\ \overline{\mathrm{F}} \textbf{ end}}$$

Finally, here is the rule for program verification:

$$\frac{\begin{array}{l}\forall \mathrm{i} \in 1..\mathrm{n} \cdot \mathrm{L_i} = \textbf{class } \mathrm{G_i} \ldots \textbf{end} \\ \Gamma = \mathit{specs}(\mathrm{L_1} \ldots \mathrm{L_n}) \\ \forall \mathrm{i} \in 1..\mathrm{n} \cdot \ \mathit{apf}(\mathrm{L_i});\Gamma \vdash_c \mathrm{L_i} \\ \mathrm{true};\Gamma \vdash_s \{\mathrm{true}\}\mathrm{s}\{\mathrm{true}\}\end{array}}{\vdash_p \mathrm{L_1} \ldots \mathrm{L_n}\ \mathrm{s}}$$

The function *apf* translates the abstract predicate family definitions of a class into a formula – its apf assumptions. It is adapted from [58] for tagged arguments:

$$\mathit{apf}(\textbf{class G} \ldots \textbf{define } \mathrm{D_1}\ \mathrm{D_2} \ldots \mathrm{D_n} \textbf{ feature} \ldots \textbf{end}) \stackrel{\text{def}}{=} \mathit{apf}_{\mathrm{G}}(\mathrm{D_1}) \wedge \ldots \wedge \mathit{apf}_{\mathrm{G}}(\mathrm{D_n})$$

$$\mathit{apf}_{\mathrm{G}}(\mathrm{x}.\mathrm{p}_G(\mathrm{Y}) \textbf{ as } \mathrm{P}) \stackrel{\text{def}}{=} \mathit{FtoE}(\mathrm{p,G,Y}) \wedge \mathit{EtoD}(\mathrm{x}.\mathrm{p}_G(\mathrm{Y}) \textbf{ as } \mathrm{P}) \wedge (\forall \mathrm{x} <: \mathrm{G} \cdot \mathit{TR}(\mathrm{p,x,Y}))$$

$$\mathit{FtoE}(\mathrm{p,G},\overline{\mathrm{t:y}}) \stackrel{\text{def}}{=} \forall \mathrm{x},\overline{\mathrm{y}} \cdot \mathrm{x} : \mathrm{G} \Rightarrow [\mathrm{x}.\mathrm{p}(\overline{\mathrm{t:y}}) \Leftrightarrow \mathrm{x}.\mathrm{p}_G(\overline{\mathrm{t:y}})]$$

$$\mathit{EtoD}(\mathrm{x}.\mathrm{p}_G(\overline{\mathrm{t:y}}) \textbf{ as } \mathrm{P}) \stackrel{\text{def}}{=} \forall \mathrm{x},\overline{\mathrm{y}} \cdot \mathrm{x}.\mathrm{p}_G(\overline{\mathrm{t:y}}) \Leftrightarrow \mathrm{P}$$

$$\mathit{TR}(\mathrm{p,x},\overline{\mathrm{t:y}}) \stackrel{\text{def}}{=} \bigwedge_{\overline{\mathrm{t':y'}} + \overline{\mathrm{t''':y''}} \equiv \overline{\mathrm{t:y}}} \forall \overline{\mathrm{y'}} \cdot \mathrm{x}.\mathrm{p}(\overline{\mathrm{t':y'}}) \Leftrightarrow \mathrm{x}.\mathrm{p}(\overline{\mathrm{t':y'}} \ + \ \overline{\mathrm{t'':\_}})$$

2.8 Useful lemmas

Lemmas 2.1 and 2.2 are frequently used in proofs of Behavioural Subtyping and Inheritance:

Lemma 2.1. $\Delta \vdash (\textbf{also}_{\mathrm{i}\in\mathrm{I}}\ \{\mathrm{P_i}\}\_\{\mathrm{Q_i}\}) \Longrightarrow \{\mathrm{P_k}\}\_\{\mathrm{Q_k}\}$ for all $\mathrm{k} \in \mathrm{I}$.

Lemma 2.2. If $\Delta \vdash \{\mathrm{P}\}\_\{\mathrm{Q}\} \Longrightarrow \{\mathrm{S_i}\}\_\{\mathrm{T_i}\}$ for all $\mathrm{i} \in \mathrm{I}$, then $\Delta \vdash \{\mathrm{P}\}\_\{\mathrm{Q}\} \Longrightarrow (\textbf{also}_{\mathrm{i}\in\mathrm{I}}\ \{\mathrm{S_i}\}\_\{\mathrm{T_i}\})$.

For Body Verification:

Lemma 2.3. If $\Delta;\Gamma \vdash_s \{\mathrm{S_i}\}\mathrm{s}\{\mathrm{T_i}\}$ for all $\mathrm{i} \in \mathrm{I}$, then under assumptions Δ and Γ, statement s satisfies $(\textbf{also}_{\mathrm{i}\in\mathrm{I}}\ \{\mathrm{S_i}\}\_\{\mathrm{T_i}\})$.

Chapter 3

Reasoning about related abstractions

The use of data abstractions is a hallmark of OO programming. A class is a typical example of such an abstraction. In interface or general multiple inheritance hierarchies, a class can combine and maintain several abstractions offered by its parents. Although most examples of this chapter involve abstractions in connection with inheritance, not all data abstractions are directly coupled with language constructs. Classes use them for various purposes: to simplify how clients manipulate a class, to separate various concepts that are combined in a class, or to encourage or enforce particular call protocols. For example, a complex object with a long initialisation phase can use the abstractions 'initialising' and 'ready', with methods applicable to an 'initialising' object, others to a 'ready' object, and some to both. Or algorithms can manipulate a mutable data structure under an 'immutable' abstraction, even if there is no interface making this explicit.

Relationships between data abstractions are important when reasoning about OO code. This chapter explores the problem of relating abstractions in an information hiding setting, where implementation details of abstractions are hidden from clients. Suppliers must therefore express and fulfill relationships between abstractions. If a class offers 'student' and 'person' abstractions (by using inheritance, or other means), for example, it might allow clients to convert a 'student' abstraction into a 'person' one. Clients can then manipulate the 'person' abstraction by calling e.g. library routines. After the manipulation, they might be allowed to convert back and assume that the number of exams the 'student' has taken is still the same. Specification mechanisms are needed to express and enforce the programmer's intentions about such relationships. This is especially important in multiple inheritance

hierarchies where classes combine multiple abstractions in complicated ways.

Abstract predicate families are a flexible mechanism for data abstraction in OO specifications. An apf P provides a predicate name for an abstraction; each class C can define an *entry* predicate P_C. The definition of P_C describes how class C implements the apf P, and is hidden from other classes. For example, apfs S and P can be used to provide an abstraction of students and persons in a program respectively. Class STUDENT can define the entries $\mathrm{S}_{STUDENT}$ and $\mathrm{P}_{STUDENT}$, while other classes can define their apf entries differently. The predicate x.S(age: a, exm: e) describes object x under the 'student' abstraction: its age is 'a' and the number of exams taken is 'e'. If the dynamic type of x is STUDENT, then class STUDENT can use the fact that

$$\mathrm{x.S(age: a, exm: e)} \Leftrightarrow \mathrm{x.S}_{STUDENT}\mathrm{(age: a, exm: e)}$$

In other words, the dynamic type of the first argument of an apf predicate (in this case the dynamic type of x) determines which apf entry applies. Apf predicates can therefore be seen to mirror dynamic dispatch of OO programs in the logic. The apf mechanism is modular and exercises information hiding: only the class defining an apf entry knows the definition and can relate its entry to the apf predicate.

The relationship described before, namely that a 'student' abstraction can be converted into a 'person' one, can be expressed as follows:

$$\mathrm{x.S(age: a, exm: e)} \Rightarrow \mathrm{x.P(age: a)}$$

Allowing the back conversion without affecting the number of exams requires a stronger property that uses separation logic's $*$-connective:

$$\mathrm{x.S(age: a, exm: e)} \Leftrightarrow [\mathrm{x.P(age: a)} * \mathrm{x.RestStoP(exm: e)}] \qquad \text{(A)}$$

where RestStoP abstracts the parts of a 'student' abstraction that are independent from and not included in a 'person' one. With this property, a client can now reason as the following proof outline shows:

$\{\mathrm{x.S(age: a, exm: e)}\}$
$\{\mathrm{x.P(age: a)} * \mathrm{x.RestStoP(exm: e)}\}$
  $\{\mathrm{x.P(age: a)}\}$
    // Manipulation of 'person' abstraction by library routines.
  $\{\mathrm{x.P(age: a + 1)}\}$
$\{\mathrm{x.P(age: a + 1)} * \mathrm{x.RestStoP(exm: e)}\}$
$\{\mathrm{x.S(age: a + 1, exm: e)}\}$

The Frame rule of separation logic guarantees that the disjoint x.RestStoP(exm: e) remains unchanged. In essence, the client uses property (A) in combination with the Frame rule to infer an S-based specification for the library manipulation. It is not necessary to re-specify or re-verify the library – knowledge of the relationship saves specification overhead and keeps reasoning modular.

To which objects the property (A) applies is a design choice: a programmer might express that *selected* classes in a heterogeneous hierarchy fulfill the relationship, or that *all* classes in a homogeneous hierarchy fulfill it. We introduce two general specification mechanisms for the two cases: *export clauses* to express properties that hold for individual classes, and *axiom clauses* to describe properties of entire hierarchies. If class STUDENT specifies

export
∀x,a,e· x : STUDENT ⇒ [x.S(age: a, exm: e) ⇔
[x.P(age: a) ∗ x.RestStoP(exm: e)]]
where {}

then a client must know that an object's dynamic type is exactly STUDENT before using the information in reasoning. On the other hand, if class STUDENT specifies

axiom
S_P: ∀a,e· S(age: a, exm: e) ⇔ [P(age: a) ∗ RestStoP(exm: e)]

then clients can use the stronger implication

∀x,a,e· x <: STUDENT ⇒
[x.S(age: a, exm: e) ⇔ [x.P(age: a) ∗ x.RestStoP(exm: e)]]

Knowledge that the object's dynamic type is a subtype of STUDENT (including STUDENT) suffices to use the relationship. This is much more convenient for clients: a sound OO type system will guarantee that if a variable has static type STUDENT and references an object, then the object's dynamic type will always be a subtype of STUDENT.

Axiom clauses offer a general facility to constrain the implementation of abstractions in subclasses. For example, class STUDENT can express that the number of exams a 'student' has taken is always non-negative, and that all subclasses should use its implementation of the 'student' abstraction:

axiom
exm_non_neg: ∀a,e· S(age: a, exm: e) ⇒ 0 ≤ e
S_constraint: ∀a,e· S(age: a, exm: e) ⇔ $S_{STUDENT}$(age: a, exm: e)

Axiom clause 'exm_non_neg' guarantees clients that 0 ≤ e whenever they know x.S(age: a, exm: e) and x <: STUDENT. Subclass representation constraints such as the one expressed in the 'S_constraint' clause are useful for

ensuring safe interaction between statically and dynamically dispatched calls on the same object – a pervasive pattern in OO programs (see Section 5.5 of [58]).

The claims made in export and axiom specifications must be checked to obtain sound reasoning. Whether or not a class fulfills axiom clauses often depends on properties of particular other classes, such as its parents. For this reason our proof system has a layered assumption structure: axiom verification can use export information of all classes in a program, and method verification can additionally use axiom information. Several examples will show how export and axiom clauses are verified and applied in verification.

This chapter extends the proof system of Parkinson and Bierman (see chapter 2) with export/axiom clauses, abstract classes, abstract methods and shared multiple inheritance[1] where fields and methods of common ancestor classes are not replicated in the descendant [26]. Apart from the use of apfs to support abstraction and information hiding, Parkinson and Bierman's system has the attractive property that it can verify a wide range of inheritance uses and abuses. Flexible handling of inheritance is vital in a proof system for multiple inheritance, since classes often interrelate methods and data from parents in complicated ways: methods can intertwine ancestor abstractions as the example in Section 3.1.1 shows, or abstractions can share parts in the case of diamond inheritance. Export and axiom clauses provide flexible ways to relate abstractions, thereby giving clients an integrated view of how methods affect abstractions from different inheritance paths. Very few proof systems exist for multiple inheritance, and no proof system we know of can facilitate reasoning about multiple related abstractions at the same level of abstraction as ours.

We implemented our proof system in MultiStar – a fully automatic verification tool. MultiStar has a two-tier architecture: a GUI front-end that translates Eiffel code and specifications into a simpler form for verification, and a language-independent back-end based on jStar [23] for reasoning. The front-end uses specifications written inside classes. Future front-ends for e.g. Java and C# can reuse the MultiStar back-end: with its support for interface inheritance, export/axiom clauses, abstract classes and abstract methods, a wide range of programs can be verified. As we shall see, export and axiom clauses can also be useful in single-inheritance situations.

All the examples presented in this chapter have been verified with MultiStar. To demonstrate the flexibility of our approach, we also used MultiStar to verify the iterator hierarchy of the Gobo data structure library [29]. The complete code and specifications of the examples and Gobo case study are

[1] Inheritance with *virtual base classes* in C++ terminology [27].

available online [28].

Chapter outline Several examples illustrating the new specification mechanisms and proof system follow in Section 3.1. Section 3.2 presents the MultiStar tool, and Section 3.3 reports on the case study with Gobo iterator classes. A formal exposition of our proof system appears in Section 3.4. Section 3.5 mentions related work and concludes. Appendix A contains an overview of the formal semantics of our proof system and a proof of soundness.

3.1 Examples

A class may inherit from several parent classes. This is indicated by listing the names of all parents in its **inherit** section. The new **export** and **axiom** sections respectively contain the export and axiom clauses of a class. As usual, empty sections will simply be omitted.

We use the method specification shorthands of [58]: if only a static specification is listed, the dynamic specification is assumed to be exactly the same, and if only a dynamic specification is listed, then a static specification is derived by replacing each apf predicate whose first argument is **Current** with the entry predicate of the class. In other words, if the shorthand is used in class C, then $\mathrm{p}(\overline{\mathrm{t}:\mathrm{e}})$ is replaced with $\mathrm{p}_C(\overline{\mathrm{t}:\mathrm{e}})$. Specifications of non-constructor methods are furthermore propagated down the hierarchy: if a class does not explicitly list an inherited method, then it is assumed to have the same static and dynamic specifications as determined for the parent class. To avoid ambiguity, we require that if the method is available in multiple parents, then they must all have identical specifications for it.

The examples do not discuss details that are uninteresting from the perspective of this thesis, such as proofs of correctness of simple ancestor classes. The paper of Parkinson and Bierman [58] contains several examples that illustrate the basics.

3.1.1 Intertwining ancestor abstractions

Classes CELL and COUNTER are shown in Figure 3.1. CELL models mutable integer-valued cells and uses apf Cell, while COUNTER uses apf Cn. The apfs provide logical abstractions of mutable cells and counters respectively. Class CCELL in Figure 3.2, the focus of this example, inherits from CELL and COUNTER. It intertwines the functionality of its parents by overriding *set_value* to store the value and increment the count. It uses apf Cc to

```
class CELL
define x.Cell_CELL(val: v) as x.value ↪ v
feature
    introduce CELL(v: INT)
    dynamic {value ↪ _}_{Cell(val: v)}
    do value := v end

    introduce value(): INT
    dynamic {Cell(val: v)}_{Cell(val: v) * Result = v}
    do Result := value end

    introduce set_value(v: INT)
    dynamic {Cell(val: _)}_{Cell(val: v)}
    do value := v end

    value: INT
end

class COUNTER
define x.Cn_COUNTER(cnt: c) as x.count ↪ c
feature
    introduce COUNTER()
    dynamic {count ↪ _}_{Cn(cnt: 0)}
    do count := 0 end

    introduce count(): INT
    dynamic {Cn(cnt: c)}_{Cn(cnt: c) * Result = c}
    do Result := count end

    introduce increment()
    dynamic {Cn(cnt: c)}_{Cn(cnt: c + 1)}
    do tmp: INT. tmp := count; count := tmp + 1 end

    count: INT
end
```

Figure 3.1: The CELL and COUNTER classes.

```
class CCELL inherit CELL COUNTER
define
x.Cell_CCELL(val: v, cnt: c) as x.Cc_CCELL(val: v, cnt: c)
x.Cn_CCELL(cnt: c) as x.Cn_COUNTER(cnt: c)
x.Cc_CCELL(val: v, cnt: c) as x.Cell_CELL(val: v) * x.Cn_COUNTER(cnt: c)
export
    ∀x· x : CCELL ⇒ [∀c,v· x.Cc(val: v, cnt: c) ⇔ x.Cell(val: v, cnt: c) ⇔
    (x.Cn(cnt: c) * Rest(x,v))] where { Rest(x,v) = x.Cell_CELL(val: v) }
feature
    introduce CCELL(v: INT)
    dynamic {value ↪ _ * count ↪ _}_{Cc(val: v, cnt: 0)}
    do Precursor{CELL}(v); Precursor{COUNTER}() end

    inherit value(): INT
    dynamic {Cc(val: v, cnt: c)}_{Cc(val: v, cnt: c) * Result = v}
        also {Cell(val: v, cnt: c)}_{Cell(val: v, cnt: c) * Result = v}

    override set_value(v: INT)
    dynamic {Cc(val: _, cnt: c)}_{Cc(val: v, cnt: c + 1)}
        also {Cell(val: _, cnt: c)}_{Cell(val: v, cnt: c + 1)}
    do CCELL::increment(); CELL::set_value(v) end

    inherit count(): INT
    dynamic {Cc(val: v, cnt: c)}_{Cc(val: v, cnt: c) * Result = c}
        also {Cn(cnt: c)}_{Cn(cnt: c) * Result = c}

    inherit increment()
    dynamic {Cc(val: v, cnt: c)}_{Cc(val: v, cnt: c + 1)}
        also {Cn(cnt: c)}_{Cn(cnt: c + 1)}
end

// In an arbitrary class or library:
use_counter(c: COUNTER)
    dynamic {c.Cn(cnt: v)}_{c.Cn(cnt: v + 10)}

use_cell(c: CELL, v: INT)
    dynamic {c.Cell(val: _)}_{c.Cell(val: v)}
```

Figure 3.2: The CCELL class and two library methods.

provide an abstraction of such objects in the logic, and 'grows' Cell_{CCELL} to accommodate method *set_value*, as we shall see.

The single export clause of CCELL relates the Cc, Cell and Cn abstractions. Only the predicate in front of **where** is exported for reasoning. Predicate definitions following **where** are used only to verify the clause and allow a class to hide implementation/representation details in its interface without introducing new predicate families. For example, if we modify the representation of class CCELL, then the definition of Rest(x,v) can be changed without invalidating correctness proofs of client code.

To verify an export clause, we must prove that the exported predicate follows from the standard apf assumptions of the class and the predicate definitions after the **where** keyword. The proof for CCELL's export clause is trivial (Section 2.7 describes the standard apf assumptions of a class). Note that export clauses are not verified for a particular dynamic type, since the standard apf assumptions of a class do not assume a particular one. It is therefore sound to use exported information to verify axiom clauses and methods of other classes, as we will do in later examples. However, nothing prevents the user from writing exported predicates as implications where the antecedent is of the form x : Type (see the export clause of CCELL). In this case information in the consequent can be applied only to objects satisfying this type constraint.

For the constructor we have to prove that its body satisfies the static specification (note that a **Precursor** call is syntactic sugar for a direct call):

$\{\text{value} \hookrightarrow \_ * \text{count} \hookrightarrow \_\}$
CELL::CELL(v)
$\{\text{Cell}_{CELL}(\text{val: v}) * \text{count} \hookrightarrow \_\}$
Precursor{COUNTER}()
$\{\text{Cell}_{CELL}(\text{val: v}) * \text{Cn}_{COUNTER}(\text{cnt: 0})\}$
$\{\text{Cc}_{CCELL}(\text{val: v, cnt: 0})\}$

The constructor body simply passes the needed fields to parent constructors and treats their internal representations abstractly thereafter.[2]

Method *value* is respecified in CCELL with Cell and Cc specificiations. Since it inherits the body from CELL, we must prove that the new static specification is satisfied assuming the body's static specification of CELL. This method proof obligation is called Inheritance in the formalisation. By applying the Frame rule (with $\text{Cn}_{COUNTER}(\text{cnt: c})$) and then the rule of Con-

[2]To simplify the formal presentation of proofs and make them more transparent, we mention fields explicitly in constructor preconditions. MultiStar injects them automatically – see Section 3.2.1 for more discussion.

sequence, we can derive each **also**-ed static specification, which is sufficient to conclude the proof by Lemma 2.2. Another proof obligation for *value* is Behavioural subtyping, where we must show that the dynamic specification listed in CELL follows from the new one, i.e. that CCELL maintains the old specification. For the proof, we first 'choose' the Cell dynamic spec with Lemma 2.1, and then remove the cnt tag by applying the Auxiliary Variable Elimination and Consequence rules. The application of Auxiliary Variable Elimination quantifies the variable c existentially in the pre- and postcondition, and the application of Consequence uses tag reduction information which is part of CCELL's standard apf assumptions.

Behavioural subtyping of *set_value* is similar. For its Body verification obligation, we must prove that both Cell_{CCELL} and Cc_{CCELL} static specifications are satisfied. The proof proceeds as follows:

$\{\mathrm{Cell}_{CCELL}(\mathrm{val}:\_,\ \mathrm{cnt}:\mathrm{c})\}$
  CCELL::increment()
$\{\mathrm{Cell}_{CCELL}(\mathrm{val}:\_,\ \mathrm{cnt}:\mathrm{c}+1)\}$
$\{\mathrm{Cell}_{CELL}(\mathrm{val}:\_) * \mathrm{Cn}_{COUNTER}(\mathrm{cnt}:\mathrm{c}+1)\}$
  CELL::set_value(v)
$\{\mathrm{Cell}_{CELL}(\mathrm{val}:\mathrm{v}) * \mathrm{Cn}_{COUNTER}(\mathrm{cnt}:\mathrm{c}+1)\}$
$\{\mathrm{Cell}_{CCELL}(\mathrm{val}:\mathrm{v},\ \mathrm{cnt}:\mathrm{c}+1)\}$

An application of Consequence proves the other **also**-ed static spec and Lemma 2.3 completes the proof. As the body operates on state described by Cell_{CELL} and $\mathrm{Cn}_{COUNTER}$, the proof obligations and separation logic's faulting semantics demand that we 'grow' Cell_{CCELL} to include both state parcels.

Now consider the two library routines at the bottom of Figure 3.2. The export clause contains the necessary information to prove the two triples:

$\{\mathrm{true}\}$ cc := **new** CCELL(5); use_counter(cc) $\{\mathrm{cc.Cc}(\mathrm{val}:5, \mathrm{cnt}:10)\}$
$\{\mathrm{true}\}$ cc := **new** CCELL(5); use_cell(cc,20) $\{\mathrm{cc.Cc}(\mathrm{val}:20, \mathrm{cnt}:\_)\}$

The proof of the second triple reduces and expands tags according to standard apf rules:

$\{\mathrm{true}\}$
  cc := **new** CCELL(5)
$\{\mathrm{cc} : \mathrm{CCELL} * \mathrm{cc.Cc}(\mathrm{val}:5, \mathrm{cnt}:0)\}$
$\{\mathrm{cc} : \mathrm{CCELL} * \mathrm{cc.Cell}(\mathrm{val}:5, \mathrm{cnt}:0)\}$
$\{\mathrm{cc} : \mathrm{CCELL} * \mathrm{cc.Cell}(\mathrm{val}:5, \mathrm{cnt}:\_)\}$
$\{\mathrm{cc} : \mathrm{CCELL} * \mathrm{cc.Cell}(\mathrm{val}:5)\}$
  use_cell(cc,20)
$\{\mathrm{cc} : \mathrm{CCELL} * \mathrm{cc.Cell}(\mathrm{val}:20)\}$

{cc : CCELL ∗ cc.Cell(val: 20, cnt: _)}
{cc.Cc(val: 20, cnt: _)}

Information about cnt is lost in the postcondition, which is unavoidable because *use_cell* could call *set_value* more than once. In a version of CCELL where Cn_{CCELL} is defined to include the Cell_{CELL} state and the equivalence of Cc, Cn and Cell is exported, information about val will likewise be lost in the first triple. Also note that dynamic type information is required to use the exported relationships, since the subclasses of CCELL are not obliged to implement them.

3.1.2 Access control and call protocols

Our proof system can enforce interesting access control patterns in verified programs. Consider class CCEL2 in Figure 3.3 which has the same executable code as CCELL but different specifications. Its export clause relates the Cell and Cn abstractions in a one-directional way. The constructor produces a Cell apf predicate with which methods *value*, *set_value* and *count* can be called. Verified clients cannot call *increment* with the Cell predicate. They must use exported information to get a Cn predicate, yet they lack information to change back after the call: no export or axiom clause is available to do this, and every method producing a Cell predicate requires one. The following proof attempt where cc2 : CCEL2 shows the problem:

{cc2.Cell(val: v, cnt: c)}
{cc2.Cn(cnt: c)}
    cc2.increment()
{cc2.Cn(cnt: c + 1)}
{???}
{cc2.Cell(val: _, cnt: _)} // The weakest requirement of *set_value*.
    cc2.set_value(10)
{cc2.Cell(val: _, cnt: _)}

While the client has a Cell predicate, the argument tagged by cnt and returned by *count* reflects precisely how many times the value has been set. If the client tries to manipulate the count by calling *increment*, then it can never regain the needed capability to call *value* and *set_value*, and must forever treat the object as a simple counter in the code. The combination of abstract predicate relationships and method specifications enforces this protocol in verified code.

class CCEL2 **inherit** CELL COUNTER
define
x.Cell_{CCEL2}(val: v, cnt: c) **as** x.Cell_{CELL}(val: v) $*$ x.$\text{Cn}_{COUNTER}$(cnt: c)
x.Cn_{CCEL2}(cnt: c) **as** x.$\text{Cn}_{COUNTER}$(cnt: c)
export
    $\forall$x$\cdot$ x : CCEL2 $\Rightarrow$ [$\forall$v,c$\cdot$ x.Cell(val: v, cnt: c) $\Rightarrow$ x.Cn(cnt: c)] **where** {}
feature
    introduce CCEL2(v: INT)
    dynamic {value $\hookrightarrow$ _ $*$ count $\hookrightarrow$ _}_{Cell(val: v, cnt: 0)}
    do Precursor{CELL}(v); **Precursor**{COUNTER}() **end**

    inherit value(): INT
    dynamic {Cell(val: v, cnt: c)}_{Cell(val: v, cnt: c) $*$ **Result** = v}

    override set_value(v: INT)
    dynamic {Cell(val: _, cnt: c)}_{Cell(val: v, cnt: c + 1)}
    do CCEL2::increment(); CELL::set_value(v) **end**

    inherit count(): INT
    dynamic {Cell(val: v, cnt: c)}_{Cell(val: v, cnt: c) $*$ **Result** = c}
        also {Cn(cnt: c)}_{Cn(cnt: c) $*$ **Result** = c}
end

Figure 3.3: The CCEL2 class.

```
class PERSON
define x.P_PERSON(age: a) as x.age ↪ a
export ∀x,a· x.P_PERSON(age: a) ⇔ x.age ↪ a where {}
feature
    introduce PERSON(a: INT)
    dynamic {age ↪ _}_{P(age: a)}
    do age := a end

    introduce age(): INT
    dynamic {P(age: a)}_{P(age: a) ∗ Result = a}
    do Result := age end

    introduce set_age(a: INT)
    dynamic {P(age: _)}_{P(age: a)}
    do age := a end

    introduce celebrate_birthday()
    static {P(age: a)}_{P(age: a + 1)}
    do tmp: INT. tmp := age(); tmp := tmp+1; set_age(tmp) end

    age: INT
end
```

Figure 3.4: The PERSON class.

3.1.3 Diamond inheritance

Verification of multiple inheritance requires proper handling of data from several parent classes. Diamond inheritance complicates matters because common ancestor fields are shared. This is unproblematic for our proof system, although the abstraction of the shared data is typically lost. Diamond inheritance can moreover require relationships between several abstractions, which this example achieves with axiom clauses.

An axiom clause consists of a name and a predicate. The name identifies the clause and allows subclasses to refine the predicate. We propagate axiom clauses down the hierarchy to save specification overhead: if a class does not list an axiom clause with the same name as one in a parent, then it is assumed to list an identical clause. To avoid ambiguity in the presence of multiple inheritance, we require that if clauses with the same name are present in multiple parents, then they must all be identical. An axiom clause copied in

class STUDENT **inherit** PERSON **define**
x.$P_{STUDENT}$(age: a) **as** x.P_{PERSON}(age: a)
x.$S_{STUDENT}$(age: a, exm: e) **as** x.$P_{STUDENT}$(age: a) $*$ x.exams $\hookrightarrow$ e
x.$RestStoP_{STUDENT}$(exm: e) **as** x.exams $\hookrightarrow$ e
export $\forall$x,a,e· [x.P_{PERSON}(age: a) $*$ x.$RestStoP_{STUDENT}$(exm: e)] $\Leftrightarrow$
x.$S_{STUDENT}$(age: a, exm: e) **where** {}
axiom S_P: $\forall$a, e· S(age: a, exm: e) $\Leftrightarrow$ [P(age: a) $*$ RestStoP(exm: e)]
feature
introduce STUDENT(a: INT, e: INT)
dynamic {age $\hookrightarrow$ _ $*$ exams $\hookrightarrow$ _}_{S(age: a, exm: e)}
do Precursor{PERSON}(a); exams := e **end**

introduce exams(): INT
dynamic {S(age: a, exm: e)}_{S(age: a, exm: e) $*$ **Result** = e}
do Result := exams **end**

introduce take_exam()
dynamic {S(age: a, exm: e)}_{S(age: a, exm: e + 1)}
do tmp: INT. tmp := exams; exams := tmp + 1 **end**

exams: INT
end

Figure 3.5: The STUDENT class.

class SMUSICIAN **inherit** STUDENT MUSICIAN
define
x.$P_{SMUSICIAN}$(age: a) **as** x.P_{PERSON}(age: a)
x.$S_{SMUSICIAN}$(age: a, exm: e) **as** x.$S_{STUDENT}$(age: a, exm: e)
x.$M_{SMUSICIAN}$(age: a, pfm: p) **as** x.$M_{MUSICIAN}$(age: a, pfm: p)
x.$SM_{SMUSICIAN}$(age: a, exm: e, pfm: p) **as** x.P_{PERSON}(age: a) ∗
x.$RestStoP_{STUDENT}$(exm: e) ∗ x.$RestMtoP_{MUSICIAN}$(pfm: p)
x.$RestStoP_{SMUSICIAN}$(exm: e) **as** x.$RestStoP_{STUDENT}$(exm: e)
x.$RestMtoP_{SMUSICIAN}$(pfm: p) **as** x.$RestMtoP_{MUSICIAN}$(pfm: p)
x.$RestSMtoS_{SMUSICIAN}$(pfm: p) **as** x.$RestMtoP_{MUSICIAN}$(pfm: p)
x.$RestSMtoM_{SMUSICIAN}$(exm: e) **as** x.$RestStoP_{STUDENT}$(exm: e)
axiom
SM_S: ∀a, e, p· SM(age: a, exm: e, pfm: p) ⇔
[S(age: a, exm: e) ∗ RestSMtoS(pfm: p)]
SM_M: ∀a, e, p· SM(age: a, exm: e, pfm: p) ⇔
[M(age: a, pfm: e) ∗ RestSMtoM(exm: e)]
feature
introduce SMUSICIAN(a: INT, e: INT, p: INT)
dynamic {age ↪ _ ∗ exams ↪ _ ∗ performances ↪ _}_
{SM(age: a, exm: e, pfm: p)}
do **Precursor**{STUDENT}(a,e); **Precursor**{MUSICIAN}(a,p) **end**

introduce do_exam_performance()
static {SM(age: a, exm: e, pfm: p)}_{SM(age: a, exm: e + 1, pfm: p + 1)}
do take_exam(); perform() **end**
end

Figure 3.6: The SMUSICIAN class.

this way is not refined in the subclass and automatically consistent with its parent versions. In the general case where a subclass refines an axiom clause, Parent consistency must be proven as indicated in the later formalisation.

The focus of this example is class SMUSICIAN, shown in Figure 3.6. It inherits from STUDENT and MUSICIAN, both which inherit from PERSON. The STUDENT and PERSON classes are shown in Figures 3.4 and 3.5; MUSICIAN is similar to STUDENT and not shown. A diamond is formed with PERSON at the top, and an instance of SMUSICIAN has one 'age' field, one *set_age* method, etc. under shared multiple inheritance semantics. The classes use axiom clauses to specify relationships between abstractions P, S, M and SM.

Since SMUSICIAN is non-abstract, we must prove that axiom SM_S holds for its direct instances. This proof obligation for axiom clauses is called Implication in the formalisation that will follow. It holds indeed, since under the standard apf assumptions of SMUSICIAN, exported information of all classes, and the assumption **Current** : SMUSICIAN, we have:

$\mathrm{SM}(\mathrm{age}:\mathrm{a}, \mathrm{exm}:\mathrm{e}, \mathrm{pfm}:\mathrm{p})$
$\Leftrightarrow$ // Standard apf assumptions, **Current** : SMUSICIAN
$\mathrm{SM}_{SMUSICIAN}(\mathrm{age}:\mathrm{a}, \mathrm{exm}:\mathrm{e}, \mathrm{pfm}:\mathrm{p})$
$\Leftrightarrow$ // Standard apf assumptions.
$\mathrm{P}_{PERSON}(\mathrm{age}:\mathrm{a}) * \mathrm{RestStoP}_{STUDENT}(\mathrm{exm}:\mathrm{e}) *$
$\mathrm{RestMtoP}_{MUSICIAN}(\mathrm{pfm}:\mathrm{p})$
$\Leftrightarrow$ // Exported information from STUDENT.
$\mathrm{S}_{STUDENT}(\mathrm{age}:\mathrm{a}, \mathrm{exm}:\mathrm{e}) * \mathrm{RestMtoP}_{MUSICIAN}(\mathrm{pfm}:\mathrm{p})$
$\Leftrightarrow$ // Standard apf assumptions.
$\mathrm{S}_{SMUSICIAN}(\mathrm{age}:\mathrm{a}, \mathrm{exm}:\mathrm{e}) * \mathrm{RestSMtoS}_{SMUSICIAN}(\mathrm{pfm}:\mathrm{p})$
$\Leftrightarrow$ // Standard apf assumptions, **Current** : SMUSICIAN
$\mathrm{S}(\mathrm{age}:\mathrm{a}, \mathrm{exm}:\mathrm{e}) * \mathrm{RestSMtoS}(\mathrm{pfm}:\mathrm{p})$

The export clause in STUDENT is not closely connected to multiple inheritance. In fact, any class C inheriting from STUDENT which defines P_C to be P_{PERSON} and S_C to be $\mathrm{S}_{STUDENT}$ will need the export clause to prove Implication of S_P, which we omit here for SMUSICIAN. Axiom verification frequently requires exported information of this kind. What is vital about the export clause in the shared multiple inheritance setting is that it isolates the shared ancestor state of SMUSICIAN, namely P_{PERSON}. This allows SMUSICIAN to relate ancestor abstractions in a fairly abstract way. Only the constructor's Body verification proof needs the export clause in PERSON[3]:

[3]Unless a class manipulates fields of its ancestors directly, this export clause would not be needed in languages where constructors of common ancestor classes cannot be called more than once.

$\{\text{age} \hookrightarrow \_ * \text{exams} \hookrightarrow \_ * \text{performances} \hookrightarrow \_\}$
Precursor{STUDENT}(a,e)
$\{\text{S}_{STUDENT}(\text{age: a, exm: e}) * \text{performances} \hookrightarrow \_\}$
$\{\text{P}_{PERSON}(\text{age: a}) * \text{RestStoP}_{STUDENT}(\text{exm: e}) * \text{performances} \hookrightarrow \_\}$
$\{\text{age} \hookrightarrow \text{a} * \text{RestStoP}_{STUDENT}(\text{exm: e}) * \text{performances} \hookrightarrow \_\}$
Precursor{MUSICIAN}(a,p)
$\{\text{M}_{MUSICIAN}(\text{age: a, pfm: p}) * \text{RestStoP}_{STUDENT}(\text{exm: e})\}$
$\{\text{P}_{PERSON}(\text{age: a}) * \text{RestMtoP}_{MUSICIAN}(\text{pfm: p}) *$
$\text{RestStoP}_{STUDENT}(\text{exm: e})\}$
$\{\text{SM}_{SMUSICIAN}(\text{age: a, exm: e, pfm: p})\}$

Note that class SMUSICIAN would not have needed exported information if it ignored the parent constructors and simply overrode everything. The same is true for proof systems with less abstraction where method bodies are reverified in subclasses.

Since **Current** in SMUSICIAN will always reference an object whose dynamic type is a subtype of SMUSICIAN, the Body verification proof of *do_exam_performance* can use axiom information to infer SM-specs for *take_exam* and *perform*:

$\{\text{SM}(\text{age: a}, \text{exm: e}, \text{pfm: p})\}$
$\{\text{S}(\text{age: a}, \text{exm: e}) * \text{RestSMtoS}(\text{pfm: p})\}$
take_exam()
$\{\text{S}(\text{age: a}, \text{exm: e} + 1) * \text{RestSMtoS}(\text{pfm: p})\}$
$\{\text{SM}(\text{age: a}, \text{exm: e} + 1, \text{pfm: p})\}$
$\{\text{M}(\text{age: a}, \text{pfm: p}) * \text{RestSMtoM}(\text{exm: e} + 1)\}$
perform()
$\{\text{M}(\text{age: a}, \text{pfm: p} + 1) * \text{RestSMtoM}(\text{exm: e} + 1)\}$
$\{\text{SM}(\text{age: a}, \text{exm: e} + 1, \text{pfm: p} + 1)\}$

The specification overhead incurred by axiom clauses is offset by specification inference gains: SM, S and M specifications can be inferred for *age*, *set_age* and *celebrate_birthday*, while SM specifications can be inferred for *exams*, *take_exam*, *performances* and *get_performance* – a total of 13 specifications for methods of SMUSICIAN. These inferred specifications are guaranteed to be implemented by all subclasses, and no dynamic type information is needed to use them[4]. Yet the system is still flexible – a subclass can always choose

[4]The technique used by Chin et al. [14] of inheriting static method specifications and deriving dynamic specifications from them implements "internal specification inference", i.e. a class infers and publishes dynamic specifications for its methods which external clients can use. In contrast to this, the technique of equipping classes with export/axiom clauses implements "external specification inference", i.e. clients infer specifications for methods based on published export/axiom information. A benefit of the

to satisfy such constraints vacuously by defining selected apf entries as false. Class DCell in [58] provides an example of this.

3.2 MultiStar

This section sketches notable aspects of the MultiStar implementation. MultiStar has a two-tier architecture: a front-end that translates Eiffel programs and specifications into a simpler form for verification, and a language-independent back-end based on jStar [23] which implements our proof system.

3.2.1 Front-end

The front-end provides a graphical user interface within the EVE integrated development environment, and is part of the standard EVE download [28]. It translates Eiffel code and specifications into the back-end's input format, and provides access to verification results. Verification is triggered by picking and dropping an annotated class on the MultiStar tool. Class annotations consist of apf entry definitions, export/axiom clauses and method specifications.

To simplify the proofs and formalisation in this chapter, constructor preconditions explicitly mention fields and break information hiding. The front-end translation of MultiStar injects them automatically. For example, a user would write the specification of CCELL's constructor as

dynamic $\{\text{true}\}\_\{\text{Cc}(\text{val}:\text{v}, \text{cnt}:0)\}$

instead of

dynamic $\{\text{value} \hookrightarrow \_ * \text{count} \hookrightarrow \_\}\_\{\text{Cc}(\text{val}:\text{v}, \text{cnt}:0)\}$

In detail, the front-end facilitates this by:

1. Using jStar's **new** statement, whose specification is given by the triple $\{\text{true}\}\text{x} := \textbf{new}\ \text{C}\{\text{x} : \text{C}\}$. This allows us to omit the fields in the dynamic precondition[5].

2. Adding all fields (including ancestor ones) to the static precondition when checking Body verification, and consuming all fields of a parent

external approach is that clients can infer valid specifications for library code without re-verifying it. For example, knowing only the inferred specifications for methods of class SMUSICIAN is not enough for proving the triple $\{\text{x.SM}(\text{age}:\text{a}, \text{exm}:\text{e}, \text{pfm}:\text{p})\}\text{use\_student}(\text{x})\{\text{x.SM}(\text{age}:\text{a}+1, \text{exm}:\text{e}+4, \text{pfm}:\text{p})\}$ without looking at the implementation of the library routine use_student(st: STUDENT) whose dynamic specification is $\{\text{st.S}(\text{age}:\text{a}, \text{exm}:\text{e})\}\_\{\text{st.S}(\text{age}:\text{a}+1, \text{exm}:\text{e}+4)\}$.

[5]The dynamic specification of the constructor is used for object initialisation. $\text{x} := \textbf{new}\ \text{C}(\bar{\text{e}})$ abbreviates $\text{x} := \textbf{new}\ \text{C}; \text{x.C}(\bar{\text{e}})$ if x is not free in $\bar{\text{e}}$.

class and its ancestors right before the parent constructor is called. This is communicated to the back-end by emitting special instructions, and allows us to omit the fields in the static precondition.

The Dynamic dispatch proof obligation, which checks that the static and dynamic specifications are consistent with each other, is unaffected because fields are omitted in both static and dynamic preconditions. In languages where no fields are shared by ancestors, or constructors of common ancestors are called only once, the manipulation does not have to add ancestor fields to the static precondition and consume fields when a parent constructor is called. This is the approach jStar uses for Java verification. A front-end for C++ can use a similar technique because every ancestor constructor is called exactly once when virtual base classes are used [27].

3.2.2 Back-end

The MultiStar back-end extends jStar with support for export and axiom clauses, abstract classes and multiple inheritance. The latter two demand generalised proof obligation checking for methods.

Export and axiom clauses

The background theory used by the jStar theorem prover is encoded as a list of sequent rules. A sequent is of the form $P \mid Q \vdash R$, meaning $(P * Q) \Rightarrow (P * R)$. Each sequent rule has the form

$A \mid B \vdash C$
if
$D \mid E \vdash F$

If the prover is trying to prove a sequent that matches the rule's conclusion $A \mid B \vdash C$, it suffices to prove the premise sequent $D \mid E \vdash F$ where the appropriate substitutions due to matching have been made. A new proof goal is thus obtained, and the proof is complete when the goal is of the form $G \mid H \vdash$. For details the reader is referred to [23].

Exported information is written as sets of implications. Before verifying an export clause, the background theory is temporarily extended with the definitions of all predicates in its **where** part. For each definition of the form $w(x) = P$, the following two rules are generated:

$\mid w(x) \vdash$	$\mid \vdash w(x)$
if	**if**
$\mid P \vdash$	$\mid \vdash P$

After all exported implications in the clause have been checked, the definitions are removed from the background theory. After all export clauses have been verified, each exported implication $P \Rightarrow (Q_1 * \ldots * Q_n)$ is added to the background theory as a set of n rules, where rule $i \in 1..n$ has the form

$| P \vdash Q_i$
if
$Q_i \mid Q_1 * \ldots * Q_{i-1} * Q_{i+1} * \ldots * Q_n \vdash$

This rule form retains information about Q_i in its premise, and removal of Q_i from the goal sequent's right-hand side brings the proof closer to completion.

The background theory augmented with export information is then used to verify axiom clauses. The predicates in axiom clauses are written as implications. After all axiom clauses have been verified, an axiom implication $P \Rightarrow (Q_1 * \ldots * Q_n)$ written in class C is encoded as n rules, with rule $i \in 1..n$ of the form

$| P \vdash Q_i$
if
$Q_i \mid Q_1 * \ldots * Q_{i-1} * Q_{i+1} * \ldots * Q_n \vdash x <: C$

where x is the pattern variable substituted for **Current**.

The background theory augmented with export and axiom information is then used for method verification.

Method proof obligations

The back-end accommodates abstract classes and abstract methods in addition to shared multiple inheritance. An abstract method has no body and hence no static specification. The back-end takes this into account when expanding specification shorthands. After shorthand expansion, verification of method m in class C proceeds as follows (the formalisation contains details about the proof obligations):

- If m has a static specification and C can be instantiated (i.e. is non-abstract), then check Dynamic dispatch.
- If m has a body in C, then check Body verification.
- Always check Behavioural subtyping. This succeeds trivially if m is introduced in C: the set of dynamic specifications for m in C's parents is empty, and therefore all its elements are preserved by the new specification.
- If m has a static specification but no body in C, then check Inheritance.

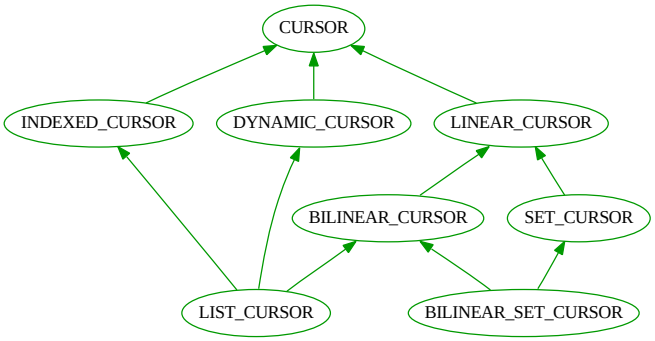


Figure 3.7: The core Gobo cursor hierarchy.

The treatment subsumes interface inheritance – interfaces are treated as abstract classes with only abstract methods and no fields.

3.3 Case study

The Gobo data structure library [29] is an open-source Eiffel library covering data structures and algorithms. It contains classic data structures such as lists, stacks and sets, and provides several implementations of each structure. The library is stable and a popular choice among Eiffel developers.

Data structures such as lists and sets can be traversed with iterators. The iterator (or *cursor*) hierarchy is characterised by relatively simple algorithms and extensive use of multiple inheritance, which makes it an ideal candidate for evaluating the novel aspects of our proof system and its implementation. The core classes are shown in Figure 3.7: a LINEAR_CURSOR can traverse a data structure forwards, a BILINEAR_CURSOR can traverse both forwards and backwards, an INDEXED_CURSOR offers random data structure access with an integer position or index, and a DYNAMIC_CURSOR can modify the data structure being traversed.

We successfully verified the core cursor hierarchy of Figure 3.7 with MultiStar. The overall effort for specification and verification was five person-days. Most of the time was spent on finding and revising specifications, since we did not modify the code. Table 3.1 shows the experimental results. The total time taken by MultiStar is reported, which includes translating Eiffel code, expanding specification shorthands and checking all proof obligations.

Since iterators rely on properties of the data structures (containers) they traverse, we annotated the container classes with the required specifications. Particularly interesting are the axiom clauses that iterators demand.

Class	LOC_1	LOC_2	Time(s)
BILINEAR_CURSOR	99	124	1.306
BILINEAR_SET_CURSOR	44	50	0.841
CURSOR	130	158	1.039
DYNAMIC_CURSOR	50	66	1.070
INDEXED_CURSOR	46	57	0.698
LINEAR_CURSOR	98	123	1.327
LIST_CURSOR	238	271	1.643
SET_CURSOR	38	44	0.738
8 classes	743	893	8.662

Table 3.1: Experimental results of the Gobo iterator case study. LOC_1 and LOC_2 denote the lines of code before and after specification respectively. MultiStar was executed on a 2.53 GHz Intel Core 2 Duo with 4 GB RAM.

Consider for example the simplified extract of DYNAMIC_CURSOR in Figure 3.8. The [G] denotes that DYNAMIC_CURSOR has a generic parameter G. Method *swap* takes another cursor referencing the same container, and additionally requires that there are data elements (items) at both cursor positions (the cursors are not 'off'). The Body verification proof of *swap* uses several properties of containers that can be expressed as axioms, including the following one:

$$
\begin{array}{l}
\forall\ \mathrm{r1,r2,iter1,iter2,i,c1,c2,c3}\ \cdot \\
[\mathrm{ItemAt}(\mathrm{res}{:}\,\mathrm{r1}, \mathrm{ref}{:}\,\mathrm{iter1}, \mathrm{iters}{:}\,\mathrm{i}, \mathrm{content}{:}\,\mathrm{c1})\ *\\
\ \mathrm{ItemAt}(\mathrm{res}{:}\,\mathrm{r2}, \mathrm{ref}{:}\,\mathrm{iter2}, \mathrm{iters}{:}\,\mathrm{i}, \mathrm{content}{:}\,\mathrm{c1})\ *\\
\ \mathrm{Replaced}(\mathrm{ref}{:}\,\mathrm{iter1}, \mathrm{value}{:}\,\mathrm{r2}, \mathrm{newcontent}{:}\,\mathrm{c2}, \mathrm{oldcontent}{:}\,\mathrm{c1}, \mathrm{iters}{:}\,\mathrm{i})\ *\\
\ \mathrm{Replaced}(\mathrm{ref}{:}\,\mathrm{iter2}, \mathrm{value}{:}\,\mathrm{r1}, \mathrm{newcontent}{:}\,\mathrm{c3}, \mathrm{oldcontent}{:}\,\mathrm{c2}, \mathrm{iters}{:}\,\mathrm{i})] \\
\Rightarrow \\
\mathrm{Swapped}(\mathrm{ref1}{:}\,\mathrm{iter1}, \mathrm{ref2}{:}\,\mathrm{iter2}, \mathrm{iters}{:}\,\mathrm{i}, \mathrm{oldcontent}{:}\,\mathrm{c1}, \mathrm{newcontent}{:}\,\mathrm{c3})
\end{array}
$$

This invariant property relates the ItemAt, Replaced and Swapped abstractions. Informally, it states that if a data structure has items r1 and r2 at iterators iter1 and iter2, and we replace the item at iter1 with r2 and the item at iter2 with r1, then the resulting data structure has the same contents as the original one except that the items at iter1 and iter2 have been swapped. The example involves neither multiple inheritance nor splitting of superclass and subclass state, and illustrates the generality and expressive power of axiom clauses.

The complete specifications and code of the case study are included in the download [28].

```
abstract class DYNAMIC_CURSOR [G] inherit CURSOR [G]
feature
    introduce abstract replace(v: G) dynamic
    {Cursor(ds: d) ∗ d.DS(content: c1, iters: i) ∗
      d.IsOff(res: False, ref: Current, iters: i, content: c1)}_
    {Cursor(ds: d) ∗ d.DS(content: c2, iters: i) ∗
     d.Replaced(ref: Current, value: v, newcontent: c2, oldcontent: c1, iters: i)}

    inherit item(): G static
    {Cursor(ds: d) ∗ d.DS(content: c, iters: i) ∗
      d.IsOff(res: False, ref: Current, iters: i, content: c)}_
    {Cursor(ds: d) ∗ d.DS(content: c, iters: i) ∗
      d.ItemAt(res: Result, ref: Current, iters: i, content: c)}

    introduce swap(other: DYNAMIC_CURSOR [G]) static
    {Cursor(ds: d) ∗ d.DS(content: c1, iters: i) ∗ other.Cursor(ds: d) ∗
      d.IsOff(res: False, ref: Current, iters: i, content: c1) ∗
      d.IsOff(res: False, ref: other, iters: i, content: c1)}_
    {Cursor(ds: d) ∗ d.DS(content: c2, iters: i) ∗ other.Cursor(ds: d) ∗
     d.Swapped(ref1: Current, ref2: other, iters: i, oldcontent: c1, newcontent: c2)}
    do
        v: G. w: G.
        v := item(); w := other.item();
        replace(w); replace(v)
    end
end
```

Figure 3.8: A simplified extract of DYNAMIC_CURSOR

3.4 Formalisation

This section contains a formal presentation of the programming language and its proof system. It extends the material of chapter 2 with abstract classes, multiple inheritance, and export and axiom specifications. To keep things simple, the presentation concentrates on the new extensions.

3.4.1 Language syntax

The grammar of the extended kernel language appears in Figure 3.9. New and changed productions are highlighted: abstract classes, multiple parents, export and axiom clauses, and abstract methods are now supported. The letter p ranges over apf names, w over auxiliary predicate names, and a over axiom names.

We assume the absence of clashes when names are introduced. This guarantees that axioms, methods and fields with the same name in parent classes stem from common ancestors.

The shared semantics of multiple inheritance is used, which is popular in Eiffel [26] and known as inheritance with *virtual base classes* in C++ [27]. Common ancestor fields are shared, and method overriding overrides all ancestor versions. To avoid ambiguity, a class can inherit a method only if its body (if there is one) is the same along all inheritance paths. Direct method calls can encode language mechanisms which allow a particular ancestor implementation to be chosen, so no generality is lost.

3.4.2 Operational semantics

The shared semantics of multiple inheritance ensures that 1) only dynamic type information is needed at runtime (in contrast to what 'select' clauses of Eiffel's replicated inheritance demand), and 2) the usual semantics of casts can be adopted (in contrast to replicated inheritance in C++, where casting can change pointer values [27]).

The operational semantics is therefore similar to Parkinson's semantics of Java [57]. Configurations contain a stack, a heap and a sequence of statements under execution. The stack maps variables to values which include object ids. The heap maps object ids to records containing a dynamic type G and field-value mappings.

L ::= Ab **class** G **inherit** $\overline{\text{H}}$ **define** $\overline{\text{D}}$ **export** $\overline{\text{E}}$ **axiom** $\overline{\text{A}}$ **feature** $\overline{\text{M}}$ $\overline{\text{F}}$ **end**	
Ab ::= **abstract** | ϵ	
D ::= x.$\text{p}_G(\overline{\text{t:y}})$ **as** P	*Define clause*
E ::= P **where** $\{\overline{\text{W}}\}$	*Export clause*
W ::= w($\overline{\text{x}}$) = P	*Where clause*
A ::= a: P	*Axiom clause*
M ::= **introduce** m(Args) Rt Sd Ss B	*Method declaration*
| **override** m(Args) Rt Sd Ss B	
| **inherit** m(Args) Rt Sd Ss	
| **introduce abstract** m(Args) Rt Sd	
| **inherit abstract** m(Args) Rt Sd	
F ::= f: Type	*Field declaration*
Sd ::= **dynamic** Spec	*Dynamic specification*
Ss ::= **static** Spec	*Static specification*
Spec ::= {P}_{Q} | {P}_{Q} **also** Spec	*Specification*
B ::= **do** s **end**	*Method body*
s ::= **skip**	*Statement*
| x := e	*Assignment*
| s ; s	*Sequential compostition*
| **if** e **then** s **else** s **end**	*Conditional*
| **while** e **do** s **end**	*Loop*
| x: Type. s	*Local variable block*
| x := **new** G	*Object allocation*
| x := y.f	*Field lookup*
| x.f := e	*Field update*
| x := y.m($\overline{\text{e}}$) | y.m($\overline{\text{e}}$)	*Dynamically dispatched call*
| x := y.G::m($\overline{\text{e}}$) | y.G::m($\overline{\text{e}}$)	*Direct method call*
e ::= x | e + e | e = e | **Void** | 0 | 1 | 2 ...	*Expression*
Type ::= INT | BOOL | G	
Args ::= $\overline{\text{x: Type}}$	*Formal arguments*
Rt ::= ϵ | : Type	*Return type*

Figure 3.9: The grammar of the extended kernel language.

3.4.3 Logic syntax and semantics

The assertion language is extended with auxiliary predicates. These predicates are defined in the **where** part of export clauses.

$$
\begin{array}{lll}
P, Q, S, T, \Delta ::= \forall x{\cdot}P \mid P \Rightarrow Q \mid \mathrm{false} \mid e = e' \mid x : G \mid x <: G \mid x.f \hookrightarrow e \mid P * Q & & \\
\quad \mid x.p(\overline{t{:}\,e}) & & \textit{Apf predicate} \\
\quad \mid x.p_G(\overline{t{:}\,e}) & & \textit{Apf entry} \\
\quad \mid w(\overline{x}) & & \textit{Auxiliary predicate}
\end{array}
$$

3.4.4 Specification refinement

The formalisation of specification refinement of Section 2.3 remains valid here. However, the assumptions Δ now contain the standard apf assumptions of a class as well as export and axiom information of all other classes.

3.4.5 The specification environment

The specification environment Γ is extended to map axiom names to their associated predicates for all classes in a program:

$$
\begin{array}{lll}
\Gamma ::= G.a \mapsto P & & \textit{Axiom specification} \\
\quad \mid G.m \mapsto (\overline{x}, \{P\}\_\{Q\}) & & \textit{Method dynamic specification} \\
\quad \mid G{::}m \mapsto (\overline{x}, \{S\}\_\{T\}) & & \textit{Method static specification} \\
\quad \mid \overline{\Gamma} & &
\end{array}
$$

For well-formed programs, Γ is still guaranteed to be a partial function, and we write $\Gamma(G.a) = P$ for $G.a \mapsto P \in \Gamma$.

3.4.6 Export information verification

A class can make information about itself available to other classes in an export clause. Export clauses are frequently used to specify relationships between apfs or their entries, and to expose apf entry definitions. Information can be hidden in predicates defined after the keyword **where**: the definitions are not exported, so other classes must treat these predicates abstractly.

Export information must be verified since other classes use it for reasoning. Under the predicate definitions following **where**, the assumptions about a class must imply exported information. This is captured by the following proof rule:

$$
\frac{[\Delta \wedge (\forall \overline{x_1} \cdot w_1(\overline{x_1}) \Leftrightarrow Q_1) \wedge \ldots \wedge (\forall \overline{x_n} \cdot w_n(\overline{x_n}) \Leftrightarrow Q_n)] \Rightarrow P}{\Delta \vdash_e P \ \textbf{where} \ \{w_1(\overline{x_1}) = Q_1; \ldots; w_n(\overline{x_n}) = Q_n\}}
$$

Since the assumptions about a class do not include assumptions about the exact dynamic type of **Current**, export information can be used to verify axioms and methods of other classes in a program.

3.4.7 Axiom verification

Information about a class and all its subclasses can be made available in an axiom clause. This knowledge can be used later to verify methods. To simplify the treatment, we require that a class explicitly lists all axiom clauses applicable to it (in MultiStar and the examples, specification shorthands achieve this).

In the rule for axiom verification, the assumptions Δ include information about class G and export information from all other classes. A subclass must preserve all axioms of its parents and may refine the predicate associated with an axiom name (the Parent consistency [P.c.] obligation). A non-abstract class must also show that the predicate holds for its direct instances (the Implication [Imp.] obligation).

$$\frac{\begin{array}{ll}\forall \mathrm{i} \in \mathrm{I} \cdot\ \Gamma(\mathrm{H_i.a}) = \mathrm{Q_i} \ \wedge\ \forall \mathrm{j} \in (1..\mathrm{n} \setminus \mathrm{I}) \cdot\ \mathrm{H_j.a} \notin \mathit{dom}(\Gamma) & \\ (\Delta \wedge \mathrm{P}) \Rightarrow \bigwedge_{\mathrm{i} \in \mathrm{I}} \mathrm{Q_i} & \text{[P.c.]} \\ \mathrm{Ab} \neq \epsilon \ \vee\ (\Delta \wedge \textbf{Current} : \mathrm{G}) \Rightarrow \mathrm{P} & \text{[Imp.]}\end{array}}{\Delta; \Gamma \vdash_a \text{a: P in Ab G parents } \mathrm{H_1} \ldots \mathrm{H_n}}$$

3.4.8 Statement verification

The assumptions Δ used to verify statements contain information about the enclosing class as well as export and axiom information from all other classes. The rules remain the same as those in Section 2.5.

3.4.9 Method verification

The rules for method verification of Section 2.6 are extended here to deal with multiple inheritance and abstract classes and methods. As for statement verification, the assumptions Δ used to verify method definitions contain information about the method's enclosing class as well as export and axiom information of all other classes.

A newly introduced method's static and dynamic specifications must be consistent if the class is not abstract:

$$\frac{\begin{array}{l} \mathrm{B} = \textbf{do}\ \mathrm{s}\ \textbf{end} \\ \mathrm{Sd} = \textbf{dynamic}\ \{\mathrm{P_G}\}\text{-}\{\mathrm{Q_G}\} \\ \mathrm{Ss} = \textbf{static}\ \{\mathrm{S_G}\}\text{-}\{\mathrm{T_G}\} \\ \mathrm{Ab}{\neq}\epsilon\ \vee\ \Delta \vdash \{\mathrm{S_G}\}\text{-}\{\mathrm{T_G}\} \overset{\textbf{Current}:\mathrm{G}}{\Longrightarrow} \{\mathrm{P_G}\}\text{-}\{\mathrm{Q_G}\} \qquad [\text{D.d.}] \\ \Delta;\Gamma \vdash_s \{\mathrm{S_G}\}\mathrm{s}\{\mathrm{T_G}\} \qquad [\text{B.v.}] \end{array}}{\Delta;\Gamma \vdash_m \textbf{introduce}\ \mathrm{m(Args)\ Rt\ Sd\ Ss\ B\ in\ Ab\ G\ parents}\ \overline{\mathrm{H}}}$$

An abstract method can be introduced without any proof obligations, since there is only a dynamic specification and no method body.

$$\frac{}{\Delta;\Gamma \vdash_m \textbf{introduce abstract}\ \mathrm{m(Args)\ Rt\ Sd\ in\ Ab\ G\ parents}\ \overline{\mathrm{H}}}$$

The next rule is used whenever an abstract method is implemented or a method body is redefined. The $\mathrm{H_1} \ldots \mathrm{H_n}$ are the immediate superclasses of G.

$$\frac{\begin{array}{l} \forall \mathrm{i} \in \mathrm{I}\cdot\ \Gamma(\mathrm{H_i.m}) = (\overline{\mathrm{x}},\{\mathrm{P_{H_i}}\}\text{-}\{\mathrm{Q_{H_i}}\}) \\ \forall \mathrm{j} \in (1..\mathrm{n} \setminus \mathrm{I})\cdot\ \mathrm{H_j.m} \notin \mathit{dom}(\Gamma) \\ \mathrm{B} = \textbf{do}\ \mathrm{s}\ \textbf{end} \\ \mathrm{Sd} = \textbf{dynamic}\ \{\mathrm{P_G}\}\text{-}\{\mathrm{Q_G}\} \\ \mathrm{Ss} = \textbf{static}\ \{\mathrm{S_G}\}\text{-}\{\mathrm{T_G}\} \\ \Delta \vdash \{\mathrm{P_G}\}\text{-}\{\mathrm{Q_G}\} \Longrightarrow (\textbf{also}_{\mathrm{i}\in\mathrm{I}}\ \{\mathrm{P_{H_i}}\}\text{-}\{\mathrm{Q_{H_i}}\}) \qquad [\text{B.s.}] \\ \mathrm{Ab}{\neq}\epsilon\ \vee\ \Delta \vdash \{\mathrm{S_G}\}\text{-}\{\mathrm{T_G}\} \overset{\textbf{Current}:\mathrm{G}}{\Longrightarrow} \{\mathrm{P_G}\}\text{-}\{\mathrm{Q_G}\} \qquad [\text{D.d.}] \\ \Delta;\Gamma \vdash_s \{\mathrm{S_G}\}\mathrm{s}\{\mathrm{T_G}\} \qquad [\text{B.v.}] \end{array}}{\Delta;\Gamma \vdash_m \textbf{override}\ \mathrm{m(Args)\ Rt\ Sd\ Ss\ B\ in\ Ab\ G\ parents}\ \mathrm{H_1} \ldots \mathrm{H_n}}$$

When a non-abstract method is inherited, its static specification must follow from those in parents. The Inheritance [Inh.] obligation enforces it.

$$\frac{\begin{array}{l} \forall \mathrm{i} \in \mathrm{I}\cdot\ \Gamma(\mathrm{H_i.m}) = (\overline{\mathrm{x}},\{\mathrm{P_{H_i}}\}\text{-}\{\mathrm{Q_{H_i}}\}) \\ \forall \mathrm{k} \in (1..\mathrm{n} \setminus \mathrm{I})\cdot\ \mathrm{H_k.m} \notin \mathit{dom}(\Gamma) \\ \forall \mathrm{j} \in \mathrm{J}\cdot\ \Gamma(\mathrm{H_j{::}m}) = (\overline{\mathrm{x}},\{\mathrm{S_{H_j}}\}\text{-}\{\mathrm{T_{H_j}}\}) \\ \forall \mathrm{l} \in (1..\mathrm{n} \setminus \mathrm{J})\cdot\ \mathrm{H_l{::}m} \notin \mathit{dom}(\Gamma) \\ \mathrm{Sd} = \textbf{dynamic}\ \{\mathrm{P_G}\}\text{-}\{\mathrm{Q_G}\} \\ \mathrm{Ss} = \textbf{static}\ \{\mathrm{S_G}\}\text{-}\{\mathrm{T_G}\} \\ \Delta \vdash \{\mathrm{P_G}\}\text{-}\{\mathrm{Q_G}\} \Longrightarrow (\textbf{also}_{\mathrm{i}\in\mathrm{I}}\ \{\mathrm{P_{H_i}}\}\text{-}\{\mathrm{Q_{H_i}}\}) \qquad [\text{B.s.}] \\ \Delta \vdash (\textbf{also}_{\mathrm{j}\in\mathrm{J}}\ \{\mathrm{S_{H_j}}\}\text{-}\{\mathrm{T_{H_j}}\}) \Longrightarrow \{\mathrm{S_G}\}\text{-}\{\mathrm{T_G}\} \qquad [\text{Inh.}] \\ \mathrm{Ab}{\neq}\epsilon\ \vee\ \Delta \vdash \{\mathrm{S_G}\}\text{-}\{\mathrm{T_G}\} \overset{\textbf{Current}:\mathrm{G}}{\Longrightarrow} \{\mathrm{P_G}\}\text{-}\{\mathrm{Q_G}\} \qquad [\text{D.d.}] \end{array}}{\Delta;\Gamma \vdash_m \textbf{inherit}\ \mathrm{m(Args)\ Rt\ Sd\ Ss\ in\ Ab\ G\ parents}\ \mathrm{H_1} \ldots \mathrm{H_n}}$$

The next rule applies whenever an abstract method is inherited or a non-abstract method is inherited and made abstract. Such a method has no static specification, so only the consistency of its dynamic specification w.r.t. those in parent classes is required with the Behavioural subtyping proof obligation.

$$\frac{\begin{array}{l}\forall \mathrm{i} \in \mathrm{I} \cdot\ \Gamma(\mathrm{H_i.m}) = (\overline{\mathrm{x}}, \{\mathrm{P_{Hi}}\}\_\{\mathrm{Q_{Hi}}\}) \\ \forall \mathrm{j} \in (1..\mathrm{n} \setminus \mathrm{I}) \cdot\ \mathrm{H_j.m} \notin \mathit{dom}(\Gamma) \\ \mathrm{Sd} = \textbf{dynamic}\ \{\mathrm{P_G}\}\_\{\mathrm{Q_G}\} \\ \Delta \vdash \{\mathrm{P_G}\}\_\{\mathrm{Q_G}\} \Longrightarrow (\textbf{also}_{\mathrm{i}\in\mathrm{I}}\ \{\mathrm{P_{Hi}}\}\_\{\mathrm{Q_{Hi}}\}) \qquad \text{[B.s.]}\end{array}}{\Delta; \Gamma \vdash_m \textbf{inherit abstract}\ \mathrm{m(Args)\ Rt\ Sd\ in\ Ab\ G\ parents\ H_1 \ldots H_n}}$$

Several intuitive relationships hold between the rules. For example, introducing a non-abstract method is the same as overriding the version of parent classes $\mathrm{I} = \emptyset$. Likewise, introducing an abstract method is the same as inheriting it from the parent set $\mathrm{I} = \emptyset$. Section 3.2.2 gives yet another perspective the rationale behind the proof obligations of the various rules.

3.4.10 Class and program verification

For class verification, different assumptions are used to verify the various class sections. The formula Δ_{APF} contains class-specific information and is used to verify export clauses. The assumptions Δ_E contain export information from all classes, and are used together with Δ_{APF} to verify axioms. The formula Δ_A contains axiom information of all classes, and is used with Δ_{APF} and Δ_E in method definition verification.

$$\frac{\begin{array}{l}\forall \mathrm{E_i} \in \overline{\mathrm{E}} \cdot \Delta_{APF} \vdash_e \mathrm{E_i} \\ \forall \mathrm{A_i} \in \overline{\mathrm{A}} \cdot (\Delta_{APF} \wedge \Delta_E); \Gamma \vdash_a \mathrm{A_i\ in\ Ab\ G\ parents\ } \overline{\mathrm{H}} \\ \forall \mathrm{M_i} \in \overline{\mathrm{M}} \cdot (\Delta_{APF} \wedge \Delta_E \wedge \Delta_A); \Gamma \vdash_m \mathrm{M_i\ in\ Ab\ G\ parents\ } \overline{\mathrm{H}}\end{array}}{\begin{array}{l}\Delta_{APF}, \Delta_E, \Delta_A; \Gamma \vdash_c \\ \mathrm{Ab}\ \textbf{class}\ \mathrm{G}\ \textbf{inherit}\ \overline{\mathrm{H}}\ \textbf{define}\ \overline{\mathrm{D}}\ \textbf{export}\ \overline{\mathrm{E}}\ \textbf{axiom}\ \overline{\mathrm{A}}\ \textbf{feature}\ \overline{\mathrm{M}}\ \overline{\mathrm{F}}\ \textbf{end}\end{array}}$$

Finally, here is the rule for program verification:

$$\frac{\begin{array}{l}\forall \mathrm{i} \in 1..\mathrm{n} \cdot\ \mathrm{L_i} = \ldots \textbf{class}\ \mathrm{G_i} \ldots \textbf{export}\ \overline{\mathrm{E_i}}\ \textbf{axiom}\ \overline{\mathrm{A_i}}\ \textbf{feature} \ldots \textbf{end} \\ \Delta_E = \bigwedge_{\mathrm{i}\in 1..\mathrm{n}} \bigwedge_{\mathrm{E_{i_k}} \in \overline{\mathrm{E_i}}} \mathit{exportinfo}(\mathrm{E_{i_k}}) \\ \Delta_A = \bigwedge_{\mathrm{i}\in 1..\mathrm{n}} \bigwedge_{\mathrm{A_{i_k}} \in \overline{\mathrm{A_i}}} \mathit{axiominfo}(\mathrm{G_i}, \mathrm{A_{i_k}}) \\ \Gamma = \mathit{specs}(\mathrm{L_1} \ldots \mathrm{L_n}) \\ \forall \mathrm{i} \in 1..\mathrm{n} \cdot\ \mathit{apf}(\mathrm{L_i}),\ \Delta_E, \Delta_A; \Gamma \vdash_c \mathrm{L_i} \\ \Delta_E \wedge \Delta_A; \Gamma \vdash_s \{\mathrm{true}\}\mathrm{s}\{\mathrm{true}\}\end{array}}{\vdash_p \mathrm{L_1} \ldots \mathrm{L_n\ s}}$$

exportinfo(P **where** ...) $\stackrel{\text{def}}{=}$ P
axiominfo(G, a: P) $\stackrel{\text{def}}{=}$ $\forall$x <: G · P[x/**Current**], where x is fresh.

Predicate definitions following the **where** keyword are hidden by *exportinfo*, and the definition of *axiominfo* reflects the fact that subclasses preserve axioms.

The function *apf* translates the abstract predicate family definitions of a class into its standard apf assumptions in the same way as before.

MultiStar and the examples assume that every class implicitly exports tag reduction information. In other words, for every entry (x.p_G(Y) **as** P) in the **define** section of a class G, ($\forall$x <: G · *TR*(p,x,Y) **where** {}) is implicitly exported.

Theorem. The program verification rule is sound. (The proof, sketched in Appendix A, depends on the layered assumption structure of export and axiom clauses that avoids circularity in reasoning[6].)

3.5 Conclusions and related work

The presented proof system supports two complementary mechanisms that can express relationships between abstractions in the logic. Such relationships are pervasive in OO programs, and facilitate flexible client reasoning, access control, specification inference, and constraints on the implementation of abstractions. Moreover, the system offers a sound way to verify various forms and uses of shared multiple inheritance. By virtue of extending Parkinson and Bierman's system, the examples in [58] illustrate that it can also deal with behaviour extension, restriction and modification, as well as representation replacement in subclasses. It is modular and every method body is verified only once. MultiStar implements these features in an automatic tool that, as the Gobo case study shows, holds good promise for verifying real-world software.

We are not aware of any other proof system or tool that can verify our examples and case study. Nevertheless, there are many relationships with other work:

[6]The layered assumption structure does not rule out axioms that depend on other axioms. If we want an axiom Q in class C_2 that depends on axiom P in class C_1, we can write the axiom ($\forall$x <: C_1· P[x/**Current**]) $\Rightarrow$ Q in class C_2 instead, where x is a fresh variable. After axiom verification, method and statement verification can use ($\forall$y <: C_2· Q[y/**Current**]) where y is fresh. The proof system enables the specification and verification of invariants for aggregate structures involving multiple objects of different types.

Class invariants	Axiom clauses
Related to object consistency	No notion of object consistency
Hold at particular program points	Hold everywhere
Expressed i.t.o. fields and pure method calls	Expressed as logical predicates
Constrain operations	Constrain logical abstractions of data
Verification involves methods	Verification cannot involve methods

Table 3.2: The main differences between class invariants and axiom clauses.

Axiom clauses We do not know of any existing specification mechanisms that are closely related to axiom clauses, in the sense that they describe and enforce relationships between logical abstractions of data in a similar way for inheritance hierarchies.

Class invariants form the basis of several OO specification and verification approaches, including Spec# [3] and JML [40]. Two main flavours of class invariants exist: private invariants, as exemplified by the object invariants of Spec#, and public invariants [41], which include JML's derived invariants and the invariants of Jacobs and Piessens for describing relationships between inspector methods [33]. Class invariants, like axiom clauses, constrain subclasses. However, there are several important differences between them. Class invariants either define exactly when an object is consistent (private invariants), or describe abstract properties of consistent objects (public invariants). Axiom clauses have no notion of object consistency. Class invariants are expected to hold at particular points in a program and may be broken at others, according to the employed invariant protocol [25]. Axiom clauses are true invariants in the sense that they hold everywhere. The relationships they describe cannot be violated by assignment statements, and hence there are no problems with e.g. method callbacks [44, 39]. Class invariants are expressed in terms of fields (including model fields) and pure method calls. Public invariants constrain operations that must establish or preserve them. Private invariants constrain the pure methods they use and indirectly also other operations that depend on the private invariant. Axioms are expressed as logical predicates, and they constrain logical abstractions of data. The verification of private invariants involves the inspection of method bodies (for the proof obligations see e.g. [45]), and the verification of public invariants involves private invariants and the specifications and/or bodies of pure methods. Axiom clauses are verified prior to methods and do not depend on methods in any way. The main differences between class invariants and axiom clauses are summarised in Table 3.2.

Export clauses There is some correspondence between the class axioms of Kassios [37, 36] and apfs/export clauses. Class axioms can be used to axiomatise the specification and program attributes of a class. In a class implementation, class axioms typically describe abstract state (represented by specification attributes) as a function of the concrete implementation state (represented by program attributes). This corresponds loosely to the way apf entry definitions relate the concrete state to apf arguments. Furthermore, class axioms can describe consequences of a specification attribute such as a class invariant, which typically include framing properties and relationships between other specification attributes. This corresponds somewhat to export clauses that describe properties of apf arguments or relationships between them. The framework of class axioms does not include inheritance, so despite the similar names, it is appropriate to compare them with export and not axiom clauses. Class axioms can also define method specifications/implementations, which apfs and export clauses are incapable of.

In jStar [23], the modifier 'export' can be added to an apf entry definition to expose it to all other classes. Even though our export clauses are more general and flexible than this mechanism, MultiStar supports it as a useful shorthand.

The rules for lossless casting by Chin et al. [14] describe relationships between predicates that provide full and partial views of objects. A view predicate describes the contents of the fields of an object directly: a full view of object o provides full knowledge of all o's fields, while a partial view with respect to class C describes only values of fields introduced by C and its ancestors. View predicates and relationships between them are generated automatically. The relationships do not have to be verified and do not constrain subclasses.

Krishnaswami et al. use so-called 'static specifications' in [38] to specify relationships between abstract predicates. Although not presented in an OO context, these relationships must be satisfied by implementations and are thus related to our export clauses.

The lemma functions of VeriFast [34] record proofs of relationships between predicates. The relationships are then used in reasoning; the proof of the Composite pattern in [35] provides a good example. Lemma functions, like export clauses, do not constrain subclasses.

Multiple inheritance Surprisingly few systems exist for reasoning about multiple inheritance. The system in [46] also uses separation logic, but without abstraction mechanisms such as apfs. Most of the paper is devoted to elementary separation logic proof rules that also apply in a single-inheritance

context. Diamond inheritance is never treated, and the bodies of inherited methods are re-verified in subclasses.

The focus of [24] is on behavioural subtyping. It proposes to verify behavioural subtyping of methods lazily, i.e. only to the extent demanded by client code. Supplier code is then continually re-verified as a client's use of it grows.

The restricted form of interface inheritance is easily handled by our proof system: an interface is simply an abstract class with only abstract methods and no fields. Many verification tools for OO programs, including Spec# [3] and the JML toolset [9], provide support for specifying and verifying interface inheritance. Both Spec# and JML use pure expressions of the programming language for specification, and follow a class invariant-based approach to verification.

Chapter 4

Verifying executable contracts

Many conventional OO program specification approaches, including Eiffel [26], Spec# [3] and JML [40], use Boolean expressions of the programming language to specify routine preconditions and postconditions, class invariants, and other assertions which hold at particular points such as loop invariants. Explaining the operational meaning of such executable assertions to programmers is easy, as they already know the meaning of programming language expressions. Even if such specifications are not formally verified, their runtime checking and value for testing provide significant benefits. Their popularity in software development is therefore not surprising.

Verifying that executable specifications will always hold at runtime is not easy. Expressions used for assertions typically include calls to methods which depend on the heap and which may also cause side-effects such as heap mutation. Consider for example a class SLIST which implements a sorted list of integers. The postcondition of method insert(i) can be specified as has(i), meaning that the list has or contains an element i. Yet has(i) might allocate a new iterator, traverse the linked structure, and temporarily affect bookkeeping of active iterators. Translating expressions such as has(i) into logical formulae for static verification is hard. Moreover, programming language expressions might have preconditions and are generally not guaranteed to terminate. This further complicates matters for verification.

In contrast to executable specifications, separation logic predicates are not computations. They specify the effects of computations in terms of program states, i.e. the contents of the stack and heap, and not in terms of outcomes of further computations. Proof systems based on separation logic (such as the ones we have seen) are promising for OO verification and can

successfully verify common programming patterns such as the Visitor [23] and Composite [35]. Unfortunately, the semantics of separation logic is not yet widely known by OO programmers.

How separation logic specifications can be used to tame executable ones is the topic of this chapter. The guarantee is simple: if a program satisfies its separation logic specification and the executable assertions are verified with the separation logic ones, then all executable assertions are guaranteed to hold at runtime. We do not verify the program with respect to its executable assertions, we verify the program *and* its executable assertions with respect to a separation logic specification. For example, if a program containing class SLIST satisifies its separation logic specification, and the separation logic postcondition of insert(i) is Q, then we are guaranteed that the executable postcondition, has(i), will always hold at runtime if we can prove the *connecting implication* $Q \Rrightarrow has(i)$. A connecting implication connects the world of separation logic predicates with the one of Boolean programming language expressions. It is proved in this case by deriving the Hoare-style triple $\{Q\}v := has(i)\{Q \wedge v = True\}$, where v is a fresh variable. A novel notion called *relative purity* is embedded in connecting implications and embraces side-effects in expressions such as has(i). The logical framework depends on the non-faulting semantics of separation logic triples and its $*$-connective, as we shall see.

We believe that executable and separation logic specifications can be complementary in OO software development. The proposed formal framework accommodates both ordinary programmers and proof experts. Programmers can express their intentions in the form of executable assertions and use runtime checking to identify faults. While software designs are still evolving, it is probably also easier to change executable assertions than more elaborate specifications. We envisage that in a second phase, once the software has stabilised and many faults have been removed, proof experts would annotate the critical parts with separation logic for verification. At this point the executable assertions do not have to be discarded: our framework integrates the specification approaches and can verify whether the expectations recorded in executable assertions are fulfilled. Problems in the verification of executable assertions can indicate discrepancies between the two types of specifications, e.g. a misunderstanding by the separation logic specifier as to when a routine may be called. Executable and separation logic specifications can therefore complement each other even in verification.

This chapter makes several contributions in the area of OO specification and verification:

1. It describes a simple technique based on connecting implications for

verifying executable preconditions and postconditions. It gives executable assertions a semantics based on their semantics as expressions. The formalisation can even be applied to contracts for non-OO non-garbage collected languages such as C.

2. It presents simple techniques to verify class invariants. If an invariant is specified with a separation logic predicate, then properties of the invariant, such as the fact that it holds in all visible states [52], often follow as a consequence. The framework can help to devise flexible and sound class invariant protocols.

3. It illustrates the framework's applicability to model-based specifications [12], where model classes and model queries are used to strengthen contracts [63].

4. It shows how the novel notion of relative purity tolerates side-effects in executable assertions to a high degree.

The techniques are well-suited to separation logic proof tools. We implemented them in the MultiStar tool [65] and verified the examples automatically.

Chapter outline A discussion of background material follows in Section 4.1. Precondition and postcondition verification are the topics of Sections 4.2 and 4.3 respectively. Section 4.4 contains an exposition on class invariants. The verification techniques are then applied to model-based specifications in Section 4.5. Relative purity and predicate extraction are considered in Section 4.6. A discussion of the MultiStar implementation follows in Section 4.7. Finally, Section 4.8 concludes and mentions related work.

4.1 Background

This chapter presents the verification framework in an abstract setting, which can be instantiated with concrete languages and proof systems. The abstract setting is not committed to a particular separation logic, proof system or language type. It is even applicable to non-OO non-garbage-collected languages such as C. The presentation uses the OO language and proof system of Chapter 2 to illustrate the abstract ideas with concrete examples.

4.1.1 The abstract setting: triples and footprints

An abstract triple is of the form {P}s{Q}. The partial correctness meaning of a triple is as follows: statement s does not fault when executed in a state satisfying P, and if it terminates then the resulting state satisfies Q. Faulting occurs when s accesses unallocated memory. In OO terms, 'unallocated memory' means heap storage which is not necessarily present in the initial state[1] and not allocated by s before being accessed. O'Hearn [54] uses the term *footprint* of s to describe the minimal state from which s can be executed safely. So s when executed in a state satisfying P will not fault if P describes at least the footprint of s. The notion of footprint has been expanded since [59], yet this is of little concern here.

4.1.2 The concrete setting

The OO language and proof system of Chapter 2 are convenient for illustrating the abstract concepts. To improve readability, we drop empty argument lists in method declarations and calls, and adhere to the uniform access principle [26] where queries can be implemented by fields or result-returning methods. As usual in Hoare-style logics, predicates may use inductive data types and functions involving them. This chapter employs only sequences of integers, where α and β are sequence-valued variables. ϵ denotes the empty sequence, and [e] denotes the singleton sequence whose only element is e. The length of α is written $|\alpha|$, the i'th element of α is α_{i}, and $\alpha +\!\!+ \beta$ denotes the sequence obtained by appending α and β.

The Boolean expressions of the executable contract language can be more complex than the expressions of the kernel language of Chapter 2. In particular, feature calls are allowed in contract expressions. To facilitate formal reasoning about executable assertions, the function $(\!|\mathrm{E}|\!)_{\mathrm{v}}$ yields a kernel language statement which captures the meaning of the contract expression E in the variable v. Said otherwise, $(\!|\mathrm{E}|\!)_{\mathrm{v}}$ translates the sugared statement v := E into an appropriate kernel language statement. Formally, assume that v has the same static type as E and that t, t_1, ... t_{n} are fresh variables:

$$
\begin{array}{rcl}
(\!|\mathrm{x}|\!)_{\mathrm{v}} & \stackrel{\mathrm{def}}{=} & \mathrm{v} := \mathrm{x}, \text{ if x is a variable or a constant.} \\
(\!|\mathrm{e}_0.\mathrm{f}(\mathrm{e}_1,\ldots,\mathrm{e}_{\mathrm{n}})|\!)_{\mathrm{v}} & \stackrel{\mathrm{def}}{=} & (\!|\mathrm{e}_0|\!)_{\mathrm{t}_0};\ \ldots;\ (\!|\mathrm{e}_{\mathrm{n}}|\!)_{\mathrm{t}_{\mathrm{n}}};\ \mathrm{v} := \mathrm{t}_0.\mathrm{f}(\mathrm{t}_1,\ldots,\mathrm{t}_{\mathrm{n}}) \\
(\!|\mathbf{not}\ \mathrm{e}|\!)_{\mathrm{v}} & \stackrel{\mathrm{def}}{=} & (\!|\mathrm{e}|\!)_{\mathrm{t}};\ \mathrm{v} := \mathbf{not}\ \mathrm{t} \\
(\!|\mathrm{e}_1 \mathrm{op}\ \mathrm{e}_2|\!)_{\mathrm{v}} & \stackrel{\mathrm{def}}{=} & (\!|\mathrm{e}_1|\!)_{\mathrm{t}_1};\ (\!|\mathrm{e}_2|\!)_{\mathrm{t}_2};\ \mathrm{v} := \mathrm{t}_1\ \mathrm{op}\ \mathrm{t}_2, \text{ if op} \in \{\mathbf{and}, +, -, =, >=\}.
\end{array}
$$

[1] Separation logic systems do not rely on well-formedness of the heap.

The translation function fixes the semantics of contract language expressions. For example, if we want the contract language to use short-circuit evaluation of e_1 **and** e_2 instead, then its translation would involve a conditional statement. The definition above is sufficient for our purposes, and we use it in the examples that follow.

4.2 Precondition verification

Given a separation logic predicate P and an executable assertion B which should hold at the same program point, the question is whether B somehow follows from P. To this end we define the connecting implication $P \Rrightarrow B$, which informally means that P is sufficient to evaluate B to True:

$P \Rrightarrow B \stackrel{\text{def}}{=} \{P\}v := B\{P \wedge v = \text{True}\}$, where v is a fresh variable.

The intuition behind the definition is that in every state satisfying P, we can evaluate B into v without faulting, and if this computation terminates, then P will hold in the resulting state and v will have value True. The routine body relies on P, so it must be re-established by the evaluation of B. This enforces a notion of purity on B, i.e. it prevents B from performing certain kinds of side-effects while allowing others. Section 4.6.1 contains more about this.

In the concrete setting of the OO proof system, there is a sufficient condition[2] for $P \Rrightarrow B$:

$$\boxed{\text{If } \Delta;\Gamma \vdash_s \{P\}(\!|B|\!)_v\{P * v = \text{True}\}, \text{ then } P \Rrightarrow B \text{ under } \Delta \text{ and } \Gamma.}$$

In intuitionistic separation logic, which is used for garbage-collected OO languages such as our concrete one, $(P * e = e') \Leftrightarrow (P \wedge e = e')$. The $\Delta;\Gamma$ used to verify a B appearing in class C is the same $\Delta;\Gamma$ under which statements and methods of C are verified. In the examples of this chapter we omit explicit reference to Δ and Γ, since none of them uses the additional logical assumptions in Δ, and method specifications in Γ can be read directly from code listings.

Examples. Consider the method preconditions of class SLIST in Figure 4.1.

1. The executable precondition of the constructor *SLIST* and features *insert*, *has*, *count* and *is_empty* is True. For any separation logic predicate P it holds that $P \Rrightarrow \text{True}$:

[2]Whether or not this condition is also necessary depends on the completeness of the proof system.

class SLIST
define x.L_{SLIST}(l:α) **as** ...
axiom
L_sorted: $\forall\alpha\cdot$ L(l:α) $\Rightarrow$ [$\forall$i,j$\in$ (1..|α|)$\cdot$ i $<$ j $\Rightarrow$ $\alpha_{\mathrm{i}} \leq \alpha_{\mathrm{j}}$]
feature
SLIST
dynamic {True}_{L(l:ϵ)}
executable {True}_{is_empty}

insert(i: INT)
dynamic {L(l:α)}_{$\exists\alpha_{\mathrm{F}},\alpha_{\mathrm{S}}\cdot$ L(l:$\alpha_{\mathrm{F}} +\!\!+ [\mathrm{i}] +\!\!+ \alpha_{\mathrm{S}}$) $*$ $\alpha_{\mathrm{F}} +\!\!+ \alpha_{\mathrm{S}} = \alpha$}
executable {True}_{has(i) **and** count = **old**(count) + 1}

remove_first
dynamic {L(l:[e] $+\!\!+$ α)}_{L(l:α)}
executable {**not** is_empty}_{count = **old**(count) $-$ 1}

first: INT
dynamic {L(l:[e] $+\!\!+$ α)}_{L(l:[e] $+\!\!+$ α) $*$ **Result** = e}
executable {**not** is_empty}_{**not** is_empty}

has(i: INT): BOOL
dynamic {L(l:α)}_{L(l:α) $*$ **Result** = ($\exists j \in 1..|\alpha| \cdot \alpha_j$ = i)}
executable {True}_{True}

count: INT
dynamic {L(l:α)}_{L(l:α) $*$ **Result** = $|\alpha|$}
executable {True}_{True}

is_empty: BOOL
dynamic {L(l:α)}_{L(l:α) $*$ **Result** = ($\alpha = \epsilon$)}
executable {True}_{True}
invariant
count_non_negative: {L(l:α)} count >= 0
empty_definition: {L(l:α)} is_empty = (count = 0)
end

Figure 4.1: The interface of a sorted list class.

$$\dfrac{\dfrac{\dfrac{\dfrac{}{\Delta;\Gamma \vdash_s \{\mathrm{True} = \mathrm{True}\}\mathrm{v} := \mathrm{True}\{\mathrm{v} = \mathrm{True}\}}\ \text{Assigment axiom}}{\Delta;\Gamma \vdash_s \{\mathrm{True}\}\mathrm{v} := \mathrm{True}\{\mathrm{v} = \mathrm{True}\}}\ \text{Consequence}}{\Delta;\Gamma \vdash_s \{\mathrm{True} * \mathrm{P}\}\mathrm{v} := \mathrm{True}\{\mathrm{v} = \mathrm{True} * \mathrm{P}\}}\ \text{Frame rule}}{\Delta;\Gamma \vdash_s \{\mathrm{P}\}\mathrm{v} := \mathrm{True}\{\mathrm{P} * \mathrm{v} = \mathrm{True}\}}\ \text{Consequence}$$

The side-condition of the Frame rule is satisfied: $\mathit{modifies}(\mathrm{v} := \mathrm{True}) = \{\mathrm{v}\}$ and $\{\mathrm{v}\} \cap \mathrm{FV}(\mathrm{P}) = \emptyset$ (remember that v is fresh). The second application of Consequence used the commutativity of $*$ and the fact that $\mathrm{P} \Rightarrow (\mathrm{P} * \mathrm{True})$ in intuitionistic separation logic.

Instead of such detailed proofs, the rest of the chapter uses proof outlines where statements are interspersed between assertions. Explicitly mentioning the freshness of certain variables is omitted if it is clear from the context. The above proof looks as follows in outline form:

$\{\mathrm{P}\}$
    $\mathrm{v} := \mathrm{True}$
$\{\mathrm{P} * \mathrm{v} = \mathrm{True}\}$

2. The preconditions of *remove_first* and *first* are identical. Here is a proof of $\mathrm{L}(\mathrm{l}{:}\,[\mathrm{e}] +\!\!+ \alpha) \Rrightarrow$ **not** is_empty:

$\{\mathrm{L}(\mathrm{l}{:}\,[\mathrm{e}] +\!\!+ \alpha)\}$
    $\mathrm{t} :=$ is_empty
$\{\mathrm{L}(\mathrm{l}{:}\,[\mathrm{e}] +\!\!+ \alpha) * \mathrm{t} = ([e] +\!\!+ \alpha = \epsilon)\}$
$\{\mathrm{L}(\mathrm{l}{:}\,[\mathrm{e}] +\!\!+ \alpha) * \mathrm{t} = \mathrm{False}\}$
    $\mathrm{v} :=$ **not** t
$\{\mathrm{L}(\mathrm{l}{:}\,[\mathrm{e}] +\!\!+ \alpha) * \mathrm{t} = \mathrm{False} * \mathrm{v} = \neg\mathrm{t}\}$
$\{\mathrm{L}(\mathrm{l}{:}\,[\mathrm{e}] +\!\!+ \alpha) * \mathrm{v} = \mathrm{True}\}$

3. The executable precondition does not always follow from the separation logic one. Suppose that the executable precondition of *has* is not True but instead **not** is_empty. A proof attempt of $\mathrm{L}(\mathrm{l}{:}\,\alpha) \Rrightarrow$ (**not** is_empty) proceeds as follows:

$\{\mathrm{L}(\mathrm{l}{:}\,\alpha)\}$
    $\mathrm{t} :=$ is_empty
$\{\mathrm{L}(\mathrm{l}{:}\,\alpha) * \mathrm{t} = (\alpha = \epsilon)\}$
    $\mathrm{v} :=$ **not** t
$\{\mathrm{L}(\mathrm{l}{:}\,\alpha) * \mathrm{t} = (\alpha = \epsilon) * \mathrm{v} = \neg\mathrm{t}\}$
$\{\mathrm{L}(\mathrm{l}{:}\,\alpha) * \mathrm{v} = (\alpha \neq \epsilon)\}$

Since $\alpha \neq \epsilon$ is not a consequence, the proof cannot conclude with $L(l{:}\,\alpha) * v = True$. The separation logic specification is too weak to demonstrate the executable one.[3]

Such a clash of specifications has several potential causes. The executable assertion is maybe overly restrictive, or it captures important semantic properties of the domain that the separation logic one ignores. If the issue is not resolved, then there is of course no guarantee that the executable assertion will hold at runtime. □

The rest of the chapter contains more examples of the form P ⇛ B.

4.3 Postcondition verification

Executable postconditions for routines may contain so-called *old-expressions*. The old-expression **old**(E) denotes the value of expression E in the state right before the routine is executed, i.e. the state which satisfies the precondition.

Given two separation logic predicates P and Q and an executable assertion B which may contain old-expressions, the connecting implication P,Q ⇛ B informally means that any pre-state satisfying P and any post-state satisfying Q are sufficient to evaluate B to True:

Let **old**(E_1),...,**old**(E_n) be the list of old-expressions in B and let v, v_1,...,v_n be fresh variables.

$$P,Q \Rrightarrow B \stackrel{\text{def}}{=} \exists R\cdot \quad \{P\}(v_1 := E_1, \ldots, v_n := E_n)\{P * R\} \text{ and } \{Q * R\}v := B[v_1; \ldots; v_n]\{Q \wedge v = True\}$$

$$B[v_1;\ldots;v_n] \stackrel{\text{def}}{=} B[v_1/\textbf{old}(E_1),\ldots,v_n/\textbf{old}(E_n)]$$

Intuitively, R contains the result of evaluating (v_1:=E_1,...,v_n:=E_n) in the pre-state where P holds. Old-expression evaluation must re-establish P because the routine body relies on it. The existence of old-expression results depends on runtime checking, so R is not contained in Q and the body is not allowed to access state described by R. The body is verified only with respect to P and Q, which guarantees that it will establish Q upon termination and never touch R. So Q * R holds right after the body's execution. Evaluating $B[v_1;\ldots;v_n]$ in this state should yield True and re-establish Q, since clients

[3] In fact we have shown that (**not** is_empty) evaluates to False if $\alpha = \epsilon$, so the only way it can also evaluate to True is when it loops forever. A verified implementation of SLIST (and classes transitively used by it) for which (**not** is_empty) terminates when executed in some initial state satisfying $L(l{:}\,\alpha)$ comprises a counterexample for $L(l{:}\,\alpha) \Rrightarrow$ (**not** is_empty).

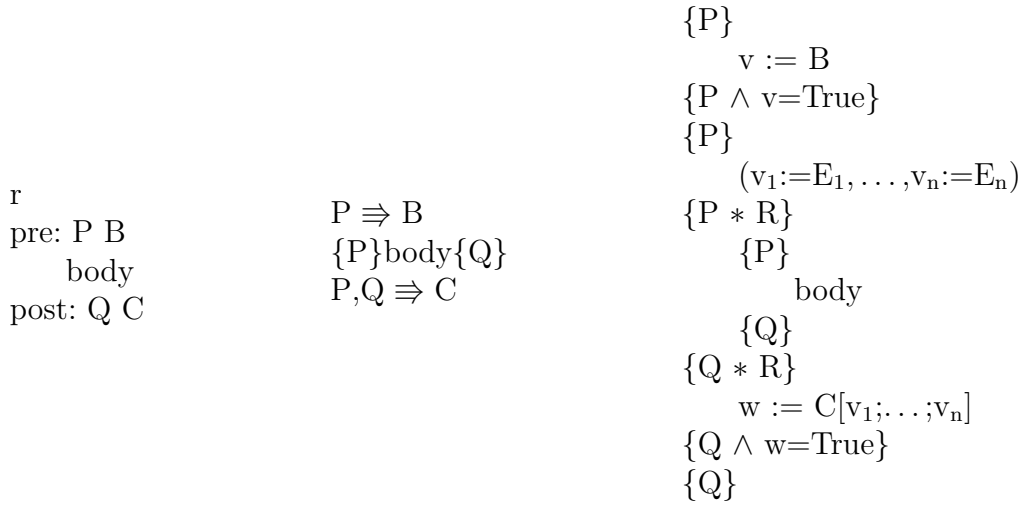


Figure 4.2: (a) Routine r and its specification. (b) Resulting proof obligations. (c) Proof obligations as triples in a proof outline.

depend on the fact that Q holds. Figure 4.2 summarises the proof obligations of a routine.

The predicate R will almost certainly have $v_1,\ldots,v_n$ as free variables, but should not have free variables which are modified by the routine body. This allows the transfer of R over the routine body with the Frame rule.

An assertion language's semantics will stipulate how to prove a triple of the form $\{P\}(v_1 := E_1, \ldots, v_n := E_n)\{Q\}$. For example, if old-expressions are evaluated in an unspecified order, then $\{P\}s\{Q\}$ must be proved for every permutation s of $(v_1{:=}E_1,\ldots,v_n{:=}E_n)$.

In our concrete OO setting, the assertion language specification states that old-expressions will be evaluated in an arbitrary order. The following condition is sufficient (but not necessary[4]) for concluding P,Q ⇛ B:

> If $\forall i \in (1..n)\cdot\ \Delta;\Gamma \vdash_s \{P\}(\!|E_i|\!)_{v_i}\{P * R_i\}$ with $\mathbf{Result} \notin FV(R_i)$
> and $\Delta;\Gamma \vdash_s \{Q * R_1 * \ldots * R_n\}v := B[v_1;\ldots;v_n]\{Q * v = True\}$
> then P,Q ⇛ B under Δ and Γ.

Once again, an assertion B appearing in class C is verified with the same $\Delta;\Gamma$ as the statements and methods of C.

[4]Suppose we take $(\exists e\cdot\ o.f \hookrightarrow e * even(e))$ for P and o.f++ for both E_1 and E_2. Since there exists no R_i such that $\Delta;\Gamma \vdash_s \{P\}(\!|E_i|\!)_{v_i}\{P * R_i\}$ for $i \in 1..2$, the rule cannot prove P,True ⇛ odd(**old**(E_1)+**old**(E_2)).

Relationships between P,Q $\Rrightarrow$ B and Q $\Rrightarrow$ B exist, which are convenient in proofs because many executable postconditions contain no old-expressions.

1. If P,Q $\Rrightarrow$ B then Q $\Rrightarrow$ B whenever B contains no old-expression.

2. If Q $\Rrightarrow$ B then P,Q $\Rrightarrow$ B for any P.

So P,Q $\Rrightarrow$ B iff Q $\Rrightarrow$ B whenever B contains no old-expression.

Examples. Consider class SLIST in Figure 4.1.

1. For *remove_first*, it is the case that $\mathrm{L}(\mathrm{l}\colon [\mathrm{e}] +\!\!+ \alpha)$,$\mathrm{L}(\mathrm{l}\colon \alpha) \Rrightarrow (\mathrm{count} = \mathbf{old}(\mathrm{count}) - 1)$. The executable postcondition contains one old-expression and $\mathrm{E}_1 = \mathrm{count}$. Here is the first part of the proof:

   $\{\mathrm{L}(\mathrm{l}\colon [\mathrm{e}] +\!\!+ \alpha)\}$
   $\quad \mathrm{t1} := \mathrm{count}$
   $\{\mathrm{L}(\mathrm{l}\colon [\mathrm{e}] +\!\!+ \alpha) * \mathrm{t1} = |[e] +\!\!+ \alpha|\}$

   Notice that $\mathrm{FV}(\mathrm{t1} = |[e] +\!\!+ \alpha|) = \{\mathrm{t1}, e, \alpha\}$. Using $(\mathrm{t1} = |[e] +\!\!+ \alpha|)$ for R_1 completes the proof:

   $\{\mathrm{L}(\mathrm{l}\colon \alpha) * \mathrm{t1} = |[e] +\!\!+ \alpha|\}$
   $\quad \mathrm{t2} := \mathrm{count}$
   $\{\mathrm{L}(\mathrm{l}\colon \alpha) * \mathrm{t2} = |\alpha| * \mathrm{t1} = |[e] +\!\!+ \alpha|\}$
   $\{\mathrm{L}(\mathrm{l}\colon \alpha) * \mathrm{t2} = |\alpha| * \mathrm{t1} = |\alpha| + 1\}$
   $\quad \mathrm{t3} := \mathrm{t1} - 1$
   $\{\mathrm{L}(\mathrm{l}\colon \alpha) * \mathrm{t2} = |\alpha| * \mathrm{t1} = |\alpha| + 1 * \mathrm{t3} = |\alpha|\}$
   $\{\mathrm{L}(\mathrm{l}\colon \alpha) * \mathrm{t2} = |\alpha| * \mathrm{t3} = |\alpha|\}$
   $\quad \mathrm{v} := \mathrm{t2} = \mathrm{t3}$
   $\{\mathrm{L}(\mathrm{l}\colon \alpha) * \mathrm{t2} = |\alpha| * \mathrm{t3} = |\alpha| * \mathrm{v} = (\mathrm{t2} = \mathrm{t3})\}$
   $\{\mathrm{L}(\mathrm{l}\colon \alpha) * \mathrm{v} = \mathrm{True}\}$

2. The executable postcondition of the constructor *SLIST* contains no old-expression, so it suffices to prove $\mathrm{L}(\mathrm{l}\colon \epsilon) \Rrightarrow$ is_empty:

   $\{\mathrm{L}(\mathrm{l}\colon \epsilon)\}$
   $\quad \mathrm{v} := \mathrm{is\_empty}$
   $\{\mathrm{L}(\mathrm{l}\colon \epsilon) * \mathrm{v} = (\epsilon = \epsilon)\}$
   $\{\mathrm{L}(\mathrm{l}\colon \epsilon) * \mathrm{v} = \mathrm{True}\}$

3. The executable postcondition of *first* contains no old-expression, so we only need $(\mathrm{L}(\mathrm{l}\colon [\mathrm{e}] +\!\!+ \alpha) * \mathbf{Result} = \mathrm{e}) \Rrightarrow \mathbf{not}$ is_empty. Example 2 in the previous section established the inner triple of the proof:

$\{\mathrm{L}(\mathrm{l}{:}\,[\mathrm{e}] +\!\!+ \alpha) * \mathbf{Result} = \mathrm{e}\}$
  $\{\mathrm{L}(\mathrm{l}{:}\,[\mathrm{e}] +\!\!+ \alpha)\}$
    $(\!|\mathbf{not}\ \mathrm{is\_empty}|\!)_{\mathrm{v}}$
  $\{\mathrm{L}(\mathrm{l}{:}\,[\mathrm{e}] +\!\!+ \alpha) * \mathrm{v} = \mathrm{True}\}$
$\{\mathrm{L}(\mathrm{l}{:}\,[\mathrm{e}] +\!\!+ \alpha) * \mathrm{v} = \mathrm{True} * \mathbf{Result} = \mathrm{e}\}$
$\{\mathrm{L}(\mathrm{l}{:}\,[\mathrm{e}] +\!\!+ \alpha) * \mathbf{Result} = \mathrm{e} * \mathrm{v} = \mathrm{True}\}$

The Frame rule is key to the proof.

4. Consider the postcondition of *insert*. It contains one old-expression, namely **old**(count):

$\{\mathrm{L}(\mathrm{l}{:}\,\alpha)\}$
  $\mathrm{t1} := \mathrm{count}$
$\{\mathrm{L}(\mathrm{l}{:}\,\alpha) * \mathrm{t1} = |\alpha|\}$

The second part of the proof consists of small pieces which are put together.

$\{\mathrm{L}(\mathrm{l}{:}\,\alpha_{\mathrm{F}} +\!\!+ [\mathrm{i}] +\!\!+ \alpha_{\mathrm{S}})\}$
  $\mathrm{t2} := \mathrm{has(i)}$
$\{\mathrm{L}(\mathrm{l}{:}\,\alpha_{\mathrm{F}} +\!\!+ [\mathrm{i}] +\!\!+ \alpha_{\mathrm{S}}) * \mathrm{t2} = (\exists j \in 1..|\alpha_{\mathrm{F}} +\!\!+ [\mathrm{i}] +\!\!+ \alpha_{\mathrm{S}}|\cdot(\alpha_{\mathrm{F}} +\!\!+ [\mathrm{i}] +\!\!+ \alpha_{\mathrm{S}})_j = \mathrm{i})\}$
$\{\mathrm{L}(\mathrm{l}{:}\,\alpha_{\mathrm{F}} +\!\!+ [\mathrm{i}] +\!\!+ \alpha_{\mathrm{S}}) * \mathrm{t2} = \mathrm{True}\}$

Applying Frame and Consequence to this triple yields Piece 1:

$\{\mathrm{L}(\mathrm{l}{:}\,\alpha_{\mathrm{F}} +\!\!+ [\mathrm{i}] +\!\!+ \alpha_{\mathrm{S}}) * \alpha_{\mathrm{F}} +\!\!+ \alpha_{\mathrm{S}} = \alpha * \mathrm{t1} = |\alpha|\}$
  $\mathrm{t2} := \mathrm{has(i)}$
$\{\mathrm{L}(\mathrm{l}{:}\,\alpha_{\mathrm{F}} +\!\!+ [\mathrm{i}] +\!\!+ \alpha_{\mathrm{S}}) * \alpha_{\mathrm{F}} +\!\!+ \alpha_{\mathrm{S}} = \alpha * \mathrm{t1} = |\alpha| * \mathrm{t2} = \mathrm{True}\}$

Next, we prove

$\{\mathrm{L}(\mathrm{l}{:}\,\alpha_{\mathrm{F}} +\!\!+ [\mathrm{i}] +\!\!+ \alpha_{\mathrm{S}}) * \alpha_{\mathrm{F}} +\!\!+ \alpha_{\mathrm{S}} = \alpha * \mathrm{t1} = |\alpha|\}$
  $\mathrm{t3} := \mathrm{count}$
$\{\mathrm{L}(\mathrm{l}{:}\,\alpha_{\mathrm{F}} +\!\!+ [\mathrm{i}] +\!\!+ \alpha_{\mathrm{S}}) * \mathrm{t3} = |\alpha_{\mathrm{F}} +\!\!+ [\mathrm{i}] +\!\!+ \alpha_{\mathrm{S}}| * \alpha_{\mathrm{F}} +\!\!+ \alpha_{\mathrm{S}} = \alpha * \mathrm{t1} = |\alpha|\}$
$\{\mathrm{L}(\mathrm{l}{:}\,\alpha_{\mathrm{F}} +\!\!+ [\mathrm{i}] +\!\!+ \alpha_{\mathrm{S}}) * \mathrm{t3} = |\alpha| + 1 * \alpha_{\mathrm{F}} +\!\!+ \alpha_{\mathrm{S}} = \alpha * \mathrm{t1} = |\alpha|\}$
  $\mathrm{t4} := \mathrm{t1} + 1$
$\{\mathrm{L}(\mathrm{l}{:}\,\alpha_{\mathrm{F}} +\!\!+ [\mathrm{i}] +\!\!+ \alpha_{\mathrm{S}}) * \mathrm{t3} = |\alpha|{+}1 * \alpha_{\mathrm{F}} +\!\!+ \alpha_{\mathrm{S}} = \alpha * \mathrm{t1} = |\alpha| * \mathrm{t4} = \mathrm{t1} + 1\}$
$\{\mathrm{L}(\mathrm{l}{:}\,\alpha_{\mathrm{F}} +\!\!+ [\mathrm{i}] +\!\!+ \alpha_{\mathrm{S}}) * \mathrm{t3} = |\alpha| + 1 * \alpha_{\mathrm{F}} +\!\!+ \alpha_{\mathrm{S}} = \alpha * \mathrm{t4} = |\alpha| + 1\}$
  $\mathrm{t5} := \mathrm{t3} = \mathrm{t4}$
$\{\mathrm{L}(\mathrm{l}{:}\,\alpha_{\mathrm{F}} +\!\!+ [\mathrm{i}] +\!\!+ \alpha_{\mathrm{S}}) * \mathrm{t3} = |\alpha|{+}1 * \alpha_{\mathrm{F}} +\!\!+ \alpha_{\mathrm{S}} = \alpha * \mathrm{t4} = |\alpha|{+}1 * \mathrm{t5} = (\mathrm{t3}{=}\mathrm{t4})\}$
$\{\mathrm{L}(\mathrm{l}{:}\,\alpha_{\mathrm{F}} +\!\!+ [\mathrm{i}] +\!\!+ \alpha_{\mathrm{S}}) * \alpha_{\mathrm{F}} +\!\!+ \alpha_{\mathrm{S}} = \alpha * \mathrm{t5} = \mathrm{True}\}$

Applying Frame establishes Piece 2:

$\{\mathrm{L}(\mathrm{l}{:}\,\alpha_\mathrm{F} +\!\!+ [\mathrm{i}] +\!\!+ \alpha_\mathrm{S}) * \alpha_\mathrm{F} +\!\!+ \alpha_\mathrm{S} = \alpha * \mathrm{t1} = |\alpha| * \mathrm{t2} = \mathrm{True}\}$
t3 := count
t4 := t1 + 1
t5 := t3 = t4
$\{\mathrm{L}(\mathrm{l}{:}\,\alpha_\mathrm{F} +\!\!+ [\mathrm{i}] +\!\!+ \alpha_\mathrm{S}) * \alpha_\mathrm{F} +\!\!+ \alpha_\mathrm{S} = \alpha * \mathrm{t5} = \mathrm{True} * \mathrm{t2} = \mathrm{True}\}$

Here is Piece 3:

$\{\mathrm{L}(\mathrm{l}{:}\,\alpha_\mathrm{F} +\!\!+ [\mathrm{i}] +\!\!+ \alpha_\mathrm{S}) * \alpha_\mathrm{F} +\!\!+ \alpha_\mathrm{S} = \alpha * \mathrm{t5} = \mathrm{True} * \mathrm{t2} = \mathrm{True}\}$
v := t2 **and** t5
$\{\mathrm{L}(\mathrm{l}{:}\,\alpha_\mathrm{F} +\!\!+ [\mathrm{i}] +\!\!+ \alpha_\mathrm{S}) * \alpha_\mathrm{F} +\!\!+ \alpha_\mathrm{S} = \alpha * \mathrm{t5} = \mathrm{True} * \mathrm{t2} = \mathrm{True} * \mathrm{v} = (\mathrm{t2}\ \mathrm{and}\ \mathrm{t5})\}$
$\{\mathrm{L}(\mathrm{l}{:}\,\alpha_\mathrm{F} +\!\!+ [\mathrm{i}] +\!\!+ \alpha_\mathrm{S}) * \alpha_\mathrm{F} +\!\!+ \alpha_\mathrm{S} = \alpha * \mathrm{v} = \mathrm{True}\}$

The proof is completed by putting the three pieces together, eliminating α_F and α_S with the Auxiliary variable elimination rule, and applying Consequence to move (t1 = $|\alpha|$) and (v = True) out from under the quantifiers to the top level in the precondition and postcondition respectively. □

4.4 Class invariant verification

Consider an executable assertion B which should hold at program point *pp*. If the separation logic predicate P holds at *pp* and P ⇛ B, then B is verified for *pp*. This technique can be used to verify executable assert statements and loop invariants, for example.

Class invariants[5] (henceforth simply called invariants) are verified in a similar way. Since an invariant protocol [64] always specifies the points in a program where an invariant, say B, should hold, B is verified if it is verified for all such points.

Another technique is to annotate an invariant B with a separation logic predicate P which characterises the states in which the invariant should hold. If P ⇛ B, then B is verified. This offers a flexible scheme if annotations are given on the level of individual invariant clauses.

Examples. Consider the following invariant clauses, reproduced from the bottom of class SLIST in Figure 4.1:[6]

[5] Also called *object invariants*.

[6] These are public invariants, as opposed to private representation invariants (which is how the term 'invariant' is used in the Spec# literature, e.g. [45]). Private invariants can be verified in a similar way.

invariant
count_non_negative: $\{L(l{:}\,\alpha)\}$ count >= 0
empty_definition: $\{L(l{:}\,\alpha)\}$ is_empty = (count = 0)

1. For the clause named *count_non_negative*, $L(l{:}\,\alpha) \Rrightarrow$ (count >= 0) holds:

$\{L(l{:}\,\alpha)\}$
t := count
$\{L(l{:}\,\alpha) * t = |\alpha|\}$
v := t >= 0
$\{L(l{:}\,\alpha) * t = |\alpha| * v = (t \geq 0)\}$
$\{L(l{:}\,\alpha) * v = True\}$

2. The invariant clause *empty_definition* is verified similarly:

$\{L(l{:}\,\alpha)\}$
t1 := is_empty
$\{L(l{:}\,\alpha) * t1 = (\alpha = \epsilon)\}$
t2 := count
$\{L(l{:}\,\alpha) * t2 = |\alpha| * t1 = (\alpha = \epsilon)\}$
t3 := t2 = 0
$\{L(l{:}\,\alpha) * t2 = |\alpha| * t1 = (\alpha = \epsilon) * t3 = (t2 = 0)\}$
$\{L(l{:}\,\alpha) * t1 = (\alpha = \epsilon) * t3 = (|\alpha| = 0)\}$
v := t1 = t3
$\{L(l{:}\,\alpha) * t1 = (\alpha = \epsilon) * t3 = (|\alpha| = 0) * v = (t1 = t3)\}$
$\{L(l{:}\,\alpha) * v = ((\alpha = \epsilon) = (|\alpha| = 0))\}$
$\{L(l{:}\,\alpha) * v = True\}$

□

The fact that the structural rules of separation logic are sound provides a simple way to show that a verified invariant clause holds at a particular program point. The following rules can also be used when verifying other executable assertions:[7] [8]

[7] Similar rules can be given for P,Q $\Rrightarrow$ B.

[8] These rules are independent of the structure of B. Rules also exist which do depend on its structure. For example, if $P_1 \Rrightarrow B_1$ and $P_2 \Rrightarrow B_2$, then $(P_1 * P_2) \Rrightarrow (B_1$ **par_and** $B_2)$. This rule allows concurrency in executable assertions.

$$\frac{P \Rrightarrow B}{(P * Q) \Rrightarrow B}\ \text{Frame}'$$

$$\frac{P \Leftrightarrow Q \qquad P \Rrightarrow B}{Q \Rrightarrow B}\ \text{Conseq}'$$

$$\frac{P \Rrightarrow B}{(\exists x \cdot P) \Rrightarrow B}\ \text{AuxVarElim}'$$

$$\frac{P \Rrightarrow B \qquad Q \Rrightarrow B}{(P \vee Q) \Rrightarrow B}\ \text{Disj}'$$

and others.

The Frame′ rule does not need any side-condition, since the translation of B will modify only fresh variables, i.e. variables not free in Q. Note the way Conseq′ is written: if P ⇛ B and Q ⇒ P, then Q ⇛ B does *not* necessarily hold (example 3 in Section 4.6.1 provides a counterexample). In the AuxVarElim′ rule, the variable x may not be free in B.

Example. If a program containing class SLIST from Figure 4.1 is verified, then it follows as a consequence of the separation logic specification that both invariant clauses hold in the visible states [52] of SLIST. In other words, both clauses are established by the constructor and hold on entry and exit of all other public features. Since each clause B is verified, we know $L(l{:}\alpha) \Rrightarrow B$. In the following proofs the '⇛ B' part is omitted.

1. Upon exit from *SLIST*:

$$\cfrac{\cfrac{\cfrac{\cfrac{\cfrac{L(l{:}\alpha)}{L(l{:}\alpha) * \alpha = \epsilon}\ \text{Frame}'}{L(l{:}\epsilon) * \alpha = \epsilon}\ \text{Conseq}'}{\exists\alpha \cdot L(l{:}\epsilon) * \alpha = \epsilon}\ \text{AuxVarElim}'}{L(l{:}\epsilon) * \text{True}}\ \text{Conseq}'}{L(l{:}\epsilon)}\ \text{Conseq}'$$

2. Upon entry to *remove_first*:

$$\dfrac{\dfrac{\dfrac{\dfrac{\dfrac{\dfrac{\dfrac{\dfrac{\dfrac{\mathrm{L}(\mathrm{l}\colon\alpha)}{\mathrm{L}(\mathrm{l}\colon\alpha) * \alpha = \beta}\,\mathrm{Frame}'}{\mathrm{L}(\mathrm{l}\colon\beta) * \alpha = \beta}\,\mathrm{Conseq}'}{\exists\alpha\cdot\, \mathrm{L}(\mathrm{l}\colon\beta) * \alpha = \beta}\,\mathrm{AuxVarElim}'}{\mathrm{L}(\mathrm{l}\colon\beta)}\,\mathrm{Conseq}'}{\mathrm{L}(\mathrm{l}\colon\beta) * \beta = [e] +\!\!+ \alpha}\,\mathrm{Frame}'}{\mathrm{L}(\mathrm{l}\colon[\mathrm{e}] +\!\!+ \alpha) * \beta = [e] +\!\!+ \alpha}\,\mathrm{Conseq}'}{\exists\beta\cdot\, \mathrm{L}(\mathrm{l}\colon[\mathrm{e}] +\!\!+ \alpha) * \beta = [e] +\!\!+ \alpha}\,\mathrm{AuxVarElim}'}{\mathrm{L}(\mathrm{l}\colon[\mathrm{e}] +\!\!+ \alpha)}\,\mathrm{Conseq}'$$

3. For the exit of *insert*, we start with L(l: β) which was derived in the previous proof.

$$\dfrac{\dfrac{\dfrac{\dfrac{\dfrac{\mathrm{L}(\mathrm{l}\colon\beta)}{\mathrm{L}(\mathrm{l}\colon\beta) * \beta = (\alpha_\mathrm{F} +\!\!+ [\mathrm{i}] +\!\!+ \alpha_\mathrm{S}) * \alpha_\mathrm{F} +\!\!+ \alpha_\mathrm{S} = \alpha}\,\mathrm{Frame}'}{\exists\beta\cdot\, \mathrm{L}(\mathrm{l}\colon\beta) * \beta = (\alpha_\mathrm{F} +\!\!+ [\mathrm{i}] +\!\!+ \alpha_\mathrm{S}) * \alpha_\mathrm{F} +\!\!+ \alpha_\mathrm{S} = \alpha}\,\mathrm{AuxVarElim}'}{\mathrm{L}(\mathrm{l}\colon\alpha_\mathrm{F} +\!\!+ [\mathrm{i}] +\!\!+ \alpha_\mathrm{S}) * \alpha_\mathrm{F} +\!\!+ \alpha_\mathrm{S} = \alpha}\,\mathrm{Conseq}'}{\exists\alpha_\mathrm{S}\cdot\, \mathrm{L}(\mathrm{l}\colon\alpha_\mathrm{F} +\!\!+ [\mathrm{i}] +\!\!+ \alpha_\mathrm{S}) * \alpha_\mathrm{F} +\!\!+ \alpha_\mathrm{S} = \alpha}\,\mathrm{AuxVarElim}'}{\exists\alpha_\mathrm{F}, \alpha_\mathrm{S}\cdot\, \mathrm{L}(\mathrm{l}\colon\alpha_\mathrm{F} +\!\!+ [\mathrm{i}] +\!\!+ \alpha_\mathrm{S}) * \alpha_\mathrm{F} +\!\!+ \alpha_\mathrm{S} = \alpha}\,\mathrm{AuxVarElim}'$$

□

In effect we create fine-grained invariant protocols by annotating invariant clauses: the predicate abstractly specifies the program points where they should hold, and invariants can be verified, i.e. guaranteed, to hold there. One can refine this idea to create more sophisticated invariant protocols.

Examples.

1. Here is an invariant protocol for single-inheritance programs where subclasses can refine individual invariant clauses. Γ maps a class name and invariant clause name to the invariant specification predicate and Boolean expression. Two rules distinguish the case where class C introduces an invariant clause named inv from the case where C inherits and refines inv from its parent C′:

$$\dfrac{\forall \mathrm{i} \in (1..\mathrm{n})\cdot\, (\mathrm{P} * \textbf{Current} : \mathrm{C}) \Rrightarrow \mathrm{B_i}}{\Gamma \vdash_i \textbf{introduce}\ \mathrm{inv}\colon \{\mathrm{P}\}\ \langle \mathrm{B_1}\rangle\ \textbf{and}\ \ldots\ \textbf{and}\ \langle \mathrm{B_n}\rangle\ \mathrm{in\ C}}$$

$$\frac{\begin{array}{l} C' \prec_1 C \\ \Gamma(C.\mathrm{inv}) = \{P\}\ B \\ B = \langle B_1 \rangle\ \textbf{and}\ \ldots\ \textbf{and}\ \langle B_n \rangle \\ P \Leftrightarrow (P' * R) \\ \forall i \in (1..n) \cdot\ (P' * \textbf{Current} : C') \Rrightarrow B_i \\ \forall j \in (1..m) \cdot\ (P' * \textbf{Current} : C') \Rrightarrow B'_j \end{array}}{\Gamma \vdash_i \textbf{inherit}\ \mathrm{inv}\colon \{P'\}\ B\ \textbf{and}\ \langle B'_1 \rangle\ \textbf{and}\ \ldots\ \textbf{and}\ \langle B'_m \rangle\ \text{in}\ C'}$$

The above invariant protocol guarantees that whenever 'inv: {P} B' appears in class C, then $\forall x <: C \cdot P[x/\textbf{Current}] \Rrightarrow B[x/\textbf{Current}]$ where x is fresh. Enclosing an expression in angle-brackets provides it with a side-effect scope. The next example explains why it works.

2. The previous invariant protocol is an instance of a more general scheme. Here, invariant clauses are equipped with both a dynamic and a static specification predicate. When a class C introduces an invariant clause:

   introduce inv: {P} {S} $\langle B_1 \rangle$ **and** ... **and** $\langle B_n \rangle$

   then it is considered a shorthand for introducing the methods C_inv_i, where i ∈ 1..n:

   introduce C_inv_i: BOOL
   dynamic {P}_{P ∗ **Result** = True}
   static {S}_{S ∗ **Result** = True}
   do $(\!|B_i|\!)_{\textbf{Result}}$ **end**

   When the direct subclass C′ refines the invariant clause by listing:

   inherit inv: {P′} {S′} B **and** $\langle B'_1 \rangle$ **and** ... **and** $\langle B'_m \rangle$

   where $B = \langle B_1 \rangle$ **and** ... **and** $\langle B_n \rangle$, then it is considered a shorthand for overriding all the methods C_inv_i, where i ∈ 1..n:

   override C_inv_i: BOOL
   dynamic {P′}_{P′ ∗ **Result** = True}
   static {S′}_{S′ ∗ **Result** = True}
   do $(\!|B_i|\!)_{\textbf{Result}}$ **end**

   and introducing the methods C′_inv_j, where j ∈ 1..m:

   introduce C′_inv_j: BOOL
   dynamic {P′}_{P′ ∗ **Result** = True}
   static {S′}_{S′ ∗ **Result** = True}
   do $(\!|B'_j|\!)_{\textbf{Result}}$ **end**

Looking at the method verification rules of 2.6, we see that class C must satisfy:

$\Delta \vdash \{S\}\_\{S * \mathbf{Result} = \text{True}\} \overset{\mathbf{Current}:C}{\Longrightarrow} \{P\}\_\{P * \mathbf{Result} = \text{True}\}$
$\forall i \in (1..n) \cdot S \Rrightarrow B_i$

and class C′ must satisfy:

$\Delta \vdash \{P'\}\_\{P' * \mathbf{Result} = \text{True}\} \Longrightarrow \{P\}\_\{P * \mathbf{Result} = \text{True}\}$
$\Delta \vdash \{S'\}\_\{S' * \mathbf{Result} = \text{True}\} \overset{\mathbf{Current}:C'}{\Longrightarrow} \{P'\}\_\{P' * \mathbf{Result} = \text{True}\}$
$\forall i \in (1..n) \cdot S' \Rrightarrow B_i$
$\forall j \in (1..m) \cdot S' \Rrightarrow B'_j$

By taking $S = P * \mathbf{Current} : C$, and $S' = P' * \mathbf{Current} : C'$, it is simple to prove that the two rules of the previous example are theorems. It is also easy to verify the protocol's guarantee.

3. Instead of overriding all the methods C_inv_i in class C′, as in the previous example, a protocol might choose to inherit them:

   inherit C_inv_i: BOOL
   dynamic $\{P'\}\_\{P' * \mathbf{Result} = \text{True}\}$
   static $\{S'\}\_\{S' * \mathbf{Result} = \text{True}\}$

   Doing so changes the proof obligations of class C′ – instead of:

   $\forall i \in (1..n) \cdot S' \Rrightarrow B_i$

   we must establish that:

   $\Delta \vdash \{S\}\_\{S * \mathbf{Result} = \text{True}\} \Longrightarrow \{S'\}\_\{S' * \mathbf{Result} = \text{True}\}$ □

4.5 Model-based specifications

Since executable specifications are frequently not very expressive, model classes and model-based contracts are sometimes used to strengthen them [12, 63].

Example. Consider the interface of model class SEQUENCE in Figure 4.3. It provides an abstraction of immutable sequences for specification purposes. Class SLIST can be specified in terms of SEQUENCE, as the interface extract in Figure 4.4 shows. Note that SLIST now has a model query, namely *model*, which returns the immutable sequence abstraction of an SLIST instance at the point when it is called. A comparison of *remove_first*'s specification in Figures 4.1 and 4.4 shows that the model-based specification involves the

class SEQUENCE
define x.$\mathrm{SEQ}_{SEQUENCE}(\mathrm{s}{:}\,\alpha)$ **as** ...
feature
SEQUENCE
dynamic $\{\mathrm{True}\}\_\{\mathrm{SEQ}(\mathrm{s}{:}\,\epsilon)\}$

cons(i: INT): SEQUENCE
dynamic $\{\mathrm{SEQ}(\mathrm{s}{:}\,\alpha)\}\_\{\mathrm{SEQ}(\mathrm{s}{:}\,\alpha) * \mathbf{Result}.\mathrm{SEQ}(\mathrm{s}{:}\,[\mathrm{i}] +\!\!+ \alpha)\}$

head: INT
dynamic $\{\mathrm{SEQ}(\mathrm{s}{:}\,[\mathrm{e}] +\!\!+ \alpha)\}\_\{\mathrm{SEQ}(\mathrm{s}{:}\,[\mathrm{e}] +\!\!+ \alpha) * \mathbf{Result} = \mathrm{e}\}$

tail: SEQUENCE
dynamic $\{\mathrm{SEQ}(\mathrm{s}{:}\,[\mathrm{e}] +\!\!+ \alpha)\}\_\{\mathrm{SEQ}(\mathrm{s}{:}\,[\mathrm{e}] +\!\!+ \alpha) * \mathbf{Result}.\mathrm{SEQ}(\mathrm{s}{:}\,\alpha)\}$

is_nil: BOOL
dynamic $\{\mathrm{SEQ}(\mathrm{s}{:}\,\alpha)\}\_\{\mathrm{SEQ}(\mathrm{s}{:}\,\alpha) * \mathbf{Result} = (\alpha = \epsilon)\}$

eq(o: SEQUENCE): BOOL
dynamic $\{\mathrm{SEQ}(\mathrm{s}{:}\,\alpha) * \mathrm{o}.\mathrm{SEQ}(\mathrm{s}{:}\,\beta)\}\_$
$\{\mathrm{SEQ}(\mathrm{s}{:}\,\alpha) * \mathrm{o}.\mathrm{SEQ}(\mathrm{s}{:}\,\beta) * \mathbf{Result} = (\alpha = \beta)\}$
end

Figure 4.3: The interface of model class SEQUENCE.

```
class SLIST
    ...
feature
    ...
    model: SEQUENCE
    dynamic {L(l: α)}_{L(l: α) * Result.SEQ(s: α)}
    executable {True}_{True}

    remove_first
    dynamic {L(l: [e] ++ α)}_{L(l: α)}
    executable {not model.is_nil}_{model.eq(old(model).tail)}
    ...
invariant
    ...
    empty_inv: {L(l: α)} is_empty = model.is_nil
end
```

Figure 4.4: An extract from class SLIST which uses model-based contracts.

element values stored in an SLIST instance and not just their number. □

The following three problems are typically not easy to solve with conventional techniques [39, 19]:

1. Devising semantics for model classes and proving their implementations correct.

2. Giving a semantics to model queries, such as *model* in Figure 4.4, and proving their implementations correct.

3. Verifying model-based specifications, such as the model-based contract of *remove_first* and the model-based invariant clause *empty_inv* in Figure 4.4.

The first two problems can be solved with separation logic. Figures 4.3 and 4.4 show separation logic specifications for a model class and model query. A conventional separation logic proof system, such as the one of Chapter 2, can be used to verify their implementations. The third problem can then be addressed with the framework of this chapter.

Examples. Consider the model-based specifications of class SLIST in Figure 4.4. Suppose that omitted features have the same separation logic specifications as in Figure 4.1.

1. For the invariant clause *empty_inv*:

$\{\mathrm{L}(\mathrm{l}{:}\,\alpha)\}$
  t1 := is_empty
$\{\mathrm{L}(\mathrm{l}{:}\,\alpha) * \mathrm{t1} = (\alpha = \epsilon)\}$
  t2 := model
$\{\mathrm{L}(\mathrm{l}{:}\,\alpha) * \mathrm{t2.SEQ}(\mathrm{s}{:}\,\alpha) * \mathrm{t1} = (\alpha = \epsilon)\}$
  t3 := t2.is_nil
$\{\mathrm{L}(\mathrm{l}{:}\,\alpha) * \mathrm{t2.SEQ}(\mathrm{s}{:}\,\alpha) * \mathrm{t3} = (\alpha = \epsilon) * \mathrm{t1} = (\alpha = \epsilon)\}$
$\{\mathrm{L}(\mathrm{l}{:}\,\alpha) * \mathrm{t3} = (\alpha = \epsilon) * \mathrm{t1} = (\alpha = \epsilon)\}$
  v := t1 = t3
$\{\mathrm{L}(\mathrm{l}{:}\,\alpha) * \mathrm{t3} = (\alpha = \epsilon) * \mathrm{t1} = (\alpha = \epsilon) * \mathrm{v} = (\mathrm{t1} = \mathrm{t3})\}$
$\{\mathrm{L}(\mathrm{l}{:}\,\alpha) * \mathrm{v} = \mathrm{True}\}$

2. We next verify the postcondition of *remove_first* which contains one old-expression:

$\{\mathrm{L}(\mathrm{l}{:}\,[\mathrm{e}] +\!\!+ \alpha)\}$
  v1 := model
$\{\mathrm{L}(\mathrm{l}{:}\,[\mathrm{e}] +\!\!+ \alpha) * \mathrm{v1.SEQ}(\mathrm{s}{:}\,[\mathrm{e}] +\!\!+ \alpha)\}$

The proof is completed by using $\mathrm{v1.SEQ}(\mathrm{s}{:}\,[\mathrm{e}] +\!\!+ \alpha)$ for R_1:

$\{\mathrm{L}(\mathrm{l}{:}\,\alpha) * \mathrm{v1.SEQ}(\mathrm{s}{:}\,[\mathrm{e}] +\!\!+ \alpha)\}$
  t1 := model
$\{\mathrm{L}(\mathrm{l}{:}\,\alpha) * \mathrm{t1.SEQ}(\mathrm{s}{:}\,\alpha) * \mathrm{v1.SEQ}(\mathrm{s}{:}\,[\mathrm{e}] +\!\!+ \alpha)\}$
  t2 := v1.tail
$\{\mathrm{L}(\mathrm{l}{:}\,\alpha) * \mathrm{t1.SEQ}(\mathrm{s}{:}\,\alpha) * \mathrm{v1.SEQ}(\mathrm{s}{:}\,[\mathrm{e}] +\!\!+ \alpha) * \mathrm{t2.SEQ}(\mathrm{s}{:}\,\alpha)\}$
$\{\mathrm{L}(\mathrm{l}{:}\,\alpha) * \mathrm{t1.SEQ}(\mathrm{s}{:}\,\alpha) * \mathrm{t2.SEQ}(\mathrm{s}{:}\,\alpha)\}$
  v := t1.eq(t2)
$\{\mathrm{L}(\mathrm{l}{:}\,\alpha) * \mathrm{t1.SEQ}(\mathrm{s}{:}\,\alpha) * \mathrm{t2.SEQ}(\mathrm{s}{:}\,\alpha) * \mathrm{v} = (\alpha = \alpha)\}$
$\{\mathrm{L}(\mathrm{l}{:}\,\alpha) * \mathrm{v} = \mathrm{True}\}$

□

4.6 Relative purity and predicate extraction

4.6.1 Relative purity

Side-effects in executable specifications conventionally complicate verification, since the logical predicates extracted from them cannot be imperative. Most techniques therefore impose some form of purity, i.e. side-effect freeness, on specification expressions. Purity comes in many flavours, such as strong purity, weak purity and observational purity [18, 53].

The verification techniques of this chapter tolerate side-effects in executable specifications to a high degree. In fact, they use a new notion of purity, namely *relative* purity. An executable assertion is pure or impure with respect to a given separation logic specification:

B is pure relative to P $\stackrel{\text{def}}{=}$ $\{P\}v := B\{P\}$, where v is a fresh variable.

This means informally that B can be evaluated in a state satisfying P, and that the evaluation will preserve P. Similarly, for an executable assertion B containing the list of **old**-expressions **old**(E_1),...,**old**(E_n):

$$\text{B is pure relative to P,Q} \stackrel{\text{def}}{=} \exists R\cdot \quad \{P\}(v_1 := E_1, \ldots, v_n := E_n)\{P * R\} \text{ and } \{Q * R\}v := B[v_1; \ldots; v_n]\{Q\}$$

where v, v_1,...,v_n are fresh variables and B[v_1;...;v_n] is defined as before.

The following lemmas follow from the definitions by the rule of Consequence:

> If P $\Rrightarrow$ B, then B is pure relative to P.
>
> If P,Q $\Rrightarrow$ B, then B is pure relative to P,Q.

Examples. Consider the specification of class SLIST in Figure 4.1.

1. Suppose we replace the executable postcondition of *insert* with a hypothetical assertion B for which $(\!|B|\!)_v$ is given by:

   t := remove_first
   insert(t)
   v := has(i)

   B is pure relative to the separation logic postcondition of *insert* because one can prove[9] $(\exists \alpha_F, \alpha_S\cdot \mathrm{L}(\mathrm{l}{:}\,\alpha_F +\!\!+ [\mathrm{i}] +\!\!+ \alpha_S) * \alpha_F +\!\!+ \alpha_S = \alpha) \Rrightarrow$ B.

[9] The proof uses the axiom information expressed in L_sorted. The rule of Disjunction can combine subproofs arising from a case split on $\alpha_F = \epsilon$.

2. Suppose B is a hypothetical assertion for which $(\!|\mathrm{B}|\!)_{\mathrm{v}}$ is:

   insert(4)
   v := **not** is_empty

   B is not pure relative to $\mathrm{L}(\mathrm{l}{:}\,\alpha)$. We cannot complete the following proof attempt, since the needed implication does not necessarily hold:

   $\{\mathrm{L}(\mathrm{l}{:}\,\alpha)\}$
   insert(4)
   $\{\exists \alpha_{\mathrm{F}}, \alpha_{\mathrm{S}}{\cdot}\ \mathrm{L}(\mathrm{l}{:}\,\alpha_{\mathrm{F}} +\!\!+ [4] +\!\!+ \alpha_{\mathrm{S}}) * \alpha_{\mathrm{F}} +\!\!+ \alpha_{\mathrm{S}} = \alpha\}$
   v := **not** is_empty
   $\{\exists \alpha_{\mathrm{F}}, \alpha_{\mathrm{S}}{\cdot}\ \mathrm{L}(\mathrm{l}{:}\,\alpha_{\mathrm{F}} +\!\!+ [4] +\!\!+ \alpha_{\mathrm{S}}) * \alpha_{\mathrm{F}} +\!\!+ \alpha_{\mathrm{S}} = \alpha\}$
   // An implication is needed to establish:
   $\{\mathrm{L}(\mathrm{l}{:}\,\alpha)\}$

   A counterexample showing that B is not pure relative to P in the context of class C comprises:

   (a) a verified implementation of C and all classes transitively used by it;

   (b) an execution trace where B is evaluatated in an initial state satisfying P that either faults or terminates in a state not satisfying P.

3. We use B from the previous example and show $(\exists \alpha{\cdot}\ \mathrm{L}(\mathrm{l}{:}\,\alpha)) \Rrightarrow \mathrm{B}$:

   $\{\exists \alpha{\cdot}\ \mathrm{L}(\mathrm{l}{:}\,\alpha)\}$
   insert(4)
   $\{\exists \alpha, \alpha_{\mathrm{F}}, \alpha_{\mathrm{S}}{\cdot}\ \mathrm{L}(\mathrm{l}{:}\,\alpha_{\mathrm{F}} +\!\!+ [4] +\!\!+ \alpha_{\mathrm{S}}) * \alpha_{\mathrm{F}} +\!\!+ \alpha_{\mathrm{S}} = \alpha\}$
   v := **not** is_empty
   $\{\exists \alpha, \alpha_{\mathrm{F}}, \alpha_{\mathrm{S}}{\cdot}\ \mathrm{L}(\mathrm{l}{:}\,\alpha_{\mathrm{F}} +\!\!+ [4] +\!\!+ \alpha_{\mathrm{S}}) * \mathrm{v} = \mathrm{True} * \alpha_{\mathrm{F}} +\!\!+ \alpha_{\mathrm{S}} = \alpha\}$
   $\{\exists \alpha{\cdot}\ \mathrm{L}(\mathrm{l}{:}\,\alpha) * \mathrm{v} = \mathrm{True}\}$

   So B is pure with respect to $\exists \alpha{\cdot}\ \mathrm{L}(\mathrm{l}{:}\,\alpha)$.

   Since $\mathrm{L}(\mathrm{l}{:}\,\alpha) \Rightarrow (\exists \alpha{\cdot}\ \mathrm{L}(\mathrm{l}{:}\,\alpha))$, this example and the previous one show that if $\mathrm{P} \Rightarrow \mathrm{Q}$ and $\mathrm{Q} \Rrightarrow \mathrm{B}$, then $\mathrm{P} \Rrightarrow \mathrm{B}$ does not hold in general.

4. Let B be any executable assertion without old-expressions. B is always pure relative to False. If B never faults, then it is also pure relative to True. □

Any notion of purity which classifies side-effects as intrinsically harmful or not will rule out executable assertions that programmers might want to write. Relative purity judges whether a side-effect is harmful or not only with

respect to the properties one cares about. For example, the side-effects in B of examples 2 and 3 are harmless with respect to $\exists\alpha\cdot \mathrm{L}(\mathrm{l}{:}\,\alpha)$, but harmful with respect to $\mathrm{L}(\mathrm{l}{:}\,\alpha)$. This is so because the side-effect of inserting 4 in a sorted list will maintain a list structure (what the first purity specification states), but not a list structure with the same elements (what the second specification demands). The concept of relativity pervades verification already, since code is only correct or incorrect relative to a specification. We adopt it also for purity.

Relative purity guarantees soundness of reasoning and does not impose unnecessary constraints on executable assertions. An executable assertion is free to perform any side-effect as long as nothing happened from the perspective of the separation logic specification. Runtime assertion checking does nothing relative to the specification of a verified program. An executable assertion can even print 'Hello World!' if the specification permits it. Purity is in the eye of the asserter.

4.6.2 Predicate extraction

Programmers are conventionally encouraged to specify sets of states with computations. Hoare-style verification cannot do much with computations – it needs predicates. So conventional verification approaches, including the Spec# system [3] and JML toolset [9], have to extract predicates from computations. This can be achieved in various ways: as long as there is an agreed-upon mapping from computations to predicates, the computations can be seen as syntactic sugar for predicates.

When given an executable assertion B without old-expressions[10], one possibility is to extract a predicate P which follows from B. This is described by the connecting implication $\mathrm{B} \Rrightarrow \mathrm{P}$, which informally means that B evaluating to True is sufficient to conclude P. Here is a formal definition:

$$\mathrm{B} \Rrightarrow \mathrm{P} \stackrel{\text{def}}{=} \forall \mathrm{Q}\cdot \text{ if } \mathrm{Q} \Rrightarrow \mathrm{B} \text{ then } \mathrm{Q} \Rightarrow \mathrm{P}.$$

If $\mathrm{B} \Rrightarrow \mathrm{P}$, then $\mathrm{P} \Rrightarrow \mathrm{B}$ does not necessarily hold. However, the best predicate that can be extracted in this way, i.e. the strongest P such that $\mathrm{B} \Rrightarrow \mathrm{P}$, is precisely the weakest P such that $\mathrm{P} \Rrightarrow \mathrm{B}$. So if we define

$$\mathit{Best}(\mathrm{B}) \stackrel{\text{def}}{=} \bigvee\{\mathrm{P} \mid \mathrm{P} \Rrightarrow \mathrm{B}\}$$

then the following lemmas hold:

[10]The treatment can be generalised to executable assertions with old-expressions.

1. $Best(\mathrm{B}) \Rrightarrow \mathrm{B}$ and $\mathrm{B} \Rrightarrow Best(\mathrm{B})$, i.e. $Best(\mathrm{B}) \Lleftrightarrow \mathrm{B}$
2. $\mathrm{B} \Rrightarrow \mathrm{P}$ iff $Best(\mathrm{B}) \Rightarrow \mathrm{P}$
3. If $\mathrm{P} \Rrightarrow \mathrm{B}$ then $\mathrm{P} \Rightarrow Best(\mathrm{B})$

Since $Best(\mathrm{B}) \Lleftrightarrow \mathrm{B}$, the predicate $Best(\mathrm{B})$ can be seen as the exact logical counterpart of B.

Examples. Let us ignore inheritance and suppose an implementation of class SLIST in Figure 4.1 maintains its stored elements in a list of linked nodes.

1. If *is_empty* is implemented as a field in class SLIST, then $Best(\text{is\_empty}) = (\text{is\_empty} \hookrightarrow \text{True})$.

2. If *is_empty* tests whether a field named 'head', which points to the first node, is **Void**, then $Best(\textbf{not}\ \text{is\_empty}) = (\exists \mathrm{x} \cdot\ \text{head} \hookrightarrow \mathrm{x} * \mathrm{x} \neq \textbf{Void})$.

3. If *has* is implemented by a simple linear traversal of the nodes, then the predicate $Best(\text{has(i)})$ denotes that i is encountered inside a node before the next node pointer becomes **Void**. Note that $Best(\text{has(i)})$ is too weak to rule out a cyclic or frying-pan [8] list. Such a list does not even have to contain i in order to satisfy $Best(\text{has(i)})$, because the definition does not demand the termination of has(i). □

As these examples show, the logical counterparts of executable assertions are often very weak. Predicate extraction techniques will have to map computations into stronger predicates in order to verify method bodies. Such predicates are similar in flavour to the ones used in this chapter, and do not follow from the executable ones. Section 4.8 discusses this in more depth.

Our approach uses predicates rather than computations to characterise states. It views the separation logic specification as the primary one for verification. There is no extraction of predicates, and verification treats executable assertions as computations which should evaluate to True. We do not verify the program *using* its executable assertions, we verify the program *and* its executable assertions.

4.7 Implementation

We extended MultiStar to provide fully automatic verification of executable Eiffel assertions with respect to separation logic specifications. The implementation leverages mechanisms which are present in MultiStar and most separation logic proof tools: symbolic execution, implication checking and

frame inference. Symbolic execution and implication checking are the basic ingredients for proving connecting implications. For example, when proving $P \Rrightarrow B$, MultiStar executes $v := B$ in the symbolic state P to obtain a symbolic state P'. It then checks whether $P' \Rightarrow (P * v = \text{True})$. For executable postconditions, the results of old-expressions are inferred automatically with frame inference and not specified manually. Suppose MultiStar must prove the connecting implication $P,Q \Rrightarrow C$, where C contains a single old-expression **old**(E). Executing $v_1 := E$ in the symbolic state P yields a resulting symbolic state P'. Frame inference next determines the symbolic result R of E in $P' \vdash P * R$. Then $w := C[v_1]$ is executed in the symbolic state $Q * R$ and yields a symbolic state Q'. Finally, MultiStar checks whether $Q' \Rightarrow (Q * w = \text{True})$.

MultiStar can easily handle the examples presented in this chapter, which are also available in the EVE download [28].

4.8 Conclusions and related work

The presented framework offers a sound and simple way to verify various executable specifications with separation logic. The notion of relative purity is central to the framework and embraces side-effects in executable assertions, thereby allowing programmers more freedom of expression compared to conventional verification approaches. The framework is well-suited to separation logic proof tools and is implemented in MultiStar.

Regarding related work, separation logic [54, 60] offers local reasoning for heap-manipulating programs and is central to the framework. Its adaptation to object-orientation [57, 56, 58, 65] is especially relevant for the presentation.

Executable specifications are embodied in programming languages such as Eiffel [26] and Spec# [3]. Dedicated specification languages such as JML [40] also use them. JML includes model classes and allows model-based contracts [12]. Another library with model classes is MML [63]. Executable assertions offer several benefits in software development, including runtime checking [26, 13] and automated testing [47, 15].

The problem of obtaining the strongest P such that $B \Rrightarrow P$ amounts to finding the weakest footprint of B preserved by B which ensures B evaluates to True. This problem appears to be similar in flavour but more general than abduction [10, 30].

Conventional approaches to verification, including the Spec# system [3] and JML toolset [9], extract predicates from assertions in a different way [45, 18]:

1. They do not use footprints. Well-formedness of the heap is exploited and any chain of references can be followed. Since a method can po-

tentially modify any reachable object, the absence of footprints makes reasoning more global. Specifications must describe which objects are modified, because there might be external references to reachable objects. The *aliasing problem* arises because objects maintaining such references, or aliases, can depend on properties of the aliased objects. Approaches to the problem typically restrict or prevent aliases and/or operations on references. They include confinement, sharing and access control, ownership, immutability, uniqueness, information flow and escape analyses [16].

2. They impose purity constraints on the methods used in assertion expressions. Several notions of purity exist which prevent methods from changing the state [18, 53]. A strongly pure method has no side-effect at all. A weakly pure method does not change existing objects, but might allocate and modify new ones. An observationally pure method may modify existing objects, provided that the change is sufficiently encapsulated such that no other class can observe a change. Proving method purity becomes harder as the notion becomes more permissive. Weak purity can be proved with a combination of pointer and escape analysis [62], while a method's observational purity can be shown by proving that it simulates a weakly pure one from the perspective of other classes [53].

3. They encode pure methods and their contracts in first-order logic as uninterpreted functions and axioms respectively [18, 45, 17]. Checking well-formedness of pure method specifications is vital in this step, because inconsistent axiomatisations can result from unsatisfiable specifications or recursive specifications which are ill-founded [61]. A pure method call in an assertion is encoded as an application of the corresponding uninterpreted function. A loop, written in the stylised form of quantification over a finite domain (as in JML and Spec#), is directly encoded as a quantified formula. The reader is referred to [45] for assertion encoding details. Model class and model-based contract encoding is discussed in [19].

Chapter 5

Correctness by construction

The previous two chapters proposed extensions to the separation logic proof system: features for reasoning about related abstractions, multiple inheritance and executable contracts. Instead of adding new features, this chapter investigates correctness by construction – an alternative way to develop verified programs. Dijkstra, who advocated this approach, described it well in his Turing Award lecture [22]:

> The only effective way to raise the confidence level of a program significantly is to give a convincing proof of its correctness. But one should not first make the program and then prove its correctness, because then the requirement of providing the proof would only increase the poor programmer's burden. On the contrary: the programmer should let correctness proof and program grow hand in hand.

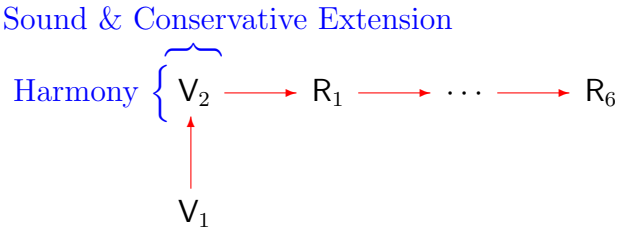
Stepwise refinement [50, 2, 31] is an important technique in correctness by construction. (It is often all that is needed, but developing correct OO code will require more machinery.) The bulk of this chapter is consequently devoted to freefinement – a new algorithm that constructs a sound refinement calculus from a verification system under certain conditions.

In fact many theories in computer science are presented, or approximated, by compositional *verification systems*. A verification system in this chapter is any formal system for establishing whether an inductively defined term, typically a program, satisfies a specification. For example, Hoare logics and type systems can be viewed as verification systems. In the case of Hoare logics, the system proves that a statement satisfies certain specifications given as preconditions and postconditions. In the case of type systems, the system proves that a term has a certain type in a type context.

Refinement systems play a similar role to verification systems, the main difference being that they relate terms to other terms, instead of terms and specifications. Another difference is that they typically include so-called specification terms. Intuitively, a term refines another if it is 'better', i.e. if it satisfies more specifications. *Refinement calculi* are formal systems for establishing refinements. For example, the calculus of Morgan [50] derives refinements between statements based on total correctness specifications. Starting from an appropriate specification statement, one can derive a correct algorithm for computing the factorial of a number by applying Morgan's refinement rules.

The work on freefinement started from the observation that a Hoare logic and a refinement calculus for a command language do not have to be independent entities: once the Hoare logic is extended with specification statements, the two systems can be accommodated in a single theory. Moreover, there is a strong relation between the two systems. This chapter explains that this relation is not a coincidence: it is possible to analyse the structure of the inference rules defining a verification system, and automatically generate a related refinement calculus. Freefinement is an algorithm that implements this transformation. Surprisingly, freefinement is not limited to Hoare logics, but can be applied to any verification system whose inference rules satisfy certain conditions. Several refinement rules proposed in the literature in different contexts arise in this way.

The freefinement algorithm works as follows. Given a term language and an accompanying verification system V_1 that satisfies certain conditions, freefinement extends the term language with specification terms and builds a verification system V_2 for extended terms. The conditions on V_1 ensure that it is possible to extend the terms without breaking the inference rules; V_2 is consequently a sound and conservative extension of V_1. Moreover, freefinement proposes a sound refinement system R that is in harmony with V_2. Harmony means that the two formal systems can interoperate smoothly. It entails, for example, that a term satisfies a specification according to V_2 if and only if it is possible to refine the specification into the term with R. In fact, proof translation between V_2 and R becomes possible because harmony is demonstrated constructively. Freefinement internally constructs the refinement calculus by 'linearising' V_2 in a series of steps. The conditions on V_1 ensure that successful linearisation is possible. According to the presentation below, at most six steps are needed for this 'refinement of refinement systems'. The situation is summarised as follows:



Freefinement requires no human intervention. The conditions it imposes are fulfilled by many program logics and type systems: examples include Hoare logic, separation logic, the simply-typed lambda calculus and System F. Freefinement defines the semantics of refinement at an abstract level: it associates each term of the extended language with a set of terms from the original language, and refinement simply reduces this set.

With freefinement, tools that are based on verification systems can readily include refinement as a complementary or alternative development style. In cases where the verification system is part of a larger system for establishing program correctness, it may well be possible to use refinement in a more elaborate calculus for constructing correct programs. The second part of this chapter shows how to use the output of freefinement in a calculus for developing correct OO code. This calculus is based on the separation logic proof system of Chapter 2, and maintains close relationships with it.

Chapter outline Section 5.1 describes the freefinement algorithm, which is applied in Section 5.2 to a simple type system for the lambda calculus and also to Hoare logic. Section 5.3 shows how the output of freefinement can be used in a calculus for constructing correct OO programs. Section 5.4 concludes with related work.

5.1 Freefinement

5.1.1 The Inputs

Freefinement requires four things as input:

1. A set of constructors $\mathbb{K}$. The constructors give rise to a term language T, where an arbitrary term t of T is defined by the grammar:

$$\mathrm{t} ::= \mathrm{C}(\mathrm{t}_1, \ldots, \mathrm{t}_n)$$

   where $\mathrm{C} \in \mathbb{K}$.

2. A set of specifications $\mathbb{S}$.

3. A binary relation $\models_{\mathsf{V}_1}$ _ *Sat* _ between terms and specifications. Intuitively, $\models_{\mathsf{V}_1}$ t *Sat* S denotes that term $t \in T$ satisfies specification $S \in \mathbb{S}$.

4. A formal system $\mathsf{V}_1(\mathbb{K}, \mathbb{S}, \models_{\mathsf{V}_1}$ _ *Sat* _$)$, which consists of a set of inference rules for proving sentences of the form t *Sat* S. Each rule of V_1 must have the form A_1 or B_1:

$$A_1 \frac{t_1\ \mathit{Sat}\ S_1 \quad \ldots \quad t_n\ \mathit{Sat}\ S_n}{C(t_1, \ldots, t_n)\ \mathit{Sat}\ S}$$
provided $\mathrm{Pred}(C, S_1, \ldots, S_n, S)$.

$$B_1 \frac{t\ \mathit{Sat}\ S_1 \quad \ldots \quad t\ \mathit{Sat}\ S_m}{t\ \mathit{Sat}\ S}$$
provided $\mathrm{Pred}(S_1, \ldots, S_m, S)$.

The t's, S's and C in the rule forms indicate where the rules of V_1 must use metavariables. Thus a rule of form A_1 has only the freedom to choose a concrete n and a definition for its proviso predicate Pred; the proviso predicate implements the side condition of the rule based on the arguments C, $S_1, \ldots, S_n$ and S. A rule of form B_1 is also a pair: a concrete m and a definition of a predicate with arguments $S_1, \ldots, S_m$ and S. Freefinement requires that the rules must be sound with respect to the following semantics:

Definition 5.1 (Semantics of the Inference Rules)**.**

5.1.1. For rules of the form A_1:
$\mathrm{Pred}(C, S_1, \ldots, S_n, S) \Rightarrow [\forall t_1, \ldots, t_n \in T \cdot \models_{\mathsf{V}_1} t_1\ \mathit{Sat}\ S_1 \wedge \ldots \wedge \models_{\mathsf{V}_1} t_n\ \mathit{Sat}\ S_n \Rightarrow \models_{\mathsf{V}_1} C(t_1, \ldots, t_n)\ \mathit{Sat}\ S]$

5.1.2. For rules of the form B_1:
$\mathrm{Pred}(S_1, \ldots, S_m, S) \Rightarrow$
$[\forall t \in T \cdot \models_{\mathsf{V}_1} t\ \mathit{Sat}\ S_1 \wedge \ldots \wedge \models_{\mathsf{V}_1} t\ \mathit{Sat}\ S_m \Rightarrow \models_{\mathsf{V}_1} t\ \mathit{Sat}\ S]$

The rule forms stipulate that the rules of V_1 must be highly compositional – a requirement that freefinement will exploit. For example, rules cannot inspect or constrain the t's that appear in premises. This will allow freefinement to reuse the rules after specification terms are added to the term language.

Consider the following three rules over $\mathbb{K} = \{0, \mathrm{succ}, \mathrm{pred}\}$ and $\mathbb{S} = \{\mathbb{N}\}$, where n is a metavariable:

$$1\ \frac{n : \mathbb{N}}{\mathrm{succ}(n) : \mathbb{N}} \qquad 2\ \frac{\mathrm{succ}(n) : \mathbb{N}}{\mathrm{pred}(\mathrm{succ}(n)) : \mathbb{N}} \qquad 3\ \frac{n : \mathbb{N}}{\mathrm{pred}(n) : \mathbb{N}}$$
provided positive(n).

Rule 1 can be written in form A_1 with $n = 1$ by defining the proviso Pred(C, S_1, S) as C = succ $\wedge$ S_1 = S = $\mathbb{N}$. Rule 2 is unacceptable, because its premise inspects the term and requires it to match succ(n). Rule 3 is also unacceptable, because it constrains the term in its proviso.

It will become clear later that the 'structural' rules of Hoare logic, such as the rule of consequence, are examples of rules of form B_1. Other rules of Hoare logic, such as the assignment axiom and rule for sequential composition, have the form A_1.

Let $\vdash_{\mathsf{V}_1} t\ \mathit{Sat}\ S$ denote that t *Sat* S is derivable with V_1. The soundness of the rules with respect to the semantics of Definition 5.1 implies the soundness of V_1:

Theorem 5.1 (Soundness of V_1)**.** $\vdash_{\mathsf{V}_1} t\ \mathit{Sat}\ S \Rightarrow \models_{\mathsf{V}_1} t\ \mathit{Sat}\ S$

Proof. By induction on the derivation of t *Sat* S:

- A rule of the form A_1 was last applied. Assume Pred(C, S_1, ..., S_n, S) and the induction hypothesis $\models_{\mathsf{V}_1} t_1\ \mathit{Sat}\ S_1 \wedge \ldots \wedge \models_{\mathsf{V}_1} t_n\ \mathit{Sat}\ S_n$. Then $\models_{\mathsf{V}_1} C(t_1, \ldots, t_n)\ \mathit{Sat}\ S$ by Definition 5.1.1.
- A rule of the form B_1 was last applied. Assume Pred(S_1, ..., S_m, S) and also the induction hypothesis $\models_{\mathsf{V}_1} t\ \mathit{Sat}\ S_1 \wedge \ldots \wedge \models_{\mathsf{V}_1} t\ \mathit{Sat}\ S_m$. From Definition 5.1.2 follows $\models_{\mathsf{V}_1} t\ \mathit{Sat}\ S$. □

Freefinement does not assume the completeness of V_1, i.e. it never assumes $\models_{\mathsf{V}_1} t\ \mathit{Sat}\ S \Rightarrow \vdash_{\mathsf{V}_1} t\ \mathit{Sat}\ S$.

5.1.2 The Extended Language and Formal System

This section extends the language T with specification terms that are useful for refinement. It gives a semantics to the resulting language U, and extends V_1 in a sound and conservative way to prove sentences of the form u *Sat* S where u ∈ U.

The Extended Language U

Suppose $\mathbb{K}$ and $\mathbb{S}$ are disjoint (if they are not, then they can always be decorated to become disjoint) and do not contain a symbol $\bigsqcup$. The extended set of constructors

$$\mathbb{K}' = \mathbb{K} \cup \mathbb{S} \cup \{\bigsqcup \text{ with arity } n \mid n \in \mathbb{N}\}$$

gives rise to an extended language U, which can also be written as:

$$u ::= C(u_1, \ldots, u_n) \mid S \mid \bigsqcup(u_1, \ldots, u_n)$$

A term of the form S is called a *spec* term, and a term of the form $\bigsqcup(u_1, \ldots, u_n)$ is called the *join* of $u_1, \ldots, u_n$. Intuitively, S is a generic term that satisfies S, and $\bigsqcup(u_1, \ldots, u_n)$ is a generic term that satisfies any S that any of the $u_1, \ldots, u_n$ satisfy. Although the details will become clear later, the reasons for adding these terms are simple: the refinement system should be able to refine spec terms into other terms for top-down development, and join terms will be important for simplifying rules of the form B_1 where $m > 1$. If there are no rules of the form B_1 where $m > 1$, then join terms and their consequent treatment can be omitted.

A couple of constructs are used for giving a semantics to U. Let X denote a subset of T, and let Y denote a subset of $\mathbb{S}$. Specs(X) is the set of all specifications that all the terms in X satisfy, and Terms(Y) is the set of terms of T that satisfy all the specifications in Y:

Definition 5.2 (Specs and Terms)**.**

- $\text{Specs}(X) \stackrel{\text{def}}{=} \{S \mid \forall t \in X \cdot \models_{V_1} t \; \mathit{Sat}\; S\}$
- $\text{Terms}(Y) \stackrel{\text{def}}{=} \{t \mid \forall S \in Y \cdot \models_{V_1} t \; \mathit{Sat}\; S\}$

An antitone Galois connection[1] exists between Specs and Terms:

Lemma 5.2. $X \subseteq \text{Terms}(Y) \Leftrightarrow Y \subseteq \text{Specs}(X)$

Proof.
$$\begin{array}{ll} & X \subseteq \text{Terms}(Y) \\ \Leftrightarrow & \{\text{definition of Terms and } \subseteq\} \\ & \quad \forall t \in X \cdot \forall S \in Y \cdot \models_{V_1} t \; \mathit{Sat}\; S \\ \Leftrightarrow & \{\text{predicate calculus}\} \\ & \quad \forall S \in Y \cdot \forall t \in X \cdot \models_{V_1} t \; \mathit{Sat}\; S \\ \Leftrightarrow & \{\text{definition of Specs and } \subseteq\} \\ & \quad Y \subseteq \text{Specs}(X) \end{array}$$
□

Antitone Galois connections have several well-known properties. For instance, (Terms ∘ Specs) and (Specs ∘ Terms) are extensive, increasing and idempotent and therefore closure operators. Freefinement relies on the following properties (their proofs appear in Appendix B):

Corollary 5.3.

5.3.1. $X \subseteq \text{Terms}(\text{Specs}(X))$

[1] Also known as an *order-reversing* or *contravariant* Galois connection.

5.3.2. $\mathrm{Terms}(\mathrm{Specs}(\mathrm{Terms}(\mathrm{Y}))) = \mathrm{Terms}(\mathrm{Y})$

5.3.3. $\mathrm{Specs}(\mathrm{X}) \subseteq \mathrm{Specs}(\mathrm{X}') \Leftrightarrow \mathrm{Terms}(\mathrm{Specs}(\mathrm{X})) \supseteq \mathrm{Terms}(\mathrm{Specs}(\mathrm{X}'))$

5.3.4. $\mathrm{Terms}(\mathrm{Y} \cup \mathrm{Y}') = \mathrm{Terms}(\mathrm{Y}) \cap \mathrm{Terms}(\mathrm{Y}')$

The following auxiliary definition provides a shorthand for the set of all terms of the form $\mathrm{C}(\mathrm{t}_1, \ldots, \mathrm{t}_n)$ where $\mathrm{t}_1 \in \mathrm{X}_1, \ldots, \mathrm{t}_n \in \mathrm{X}_n$:

$$\mathrm{C}(\mathrm{X}_1, \ldots, \mathrm{X}_n) \stackrel{\text{def}}{=} \{\mathrm{C}(\mathrm{t}_1, \ldots, \mathrm{t}_n) \mid \textstyle\bigwedge_{i \in 1..n} \mathrm{t}_i \in \mathrm{X}_i\}$$

For example, it yields a singleton set for nullary constructors:

$$\{\mathrm{C}() \mid \textstyle\bigwedge_{i \in 1..0} \mathrm{t}_i \in \mathrm{X}_i\} = \{\mathrm{C}() \mid \mathit{True}\} = \{\mathrm{C}()\}$$

The semantics of U is given by the function $[\![\_]\!]$ of type $\mathrm{U} \to \mathcal{P}(\mathrm{T})$, i.e. every term in U denotes a set of terms from T:

Definition 5.3 (Semantics of U)**.**

$$\begin{array}{rcl} [\![\mathrm{C}(\mathrm{u}_1, \ldots, \mathrm{u}_n)]\!] & \stackrel{\text{def}}{=} & \mathrm{Terms}(\mathrm{Specs}(\mathrm{C}([\![\mathrm{u}_1]\!], \ldots, [\![\mathrm{u}_n]\!]))) \\ [\![\mathrm{S}]\!] & \stackrel{\text{def}}{=} & \mathrm{Terms}(\{\mathrm{S}\}) \\ [\![\bigsqcup(\mathrm{u}_1, \ldots, \mathrm{u}_n)]\!] & \stackrel{\text{def}}{=} & \bigcap_{i \in 1..n} [\![\mathrm{u}_i]\!] \end{array}$$

If the relation $\models_{\mathsf{V}_1 \_} \mathit{Sat}\ \_$ is well-behaved in a sense that will be made precise later, then $[\![\mathrm{u}]\!]$ has a simple intuitive explanation: it denotes the set of all primitive terms, i.e. terms from T, that refine u. For a term $\mathrm{C}(\mathrm{u}_1, \ldots, \mathrm{u}_n)$, first consider $\mathrm{C}([\![\mathrm{u}_1]\!], \ldots, [\![\mathrm{u}_n]\!])$ – the set of terms of the form $\mathrm{C}(\mathrm{t}_1, \ldots, \mathrm{t}_n)$ where $\mathrm{t}_1 \in [\![\mathrm{u}_1]\!]$ (i.e. t_1 refines u_1) and … and $\mathrm{t}_n \in [\![\mathrm{u}_n]\!]$. All the specifications that all these terms implement are then collected, and any primitive term that satisfies all such specifications refines $\mathrm{C}(\mathrm{u}_1, \ldots, \mathrm{u}_n)$. The primitive terms that refine S are exactly those that satisfy S. Finally, $\bigsqcup(\mathrm{u}_1, \ldots, \mathrm{u}_n)$ is refined by any primitive term that refines all $\mathrm{u}_1, \ldots, \mathrm{u}_n$.

For all u, the set $[\![\mathrm{u}]\!]$ is a fixpoint of $\mathrm{Terms} \circ \mathrm{Specs}$ and hence a closed element:

Lemma 5.4. $\mathrm{Terms}(\mathrm{Specs}([\![\mathrm{u}]\!])) = [\![\mathrm{u}]\!]$

Proof. By induction on the structure of u:

- If u has the form $\mathrm{C}(\mathrm{u}_1, \ldots, \mathrm{u}_n)$ or S, then $[\![\mathrm{u}]\!] = \mathrm{Terms}(\mathrm{Y})$ for some Y and the result follows by Corollary 5.3.2.

- If u has the form $\bigsqcup(\mathrm{u}_1, \ldots, \mathrm{u}_n)$, assume $[\![\mathrm{u}_i]\!] = \mathrm{Terms}(\mathrm{Specs}([\![\mathrm{u}_i]\!]))$ for all $i \in 1..n$. So $[\![\bigsqcup(\mathrm{u}_1, \ldots, \mathrm{u}_n)]\!] = \bigcap_{i\in 1..n} \mathrm{Terms}(\mathrm{Specs}([\![\mathrm{u}_i]\!])) = \mathrm{Terms}(\bigcup_{i\in 1..n} \mathrm{Specs}([\![\mathrm{u}_i]\!]))$ by Corollary 5.3.4, and Corollary 5.3.2 concludes the proof. □

The rest of the chapter introduces further properties of the semantics as needed.

Extending V_1: Preliminaries

The next section will extend V_1 to obtain a formal system V_2 for proving sentences of the form u *Sat* S. The aim is to construct a sound and conservative extension of V_1. Informally, a sound extension of V_1 must have equal or more power:

Definition 5.4 (Sound Extension)**.** $\mathsf{V}_2(\mathbb{K}', \mathbb{S}', \models_{\mathsf{V}_2\text{-}} \mathit{Sat}\ \_)$ is a sound extension of $\mathsf{V}_1(\mathbb{K}, \mathbb{S}, \models_{\mathsf{V}_1\text{-}} \mathit{Sat}\ \_)$ if and only if

1. V_2 uses richer terms and specifications:
   $\mathbb{K} \subseteq \mathbb{K}'$ and $\mathbb{S} \subseteq \mathbb{S}'$

2. V_2 can prove everything that V_1 can prove:
   $\forall \mathrm{t} \in \mathrm{T},\ \mathrm{S} \in \mathbb{S} \cdot \vdash_{\mathsf{V}_1} \mathrm{t}\ \mathit{Sat}\ \mathrm{S} \Rightarrow \vdash_{\mathsf{V}_2} \mathrm{t}\ \mathit{Sat}\ \mathrm{S}$

3. V_2 uses a richer semantics:
   $\forall \mathrm{t} \in \mathrm{T},\ \mathrm{S} \in \mathbb{S} \cdot \models_{\mathsf{V}_2} \mathrm{t}\ \mathit{Sat}\ \mathrm{S} \Rightarrow \models_{\mathsf{V}_1} \mathrm{t}\ \mathit{Sat}\ \mathrm{S}$

4. V_2 is sound:
   $\forall \mathrm{u} \in \mathrm{U},\ \mathrm{S}' \in \mathbb{S}' \cdot \vdash_{\mathsf{V}_2} \mathrm{u}\ \mathit{Sat}\ \mathrm{S}' \Rightarrow \models_{\mathsf{V}_2} \mathrm{u}\ \mathit{Sat}\ \mathrm{S}'$

As a consequence, $\forall \mathrm{t} \in \mathrm{T},\ \mathrm{S} \in \mathbb{S} \cdot \vdash_{\mathsf{V}_2} \mathrm{t}\ \mathit{Sat}\ \mathrm{S} \Rightarrow \models_{\mathsf{V}_1} \mathrm{t}\ \mathit{Sat}\ \mathrm{S}$, which intuitively means that V_2 restricted to $\mathbb{K}$ and $\mathbb{S}$ is sound with respect to the semantics of V_1.

In a sound and conservative extension, the converse of requirement 2 also holds:

Definition 5.5 (Sound and Conservative Extension)**.** A formal system $\mathsf{V}_2(\mathbb{K}', \mathbb{S}', \models_{\mathsf{V}_2\text{-}} \mathit{Sat}\ \_)$ is a sound and conservative extension of $\mathsf{V}_1(\mathbb{K}, \mathbb{S}, \models_{\mathsf{V}_1\text{-}} \mathit{Sat}\ \_)$ if and only if

1. V_2 is a sound extension of V_1.

2. V_1 and V_2 restricted to $\mathbb{K}$ and $\mathbb{S}$ have equal derivability:
   $\forall \mathrm{t} \in \mathrm{T},\ \mathrm{S} \in \mathbb{S} \cdot \vdash_{\mathsf{V}_1} \mathrm{t}\ \mathit{Sat}\ \mathrm{S} \Leftrightarrow \vdash_{\mathsf{V}_2} \mathrm{t}\ \mathit{Sat}\ \mathrm{S}$

Although a sound and conservative extension cannot prove more sentences of the form t *Sat* S, it is still useful for extending the term language and installing a richer semantics. It can also extend the specifications, but the V_2 of the next section will simply use $\mathbb{S}$.

The Extended Formal System V_2

The construction of V_2 starts with the empty set of rules and proceeds in two steps:

1. For each rule of V_1, replace t's by u's and add the resulting rule. This change of metavariables yields the rule forms A_2 and B_2 in V_2:

$$\mathrm{A}_2\ \frac{\mathrm{u}_1\ \mathit{Sat}\ \mathrm{S}_1 \quad \ldots \quad \mathrm{u}_n\ \mathit{Sat}\ \mathrm{S}_n}{\mathrm{C}(\mathrm{u}_1, \ldots, \mathrm{u}_n)\ \mathit{Sat}\ \mathrm{S}}$$
provided $\mathrm{Pred}(\mathrm{C}, \mathrm{S}_1, \ldots, \mathrm{S}_n, \mathrm{S})$.

$$\mathrm{B}_2\ \frac{\mathrm{u}\ \mathit{Sat}\ \mathrm{S}_1 \quad \ldots \quad \mathrm{u}\ \mathit{Sat}\ \mathrm{S}_m}{\mathrm{u}\ \mathit{Sat}\ \mathrm{S}}$$
provided $\mathrm{Pred}(\mathrm{S}_1, \ldots, \mathrm{S}_m, \mathrm{S})$.

2. Add the following rules for spec terms and joins:

$$\textsc{Spec}\ \frac{}{\mathrm{S}\ \mathit{Sat}\ \mathrm{S}}$$

$$\textsc{Join}\ \frac{\mathrm{u}\ \mathit{Sat}\ \mathrm{S}}{\bigsqcup(\ldots, \mathrm{u}, \ldots)\ \mathit{Sat}\ \mathrm{S}}$$

By induction on the derivation, V_1 and V_2 are equivalent with respect to derivability on T, i.e. $\vdash_{\mathsf{V}_1} \mathrm{t}\ \mathit{Sat}\ \mathrm{S} \Leftrightarrow \vdash_{\mathsf{V}_2} \mathrm{t}\ \mathit{Sat}\ \mathrm{S}$. So for V_2 to be a sound and conservative extension of V_1, it will suffice to equip V_2 with a richer semantics and to prove it sound.

The *Sat* relation between U and $\mathbb{S}$ is defined as follows:

Definition 5.6 (Extended Satisfaction)**.**
$\models_{\mathsf{V}_2} \mathrm{u}\ \mathit{Sat}\ \mathrm{S} \stackrel{\text{def}}{=} \forall \mathrm{t} \in [\![\mathrm{u}]\!] \cdot \models_{\mathsf{V}_1} \mathrm{t}\ \mathit{Sat}\ \mathrm{S}$

Furthermore, the U-semantics of t contains t as an element:

Lemma 5.5 (Term Embedding)**.** $\forall \mathrm{t} \in \mathrm{T} \cdot \mathrm{t} \in [\![\mathrm{t}]\!]$

Proof. By induction on the structure of t. Suppose $\mathrm{t} = \mathrm{C}(\mathrm{t}_1, \ldots, \mathrm{t}_n)$ and assume $\mathrm{t}_1 \in [\![\mathrm{t}_1]\!], \ldots, \mathrm{t}_n \in [\![\mathrm{t}_n]\!]$. So $\mathrm{t} \in \mathrm{C}([\![\mathrm{t}_1]\!], \ldots, [\![\mathrm{t}_n]\!])$, which is a subset of $\mathrm{Terms}(\mathrm{Specs}(\mathrm{C}([\![\mathrm{t}_1]\!], \ldots, [\![\mathrm{t}_n]\!])))$ by Corollary 5.3.1. □

Therefore $\models_{\mathsf{V}_2} \mathrm{t}\ \mathit{Sat}\ \mathrm{S} \Rightarrow \models_{\mathsf{V}_1} \mathrm{t}\ \mathit{Sat}\ \mathrm{S}$ holds, and the soundness proof of V_2 establishes that V_2 is a sound and conservative extension of V_1:

Theorem 5.6 (Soundness of V_2)**.** $\vdash_{\mathsf{V}_2} \mathrm{u}\ \mathit{Sat}\ \mathrm{S} \Rightarrow \models_{\mathsf{V}_2} \mathrm{u}\ \mathit{Sat}\ \mathrm{S}$

Proof. By induction on the structure of the derivation:

- For each rule of the form A_2, assume $\mathrm{Pred}(\mathrm{C}, \mathrm{S}_1, \ldots, \mathrm{S}_n, \mathrm{S})$ and assume

$$\begin{array}{c} \forall \mathrm{t}_1 \in [\![\mathrm{u}_1]\!] \cdot \models_{\mathsf{V}_1} \mathrm{t}_1\ \mathit{Sat}\ \mathrm{S}_1 \\ \vdots \\ \forall \mathrm{t}_n \in [\![\mathrm{u}_n]\!] \cdot \models_{\mathsf{V}_1} \mathrm{t}_n\ \mathit{Sat}\ \mathrm{S}_n \end{array}$$

  So $\forall \mathrm{t}_1 \in [\![\mathrm{u}_1]\!], \ldots, \mathrm{t}_n \in [\![\mathrm{u}_n]\!] \cdot \models_{\mathsf{V}_1} \mathrm{C}(\mathrm{t}_1, \ldots, \mathrm{t}_n)\ \mathit{Sat}\ \mathrm{S}$ because the corresponding rule of the form A_1 in V_1 is sound with respect to Definition 5.1.1. So $\mathrm{S} \in \mathrm{Specs}(\mathrm{C}([\![\mathrm{u}_1]\!], \ldots, [\![\mathrm{u}_n]\!]))$ and hence $\forall \mathrm{t} \in \mathrm{Terms}(\mathrm{Specs}(\mathrm{C}([\![\mathrm{u}_1]\!], \ldots, [\![\mathrm{u}_n]\!]))) \cdot \models_{\mathsf{V}_1} \mathrm{t}\ \mathit{Sat}\ \mathrm{S}$.

- For each rule of the form B_2, assume $\mathrm{Pred}(\mathrm{S}_1, \ldots, \mathrm{S}_m, \mathrm{S})$ and assume $\forall \mathrm{t} \in [\![\mathrm{u}]\!] \cdot \models_{\mathsf{V}_1} \mathrm{t}\ \mathit{Sat}\ \mathrm{S}_1 \ \wedge\ \ldots\ \wedge\ \models_{\mathsf{V}_1} \mathrm{t}\ \mathit{Sat}\ \mathrm{S}_m$.

  Now $\forall \mathrm{t} \in [\![\mathrm{u}]\!] \cdot \models_{\mathsf{V}_1} \mathrm{t}\ \mathit{Sat}\ \mathrm{S}$ because the corresponding rule of the form B_1 in V_1 is sound with respect to Definition 5.1.2.

- Spec: $\forall \mathrm{t} \in \mathrm{Terms}(\{\mathrm{S}\}) \cdot \models_{\mathsf{V}_1} \mathrm{t}\ \mathit{Sat}\ \mathrm{S}$ by definition.

- Join: Assume $\forall \mathrm{t} \in [\![\mathrm{u}]\!] \cdot \models_{\mathsf{V}_1} \mathrm{t}\ \mathit{Sat}\ \mathrm{S}$. If $\mathrm{t} \in [\![\bigsqcup(\ldots, \mathrm{u}, \ldots)]\!]$ then $\mathrm{t} \in [\![\mathrm{u}]\!]$ and hence $\models_{\mathsf{V}_1} \mathrm{t}\ \mathit{Sat}\ \mathrm{S}$. □

Extended satisfaction has an alternative characterisation that freefinement will also use:

Lemma 5.7. $\models_{\mathsf{V}_2} \mathrm{u}\ \mathit{Sat}\ \mathrm{S} \ \Leftrightarrow\ \mathrm{S} \in \mathrm{Specs}([\![\mathrm{u}]\!])$

Proof.
$$\begin{array}{ll} & \models_{\mathsf{V}_2} \mathrm{u}\ \mathit{Sat}\ \mathrm{S} \\ \Leftrightarrow & \{\text{definition}\} \\ & \forall \mathrm{t} \in [\![\mathrm{u}]\!] \cdot \models_{\mathsf{V}_1} \mathrm{t}\ \mathit{Sat}\ \mathrm{S} \\ \Leftrightarrow & \{\text{definition of Specs}\} \\ & \mathrm{S} \in \mathrm{Specs}([\![\mathrm{u}]\!]) \end{array}$$
□

5.1.3 System V_2 and Refinement

The next section will construct several refinement systems, or calculi, that are based on V_2. These refinement systems are formal systems for proving sentences of the form $\mathrm{u} \sqsubseteq \mathrm{u}'$. The definition of the refinement relation makes the semantics of refinement precise:

Definition 5.7 (Refinement)**.** $\models \mathrm{u} \sqsubseteq \mathrm{u}' \stackrel{\text{def}}{=} [\![\mathrm{u}]\!] \supseteq [\![\mathrm{u}']\!]$

This definition leads to simple proofs, and is equivalent to several other formulations. The following theorem states one such alternative, and its proof mentions others:

Lemma 5.8 (Equivalent Characterisation of Refinement)**.**
$\models \mathrm{u} \sqsubseteq \mathrm{u}' \;\Leftrightarrow\; \forall \mathrm{S} \cdot \models_{\mathsf{V}_2} \mathrm{u}\ \mathit{Sat}\ \mathrm{S} \Rightarrow \models_{\mathsf{V}_2} \mathrm{u}'\ \mathit{Sat}\ \mathrm{S}$

Proof.
$$
\begin{array}{ll}
 & [\![\mathrm{u}]\!] \supseteq [\![\mathrm{u}']\!] \\
\Leftrightarrow & \{\text{Lemma 5.4}\} \\
 & \mathrm{Terms}(\mathrm{Specs}([\![\mathrm{u}]\!])) \supseteq \mathrm{Terms}(\mathrm{Specs}([\![\mathrm{u}']\!])) \\
\Leftrightarrow & \{\text{Corollary 5.3.3}\} \\
 & \mathrm{Specs}([\![\mathrm{u}]\!]) \subseteq \mathrm{Specs}([\![\mathrm{u}']\!]) \\
\Leftrightarrow & \{\text{definition of } \subseteq\} \\
 & \forall \mathrm{S} \cdot \mathrm{S} \in \mathrm{Specs}([\![\mathrm{u}]\!]) \Rightarrow \mathrm{S} \in \mathrm{Specs}([\![\mathrm{u}']\!]) \\
\Leftrightarrow & \{\text{Lemma 5.7}\} \\
 & \forall \mathrm{S} \cdot \models_{\mathsf{V}_2} \mathrm{u}\ \mathit{Sat}\ \mathrm{S} \Rightarrow \models_{\mathsf{V}_2} \mathrm{u}'\ \mathit{Sat}\ \mathrm{S}
\end{array}
$$
□

If $\models_{\mathsf{V}_1} \_\ \mathit{Sat}\ \_$ is well-behaved, then there is also another explanation for defining $\models \mathrm{u} \sqsubseteq \mathrm{u}'$ as $[\![\mathrm{u}]\!] \supseteq [\![\mathrm{u}']\!]$: u' refines u iff every primitive term that refines u' also refines u. Put differently, u' refines u iff u' constrains the set of eventual primitive terms that refinement can produce to the same or higher degree compared to u. So u can be seen as a placeholder for any of the primitive terms in $[\![\mathrm{u}]\!]$, and the role of refinement is to reduce the uncertainty.

Many examples of refinements will follow later, so here is a small one: a join term implements the least upper bound (join) of its immediate subterms with respect to $\sqsubseteq$, hence the name. In particular:

1. $\forall i \in 1..n \cdot \models \mathrm{u}_i \sqsubseteq \bigsqcup(\mathrm{u}_1, \ldots, \mathrm{u}_n)$
2. If $(\forall i \in 1..n \cdot \models \mathrm{u}_i \sqsubseteq \mathrm{u})$, then $\models \bigsqcup(\mathrm{u}_1, \ldots, \mathrm{u}_n) \sqsubseteq \mathrm{u}$.

The notation $\mathrm{u} \equiv \mathrm{u}'$ is a shorthand for $[\![\mathrm{u}]\!] = [\![\mathrm{u}']\!]$, which is equivalent to $\models \mathrm{u} \sqsubseteq \mathrm{u}' \wedge \models \mathrm{u}' \sqsubseteq \mathrm{u}$.

A refinement system R will be sound if and only if $\vdash_{\mathsf{R}} \mathrm{u} \sqsubseteq \mathrm{u}'$ implies $\models \mathrm{u} \sqsubseteq \mathrm{u}'$. In the next section, freefinement will construct several sound refinement systems where each system R is related to V_2 by the properties Harmony 1 and 2 below.

Harmony 1. If $\vdash_{\mathsf{V}_2} \mathrm{u}\ \mathit{Sat}\ \mathrm{S}$ and $\vdash_{\mathsf{R}} \mathrm{u} \sqsubseteq \mathrm{u}'$, then $\vdash_{\mathsf{V}_2} \mathrm{u}'\ \mathit{Sat}\ \mathrm{S}$.

Intuitively, Harmony 1 says that V_2 contains sufficient machinery to prove the same properties about u' that it could prove about u. In other words, R is not too powerful for V_2.

Harmony 2. If $\vdash_{\mathsf{V}_2} \mathrm{u}\ \mathit{Sat}\ \mathrm{S}$, then $\vdash_{\mathsf{R}} \mathrm{S} \sqsubseteq \mathrm{u}$.

Intuitively, Harmony 2 means that the refinement system R contains sufficient machinery to refine a specification into any term that satisfies it according to V_2. In other words, V_2 is embedded in R and hence R is not too weak.

Harmony 1 is stronger than the converse of Harmony 2:

Theorem 5.9. If V_2 and a refinement system R are related by Harmony 1, then $\vdash_{\mathsf{R}} \mathrm{S} \sqsubseteq \mathrm{u} \ \Rightarrow\ \vdash_{\mathsf{V}_2} \mathrm{u}\ \mathit{Sat}\ \mathrm{S}$.

Proof. Assume $\vdash_{\mathsf{R}} \mathrm{S} \sqsubseteq \mathrm{u}$. Since $\vdash_{\mathsf{V}_2} \mathrm{S}\ \mathit{Sat}\ \mathrm{S}$ by SPEC, it follows from Harmony 1 that $\vdash_{\mathsf{V}_2} \mathrm{u}\ \mathit{Sat}\ \mathrm{S}$. □

A refinement system R is called *harmonic* iff it satisfies Harmony 1 and 2. Harmonic refinement systems interoperate nicely with V_2. In fact, the proofs of Harmony 1 and 2 in the next section are constructive in the sense that they enable proof translation. Given a V_2-proof of $\mathrm{u}\ \mathit{Sat}\ \mathrm{S}$ and an R-proof of $\mathrm{u} \sqsubseteq \mathrm{u}'$, they describe a V_2-proof of $\mathrm{u}'\ \mathit{Sat}\ \mathrm{S}$. Based on a V_2-proof of $\mathrm{u}\ \mathit{Sat}\ \mathrm{S}$, they show how to build an R-proof for $\mathrm{S} \sqsubseteq \mathrm{u}$. Since Harmony 1 is established constructively, given an R-proof of $\mathrm{S} \sqsubseteq \mathrm{u}$, the proof of Theorem 5.9 shows how to build a V_2-proof for $\mathrm{u}\ \mathit{Sat}\ \mathrm{S}$.

The final refinement system that freefinement produces will also have a specific desired form. This form guarantees that refinement proofs are 'linear' developments where terms can be refined in-place. Formally, a refinement system has the desired form if the rules with premises describe either the transitivity or the monotonicity of refinement. All the other rules must be axioms, i.e. without any premise.

5.1.4 The Refinement of Refinement Systems

V_2 can be linearised in a series of steps to obtain a sound and harmonic refinement system of the desired form. At most six steps are necessary according to this presentation – the exact number depends on V_1. The steps make it easy to prove and maintain soundness and harmony, which would otherwise be more complex to establish for the final refinement calculus.

Many of the steps take a previously constructed refinement system and add or remove rules to obtain a new system. If a sound and harmonic refinement system is extended with a rule that is sound and respects Harmony 1, then the resulting system will be sound and harmonic. There is no need to prove Harmony 2 again, because the new refinement system can still derive all sentences that the old one could derive. If a rule is removed from a sound and harmonic refinement system, then the resulting system remains sound and will also be harmonic if it satisfies Harmony 2. A simple way of showing that Harmony 2 still holds is to show that any application of the old rule can be achieved by a combination of rules that remain in the system.

Getting Started: R_1

The first refinement system R_1 is obtained from V_2 by a simple syntactic transformation: each sentence u *Sat* S becomes $\mathrm{S} \sqsubseteq \mathrm{u}$. R_1 has rules of the form A_3 and B_3, a Spec rule and also a Join rule if join terms were needed:

$$\mathrm{A}_3\ \frac{\mathrm{S}_1 \sqsubseteq \mathrm{u}_1 \quad \dots \quad \mathrm{S}_n \sqsubseteq \mathrm{u}_n}{\mathrm{S} \sqsubseteq \mathrm{C}(\mathrm{u}_1, \dots, \mathrm{u}_n)}$$
provided $\mathrm{Pred}(\mathrm{C}, \mathrm{S}_1, \dots, \mathrm{S}_n, \mathrm{S})$.

$$\mathrm{B}_3\ \frac{\mathrm{S}_1 \sqsubseteq \mathrm{u} \quad \dots \quad \mathrm{S}_m \sqsubseteq \mathrm{u}}{\mathrm{S} \sqsubseteq \mathrm{u}}$$
provided $\mathrm{Pred}(\mathrm{S}_1, \dots, \mathrm{S}_m, \mathrm{S})$.

$$\textsc{Spec}\ \frac{}{\mathrm{S} \sqsubseteq \mathrm{S}}$$

$$\textsc{Join}\ \frac{\mathrm{S} \sqsubseteq \mathrm{u}}{\mathrm{S} \sqsubseteq \bigsqcup(\dots, \mathrm{u}, \dots)}$$

V_2 and R_1 are isomorphic: a proof of u *Sat* S in V_2 corresponds to a proof of $\mathrm{S} \sqsubseteq \mathrm{u}$ in R_1 and vice versa, so $\vdash_{\mathsf{V}_2} \mathrm{u}\ \mathit{Sat}\ \mathrm{S} \Leftrightarrow \vdash_{\mathsf{R}_1} \mathrm{S} \sqsubseteq \mathrm{u}$. The soundness proof of R_1 relies on the following equivalence:

Lemma 5.10. $\models_{\mathsf{V}_2} \mathrm{u}\ \mathit{Sat}\ \mathrm{S} \Leftrightarrow \models \mathrm{S} \sqsubseteq \mathrm{u}$

Proof.
$$\begin{array}{ll} & \models_{\mathsf{V}_2} \mathrm{u}\ \mathit{Sat}\ \mathrm{S} \\ \Leftrightarrow & \{\text{Lemma 5.7}\} \\ & \{\mathrm{S}\} \subseteq \mathrm{Specs}([\![\mathrm{u}]\!]) \\ \Leftrightarrow & \{\text{Lemma 5.2}\} \\ & [\![\mathrm{u}]\!] \subseteq \mathrm{Terms}(\{\mathrm{S}\}) \end{array}$$
□

Theorem 5.11 (Soundness of R_1)**.** $\vdash_{\mathsf{R}_1} \mathrm{u} \sqsubseteq \mathrm{u}' \Rightarrow \models \mathrm{u} \sqsubseteq \mathrm{u}'$

Proof. If $\vdash_{\mathsf{R}_1} \mathrm{u} \sqsubseteq \mathrm{u}'$, then u has the form S and $\vdash_{\mathsf{V}_2} \mathrm{u}'$ *Sat* S. The soundness of V_2 implies $\models_{\mathsf{V}_2} \mathrm{u}'$ *Sat* S, and Lemma 5.10 in turn implies $\models \mathrm{S} \sqsubseteq \mathrm{u}'$. □

Theorem 5.12. R_1 is harmonic.

Proof. Harmony 2 holds by construction. For Harmony 1, assume $\vdash_{\mathsf{R}_1} \mathrm{u} \sqsubseteq \mathrm{u}'$. Then u has the form S″ and $\vdash_{\mathsf{V}_2} \mathrm{u}'$ *Sat* S″ by construction. That $\vdash_{\mathsf{V}_2} \mathrm{u}$ *Sat* S′ (i.e. $\vdash_{\mathsf{V}_2} \mathrm{S}''$ *Sat* S′) implies $\vdash_{\mathsf{V}_2} \mathrm{u}'$ *Sat* S′ for all S′ follows by induction on the derivation of S″ *Sat* S′:

- Spec: S′ and S″ are the same. Since $\vdash_{\mathsf{V}_2} \mathrm{u}'$ *Sat* S″, it holds that $\vdash_{\mathsf{V}_2} \mathrm{u}'$ *Sat* S′.
- For each rule of the form B_2: S and S′ are the same. Assume Pred(S_1, …, S_m, S), $\vdash_{\mathsf{V}_2} \mathrm{u}$ *Sat* S_1, …, $\vdash_{\mathsf{V}_2} \mathrm{u}$ *Sat* S_m, and by the induction hypothesis also $\vdash_{\mathsf{V}_2} \mathrm{u}'$ *Sat* S_1, …, $\vdash_{\mathsf{V}_2} \mathrm{u}'$ *Sat* S_m. So the rule being considered is applicable and $\vdash_{\mathsf{V}_2} \mathrm{u}'$ *Sat* S. Hence $\vdash_{\mathsf{V}_2} \mathrm{u}'$ *Sat* S′. □

Note: if V_2 has only rules of the form A_2 where $n = 0$ and/or rules of the form B_2 where $m = 0$, then R_1 is a refinement system of the desired form and freefinement stops.

Adding Transitivity: R_2

The refinement system R_2 extends R_1 with the rule Trans which states that refinement is transitive:

$$\textsc{Trans}\ \frac{\mathrm{u}_1 \sqsubseteq \mathrm{u}_2 \qquad \mathrm{u}_2 \sqsubseteq \mathrm{u}_3}{\mathrm{u}_1 \sqsubseteq \mathrm{u}_3}$$

Trans is sound because $\supseteq$ is transitive, and it maintains Harmony 1 since implication is transitive. So R_2 is sound and harmonic.

Simplification: R_3

The presence of Spec and Trans in R_2 allows the simplification of rules of the form B_3 with $m = 1$:

$$\mathrm{B}_3\ \frac{\mathrm{S}_1 \sqsubseteq \mathrm{u}}{\mathrm{S} \sqsubseteq \mathrm{u}}$$
provided Pred(S_1, S).

For an arbitrary rule of this form, consider the derivation

$$\textsc{Spec}\ \frac{}{\mathrm{S}_1 \sqsubseteq \mathrm{S}_1}$$
$$\mathrm{B}_3\ \frac{\mathrm{S}_1 \sqsubseteq \mathrm{S}_1}{\mathrm{S} \sqsubseteq \mathrm{S}_1}$$
provided $\mathrm{Pred}(\mathrm{S}_1, \mathrm{S})$.

By virtue of having been derived, the new rule

$$\mathrm{B}_3\ \frac{}{\mathrm{S} \sqsubseteq \mathrm{S}_1}$$
provided $\mathrm{Pred}(\mathrm{S}_1, \mathrm{S})$.

is sound and maintains Harmony 1, and can therefore be added to R_2 to obtain a sound and harmonic refinement system. In fact, it can replace the old version without breaking Harmony 2, since removing the old version will not decrease the derivable set of sentences: every application of the old B_3 can be changed into:

$$\textsc{Trans}\ \frac{\mathrm{B}_3\ \dfrac{}{\mathrm{S} \sqsubseteq \mathrm{S}_1} \qquad \mathrm{S}_1 \sqsubseteq \mathrm{u}}{\mathrm{S} \sqsubseteq \mathrm{u}}$$

since $\mathrm{Pred}(\mathrm{S}_1, \mathrm{S})$ is guaranteed.

The refinement system R_3 is the same as R_2, except that the rules of the form B_3 with $m = 1$ are replaced by their simplified versions. R_3 is sound and harmonic.

Note: if V_2 has only rules of the form A_2 where $n = 0$ and rules of the form B_2 where $m \leq 1$, then R_3 is a refinement system of the desired form and freefinement stops.

Adding Monotonicity: R_4

All the constructors of U are monotone with respect to $\sqsubseteq$, i.e. the following rules are sound:

$$\text{C-}i\ \frac{\mathrm{u}_i \sqsubseteq \mathrm{u}'_i}{\mathrm{C}(\mathrm{u}_1, \ldots, \mathrm{u}_i, \ldots, \mathrm{u}_n) \sqsubseteq \mathrm{C}(\mathrm{u}_1, \ldots, \mathrm{u}'_i, \ldots, \mathrm{u}_n)}$$

$$\textsc{Join}\text{-}i\ \frac{\mathrm{u}_i \sqsubseteq \mathrm{u}'_i}{\bigsqcup(\mathrm{u}_1, \ldots, \mathrm{u}_i, \ldots, \mathrm{u}_n) \sqsubseteq \bigsqcup(\mathrm{u}_1, \ldots, \mathrm{u}'_i, \ldots, \mathrm{u}_n)}$$

Moreover, these rules maintain harmony:

Lemma 5.13. C-i maintains Harmony 1.

Proof. Assume $\forall \mathrm{S}' \cdot \vdash_{\mathsf{V}_2} \mathrm{u}_i\ \mathit{Sat}\ \mathrm{S}' \Rightarrow \vdash_{\mathsf{V}_2} \mathrm{u}'_i\ \mathit{Sat}\ \mathrm{S}'$. That $\forall \mathrm{S} \cdot \vdash_{\mathsf{V}_2} \mathrm{C}(\mathrm{u}_1, \ldots, \mathrm{u}_i, \ldots, \mathrm{u}_n)\ \mathit{Sat}\ \mathrm{S} \Rightarrow \vdash_{\mathsf{V}_2} \mathrm{C}(\mathrm{u}_1, \ldots, \mathrm{u}'_i, \ldots, \mathrm{u}_n)\ \mathit{Sat}\ \mathrm{S}$ follows by induction on the derivation of $\mathrm{C}(\mathrm{u}_1, \ldots, \mathrm{u}_i, \ldots, \mathrm{u}_n)\ \mathit{Sat}\ \mathrm{S}$:

- A_2: Suppose $\vdash_{\mathsf{V}_2} u_j$ *Sat* S_j for $j \in 1..n$, and also suppose Pred(C, S_1, …, S_n, S) holds. Since $\vdash_{\mathsf{V}_2} u'_i$ *Sat* S_i, the same rule A_2 can be applied to derive C(u_1, …, u'_i, …, u_n) *Sat* S.

- B_2: Suppose $\vdash_{\mathsf{V}_2} C(u_1, \ldots, u_i, \ldots, u_n)$ *Sat* S_j for $j \in 1..m$, and suppose Pred(S_1, …, S_m, S). The induction hypothesis is the assumption $\vdash_{\mathsf{V}_2} C(u_1, \ldots, u'_i, \ldots, u_n)$ *Sat* S_j for $j \in 1..m$. Since Pred(S_1, …, S_m, S), the same rule B_2 is applicable and hence $\vdash_{\mathsf{V}_2} C(u_1, \ldots, u'_i, \ldots, u_n)$ *Sat* S. □

Lemma 5.14. Join-i maintains Harmony 1.

Proof. Assume $\forall S' \cdot \vdash_{\mathsf{V}_2} u_i$ *Sat* $S' \Rightarrow \vdash_{\mathsf{V}_2} u'_i$ *Sat* S'. That $\forall S \cdot \vdash_{\mathsf{V}_2} \bigsqcup(u_1, \ldots, u_i, \ldots, u_n)$ *Sat* S $\Rightarrow \vdash_{\mathsf{V}_2} \bigsqcup(u_1, \ldots, u'_i, \ldots, u_n)$ *Sat* S follows by induction on the derivation of $\bigsqcup(u_1, \ldots, u_i, \ldots, u_n)$ *Sat* S:

- Join: Suppose u_j *Sat* S was the premise for some $j \in 1..n$. If $j \neq i$, then apply Join to the premise u_j *Sat* S to derive the required $\bigsqcup(u_1, \ldots, u'_i, \ldots, u_n)$ *Sat* S. If $j = i$, then by assumption $\vdash_{\mathsf{V}_2} u'_i$ *Sat* S holds, and the result follows by Join.

- B_2: Suppose $\vdash_{\mathsf{V}_2} \bigsqcup(u_1, \ldots, u_i, \ldots, u_n)$ *Sat* S_j for $j \in 1..m$, and suppose Pred(S_1, …, S_m, S). The induction hypothesis is the assumption $\vdash_{\mathsf{V}_2} \bigsqcup(u_1, \ldots, u'_i, \ldots, u_n)$ *Sat* S_j for $j \in 1..m$. Since Pred(S_1, …, S_m, S), the same rule B_2 is applicable and hence $\vdash_{\mathsf{V}_2} \bigsqcup(u_1, \ldots, u'_i, \ldots, u_n)$ *Sat* S. □

Let the notation v[u] denote a term in U whose parse tree is factored into two parts: a core tree v with a 'hole' where the sub-tree for u fits. The rule Mono packages C-i and Join-i in a single convenient form:

$$\textsc{Mono}\ \frac{u \sqsubseteq u'}{v[u] \sqsubseteq v[u']}$$

Informally, the rule Mono allows in-place refinement: if u_0 can be factored as v[u], and u′ refines u, then v[u′] refines u_0.

Mono is sound and maintains harmony because C-i and Join-i are sound and maintain harmony. The refinement system R_4 extends R_3 with Mono. It is sound and harmonic.

Simplification: R_5

The rule Mono makes it possible to simplify:

- The Join rule:

$$\textsc{Join}\ \frac{\mathrm{S} \sqsubseteq \mathrm{u}}{\mathrm{S} \sqsubseteq \bigsqcup(\ldots, \mathrm{u}, \ldots)}$$

- Rules of the form A_3 with $n \geq 1$:

$$\mathrm{A}_3\ \frac{\mathrm{S}_1 \sqsubseteq \mathrm{u}_1 \quad \ldots \quad \mathrm{S}_n \sqsubseteq \mathrm{u}_n}{\mathrm{S} \sqsubseteq \mathrm{C}(\mathrm{u}_1, \ldots, \mathrm{u}_n)}$$
$$\text{provided } \mathrm{Pred}(\mathrm{C}, \mathrm{S}_1, \ldots, \mathrm{S}_n, \mathrm{S}).$$

Consider the derivation:

$$\textsc{Join}\ \frac{\textsc{Spec}\ \dfrac{}{\mathrm{S} \sqsubseteq \mathrm{S}}}{\mathrm{S} \sqsubseteq \bigsqcup(\ldots, \mathrm{S}, \ldots)}$$

By virtue of having been derived, the simplified rule

$$\textsc{Join}\ \frac{}{\mathrm{S} \sqsubseteq \bigsqcup(\ldots, \mathrm{S}, \ldots)}$$

is sound and respects Harmony 1. It can replace the old version of Join without decreasing derivability, because any application of the old version can be achieved by:

$$\textsc{Trans}\ \frac{\textsc{Join}\ \dfrac{}{\mathrm{S} \sqsubseteq \bigsqcup(\ldots, \mathrm{S}, \ldots)} \qquad \textsc{Mono}\ \dfrac{\mathrm{S} \sqsubseteq \mathrm{u}}{\bigsqcup(\ldots, \mathrm{S}, \ldots) \sqsubseteq \bigsqcup(\ldots, \mathrm{u}, \ldots)}}{\mathrm{S} \sqsubseteq \bigsqcup(\ldots, \mathrm{u}, \ldots)}$$

Likewise, for each rule of the form A_3, the derived rule

$$\mathrm{A}_3\ \frac{}{\mathrm{S} \sqsubseteq \mathrm{C}(\mathrm{S}_1, \ldots, \mathrm{S}_n)}$$
$$\text{provided } \mathrm{Pred}(\mathrm{C}, \mathrm{S}_1, \ldots, \mathrm{S}_n, \mathrm{S}).$$

is sound and respects harmony. It makes the old version redundant, since any application of the old rule can be replaced by:

$$\mathrm{A}_3\ \frac{}{\mathrm{S} \sqsubseteq \mathrm{C}(\mathrm{S}_1, \ldots, \mathrm{S}_n)}$$
$$\mathrm{E}_1$$
$$\ddots$$
$$\mathrm{E}_n$$

where E_i is given by:

$$\textsc{Trans}\ \frac{\mathrm{S} \sqsubseteq \mathrm{C}(\mathrm{u}_1, \ldots, \mathrm{u}_{i-1}, \mathrm{S}_i, \ldots, \mathrm{S}_n) \qquad \mathrm{P}_i}{\mathrm{S} \sqsubseteq \mathrm{C}(\mathrm{u}_1, \ldots, \mathrm{u}_i, \mathrm{S}_{i+1}, \ldots, \mathrm{S}_n)}$$

and P_i is the proof tree:

$$\textsc{Mono}\ \frac{\mathrm{S}_i \sqsubseteq \mathrm{u}_i}{\mathrm{C}(\mathrm{u}_1, \ldots, \mathrm{u}_{i-1}, \mathrm{S}_i, \ldots, \mathrm{S}_n) \sqsubseteq \mathrm{C}(\mathrm{u}_1, \ldots, \mathrm{u}_i, \mathrm{S}_{i+1}, \ldots, \mathrm{S}_n)}$$

Apart from these simplifications, the refinement system R_5 is the same as R_4. It is sound and harmonic.

Note: if V_2 does not include rules of the form B_2 where $m > 1$, then R_5 has the desired form and freefinement stops.

Wrapping Up: R_6

It remains to simplify rules of the form B_3 with $m > 1$:

$$\mathrm{B}_3\ \frac{\mathrm{S}_1 \sqsubseteq \mathrm{u} \qquad \ldots \qquad \mathrm{S}_m \sqsubseteq \mathrm{u}}{\mathrm{S} \sqsubseteq \mathrm{u}}$$
$$\text{provided } \mathrm{Pred}(\mathrm{S}_1, \ldots, \mathrm{S}_m, \mathrm{S}).$$

If $\mathrm{Pred}(\mathrm{S}_1, \ldots, \mathrm{S}_m, \mathrm{S})$, then R_5 can derive:

$$\begin{array}{c} \textsc{Join} \dfrac{}{\mathrm{S}_1 \sqsubseteq \bigsqcup(\mathrm{S}_1, \ldots, \mathrm{S}_m)} \quad \cdots \quad \textsc{Join} \dfrac{}{\mathrm{S}_m \sqsubseteq \bigsqcup(\mathrm{S}_1, \ldots, \mathrm{S}_m)} \\ \mathrm{B}_3\ \dfrac{}{\mathrm{S} \sqsubseteq \bigsqcup(\mathrm{S}_1, \ldots, \mathrm{S}_m)} \end{array}$$

The derived rule

$$\mathrm{B}_3\ \frac{}{\mathrm{S} \sqsubseteq \bigsqcup(\mathrm{S}_1, \ldots, \mathrm{S}_m)}$$
$$\text{provided } \mathrm{Pred}(\mathrm{S}_1, \ldots, \mathrm{S}_m, \mathrm{S}).$$

is therefore sound and respects Harmony 1. Together with the rule:

$$\textsc{Unjoin}\ \frac{}{\bigsqcup(\mathrm{u}, \ldots, \mathrm{u}) \sqsubseteq \mathrm{u}}$$

which is trivially sound and respects Harmony 1, it can replace the old B_3 because any application of the old rule can be rewritten as:

$$\textsc{Trans}\;\dfrac{\dfrac{\dfrac{}{\mathrm{S} \sqsubseteq \bigsqcup(\mathrm{S}_1, \ldots, \mathrm{S}_m)}\,\mathrm{B}_3}{\mathrm{F}_1} \quad \ddots \quad \mathrm{F}_m \qquad \mathrm{G}}{\mathrm{S} \sqsubseteq \mathrm{u}}$$

where G is UNJOIN, F_i is given by:

$$\textsc{Trans}\;\dfrac{\mathrm{S} \sqsubseteq \bigsqcup(\overbrace{\mathrm{u}, \ldots, \mathrm{u}}^{i-1}, \mathrm{S}_i, \ldots, \mathrm{S}_m) \qquad \mathrm{Q}_i}{\mathrm{S} \sqsubseteq \bigsqcup(\mathrm{u}, \ldots, \mathrm{u}, \mathrm{S}_{i+1}, \ldots, \mathrm{S}_m)}$$

and Q_i is the proof tree:

$$\textsc{Mono}\;\dfrac{\mathrm{S}_i \sqsubseteq \mathrm{u}}{\bigsqcup(\mathrm{u}, \ldots, \mathrm{u}, \mathrm{S}_i, \ldots, \mathrm{S}_m) \sqsubseteq \bigsqcup(\underbrace{\mathrm{u}, \ldots, \mathrm{u}}_{i}, \mathrm{S}_{i+1}, \ldots, \mathrm{S}_m)}$$

R_6 is the same as R_5, except that it includes UNJOIN and replaces rules of the form B_3 where $m > 1$ with their simplified versions. R_6 is sound, harmonic and of the desired form.

5.1.5 Discussion

R_6 can be made more powerful in several ways. For example, the following generalisation of JOIN is sound and preserves Harmony 1:

$$\textsc{Join}'\;\dfrac{}{\mathrm{u} \sqsubseteq \bigsqcup(\ldots, \mathrm{u}, \ldots)}$$

The same holds for the reflexivity of refinement, which generalises SPEC, and other rules such as UNNEST:

$$\textsc{Unnest}\;\dfrac{}{\bigsqcup(\mathrm{u}_1, \ldots, \mathrm{u}_n) \sqsubseteq \bigsqcup(\mathrm{u}_1, \ldots, \mathrm{u}_{i-1}, \mathrm{u}'_1, \ldots, \mathrm{u}'_m, \mathrm{u}_{i+1}, \ldots, \mathrm{u}_n)}$$

provided $1 \leq i \leq n$ and $\mathrm{u}_i = \bigsqcup(\mathrm{u}'_1, \ldots, \mathrm{u}'_m)$.

In specific applications of freefinement, it might also be useful to add derived rules to R_6. Examples of this will follow later.

Freefinement assumes as little as possible about $\models_{\mathsf{V}_1}$ *Sat* and is consequently very generic. As one might expect, additional assumptions can help to construct more powerful refinement systems. For example, suppose 'plus' is a constructor that is commutative in the sense that

$$\forall t_1, t_2 \in T, S \in \mathbb{S} \cdot \models_{V_1} plus(t_1, t_2)\ \mathit{Sat}\ S \ \Leftrightarrow\ \models_{V_1} plus(t_2, t_1)\ \mathit{Sat}\ S$$

Then $Specs(plus([\![u_1]\!], [\![u_2]\!])) = Specs(plus([\![u_2]\!], [\![u_1]\!]))$ because

$$\begin{array}{ll} & S \in Specs(plus([\![u_1]\!], [\![u_2]\!])) \\ \Leftrightarrow & \forall t_1 \in [\![u_1]\!], t_2 \in [\![u_2]\!] \cdot \models_{V_1} plus(t_1, t_2)\ \mathit{Sat}\ S \\ \Leftrightarrow & \forall t_1 \in [\![u_1]\!], t_2 \in [\![u_2]\!] \cdot \models_{V_1} plus(t_2, t_1)\ \mathit{Sat}\ S \\ \Leftrightarrow & S \in Specs(plus([\![u_2]\!], [\![u_1]\!])) \end{array}$$

So $[\![plus(u_1, u_2)]\!] = [\![plus(u_2, u_1)]\!]$ and therefore the refinement rule $plus(u_1, u_2) \equiv plus(u_2, u_1)$ is sound. Depending on the rules of V_1, it might also preserve harmony.

As mentioned before, the semantic function $[\![\_]\!]$ and the refinement order $\sqsubseteq$ have nice interpretations when $\models_{V_1} \_\ \mathit{Sat}\ \_$ is well-behaved. Here is the definition:

Definition 5.8 (Well-behavedness)**.** $\models_{V_1} \_\ \mathit{Sat}\ \_$ is well-behaved iff
$\forall C \in \mathbb{K}, t_1, \ldots, t_n \in T, S \in \mathbb{S} \cdot \models_{V_1} C(t_1, \ldots, t_n)\ \mathit{Sat}\ S \Rightarrow$
$\forall t \in C(Terms(Specs(\{t_1\})), \ldots, Terms(Specs(\{t_n\}))) \cdot \models_{V_1} t\ \mathit{Sat}\ S$

There is also an alternative characterisation of well-behavedness:

Lemma 5.15. $\models_{V_1} \_\ \mathit{Sat}\ \_$ is well-behaved iff
$\forall C \in \mathbb{K}, t_1, \ldots, t_n \in T \cdot Terms(Specs(\{C(t_1, \ldots, t_n)\})) =$
$Terms(Specs(C(Terms(Specs(\{t_1\})), \ldots, Terms(Specs(\{t_n\})))))$

Proof. $t_i \in Terms(Specs(\{t_i\}))$ for $i \in 1..n$ by Corollary 5.3.1, so $\{C(t_1, \ldots, t_n)\} \subseteq C(Terms(Specs(\{t_1\})), \ldots, Terms(Specs(\{t_n\})))$. Hence by Corollary B.1.3 in Appendix B, $Specs(\{C(t_1, \ldots, t_n)\}) \supseteq$ $Specs(C(Terms(Specs(\{t_1\})), \ldots, Terms(Specs(\{t_n\}))))$.

Therefore:

$\models_{V_1} \_\ \mathit{Sat}\ \_$ is well-behaved
$\Leftrightarrow$
$\forall C \in \mathbb{K}, t_1, \ldots, t_n \in T, S \in \mathbb{S} \cdot \models_{V_1} C(t_1, \ldots, t_n)\ \mathit{Sat}\ S \Rightarrow$
$\forall t \in C(Terms(Specs(\{t_1\})), \ldots, Terms(Specs(\{t_n\}))) \cdot \models_{V_1} t\ \mathit{Sat}\ S$
$\Leftrightarrow$
$\forall C \in \mathbb{K}, t_1, \ldots, t_n \in T, S \in \mathbb{S} \cdot S \in Specs(\{C(t_1, \ldots, t_n)\}) \Rightarrow$
$S \in Specs(C(Terms(Specs(\{t_1\})), \ldots, Terms(Specs(\{t_n\}))))$
$\Leftrightarrow$
$\forall C \in \mathbb{K}, t_1, \ldots, t_n \in T \cdot Specs(\{C(t_1, \ldots, t_n)\}) \subseteq$
$Specs(C(Terms(Specs(\{t_1\})), \ldots, Terms(Specs(\{t_n\}))))$
$\Leftrightarrow$ {by the reasoning above}
$\forall C \in \mathbb{K}, t_1, \ldots, t_n \in T \cdot Specs(\{C(t_1, \ldots, t_n)\}) =$
$Specs(C(Terms(Specs(\{t_1\})), \ldots, Terms(Specs(\{t_n\}))))$

The result then follows by Corollary 5.3.3. □

Freefinement does not require well-behavedness of $\models_{\mathsf{V}_1 \_}\ \mathit{Sat}\ \_$, but the next theorem shows that the intuitions behind the definitions are simple when $\models_{\mathsf{V}_1 \_}\ \mathit{Sat}\ \_$ is well-behaved. For example, Theorem 5.16.3 says that $[\![\mathrm{u}]\!]$ is the set of all primitive terms that refine u.

Theorem 5.16. If $\models_{\mathsf{V}_1 \_}\ \mathit{Sat}\ \_$ is well-behaved, then

5.16.1. $\forall \mathrm{t} \in \mathrm{T} \cdot [\![\mathrm{t}]\!] = \mathrm{Terms}(\mathrm{Specs}(\{\mathrm{t}\}))$

5.16.2. $\forall \mathrm{t} \in \mathrm{T}, \mathrm{S} \in \mathbb{S} \cdot \models_{\mathsf{V}_1} \mathrm{t}\ \mathit{Sat}\ \mathrm{S} \ \Leftrightarrow\ \models_{\mathsf{V}_2} \mathrm{t}\ \mathit{Sat}\ \mathrm{S}$

5.16.3. $\forall \mathrm{t} \in \mathrm{T}, \mathrm{u} \in \mathrm{U} \cdot \mathrm{t} \in [\![\mathrm{u}]\!] \ \Leftrightarrow\ \models \mathrm{u} \sqsubseteq \mathrm{t}$

Proof.

5.16.1. By induction on the structure of t. Suppose $\mathrm{t} = \mathrm{C}(\mathrm{t}_1, \ldots, \mathrm{t}_n)$ and assume $[\![\mathrm{t}_i]\!] = \mathrm{Terms}(\mathrm{Specs}(\{\mathrm{t}_i\}))$ for $i \in 1..n$. Then:
$[\![\mathrm{t}]\!] = \mathrm{Terms}(\mathrm{Specs}(\mathrm{C}([\![\mathrm{t}_1]\!], \ldots, [\![\mathrm{t}_n]\!])))$
$\quad = \quad$ {induction hypothesis}
$\mathrm{Terms}(\mathrm{Specs}(\mathrm{C}(\mathrm{Terms}(\mathrm{Specs}(\{\mathrm{t}_1\})), \ldots, \mathrm{Terms}(\mathrm{Specs}(\{\mathrm{t}_n\})))))$
$\quad = \quad$ {Lemma 5.15}
$\mathrm{Terms}(\mathrm{Specs}(\{\mathrm{C}(\mathrm{t}_1, \ldots, \mathrm{t}_n)\})) = \mathrm{Terms}(\mathrm{Specs}(\{\mathrm{t}\}))$.

5.16.2.
$\quad \models_{\mathsf{V}_1} \mathrm{t}\ \mathit{Sat}\ \mathrm{S}$
$\Leftrightarrow$ {definition of Specs}
$\quad \mathrm{S} \in \mathrm{Specs}(\{\mathrm{t}\})$
$\Leftrightarrow$ {Corollary B.1.7 in Appendix B}
$\quad \mathrm{S} \in \mathrm{Specs}(\mathrm{Terms}(\mathrm{Specs}(\{\mathrm{t}\})))$
$\Leftrightarrow$ {Theorem 5.16.1}
$\quad \mathrm{S} \in \mathrm{Specs}([\![\mathrm{t}]\!])$
$\Leftrightarrow$ {Lemma 5.7}
$\quad \models_{\mathsf{V}_2} \mathrm{t}\ \mathit{Sat}\ \mathrm{S}$

5.16.3. The $\Leftarrow$ proof is trivial since $\mathrm{t} \in [\![\mathrm{t}]\!]$. For $\Rightarrow$, assume $\{\mathrm{t}\} \subseteq [\![\mathrm{u}]\!]$. Then $\mathrm{Terms}(\mathrm{Specs}(\{\mathrm{t}\})) \subseteq \mathrm{Terms}(\mathrm{Specs}([\![\mathrm{u}]\!]))$ by Corollary B.1.5 in Appendix B, and $[\![\mathrm{t}]\!] \subseteq [\![\mathrm{u}]\!]$ by Theorem 5.16.1 and Lemma 5.4. □

Whether $\models_{\mathsf{V}_1 \_}\ \mathit{Sat}\ \_$ is well-behaved depends partly on the expressivity of specifications. For example, suppose

$$\mathbb{K} = \{\mathrm{x} := \mathrm{e} \mid \mathrm{e} \text{ is an arithmetic expression}\} \cup \{\_ \fatsemi \_\}$$

i.e. there is a nullary constructor x := e for all arithmetic expressions e, and a binary constructor for sequential composition. Suppose $\mathbb{S} = \{\text{Even\_x}\}$, and $\models_{\mathsf{V}_1}$t *Sat* Even_x holds iff, if t is executed in any state where x is even, then x is even in every resulting state. So $\models_{\mathsf{V}_1}$x := x + 1 ⨟ x := x + 1 *Sat* Even_x, but it is not the case that $\models_{\mathsf{V}_1}$x := x + 1 *Sat* Even_x. In fact, x := x + 1 does not satisfy any specification. This implies that Terms(Specs({x := x + 1})) = T, so x := 1 ⨟ x := 1 ∈ Terms(Specs({x := x + 1})) ⨟ Terms(Specs({x := x + 1})). But $\models_{\mathsf{V}_1}$x := 1 ⨟ x := 1 *Sat* Even_x does not hold, hence $\models_{\mathsf{V}_1\_}$ *Sat* _ is not well-behaved.

Even though $\models_{\mathsf{V}_1\_}$ *Sat* _ is not well-behaved, it is still possible to have inference rules that are amenable to freefinement, for example:

$$1\ \frac{}{\text{x := e } \textit{Sat}\ \text{Even\_x}} \quad \text{provided e} \in \{\ldots, \text{-2, 0, 2, } \ldots\}. \qquad 2\ \frac{\text{t } \textit{Sat}\ \text{Even\_x} \qquad \text{t}' \textit{ Sat}\ \text{Even\_x}}{\text{t} ⨟ \text{t}' \textit{ Sat}\ \text{Even\_x}}$$

If $\mathbb{S}$ is instead a set of specifications of the form $[P, Q]$, where P is a precondition and Q a postcondition, and

$$\models_{\mathsf{V}_1}\text{t} ⨟ \text{t}' \textit{ Sat } [P, Q] \quad \Leftrightarrow \quad \exists R \cdot \models_{\mathsf{V}_1}\text{t } \textit{Sat } [P, R] \wedge \models_{\mathsf{V}_1}\text{t}' \textit{ Sat } [R, Q]$$

then it is easy to show that this $\models_{\mathsf{V}_1\_}$ *Sat* _ is well-behaved.

The completeness of V_1 is a sufficient condition for the well-behavedness of $\models_{\mathsf{V}_1\_}$ *Sat* _:

Theorem 5.17. If V_1 is complete, then $\models_{\mathsf{V}_1\_}$ *Sat* _ is well-behaved.

Proof. If V_1 is complete, then $\models_{\mathsf{V}_1}\text{C}(\text{t}_1, \ldots, \text{t}_n)$ *Sat* S $\Leftrightarrow$ $\vdash_{\mathsf{V}_1}\text{C}(\text{t}_1, \ldots, \text{t}_n)$ *Sat* S. The well-behavedness of $\models_{\mathsf{V}_1\_}$ *Sat* _ follows by induction on the derivation of $\text{C}(\text{t}_1, \ldots, \text{t}_n)$ *Sat* S:

- For each rule of the form A_1, assume $\text{Pred}(\text{C}, \text{S}_1, \ldots, \text{S}_n, \text{S})$ and $\text{S}_i \in \text{Specs}(\{\text{t}_i\})$ for all $i \in 1..n$. So $\forall \text{t}'_i \in \text{Terms}(\text{Specs}(\{\text{t}_i\}))\cdot \models_{\mathsf{V}_1}\text{t}'_i$ *Sat* S_i for all $i \in 1..n$. The rule is sound with respect to Definition 5.1.1, hence $\forall \text{t} \in \text{C}(\text{Terms}(\text{Specs}(\{\text{t}_1\})), \ldots, \text{Terms}(\text{Specs}(\{\text{t}_n\})))\cdot\models_{\mathsf{V}_1}\text{t}$ *Sat* S.

- For each rule of the form B_1, assume $\text{Pred}(\text{S}_1, \ldots, \text{S}_m, \text{S})$ and $\forall \text{t} \in \text{C}(\text{Terms}(\text{Specs}(\{\text{t}_1\})), \ldots, \text{Terms}(\text{Specs}(\{\text{t}_n\})))\cdot\models_{\mathsf{V}_1}\text{t}$ *Sat* S_i for all $i \in 1..m$. The rule is sound w.r.t. Definition 5.1.2, so $\forall \text{t} \in \text{C}(\text{Terms}(\text{Specs}(\{\text{t}_1\})), \ldots, \text{Terms}(\text{Specs}(\{\text{t}_n\})))\cdot\models_{\mathsf{V}_1}\text{t}$ *Sat* S. □

There are several ways to establish harmony between V_2 and the refinement calculi. If a refinement calculus R includes the rule Trans and enjoys the derivability property $\vdash_{\mathsf{V}_2}$u *Sat* S $\Leftrightarrow \vdash_{\mathsf{R}}$S $\sqsubseteq$ u, then it is trivially harmonic. Including Trans in R_1 and maintaining the derivability property

at each step also yields a nice presentation: it requires more work to show that R_1 is harmonic, but the treatment of R_4 becomes simpler. This simplification also applies to the existing presentation: R_3 includes TRANS; it is harmonic and satisfies the derivability property by Theorem 5.9. Adding C-i and JOIN-i does not break the derivability property, and hence preserves harmony.

5.2 Applications

5.2.1 Lambda Calculus

The top left corner of Figure 5.1 contains a type system λ_1 for the lambda calculus. By considering pairs of the form (typing context, type) as specifications, it is possible to apply freefinement and obtain a refinement calculus for (extended) lambda terms in the spirit of Denney [20]. The inputs to freefinement are as follows:

1. $\mathbb{K} = \text{Var} \cup \{\lambda \text{x}.\ \_ \mid \text{x} \in \text{Var}\} \cup \{\_\ \_\}$
   Note that $\mathbb{K}$ defines the language T of lambda terms:

   $$\text{e} ::= \text{x} \mid \lambda \text{x}.\, \text{e} \mid \text{e}\ \ \text{e}'$$

   Here and in the following, x ranges over the set of variables Var, and e ranges over T.

2. $\mathbb{S} = \{[\Gamma; \tau] \mid \Gamma \in \text{Context} \wedge \tau \in \text{Type}\}$, where Context is the set of typing contexts and Type is the set of types that contains the type constructor $\_ \to \_$. The intended representation of a typing context Γ is a list of variable names paired with types. Variables may appear more than once in Γ, and variable lookup uses the rightmost occurrence. In the following, σ and τ range over Type, and Γ ranges over Context.

3. $\models_{\mathsf{V}_1} \_\ \mathit{Sat}\ \_$ is defined by:

   - $\models_{\mathsf{V}_1} \text{x}\ \mathit{Sat}\ [\Gamma; \tau] \ \Leftrightarrow\ \text{x} : \tau\ \in\ \Gamma$
   - $\models_{\mathsf{V}_1} \lambda \text{x}.\, \text{e}\ \mathit{Sat}\ [\Gamma; \tau] \ \Leftrightarrow\ \exists \sigma, \tau' \cdot \tau = \sigma \to \tau' \wedge \models_{\mathsf{V}_1} \text{e}\ \mathit{Sat}\ [\Gamma, \text{x} : \sigma; \tau']$
   - $\models_{\mathsf{V}_1} \text{e}\ \ \text{e}'\ \mathit{Sat}\ [\Gamma; \tau] \ \Leftrightarrow\ \exists \sigma \cdot \models_{\mathsf{V}_1} \text{e}\ \mathit{Sat}\ [\Gamma; \sigma \to \tau] \wedge \models_{\mathsf{V}_1} \text{e}'\ \mathit{Sat}\ [\Gamma; \sigma]$

4. V_1, shown in the top right corner of Figure 5.1, is obtained from λ_1 by replacing $\Gamma \vdash \text{e} : \tau$ with $\text{e}\ \mathit{Sat}\ [\Gamma; \tau]$. The rules VAR, ABS and APP are all of the form A_1 with $n = 0$, 1 and 2 respectively. For example, in the case of ABS, Pred(C, S_1, S) is defined as
   $\exists \text{x}, \Gamma, \sigma, \tau \cdot\ \text{C} = \lambda \text{x}.\ \_ \wedge\ \text{S}_1 = [\Gamma, \text{x} : \sigma; \tau] \wedge\ \text{S} = [\Gamma; \sigma \to \tau]$.

λ_1	V_1
$\textsc{Var}\ \dfrac{\mathrm{x} : \tau \ \in\ \Gamma}{\Gamma \vdash \mathrm{x} : \tau}$	$\textsc{Var}\ \dfrac{}{\mathrm{x}\ \mathit{Sat}\ [\Gamma;\tau]}$ provided $\mathrm{x} : \tau \ \in\ \Gamma$.
$\textsc{Abs}\ \dfrac{\Gamma, \mathrm{x} : \sigma \vdash \mathrm{e} : \tau}{\Gamma \vdash \lambda \mathrm{x}.\, \mathrm{e} : \sigma \to \tau}$	$\textsc{Abs}\ \dfrac{\mathrm{e}\ \mathit{Sat}\ [\Gamma, \mathrm{x} : \sigma; \tau]}{\lambda \mathrm{x}.\, \mathrm{e}\ \mathit{Sat}\ [\Gamma; \sigma \to \tau]}$
$\textsc{App}\ \dfrac{\Gamma \vdash \mathrm{e} : \sigma \to \tau \quad \Gamma \vdash \mathrm{e}' : \sigma}{\Gamma \vdash \mathrm{e}\ \mathrm{e}' : \tau}$	$\textsc{App}\ \dfrac{\mathrm{e}\ \mathit{Sat}\ [\Gamma; \sigma \to \tau] \quad \mathrm{e}'\ \mathit{Sat}\ [\Gamma; \sigma]}{\mathrm{e}\ \mathrm{e}'\ \mathit{Sat}\ [\Gamma; \tau]}$
λ_2	R_5
$\textsc{Var}\ \dfrac{\mathrm{x} : \tau \ \in\ \Gamma}{\Gamma \vdash \mathrm{x} : \tau}$	$\textsc{Var}\ \dfrac{}{[\Gamma;\tau] \sqsubseteq \mathrm{x}}$ provided $\mathrm{x} : \tau \ \in\ \Gamma$.
$\textsc{Abs}\ \dfrac{\Gamma, \mathrm{x} : \sigma \vdash \mathrm{f} : \tau}{\Gamma \vdash \lambda \mathrm{x}.\, \mathrm{f} : \sigma \to \tau}$	$\textsc{Abs}\ \dfrac{}{[\Gamma; \sigma \to \tau] \sqsubseteq \lambda \mathrm{x}.\, [\Gamma, \mathrm{x} : \sigma; \tau]}$
$\textsc{App}\ \dfrac{\Gamma \vdash \mathrm{f} : \sigma \to \tau \quad \Gamma \vdash \mathrm{f}' : \sigma}{\Gamma \vdash \mathrm{f}\ \mathrm{f}' : \tau}$	$\textsc{App}\ \dfrac{}{[\Gamma;\tau] \sqsubseteq [\Gamma; \sigma \to \tau]\ \ [\Gamma; \sigma]}$
$\textsc{Spec}\ \dfrac{}{\Gamma \vdash [\Gamma; \sigma] : \sigma}$	$\textsc{Spec}\ \dfrac{}{[\Gamma; \sigma] \sqsubseteq [\Gamma; \sigma]}$
	$\textsc{Trans}\ \dfrac{\mathrm{f}_1 \sqsubseteq \mathrm{f}_2 \qquad \mathrm{f}_2 \sqsubseteq \mathrm{f}_3}{\mathrm{f}_1 \sqsubseteq \mathrm{f}_3}$
	$\textsc{Mono}\ \dfrac{\mathrm{f} \sqsubseteq \mathrm{f}'}{\mathrm{g}[\mathrm{f}] \sqsubseteq \mathrm{g}[\mathrm{f}']}$

Figure 5.1: Freefinement and a typed lambda calculus.

Since V_1 does not contain rules of the form B_1 where $m > 1$, freefinement does not add join terms to the lambda calculus. The system λ_2 in Figure 5.1 is V_2 where f *Sat* $[\Gamma; \tau]$ is written instead as $\Gamma \vdash \mathrm{f} : \tau$. The system R_5, shown in the bottom right of Figure 5.1, is the final harmonic refinement calculus that freefinement produces.

Here is an example top-down typing derivation with R_5:

$$
\begin{array}{ll}
 & [\Gamma; (\sigma \to \tau) \to (\sigma \to \tau)] \\
\sqsubseteq & \quad \text{“Abs”} \\
 & \lambda \mathrm{x}.\, [\Gamma, \mathrm{x} : \sigma \to \tau; \sigma \to \tau] \\
\sqsubseteq & \quad \text{“Mono with Abs”} \\
 & \lambda \mathrm{x}.\, \lambda \mathrm{y}.\, [\Gamma, \mathrm{x} : \sigma \to \tau, \mathrm{y} : \sigma; \tau] \\
\sqsubseteq & \quad \text{“Mono with App”} \\
 & \lambda \mathrm{x}.\, \lambda \mathrm{y}.\, ([\Gamma, \mathrm{x} : \sigma \to \tau, \mathrm{y} : \sigma; \sigma \to \tau]\ \ [\Gamma, \mathrm{x} : \sigma \to \tau, \mathrm{y} : \sigma; \sigma]) \\
\sqsubseteq & \quad \text{“Twice Mono with Var”} \\
 & \lambda \mathrm{x}.\, \lambda \mathrm{y}.\, (\mathrm{x}\ \ \mathrm{y})
\end{array}
$$

Since R_5 is harmonic and V_2 is a sound and conservative extension of V_1, it holds that $\vdash_{\lambda_1} \Gamma \vdash \lambda \mathrm{x}.\, \lambda \mathrm{y}.\, (\mathrm{x}\ \ \mathrm{y}) : (\sigma \to \tau) \to (\sigma \to \tau)$.

One might wish to extend R_5 using knowledge particular to lambda calculus typing. It is simple to show that V_1 is complete, so

$$\vdash_{\lambda_1} \Gamma \vdash \mathrm{e} : \tau \ \Leftrightarrow\ \vdash_{\mathsf{V}_1} \mathrm{e}\ \mathit{Sat}\ [\Gamma; \tau] \ \Leftrightarrow\ \models_{\mathsf{V}_1} \mathrm{e}\ \mathit{Sat}\ [\Gamma; \tau]$$

Furthermore, by Theorems 5.17 and 5.16.2,

$$\models_{\mathsf{V}_1} \mathrm{e}\ \mathit{Sat}\ [\Gamma; \tau] \ \Leftrightarrow\ \models_{\mathsf{V}_2} \mathrm{e}\ \mathit{Sat}\ [\Gamma; \tau]$$

and because V_2 is a sound and conservative extension of V_1,

$$\vdash_{\mathsf{V}_1} \mathrm{e}\ \mathit{Sat}\ [\Gamma; \tau] \ \Leftrightarrow\ \vdash_{\mathsf{V}_2} \mathrm{e}\ \mathit{Sat}\ [\Gamma; \tau]$$

Consider the property of *preservation*:

Definition 5.9. A relation $\rightsquigarrow \subseteq \mathrm{T} \times \mathrm{T}$ satisfies preservation $\stackrel{\text{def}}{=}$
$\forall \Gamma, \tau, \mathrm{e}, \mathrm{e}' \cdot$ if $\vdash_{\lambda_1} \Gamma \vdash \mathrm{e} : \tau$ and $\mathrm{e} \rightsquigarrow \mathrm{e}'$, then $\vdash_{\lambda_1} \Gamma \vdash \mathrm{e}' : \tau$.

Theorem 5.18. If $\rightsquigarrow$ satisfies preservation, then:

5.18.1. If $\mathrm{e} \rightsquigarrow \mathrm{e}'$, then $\models \mathrm{e} \sqsubseteq \mathrm{e}'$.

5.18.2. If $\vdash_{\mathsf{V}_2} \mathrm{e}$ Sat $[\Gamma; \tau]$ and $\mathrm{e} \rightsquigarrow \mathrm{e}'$, then $\vdash_{\mathsf{V}_2} \mathrm{e}'\ \mathit{Sat}\ [\Gamma; \tau]$.

Proof. The proof of 5.18.2 is trivial. For 5.18.1:

$$
\begin{array}{ll}
 & \forall \Gamma, \tau, \mathrm{e}, \mathrm{e}' \cdot \vdash_{\lambda_1} \Gamma \vdash \mathrm{e} : \tau \ \wedge \ \mathrm{e} \rightsquigarrow \mathrm{e}' \ \Rightarrow \ \vdash_{\lambda_1} \Gamma \vdash \mathrm{e}' : \tau \\
\Leftrightarrow & \{\text{predicate logic}\} \\
 & \forall \mathrm{e}, \mathrm{e}' \cdot \mathrm{e} \rightsquigarrow \mathrm{e}' \ \Rightarrow \ (\forall \Gamma, \tau \cdot \vdash_{\lambda_1} \Gamma \vdash \mathrm{e} : \tau \ \Rightarrow \ \vdash_{\lambda_1} \Gamma \vdash \mathrm{e}' : \tau) \\
\Leftrightarrow & \{\text{equivalent terms}\} \\
 & \forall \mathrm{e}, \mathrm{e}' \cdot \mathrm{e} \rightsquigarrow \mathrm{e}' \ \Rightarrow \ (\forall \mathrm{S} \in \mathbb{S} \cdot \models_{\mathsf{V}_2} \mathrm{e} \ \mathit{Sat}\ \mathrm{S} \ \Rightarrow \ \models_{\mathsf{V}_2} \mathrm{e}' \ \mathit{Sat}\ \mathrm{S}) \\
\Leftrightarrow & \{\text{Lemma 5.8}\} \\
 & \forall \mathrm{e}, \mathrm{e}' \cdot \mathrm{e} \rightsquigarrow \mathrm{e}' \ \Rightarrow \ \models \mathrm{e} \sqsubseteq \mathrm{e}'
\end{array}
$$

□

So any relation that satisfies preservation contains only sound refinements that satisfy Harmony 1, and can augment R_5 to yield a sound and harmonic refinement system. Examples of relations that satisfy preservation include:

- The α-conversion relation.
- The β-reduction relation.
- The η-contraction relation. So $\lambda \mathrm{x}.\,(\mathrm{e}\ \mathrm{x}) \sqsubseteq \mathrm{e}$, provided x does not appear free in e.
- The relation $\leq$ on closed terms, where $\mathrm{e} \leq \mathrm{e}'$ exactly when e has fewer types than e'.

Here is a small example that uses the η-contraction extension:

$$
\begin{array}{ll}
 & \lambda \mathrm{x}.\,\lambda \mathrm{y}.\,\lambda \mathrm{z}.\,((\mathrm{x}\ \mathrm{y})\ \mathrm{z}) \\
\sqsubseteq & \{\textsc{Mono} \text{ with } \eta\text{-contraction}\} \\
 & \lambda \mathrm{x}.\,\lambda \mathrm{y}.\,(\mathrm{x}\ \mathrm{y}) \\
\sqsubseteq & \{\textsc{Mono} \text{ with } \eta\text{-contraction}\} \\
 & \lambda \mathrm{x}.\,\mathrm{x}
\end{array}
$$

5.2.2 Hoare Logic

Figure 5.2 contains system H, a Hoare logic for simple imperative programs. P is a precondition, Q a postcondition, and c a command in the Hoare triple $\{\mathrm{P}\}\mathrm{c}\{\mathrm{Q}\}$, and $\models_{\mathsf{H}}\{\mathrm{P}\}\mathrm{c}\{\mathrm{Q}\}$ is the usual partial correctness interpretation of $\{\mathrm{P}\}\mathrm{c}\{\mathrm{Q}\}$. By interpreting a specification as a pre-post pair, the rules of H do not fit the rule forms A_1 and B_1, since the proviso of AuxVarElim inspects the command c to determine the variables that it writes and reads. However, if specifications also keep track of written and read variables, then it becomes possible to apply freefinement to obtain a refinement calculus in the spirit of Morgan [50]. Here are the inputs:

1. There are constructors for assignments, sequential composition, conditionals and loops:

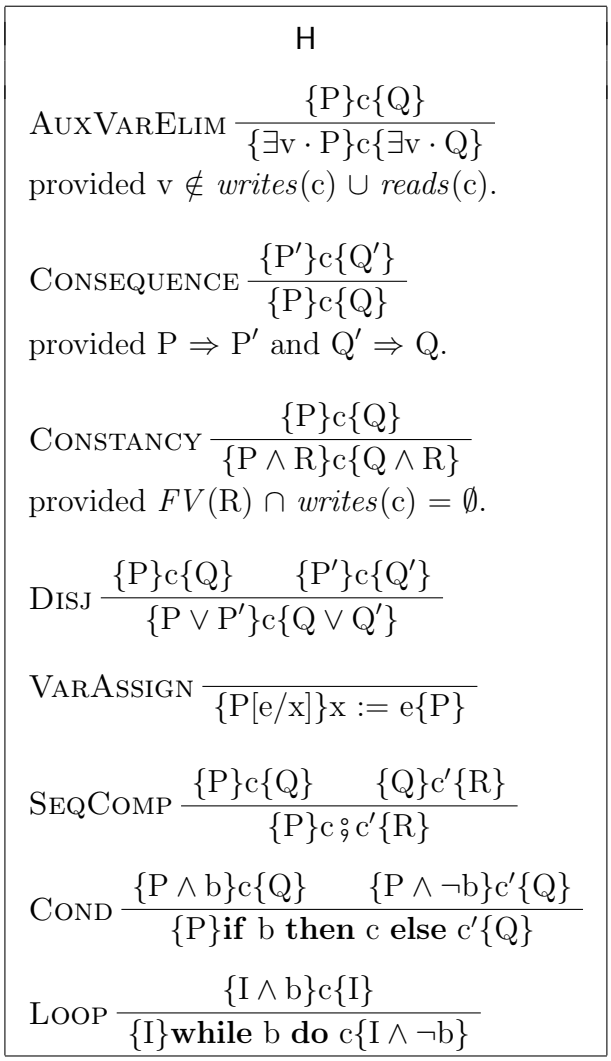


Figure 5.2: Freefinement and Hoare logic: the system H.

V_1

$$\textsc{AuxVarElim}\ \frac{\mathrm{c}\quad \mathit{Sat}\quad \overline{\mathrm{x}};\overline{\mathrm{y}}:\{\mathrm{P},\mathrm{Q}\}}{\mathrm{c}\quad \mathit{Sat}\quad \overline{\mathrm{x}};\overline{\mathrm{y}}:\{\exists \mathrm{v}\cdot \mathrm{P},\exists \mathrm{v}\cdot \mathrm{Q}\}}$$
provided $\mathrm{v}\notin \overline{\mathrm{x}}\cup\overline{\mathrm{y}}$.

$$\textsc{Consequence}\ \frac{\mathrm{c}\quad \mathit{Sat}\quad \overline{\mathrm{x}'};\overline{\mathrm{y}'}:\{\mathrm{P}',\mathrm{Q}'\}}{\mathrm{c}\quad \mathit{Sat}\quad \overline{\mathrm{x}};\overline{\mathrm{y}}:\{\mathrm{P},\mathrm{Q}\}}$$
provided $\overline{\mathrm{x}}\supseteq\overline{\mathrm{x}'}$ and $\overline{\mathrm{y}}\supseteq\overline{\mathrm{y}'}$ and $\mathrm{P}\Rightarrow\mathrm{P}'$ and $\mathrm{Q}'\Rightarrow\mathrm{Q}$.

$$\textsc{Constancy}\ \frac{\mathrm{c}\quad \mathit{Sat}\quad \overline{\mathrm{x}};\overline{\mathrm{y}}:\{\mathrm{P},\mathrm{Q}\}}{\mathrm{c}\quad \mathit{Sat}\quad \overline{\mathrm{x}};\overline{\mathrm{y}}:\{\mathrm{P}\wedge\mathrm{R},\mathrm{Q}\wedge\mathrm{R}\}}$$
provided $\mathit{FV}(\mathrm{R})\cap\overline{\mathrm{x}}=\emptyset$.

$$\textsc{Disj}\ \frac{\mathrm{c}\quad \mathit{Sat}\quad \overline{\mathrm{x}};\overline{\mathrm{y}}:\{\mathrm{P},\mathrm{Q}\}\qquad \mathrm{c}\quad \mathit{Sat}\quad \overline{\mathrm{x}};\overline{\mathrm{y}}:\{\mathrm{P}',\mathrm{Q}'\}}{\mathrm{c}\quad \mathit{Sat}\quad \overline{\mathrm{x}};\overline{\mathrm{y}}:\{\mathrm{P}\vee\mathrm{P}',\mathrm{Q}\vee\mathrm{Q}'\}}$$

$$\textsc{VarAssign}\ \frac{}{\mathrm{x}:=\mathrm{e}\quad \mathit{Sat}\quad \mathrm{x};\mathit{FV}(\mathrm{e}):\{\mathrm{P}[\mathrm{e}/\mathrm{x}],\mathrm{P}\}}$$

$$\textsc{SeqComp}\ \frac{\mathrm{c}\quad \mathit{Sat}\quad \overline{\mathrm{x}};\overline{\mathrm{y}}:\{\mathrm{P},\mathrm{Q}\}\qquad \mathrm{c}'\quad \mathit{Sat}\quad \overline{\mathrm{x}'};\overline{\mathrm{y}'}:\{\mathrm{Q},\mathrm{R}\}}{\mathrm{c}\fatsemi\mathrm{c}'\quad \mathit{Sat}\quad \overline{\mathrm{x}}\cup\overline{\mathrm{x}'};\overline{\mathrm{x}}\cup\overline{\mathrm{x}'}:\{\mathrm{P},\mathrm{R}\}}$$

$$\textsc{Cond}\ \frac{\mathrm{c}\quad \mathit{Sat}\quad \overline{\mathrm{x}};\overline{\mathrm{y}}:\{\mathrm{P}\wedge\mathrm{b},\mathrm{Q}\}\qquad \mathrm{c}'\quad \mathit{Sat}\quad \overline{\mathrm{x}'};\overline{\mathrm{y}'}:\{\mathrm{P}\wedge\neg\mathrm{b},\mathrm{Q}\}}{\textbf{if}\ \mathrm{b}\ \textbf{then}\ \mathrm{c}\ \textbf{else}\ \mathrm{c}'\quad \mathit{Sat}\quad \overline{\mathrm{x}}\cup\overline{\mathrm{x}'};\overline{\mathrm{y}}\cup\overline{\mathrm{y}'}\cup\mathit{FV}(\mathrm{b}):\{\mathrm{P},\mathrm{Q}\}}$$

$$\textsc{Loop}\ \frac{\mathrm{c}\quad \mathit{Sat}\quad \overline{\mathrm{x}};\overline{\mathrm{y}}:\{\mathrm{I}\wedge\mathrm{b},\mathrm{I}\}}{\textbf{while}\ \mathrm{b}\ \textbf{do}\ \mathrm{c}\quad \mathit{Sat}\quad \overline{\mathrm{x}};\overline{\mathrm{y}}\cup\mathit{FV}(\mathrm{b}):\{\mathrm{I},\mathrm{I}\wedge\neg\mathrm{b}\}}$$

Figure 5.3: Freefinement and Hoare logic: the system V_1.

V_2

$$\textsc{AuxVarElim}\ \frac{\mathrm{s}\quad \mathit{Sat}\quad \overline{\mathrm{x}};\overline{\mathrm{y}}:\{\mathrm{P},\mathrm{Q}\}}{\mathrm{s}\quad \mathit{Sat}\quad \overline{\mathrm{x}};\overline{\mathrm{y}}:\{\exists \mathrm{v}\cdot \mathrm{P},\exists \mathrm{v}\cdot \mathrm{Q}\}}$$
provided $\mathrm{v}\notin \overline{\mathrm{x}}\cup\overline{\mathrm{y}}$.

$$\textsc{Consequence}\ \frac{\mathrm{s}\quad \mathit{Sat}\quad \overline{\mathrm{x}'};\overline{\mathrm{y}'}:\{\mathrm{P}',\mathrm{Q}'\}}{\mathrm{s}\quad \mathit{Sat}\quad \overline{\mathrm{x}};\overline{\mathrm{y}}:\{\mathrm{P},\mathrm{Q}\}}$$
provided $\overline{\mathrm{x}}\supseteq\overline{\mathrm{x}'}$ and $\overline{\mathrm{y}}\supseteq\overline{\mathrm{y}'}$ and $\mathrm{P}\Rightarrow\mathrm{P}'$ and $\mathrm{Q}'\Rightarrow\mathrm{Q}$.

$$\textsc{Constancy}\ \frac{\mathrm{s}\quad \mathit{Sat}\quad \overline{\mathrm{x}};\overline{\mathrm{y}}:\{\mathrm{P},\mathrm{Q}\}}{\mathrm{s}\quad \mathit{Sat}\quad \overline{\mathrm{x}};\overline{\mathrm{y}}:\{\mathrm{P}\wedge\mathrm{R},\mathrm{Q}\wedge\mathrm{R}\}}$$
provided $\mathit{FV}(\mathrm{R})\cap\overline{\mathrm{x}}=\emptyset$.

$$\textsc{Disj}\ \frac{\mathrm{s}\quad \mathit{Sat}\quad \overline{\mathrm{x}};\overline{\mathrm{y}}:\{\mathrm{P},\mathrm{Q}\}\qquad \mathrm{s}\quad \mathit{Sat}\quad \overline{\mathrm{x}};\overline{\mathrm{y}}:\{\mathrm{P}',\mathrm{Q}'\}}{\mathrm{s}\quad \mathit{Sat}\quad \overline{\mathrm{x}};\overline{\mathrm{y}}:\{\mathrm{P}\vee\mathrm{P}',\mathrm{Q}\vee\mathrm{Q}'\}}$$

$$\textsc{VarAssign}\ \frac{}{\mathrm{x}:=\mathrm{e}\quad \mathit{Sat}\quad \mathrm{x};\mathit{FV}(\mathrm{e}):\{\mathrm{P}[\mathrm{e}/\mathrm{x}],\mathrm{P}\}}$$

$$\textsc{SeqComp}\ \frac{\mathrm{s}\quad \mathit{Sat}\quad \overline{\mathrm{x}};\overline{\mathrm{y}}:\{\mathrm{P},\mathrm{Q}\}\qquad \mathrm{s}'\quad \mathit{Sat}\quad \overline{\mathrm{x}'};\overline{\mathrm{y}'}:\{\mathrm{Q},\mathrm{R}\}}{\mathrm{s}\fatsemi\mathrm{s}'\quad \mathit{Sat}\quad \overline{\mathrm{x}}\cup\overline{\mathrm{x}'};\overline{\mathrm{x}}\cup\overline{\mathrm{x}'}:\{\mathrm{P},\mathrm{R}\}}$$

$$\textsc{Cond}\ \frac{\mathrm{s}\quad \mathit{Sat}\quad \overline{\mathrm{x}};\overline{\mathrm{y}}:\{\mathrm{P}\wedge\mathrm{b},\mathrm{Q}\}\qquad \mathrm{s}'\quad \mathit{Sat}\quad \overline{\mathrm{x}'};\overline{\mathrm{y}'}:\{\mathrm{P}\wedge\neg\mathrm{b},\mathrm{Q}\}}{\textbf{if}\ \mathrm{b}\ \textbf{then}\ \mathrm{s}\ \textbf{else}\ \mathrm{s}'\quad \mathit{Sat}\quad \overline{\mathrm{x}}\cup\overline{\mathrm{x}'};\overline{\mathrm{y}}\cup\overline{\mathrm{y}'}\cup\mathit{FV}(\mathrm{b}):\{\mathrm{P},\mathrm{Q}\}}$$

$$\textsc{Loop}\ \frac{\mathrm{s}\quad \mathit{Sat}\quad \overline{\mathrm{x}};\overline{\mathrm{y}}:\{\mathrm{I}\wedge\mathrm{b},\mathrm{I}\}}{\textbf{while}\ \mathrm{b}\ \textbf{do}\ \mathrm{s}\quad \mathit{Sat}\quad \overline{\mathrm{x}};\overline{\mathrm{y}}\cup\mathit{FV}(\mathrm{b}):\{\mathrm{I},\mathrm{I}\wedge\neg\mathrm{b}\}}$$

$$\textsc{Spec}\ \frac{}{\overline{\mathrm{x}};\overline{\mathrm{y}}:\{\mathrm{P},\mathrm{Q}\}\quad \mathit{Sat}\quad \overline{\mathrm{x}};\overline{\mathrm{y}}:\{\mathrm{P},\mathrm{Q}\}}$$

$$\textsc{Join}\ \frac{\mathrm{s}\quad \mathit{Sat}\quad \overline{\mathrm{x}};\overline{\mathrm{y}}:\{\mathrm{P},\mathrm{Q}\}}{\bigsqcup(\ldots,\mathrm{s},\ldots)\quad \mathit{Sat}\quad \overline{\mathrm{x}};\overline{\mathrm{y}}:\{\mathrm{P},\mathrm{Q}\}}$$

Figure 5.4: Freefinement and Hoare logic: the system V_2.

R_6

AuxVarElim $\dfrac{}{\overline{\mathrm{x}};\overline{\mathrm{y}}:\{\exists \mathrm{v}\cdot \mathrm{P}, \exists \mathrm{v}\cdot \mathrm{Q}\} \sqsubseteq \overline{\mathrm{x}};\overline{\mathrm{y}}:\{\mathrm{P},\mathrm{Q}\}}$
provided $\mathrm{v} \notin \overline{\mathrm{x}} \cup \overline{\mathrm{y}}$.

Consequence $\dfrac{}{\overline{\mathrm{x}};\overline{\mathrm{y}}:\{\mathrm{P},\mathrm{Q}\} \sqsubseteq \overline{\mathrm{x}'};\overline{\mathrm{y}'}:\{\mathrm{P}',\mathrm{Q}'\}}$
provided $\overline{\mathrm{x}} \supseteq \overline{\mathrm{x}'}$ and $\overline{\mathrm{y}} \supseteq \overline{\mathrm{y}'}$ and $\mathrm{P} \Rightarrow \mathrm{P}'$ and $\mathrm{Q}' \Rightarrow \mathrm{Q}$.

Constancy $\dfrac{}{\overline{\mathrm{x}};\overline{\mathrm{y}}:\{\mathrm{P}\wedge\mathrm{R},\mathrm{Q}\wedge\mathrm{R}\} \sqsubseteq \overline{\mathrm{x}};\overline{\mathrm{y}}:\{\mathrm{P},\mathrm{Q}\}}$
provided $FV(\mathrm{R}) \cap \overline{\mathrm{x}} = \emptyset$.

Disj $\dfrac{}{\overline{\mathrm{x}};\overline{\mathrm{y}}:\{\mathrm{P}\vee\mathrm{P}',\mathrm{Q}\vee\mathrm{Q}'\} \sqsubseteq \bigsqcup(\overline{\mathrm{x}};\overline{\mathrm{y}}:\{\mathrm{P},\mathrm{Q}\},\ \overline{\mathrm{x}};\overline{\mathrm{y}}:\{\mathrm{P}',\mathrm{Q}'\})}$

VarAssign $\dfrac{}{\mathrm{x}; FV(\mathrm{e}):\{\mathrm{P}[\mathrm{e}/\mathrm{x}],\mathrm{P}\} \sqsubseteq \mathrm{x} := \mathrm{e}}$

SeqComp $\dfrac{}{\overline{\mathrm{x}}\cup\overline{\mathrm{x}'};\overline{\mathrm{x}}\cup\overline{\mathrm{x}'}:\{\mathrm{P},\mathrm{R}\} \sqsubseteq \overline{\mathrm{x}};\overline{\mathrm{y}}:\{\mathrm{P},\mathrm{Q}\}\ ;\ \overline{\mathrm{x}'};\overline{\mathrm{y}'}:\{\mathrm{Q},\mathrm{R}\}}$

Cond $\dfrac{}{\begin{array}{l}\overline{\mathrm{x}}\cup\overline{\mathrm{x}'};\overline{\mathrm{y}}\cup\overline{\mathrm{y}'}\cup FV(\mathrm{b}):\{\mathrm{P},\mathrm{Q}\}\\ \quad\sqsubseteq \textbf{if } \mathrm{b} \textbf{ then } \overline{\mathrm{x}};\overline{\mathrm{y}}:\{\mathrm{P}\wedge\mathrm{b},\mathrm{Q}\} \textbf{ else } \overline{\mathrm{x}'};\overline{\mathrm{y}'}:\{\mathrm{P}\wedge\neg\mathrm{b},\mathrm{Q}\}\end{array}}$

Loop $\dfrac{}{\overline{\mathrm{x}};\overline{\mathrm{y}}\cup FV(\mathrm{b}):\{\mathrm{I},\mathrm{I}\wedge\neg\mathrm{b}\} \sqsubseteq \textbf{while } \mathrm{b} \textbf{ do } \overline{\mathrm{x}};\overline{\mathrm{y}}:\{\mathrm{I}\wedge\mathrm{b},\mathrm{I}\}}$

Spec $\dfrac{}{\overline{\mathrm{x}};\overline{\mathrm{y}}:\{\mathrm{P},\mathrm{Q}\} \sqsubseteq \overline{\mathrm{x}};\overline{\mathrm{y}}:\{\mathrm{P},\mathrm{Q}\}}$

Join $\dfrac{}{\overline{\mathrm{x}};\overline{\mathrm{y}}:\{\mathrm{P},\mathrm{Q}\} \sqsubseteq \bigsqcup(\ldots,\ \overline{\mathrm{x}};\overline{\mathrm{y}}:\{\mathrm{P},\mathrm{Q}\},\ \ldots)}$

Trans $\dfrac{\mathrm{s}_1 \sqsubseteq \mathrm{s}_2 \qquad \mathrm{s}_2 \sqsubseteq \mathrm{s}_3}{\mathrm{s}_1 \sqsubseteq \mathrm{s}_3}$

Mono $\dfrac{\mathrm{s} \sqsubseteq \mathrm{s}'}{\mathrm{t}[\mathrm{s}] \sqsubseteq \mathrm{t}[\mathrm{s}']}$

Unjoin $\dfrac{}{\bigsqcup(\mathrm{s},\ \ldots,\ \mathrm{s}) \sqsubseteq \mathrm{s}}$

Figure 5.5: Freefinement and Hoare logic: the system R_6.

$$\begin{aligned}\mathbb{K} \;=\; & \{\mathrm{x} := \mathrm{e} \mid \mathrm{x} \in \mathrm{Var} \;\wedge\; \mathrm{e} \in \mathrm{IntExp}\} \\ & \cup \{\_ \,;\, \_\} \\ & \cup \{\textbf{if } \mathrm{b} \textbf{ then } \_ \textbf{ else } \_ \mid \mathrm{b} \in \mathrm{BoolExp}\} \\ & \cup \{\textbf{while } \mathrm{b} \textbf{ do } \_ \mid \mathrm{b} \in \mathrm{BoolExp}\}\end{aligned}$$

2. A specification consists of two sets of variables and two assertions, written in a notation resembling Morgan's specification statement [49]:

$$\mathbb{S} = \left\{\overline{\mathrm{x}}; \overline{\mathrm{y}} : \{\mathrm{P}, \mathrm{Q}\} \mid \overline{\mathrm{x}}, \overline{\mathrm{y}} \in \mathcal{P}(\mathrm{Var}) \;\wedge\; \mathrm{P}, \mathrm{Q} \in \mathrm{Assertion}\right\}$$

3. In the specification $\overline{\mathrm{x}}; \overline{\mathrm{y}} : \{\mathrm{P}, \mathrm{Q}\}$, the $\overline{\mathrm{x}}$ and $\overline{\mathrm{y}}$ are upper bounds on the sets of variables written and read by the command respectively, the P is a precondition and the Q a postcondition:

$$\models_{\mathsf{V}_1} \mathrm{c}\ \mathit{Sat}\, \overline{\mathrm{x}}; \overline{\mathrm{y}} : \{\mathrm{P}, \mathrm{Q}\} \stackrel{\text{def}}{=} \mathit{writes}(\mathrm{c}) \subseteq \overline{\mathrm{x}} \;\wedge\; \mathit{reads}(\mathrm{c}) \subseteq \overline{\mathrm{y}} \;\wedge\; \models_{\mathsf{H}} \{\mathrm{P}\}\mathrm{c}\{\mathrm{Q}\}$$

4. V_1, shown in Figure 5.3, has the following relationship with H:

   $$\vdash_{\mathsf{V}_1} \mathrm{c}\ \mathit{Sat}\, \overline{\mathrm{x}}; \overline{\mathrm{y}} : \{\mathrm{P}, \mathrm{Q}\} \;\Leftrightarrow\; \mathit{writes}(\mathrm{c}) \subseteq \overline{\mathrm{x}} \;\wedge\; \mathit{reads}(\mathrm{c}) \subseteq \overline{\mathrm{y}} \;\wedge\; \vdash_{\mathsf{H}} \{\mathrm{P}\}\mathrm{c}\{\mathrm{Q}\}$$

   Note that:

   - The non-structural rules of H have counterparts in V_1 that embody the definitions of *writes* and *reads*. For example, the conclusion of Cond reflects that
     $\mathit{writes}(\textbf{if } \mathrm{b} \textbf{ then } \mathrm{c} \textbf{ else } \mathrm{c}') \stackrel{\text{def}}{=} \mathit{writes}(\mathrm{c}) \cup \mathit{writes}(\mathrm{c}')$ and
     $\mathit{reads}(\textbf{if } \mathrm{b} \textbf{ then } \mathrm{c} \textbf{ else } \mathrm{c}') \stackrel{\text{def}}{=} \mathit{reads}(\mathrm{c}) \cup \mathit{reads}(\mathrm{c}') \cup \mathit{FV}(\mathrm{b})$.
   - The structural rules of H that inspect c for its write and/or read sets have counterparts in V_1 that consult the specification instead. See for example the proviso of AuxVarElim.
   - Consequence in V_1 allows the enlargement of write and read sets. This loosening of the bounds is useful in refinement developments, because then the resulting code is not forced to write and read all the variables that were originally available for writing and reading.

   The V_1-counterparts of the structural rules of H are all of the form B_1. For example, $m = 1$ in the case of Constancy, and $m = 2$ for Disj. The other rules are of the form A_1. For example, $n = 2$ in the case of Cond, and $n = 1$ for Loop.

The systems V_2 and R_6 that freefinement produces appear in Figures 5.4 and 5.5. R_6 yields several derived rules that may be useful in practical refinement developments. For example, the rule:

$$\textsc{DerivedVarAssign}\ \frac{}{\overline{x};\overline{y}:\{P,Q\} \sqsubseteq z := e}$$
provided $z \in \overline{x}$ and $FV(e) \subseteq \overline{y}$ and $P \Rightarrow Q[e/z]$.

can replace VarAssign, and is similar to the assignment law of Morgan (Law 1.3 on p. 8 of [50]). Likewise, the derived rule:

$$\textsc{FollowingVarAssign}\ \frac{}{\overline{x};\overline{y}:\{P,Q\} \sqsubseteq \overline{x};\overline{y}:\{P,Q[e/z]\} \fatsemi z := e}$$
provided $z \in \overline{x}$ and $FV(e) \subseteq \overline{y}$.

is similar to the *following assignment* law of Morgan (Law 3.5 on p. 32 of [50]).

Here is an example showing that R_6 can derive a correct factorial algorithm starting with its specification:

$$\begin{array}{ll} & y,z; x,y,z:\{true, y = x!\} \\ \sqsubseteq & \text{“SeqComp”} \\ & y,z; \emptyset:\{true, y = 1 \wedge z = 0\} \fatsemi y,z; x,y,z:\{y = 1 \wedge z = 0, y = x!\} \end{array}$$

The first spec statement is refined as follows:

$$\begin{array}{ll} & y,z; \emptyset:\{true, y = 1 \wedge z = 0\} \\ \sqsubseteq & \text{“SeqComp”} \\ & y; \emptyset:\{true, y = 1\} \fatsemi z; \emptyset:\{y = 1, y = 1 \wedge z = 0\} \\ \sqsubseteq & \text{“Twice Mono with Consequence”} \\ & y; \emptyset:\{1 = 1, y = 1\} \fatsemi z; \emptyset:\{y = 1 \wedge 0 = 0, y = 1 \wedge z = 0\} \\ \sqsubseteq & \text{“Twice Mono with VarAssign”} \\ & y := 1 \fatsemi z := 0 \end{array}$$

And for the second spec statement:

$$\begin{array}{ll} & y,z; x,y,z:\{y = 1 \wedge z = 0, y = x!\} \\ \sqsubseteq & \text{“Consequence”} \\ & y,z; x,y,z:\{y = z!, y = z! \wedge \neg z{\neq}x\} \\ \sqsubseteq & \text{“Loop”} \\ & \textbf{while } z{\neq}x \textbf{ do } y,z; y,z:\{y = z! \wedge z{\neq}x, y = z!\} \\ \sqsubseteq & \text{“Mono with SeqComp”} \\ & \textbf{while } z{\neq}x \textbf{ do } z; z:\{y = z! \wedge z{\neq}x, y{\cdot}z = z!\} \fatsemi y; y,z:\{y{\cdot}z = z!, y = z!\} \\ \sqsubseteq & \text{“Mono with VarAssign”} \\ & \textbf{while } z{\neq}x \textbf{ do } z; z:\{y = z! \wedge z{\neq}x, y{\cdot}z = z!\} \fatsemi y := y{\cdot}z \\ \sqsubseteq & \text{“Mono with Consequence”} \\ & \textbf{while } z{\neq}x \textbf{ do } z; z:\{y{\cdot}(z{+}1) = (z{+}1)!, y{\cdot}z = z!\} \fatsemi y := y{\cdot}z \end{array}$$

$\sqsubseteq$ "Mono with VarAssign"
while z≠x **do** z := z+1 ⨟ y := y·z

Since $\vdash_{\mathsf{R}_6}$ y,z; x,y,z : {true, y = x!} $\sqsubseteq$ y := 1 ⨟ z := 0 ⨟ **while** z≠x **do** z := z+1 ⨟ y := y·z, it is the case that $\vdash_{\mathsf{V}_1}$ y := 1 ⨟ z := 0 ⨟ **while** z≠x **do** z := z+1 ⨟ y := y·z *Sat* y,z; x,y,z : {true, y = x!} and hence also
$\vdash_{\mathsf{H}}$ {true} y := 1 ⨟ z := 0 ⨟ **while** z≠x **do** z := z+1 ⨟ y := y·z {y = x!}.

Here is another example of using R_6; it involves join statements. The statement $\bigsqcup(\overline{x}; \overline{y} : \{P_1, Q_1\},\ \overline{x}; \overline{y} : \{P_2, Q_2\})$ is the join of the specification statements $\overline{x}; \overline{y} : \{P_1, Q_1\}$ and $\overline{x}; \overline{y} : \{P_2, Q_2\}$. Expressing it as a spec statement is simple because

$$\bigsqcup(\overline{x}; \overline{y} : \{P_1, Q_1\},\ \overline{x}; \overline{y} : \{P_2, Q_2\}) \equiv \overline{x}; \overline{y} : \{P_1,\ Q_1\} \textbf{ also } \{P_2,\ Q_2\}$$

where the definition of {P₁, Q₁} **also** {P₂, Q₂}, taken from [58], is: {(P₁ ∧ z=1) ∨ (P₂ ∧ z≠1), (Q₁ ∧ z=1) ∨ (Q₂ ∧ z≠1)} where z is fresh. R_6 can derive both directions of refinement. Firstly:

$\bigsqcup(\overline{x}; \overline{y} : \{P_1, Q_1\},\ \overline{x}; \overline{y} : \{P_2, Q_2\})$
$\sqsubseteq$ "Twice Mono with Consequence"
$\bigsqcup(\overline{x}; \overline{y} : \{\exists z\cdot\ (P_1 \wedge z{=}1 \vee P_2 \wedge z{\neq}1) \wedge z{=}1, \exists z\cdot\ (Q_1 \wedge z{=}1 \vee Q_2 \wedge z{\neq}1) \wedge z{=}1\},$
$\overline{x}; \overline{y} : \{\exists z\cdot\ (P_1 \wedge z{=}1 \vee P_2 \wedge z{\neq}1) \wedge z{\neq}1, \exists z\cdot\ (Q_1 \wedge z{=}1 \vee Q_2 \wedge z{\neq}1) \wedge z{\neq}1\})$
$\sqsubseteq$ "Twice Mono with AuxVarElim"
$\bigsqcup(\overline{x}; \overline{y} : \{(P_1 \wedge z{=}1 \vee P_2 \wedge z{\neq}1) \wedge z{=}1, (Q_1 \wedge z{=}1 \vee Q_2 \wedge z{\neq}1) \wedge z{=}1\},$
$\overline{x}; \overline{y} : \{(P_1 \wedge z{=}1 \vee P_2 \wedge z{\neq}1) \wedge z{\neq}1, (Q_1 \wedge z{=}1 \vee Q_2 \wedge z{\neq}1) \wedge z{\neq}1\})$
$\sqsubseteq$ "Twice Mono with Constancy"
$\bigsqcup(\overline{x}; \overline{y} : \{P_1 \wedge z{=}1 \vee P_2 \wedge z{\neq}1, Q_1 \wedge z{=}1 \vee Q_2 \wedge z{\neq}1\},$
$\overline{x}; \overline{y} : \{P_1 \wedge z{=}1 \vee P_2 \wedge z{\neq}1, Q_1 \wedge z{=}1 \vee Q_2 \wedge z{\neq}1\})$
$\sqsubseteq$ "Unjoin"
$\overline{x}; \overline{y} : \{P_1,\ Q_1\}$ **also** $\{P_2,\ Q_2\}$

Secondly:

$\overline{x}; \overline{y} : \{(P_1 \wedge z{=}1) \vee (P_2 \wedge z{\neq}1), (Q_1 \wedge z{=}1) \vee (Q_2 \wedge z{\neq}1)\}$
$\sqsubseteq$ "Disj"
$\bigsqcup(\overline{x}; \overline{y} : \{P_1 \wedge z{=}1, Q_1 \wedge z{=}1\},\ \overline{x}; \overline{y} : \{P_2 \wedge z{\neq}1, Q_2 \wedge z{\neq}1\})$
$\sqsubseteq$ "Twice Mono with Constancy"
$\bigsqcup(\overline{x}; \overline{y} : \{P_1, Q_1\},\ \overline{x}; \overline{y} : \{P_2, Q_2\})$

Leino and Manohar [43] mention several uses of the join of spec-like statements.

5.2.3 Discussion

The type system λ_1 considered above is very simple. Freefinement also applies to System F and other more sophisticated type systems.

Although λ_1 had only rules of the form A_1, typing rules of the form B_1 are quite common – examples include rules for subtyping and intersection types:

$$\textsc{Sub}\ \frac{\Gamma \vdash \mathrm{e} : \tau}{\Gamma \vdash \mathrm{e} : \tau'} \quad \text{provided } \tau <: \tau'. \qquad \textsc{Inter}\ \frac{\Gamma \vdash \mathrm{e} : \tau \qquad \Gamma \vdash \mathrm{e} : \tau'}{\Gamma \vdash \mathrm{e} : \tau \wedge \tau'}$$

There is no golden recipe for adapting proof systems to make them amenable to freefinement. However, enriching specifications and/or terms might help. The Hoare logic example used enriched specifications to keep track of write and read sets. Consider again the two problematic rules from before:

$$2\ \frac{\mathrm{succ(n)} : \mathbb{N}}{\mathrm{pred(succ(n))} : \mathbb{N}} \qquad 3\ \frac{\mathrm{n} : \mathbb{N}}{\mathrm{pred(n)} : \mathbb{N}} \quad \text{provided positive(n).}$$

Rule 2 can be accommodated by choosing $\mathbb{S} = \{\text{'z'},\text{'s'},\text{'p'}\} \times \{\mathbb{N}\}$. Intuitively, the specification $(\text{'s'}, \mathbb{N})$ tracks the fact that the outermost constructor is 'succ'. The rule then becomes:

$$2\ \frac{\mathrm{n} : (\text{'s'}, \mathbb{N})}{\mathrm{pred(n)} : (\text{'p'}, \mathbb{N})}$$

Rule 3 can be accommodated by choosing $\mathbb{S} = \mathbb{N} \times \{\mathbb{N}\}$. Then the sentence $\mathrm{n} : (i, \mathbb{N})$ tracks the fact that term n denotes the natural number i. The adapted rule is of the form A_1 with $n = 1$:

$$3\ \frac{\mathrm{n} : (i, \mathbb{N})}{\mathrm{pred(n)} : (i-1, \mathbb{N})} \quad \text{provided } i > 0.$$

In some cases it might be useful to enrich the term language. For example, consider the rule of concurrent separation logic [7] that removes auxiliary commands (ghost assignments):

$$\textsc{Auxiliary}\ \frac{\Gamma \vdash \{\mathrm{P}\}\mathrm{c}\{\mathrm{Q}\}}{\Gamma \vdash \{\mathrm{P}\}\mathrm{c}\backslash\mathrm{a}\{\mathrm{Q}\}} \quad \text{provided a} \in \mathit{aux}(\mathrm{c}) \text{ and a} \cap (FV(\mathrm{P}) \cup FV(\mathrm{Q})) = \emptyset.$$

This rule is not of the form A_1 or B_1, because it contains a meta-operation in the conclusion. However, if the meta-operation is turned into an explicit constructor (and specifications track auxiliaries), then the rule is of the form B_1 with $m = 1$ and freefinement can handle it.

5.3 Constructing correct OO programs

This section outlines a technique that provides correctness by construction for OO programs. It applies freefinement to the statement verification part of the separation logic proof system in Chapter 2, and combines the result with rules that deal with non-statement constructs.

The judgements for statement verification in Section 2.5 have the form $\Delta; \Gamma \vdash_s \{\mathrm{P}\}\mathrm{s}\{\mathrm{Q}\}$. No rule modifies Δ and Γ, so they are constant parameters of statement verification. Spec statements can consequently exclude Δ and Γ and retain the form $\overline{\mathrm{x}}; \overline{\mathrm{y}} : \{\mathrm{P}, \mathrm{Q}\}$ of the previous section. Freefinement also adds join statements to the language, and proceeds as in the Hoare logic example. The resulting refinement judgements have the form $\Delta; \Gamma \vdash \mathrm{s} \sqsubseteq \mathrm{s}'$. For example, the refinement counterpart of the Frame rule resembles CONSTANCY from before:

$$\frac{FV(\mathrm{R}) \cap \overline{\mathrm{x}} = \emptyset}{\Delta; \Gamma \vdash \overline{\mathrm{x}}; \overline{\mathrm{y}} : \{\mathrm{P} * \mathrm{R}, \mathrm{Q} * \mathrm{R}\} \sqsubseteq \overline{\mathrm{x}}; \overline{\mathrm{y}} : \{\mathrm{P}, \mathrm{Q}\}}$$

The ability to refine statements is useful in the context of OO programming, but correctness by construction also needs to accommodate other language constructs and concepts such as data refinement [49]. To this end, the rest of the section presents a suitably extended language and a development calculus.

5.3.1 Language and proof rules

Figure 5.6 shows the grammar of a slightly modified language. The highlighted productions generate non-executable code, which facilitate the gradual development of methods and statements. The only other difference with the language of Chapter 2 is that method declarations of the form 'Sig Sd Ss **do** s **end**' do not explicitly indicate whether they introduce or override the method. However, the earlier convention that all non-constructor methods must be listed in all subclasses makes it simple to distinguish these cases.

The proof rules of Chapter 2 are sufficient to verify code that does not use the highlighted productions. Verifying programs that are partially developed needs additional rules for the new constructs. Familiar rules apply to spec and join statements:

$$\frac{}{\Delta; \Gamma \vdash_s \{\mathrm{P}\}\overline{\mathrm{x}}; \overline{\mathrm{y}} : \{\mathrm{P}, \mathrm{Q}\}\{\mathrm{Q}\}}$$

$$\frac{\Delta; \Gamma \vdash_s \{\mathrm{P}\}\mathrm{s}\{\mathrm{Q}\}}{\Delta; \Gamma \vdash_s \{\mathrm{P}\}\bigsqcup(\ldots, \mathrm{s}, \ldots)\{\mathrm{Q}\}}$$

L ::= **class** G Inh Def Feat **end**	
Inh ::= **inherit** H	
Def ::= **define** $\overline{\text{D}}$	
Feat ::= **feature** $\overline{\text{M}}$ $\overline{\text{F}}$	
D ::= x.$\text{p}_G(\overline{\text{t:y}})$ **as** P	*Define clause*
M ::= Sig	*Method declaration*
| Sig Sd	
| Sig Sd Ss	
| Sig Sd Ss **do** s **end**	
| **inherit** Sig Sd Ss	
F ::= f: Type	*Field declaration*
Sig ::= m(Args) Rt	
Sd ::= **dynamic** Spec	*Dynamic specification*
Ss ::= **static** Spec	*Static specification*
Spec ::= {P}_{Q} | {P}_{Q} **also** Spec	*Specification*
s ::= $\overline{\text{x}};\overline{\text{y}}:\{\text{P},\text{Q}\}$	*Spec statement*
| $\bigsqcup(\text{s}_1, \ldots, \text{s}_\text{n})$	*Join statement*
| **skip**	
| x := e	*Assignment*
| s ; s	*Sequential compostition*
| **if** e **then** s **else** s **end**	*Conditional*
| **while** e **do** s **end**	*Loop*
| x: Type. s	*Local variable block*
| x := **new** G	*Object allocation*
| x := y.f	*Field lookup*
| x.f := e	*Field update*
| x := y.m($\overline{\text{e}}$) | y.m($\overline{\text{e}}$)	*Dynamically dispatched call*
| x := y.G::m($\overline{\text{e}}$) | y.G::m($\overline{\text{e}}$)	*Direct method call*
e ::= x | e + e | e = e | **Void** | 0 | 1 | 2 | ...	*Expression*
Type ::= $\overline{\text{INT}}$ | BOOL | G	
Args ::= $\overline{\text{x: Type}}$	*Formal arguments*
Rt ::= ϵ | : Type	*Return type*

Figure 5.6: The extended grammar.

A method signature is always verified:

$$\frac{}{\Delta;\Gamma \vdash_m \text{Sig in G parent H}}$$

In order to reduce duplication and improve readability, the presentation will frequently use holes or placeholders in the premises of rules. These holes can then be filled in different ways to obtain different rules. For example, the rule for verifying a method signature with a dynamic specification has a hole, $[\,\text{Hole}_{\text{B.s}}\,]$, that captures the Behavioral subtyping proof obligation:

$$\frac{[\,\text{Hole}_{\text{B.s.}}\,]}{\Delta;\Gamma \vdash_m \text{Sig Sd in G parent H}}$$

$[\,\text{Hole}_{\text{B.s.}}\,]$ can be filled in two ways:

$$\text{H.m} \notin \mathit{dom}(\Gamma)$$

$$\begin{array}{l}\text{Sd} = \textbf{dynamic}\ \text{Spec}_{\text{G}}\\ \Gamma(\text{H.m}) = (\overline{\text{u}},\text{Spec}_{\text{H}})\\ \Delta \vdash \text{Spec}_{\text{G}} \Longrightarrow \text{Spec}_{\text{H}}\end{array}$$

The rule for signatures with dynamic and static specifications also uses $[\,\text{Hole}_{\text{B.s.}}\,]$, and adds a proof obligation for Dynamic dispatch:

$$\frac{\begin{array}{l}[\,\text{Hole}_{\text{B.s.}}\,]\\ [\,\text{Hole}_{\text{D.d.}}\,]\end{array}}{\Delta;\Gamma \vdash_m \text{Sig Sd Ss in G parent H}}$$

$[\,\text{Hole}_{\text{D.d.}}\,]$ is a placeholder for:

$$\begin{array}{l}\text{Ss} = \textbf{static}\ \text{Spec}_{\text{S}}\\ \text{Sd} = \textbf{dynamic}\ \text{Spec}_{\text{D}}\\ \Delta \vdash \text{Spec}_{\text{S}} \overset{\textbf{Current}:\text{G}}{\Longrightarrow} \text{Spec}_{\text{D}}\end{array}$$

The following definition is convenient for referring to a collection of well-formed and verified classes:

$$\vdash \overline{\text{L}} \overset{\text{def}}{=} \forall\, \text{L} \in \overline{\text{L}} \cdot\ \mathit{apf}(\text{L});\mathit{specs}(\overline{\text{L}}) \vdash_c \text{L}$$

5.3.2 Development calculus

The calculus for developing correct OO programs uses judgements of the form $\overline{\mathrm{L}} \rightsquigarrow \overline{\mathrm{L}'}$. The symbol $\rightsquigarrow$ can be read as 'evolves to' or 'becomes'. To be useful, the calculus must satisfy two properties:

- Soundness: If $\vdash \overline{\mathrm{L}}$ and $\overline{\mathrm{L}} \rightsquigarrow \overline{\mathrm{L}'}$, then $\vdash \overline{\mathrm{L}'}$.
- Completeness: If $\vdash \overline{\mathrm{L}}$ then $\epsilon \rightsquigarrow^* \overline{\mathrm{L}}$.

The soundness property guarantees that the rules will maintain correctness and not introduce errors. The completeness property ensures that any correct program can be developed from scratch with the calculus. In practical terms, this means that the calculus supports the top-down development of every correct program. However, several rules will break with the top-down style and enable more flexible evolution. The rest of this section summarises the calculus.

Adding a class

Any new class needs a fresh name. A new class that does not inherit from any user-defined class inherits no method signatures:

$$\frac{\mathrm{G} \notin \textit{class-names}(\overline{\mathrm{L}}) \cup \{\mathrm{ANY}\}}{\overline{\mathrm{L}} \rightsquigarrow \overline{\mathrm{L}} + \textbf{class}\ \mathrm{G}\ \textbf{define feature end}}$$

Otherwise, it inherits from its parent the signatures of all methods that are not constructors:

$$\frac{\begin{array}{l}\mathrm{G} \notin \textit{class-names}(\overline{\mathrm{L}}) \cup \{\mathrm{ANY}\} \\ \overline{\mathrm{L}} = \overline{\mathrm{L}_1} + \textbf{class}\ \mathrm{H}\ \mathrm{Inh}\ \mathrm{Def}\ \textbf{feature}\ \overline{\mathrm{M}}\ \overline{\mathrm{F}}\ \textbf{end} \\ \overline{\mathrm{Sig}} = \textit{non-constructor-signatures}(\overline{\mathrm{M}})\end{array}}{\overline{\mathrm{L}} \rightsquigarrow \overline{\mathrm{L}} + \textbf{class}\ \mathrm{G}\ \textbf{inherit}\ \mathrm{H}\ \textbf{define feature}\ \overline{\mathrm{Sig}}\ \textbf{end}}$$

Adding an apf entry

Apf entries support data refinement: they describe how a class represents the abstract concepts/structures expressed by apf predicates. Provided that the definition is well-formed, the rule for adding an apf entry is simple:

$$\frac{\begin{array}{l}\overline{\mathrm{L}} = \overline{\mathrm{L}_1} + \textbf{class}\ \mathrm{G}\ \mathrm{Inh}\ \textbf{define}\ \overline{\mathrm{D}}\ \mathrm{Feat}\ \textbf{end} \\ \mathrm{p}_G \notin \textit{defined-apf-entries}(\overline{\mathrm{D}}) \\ \overline{\mathrm{D}'} = \overline{\mathrm{D}} + \mathrm{x}.\mathrm{p}_G(\overline{\mathrm{t}:\mathrm{y}})\ \textbf{as}\ \mathrm{P}\end{array}}{\overline{\mathrm{L}} \rightsquigarrow \overline{\mathrm{L}_1} + \textbf{class}\ \mathrm{G}\ \mathrm{Inh}\ \textbf{define}\ \overline{\mathrm{D}'}\ \mathrm{Feat}\ \textbf{end}}$$

Adding a field

A new field must have a fresh name and a legal type:

$$\frac{\begin{array}{l}\overline{\mathrm{L}} = \overline{\mathrm{L_1}} + \textbf{class}\ \mathrm{G}\ \mathrm{Inh}\ \mathrm{Def}\ \textbf{feature}\ \overline{\mathrm{M}}\ \overline{\mathrm{F}}\ \textbf{end} \\ \mathrm{classes} = \textit{ancestors}(\mathrm{G},\overline{\mathrm{L}}) + \mathrm{G} + \textit{descendants}(\mathrm{G},\overline{\mathrm{L}}) \\ \mathrm{f} \notin \textit{field-names}(\mathrm{classes},\overline{\mathrm{L}}) \\ \mathrm{Type} \in \textit{legal-types}(\overline{\mathrm{L}}) \\ \overline{\mathrm{F}'} = \overline{\mathrm{F}} + \mathrm{f}\colon \mathrm{Type}\end{array}}{\overline{\mathrm{L}} \rightsquigarrow \overline{\mathrm{L_1}} + \textbf{class}\ \mathrm{G}\ \mathrm{Inh}\ \mathrm{Def}\ \textbf{feature}\ \overline{\mathrm{M}}\ \overline{\mathrm{F}'}\ \textbf{end}}$$

Adding a constructor signature

A constructor signature has the same name as its enclosing class and no return type.

$$\frac{\begin{array}{l}\overline{\mathrm{L}} = \overline{\mathrm{L_1}} + \textbf{class}\ \mathrm{G}\ \mathrm{Inh}\ \mathrm{Def}\ \textbf{feature}\ \overline{\mathrm{M}}\ \overline{\mathrm{F}}\ \textbf{end} \\ \mathrm{Sig} = \mathrm{G}(\mathrm{Args}) \\ \text{valid-signature}(\mathrm{Sig},\mathrm{G},\overline{\mathrm{L}}) \\ \overline{\mathrm{M}'} = \overline{\mathrm{M}} + \mathrm{Sig}\end{array}}{\overline{\mathrm{L}} \rightsquigarrow \overline{\mathrm{L_1}} + \textbf{class}\ \mathrm{G}\ \mathrm{Inh}\ \mathrm{Def}\ \textbf{feature}\ \overline{\mathrm{M}'}\ \overline{\mathrm{F}}\ \textbf{end}}$$

The predicate valid-signature(Sig,$\overline{\mathrm{G}}$,$\overline{\mathrm{L}}$) holds iff

1. $\forall \mathrm{G} \in \overline{\mathrm{G}} \cdot$ *name*(Sig) $\notin$ *signature-names*(G,$\overline{\mathrm{L}}$)
2. If Sig has a return type, then it must be in *legal-types*($\overline{\mathrm{L}}$)
3. The argument names in Sig are all distinct
4. The argument types of Sig are all in *legal-types*($\overline{\mathrm{L}}$)

Adding an ordinary method signature

A new ordinary method signature gets added to a class and all its descendants, which can later decide to inherit or override the eventual implementation.

$$\frac{\begin{array}{l}\overline{\mathrm{L}} = \overline{\mathrm{L_1}} + \overline{\mathrm{L_2}} \\ \textit{class-names}(\overline{\mathrm{L_2}}) = \mathrm{G} + \textit{descendants}(\mathrm{G},\overline{\mathrm{L}}) \\ \textit{name}(\mathrm{Sig}) \notin \mathrm{CLASS\text{-}NAMESPACE} \\ \text{valid-signature}(\mathrm{Sig},\textit{class-names}(\overline{\mathrm{L_2}}),\overline{\mathrm{L}}) \\ \overline{\mathrm{L_2}} = +_{\mathrm{i}\in 1..\mathrm{n}}\ \textbf{class}\ \mathrm{H_i}\ \mathrm{Inh_i}\ \mathrm{Def_i}\ \textbf{feature}\ \overline{\mathrm{M_i}}\ \overline{\mathrm{F_i}}\ \textbf{end} \\ \overline{\mathrm{L_2'}} = +_{\mathrm{i}\in 1..\mathrm{n}}\ \textbf{class}\ \mathrm{H_i}\ \mathrm{Inh_i}\ \mathrm{Def_i}\ \textbf{feature}\ \overline{\mathrm{M_i}} + \mathrm{Sig}\ \overline{\mathrm{F_i}}\ \textbf{end}\end{array}}{\overline{\mathrm{L}} \rightsquigarrow \overline{\mathrm{L_1}} + \overline{\mathrm{L_2'}}}$$

Method manipulation

All remaining rules manipulate methods, and are based on the following rule:

$$\frac{\begin{array}{l}\overline{\mathrm{L}} = \overline{\mathrm{L_1}} + \mathrm{L} \\ \mathrm{L} = \textbf{class } \mathrm{G} \textbf{ inherit } \mathrm{H\ Def} \textbf{ feature } \overline{\mathrm{M}}\ \overline{\mathrm{F}} \textbf{ end} \\ \Gamma = \mathit{specs}(\overline{\mathrm{L}}) \\ \Delta = \mathit{apf}(\mathrm{L}) \\ \overline{\mathrm{M}} = \overline{\mathrm{M_1}} + \mathrm{M} \\ [\,\mathrm{Hole_a}\,] \\ \overline{\mathrm{M}'} = \overline{\mathrm{M_1}} + \mathrm{M}'\end{array}}{\overline{\mathrm{L}} \rightsquigarrow \overline{\mathrm{L_1}} + \textbf{class } \mathrm{G} \textbf{ inherit } \mathrm{H\ Def} \textbf{ feature } \overline{\mathrm{M}'}\ \overline{\mathrm{F}} \textbf{ end}}$$

Adding a dynamic specification

The rule for adding a dynamic specification to a method signature fills $[\,\mathrm{Hole_a}\,]$ as follows:

$$\begin{array}{l}\mathrm{M} = \mathrm{m(Args)\ Rt} \\ [\,\mathrm{Hole_{B.s.}}\,] \\ \mathrm{M}' = \mathrm{m(Args)\ Rt\ Sd}\end{array}$$

Adding a static specification

Static specifications are also related to data refinement, since they can state how an operation on abstract structures/concepts, as mentioned in the dynamic specification, is realised in more concrete terms. The rule for adding a static specification fills $[\,\mathrm{Hole_a}\,]$ with:

$$\begin{array}{l}\mathrm{M} = \mathrm{Sig\ Sd} \\ [\,\mathrm{Hole_{D.d.}}\,] \\ \mathrm{M}' = \mathrm{Sig\ Sd\ Ss}\end{array}$$

Adding a method body

The calculus supports the refinement of method bodies, so the initial body can well be a single spec statement, as this filling for $[\,\mathrm{Hole_a}\,]$ shows:

$$\begin{array}{l}\mathrm{M} = \mathrm{Sig\ Sd\ Ss} \\ \mathrm{Sig} = \mathrm{m(Args)\ Rt} \\ \mathrm{Ss} = \textbf{static } \{\mathrm{S}\}\_\{\mathrm{T}\} \\ [\,\mathrm{Hole_b}\,] \\ \overline{\mathrm{y}} = \overline{\mathrm{x}} + \mathit{names}(\mathrm{Args}) \\ \mathrm{M}' = \mathrm{Sig\ Sd\ Ss} \textbf{ do } \overline{\mathrm{x}}; \overline{\mathrm{y}} : \{\mathrm{S}, \mathrm{T}\} \textbf{ end}\end{array}$$

The purpose of $[\,\mathrm{Hole_b}\,]$ is to constrain $\overline{\mathrm{x}}$, the variable(s) that may be modified, appropriately. It has two fillings:

$$\begin{array}{l}\mathrm{Rt} = \epsilon \\ \overline{\mathrm{x}} = \epsilon\end{array}$$

$$\begin{array}{l}\mathrm{Rt} \neq \epsilon \\ \overline{\mathrm{x}} = \textbf{Result}\end{array}$$

Removing a method body

This gives the opportunity to implement the body in a different way, or to inherit a parent version. The filling for $[\,\mathrm{Hole_a}\,]$ is simple:

$$\begin{array}{l}\mathrm{M} = \mathrm{Sig\ Sd\ Ss}\ \textbf{do}\ \mathrm{s}\ \textbf{end} \\ \mathrm{M}' = \mathrm{Sig\ Sd\ Ss}\end{array}$$

This is an example of a rule that relaxes the top-down style of development.

Inheriting a method body

Inheriting a method body imposes the Inheritance proof obligation, as the filling for $[\,\mathrm{Hole_a}\,]$ shows:

$$\begin{array}{l}\mathrm{M} = \mathrm{Sig\ Sd\ Ss} \\ \mathrm{Sig} = \mathrm{m(Args)\ Rt} \\ \mathrm{Ss} = \textbf{static}\ \mathrm{Spec_G} \\ \Gamma(\mathrm{H::m}) = (\overline{\mathrm{u}},\mathrm{Spec_H}) \\ \Delta \vdash \mathrm{Spec_H} \Longrightarrow \mathrm{Spec_G} \\ \mathrm{M}' = \textbf{inherit}\ \mathrm{Sig\ Sd\ Ss}\end{array}$$

Uninheriting a method body

The next filling for $[\,\mathrm{Hole_a}\,]$ can be used to override a method's implementation instead of inheriting it:

$$\begin{array}{l}\mathrm{M} = \textbf{inherit}\ \mathrm{Sig\ Sd\ Ss} \\ \mathrm{M}' = \mathrm{Sig\ Sd\ Ss}\end{array}$$

Statement refinement

This is where the result of freefinement is incorporated in the development calculus. It fills $[\,\mathrm{Hole_a}\,]$ with:

$$\begin{array}{l}\mathrm{M} = \mathrm{Sig\ Sd\ Ss}\ \textbf{do}\ \mathrm{s}\ \textbf{end} \\ \Delta;\Gamma \vdash \mathrm{s} \sqsubseteq \mathrm{s}' \\ \mathrm{M}' = \mathrm{Sig\ Sd\ Ss}\ \textbf{do}\ \mathrm{s}'\ \textbf{end}\end{array}$$

5.4 Related Work

In his work on refinement for the lambda calculus, Denney [20] treats types as rudimentary specifications and introduces a specification construct $?_\tau$ for each type τ. Conceptually, $?_\tau$ corresponds to $[\Gamma; \tau]$ where the context Γ is left implicit. For example, consider the term $\lambda x : \sigma.\, ?_\tau$ in the context Γ. The $?_\tau$ inside the term corresponds to $[\Gamma, x : \sigma; \tau]$. Denney also considers richer specifications for lambda terms in his PhD thesis [21]. This results in a more powerful refinement calculus in which specification constructs can contain logical assumptions.

The specification statement $\overline{x} : [P, Q]$ of Morgan [49] is analogous to $\overline{x}; Var : \{P, Q\}$, since there is no restriction on the variables that the statement may read. However, his specification statement is a total correctness specification, and the accompanying refinement calculus [50] establishes total correctness. Similar refinement calculi for total correctness were proposed by Back [1, 2], Morris [51] and Hehner [31]. The books [50, 2, 31] contain many examples of how correct programs can be constructed from specifications via refinement.

Leino and Manohar [43] consider the join of Morgan's specification statements $\overline{x} : [P_1, Q_1]$ and $\overline{x} : [P_2, Q_2]$, and mention several of its uses. Free-finement adds explicit constructors for joins, and relies on the ability to join arbitrary terms from U in order to establish harmony.

Several systems in the literature provide refinement and correctness by construction for OO programs [4, 48, 6, 5, 55]. The calculus in Section 5.3 is rather minimal and not as elaborate as some published ones. Nevertheless, several characteristics set it apart from existing proposals:

1. It is faithful to the OO paradigm and popular OO languages. It accommodates references, shared mutable state, inheritance and dynamic dispatch.

2. It uses state-of-the-art mechanisms that facilitate modular and flexible reasoning, such as separation logic and abstract predicate families.

3. It is not postulated in isolation. The calculus has a clear relationship with an accepted system for verifying OO programs in a bottom-up way.

There is a relationship between observational equivalence of terms and the function Specs, because $\models_{\mathsf{V}_1} \_ \mathit{Sat} \_$ gives rise to a notion of observability from the specification point of view. In particular, two terms t and t′ are observationally equivalent in this sense iff $t \sim t'$, where

$$t \sim t' \stackrel{\text{def}}{=} \text{Specs}(\{t\}) = \text{Specs}(\{t'\})$$

It is trivial to check that $\sim$ is an equivalence relation. If $\models_{\mathsf{V}_1 \_}\ \mathit{Sat}\ \_$ is well-behaved, then $t \sim t' \Leftrightarrow [\![t]\!] = [\![t']\!]$ (i.e. $t \sim t' \Leftrightarrow t \equiv t'$) by Corollary B.1.9 in Appendix B and Theorem 5.16.1.

Chapter 6

Conclusion

The state of the art in reasoning about software has seen several recent advances. In particular, proof systems that are based on separation logic emerged to provide intuitive, flexible and modular reasoning for OO programs. Their success is due to the synergy of the following insights:

- Separation is a natural way to think about shared mutable data. It leads to intuitive and simple correctness proofs for code that manipulates references.
- Abstract predicate families befit reasoning about object-orientation. They are logical abstractions that encapsulate state and hide representation details. Subclasses can change the representation and still satisfy the inherited specifications of routines.
- Giving two specifications to each routine resolves the tension between saying in detail what the particular body achieves, and saying what the general idea of the routine is that all subclasses must respect. Using separation logic and apfs in these specifications yields elegant proofs for OO programs.

The existing work forms a solid foundation for further investigation and research, which can of course proceed in many directions. This thesis tackled the following shortcomings in previous work:

- The basic apf mechanism can be too restrictive. Code sometimes depends on how a particular class implements and/or relates these abstractions, or properties that a whole class hierarchy will fulfill. This is especially important in the presence of multiple inheritance, where a class can relate and combine several concepts in various ways.

- Separation logic verification excluded the executable contracts that are present in existing OO code. These contracts may be weak and/or perform side-effects. They nevertheless capture useful design information which can be exploited in reasoning.

- It is sometimes easier to write code that is correct in the first place instead of hacking away and trying to verify code afterwards. Supporting this requires calculi for correctness by construction and refinement.

All the proposed solutions fit in nicely with the existing work of Parkinson and others [57, 58]. In particular:

- Export and axiom clauses can specify properties of and relationships between apfs that hold for particular classes and whole class hierarchies. The information is first verified and then made available as ordinary assumptions to reason about the code. Together with other generalisations, export and axiom clauses allow flexible reasoning about multiple inheritance.

- Separation logic can be used to verify that executable contracts will always hold at runtime. Central to such reasoning is the new concept of relative purity, which embraces side-effects in executable contracts.

- Freefinement can automatically provide a refinement calculus that interoperates smoothly with the rules for statement verification. This refinement calculus fits into a larger calculus for correctness by construction, which complements the separation logic system for bottom-up verification.

The contributions of this thesis thus advance the state of the art in the verification of existing OO programs, and in the development of new programs that are correct by design. The ultimate hope is that, in the long term, such techniques will encourage and enable engineers to produce software that is more reliable.

Appendix A

Semantics of the proof system

This appendix outlines the semantics and soundness proof of the verification system of chapter 3. Since our system is a conservative extension of Parkinson and Bierman's system, the presentation closely follows the one in [58]. The most interesting difference is the treatment of export and axiom information in the soundness proof of the program verification rule (Theorem A.4 below).

The semantics of the logical formula is defined in terms of a state σ, an interpretation of predicate symbols $\mathcal{I}$, and an interpretation of logical variables $\mathcal{L}$. The interpretation $\mathcal{I}$ maps predicate names to their definitions, whereas a definition maps a list of arguments to a set of states:

$$\begin{array}{lcl} \mathcal{I} & : & \mathsf{Preds} \to (\mathsf{Vals}^* \to \mathcal{P}(\Sigma)) \\ \mathcal{L} & : & \mathsf{Vars} \to \mathsf{Vals} \end{array}$$

Predicates are defined in the standard way:

$$\sigma, \mathcal{I}, \mathcal{L} \models \mathrm{pred}(\overline{\mathrm{X}}) \Leftrightarrow \sigma \in (\mathcal{I}(\mathrm{pred})(\mathcal{L}(\overline{\mathrm{X}})))$$

Definition A.1. $\mathcal{I} \models \Delta$ iff $\sigma, \mathcal{I}, \mathcal{L} \models \Delta$ for all σ and $\mathcal{L}$.

Under mild syntactic restrictions on predicate definitions, obeyed in this thesis and detailed in [57], one can show that every set of disjoint predicate definitions is satisfiable:

Lemma A.1. For any set of definitions $\mathrm{W}_1, \ldots \mathrm{W}_\mathrm{m}$, $\mathrm{D}_1, \ldots \mathrm{D}_\mathrm{n}$ where W_i has form $\mathrm{w}_\mathrm{i}(\overline{\mathrm{x}_\mathrm{i}}) = \mathrm{Q}_\mathrm{i}$ and D_j is listed in class G_j, there exists an interpretation $\mathcal{I}$ such that $\mathcal{I} \models [\bigwedge_{\mathrm{i}\in 1..\mathrm{m}} \forall \overline{\mathrm{x}_\mathrm{i}}.\ \mathrm{w}_\mathrm{i}(\overline{\mathrm{x}_\mathrm{i}}) \Leftrightarrow \mathrm{Q}_\mathrm{i}] \wedge [\bigwedge_{\mathrm{j}\in 1..\mathrm{n}} \mathit{apf}_{\mathrm{G}_\mathrm{j}}(\mathrm{D}_\mathrm{j})]$ provided that no two distinct definitions in the set define the same predicate.

The semantics of our proof system's judgements is defined next. We do not define the semantics of $\vdash_e$ and $\vdash_a$ explicitly, since we work with their premises (valid logical formulae whose existence is guaranteed) instead. For triples, the usual partial-correctness semantics for separation logic is used: if the precondition holds in the start state, then 1) the statement will not fault (access unallocated memory, for example), and 2) if the statement terminates, then the postcondition holds in the resulting state.

Definition A.2. $\mathcal{I} \models_{\mathrm{n}} \{\mathrm{P}\}\mathrm{s}\{\mathrm{Q}\}$ iff whenever σ, $\mathcal{I}$, $\mathcal{L} \models \mathrm{P}$ then $\forall \mathrm{m} \leq \mathrm{n}\cdot$

1. σ,s $\longrightarrow^{\mathrm{m}}$ fault does not hold, and
2. if σ,s $\longrightarrow^{\mathrm{m}}$ σ',**skip** then σ', $\mathcal{I}$, $\mathcal{L} \models \mathrm{Q}$

The index n deals with mutual recursion in method definitions. $\mathcal{I} \models_{\mathrm{n}} \Gamma$ means that all methods in Γ meet their specifications when executed for up to n steps.

Definition A.3 (Method verification semantics)**.** If m in G is non-abstract, let s denote its body.

$\mathcal{I}, \Gamma \models_0 \mathrm{G.m} \mapsto (\overline{\mathrm{x}},\{\mathrm{P}\}\_\{\mathrm{Q}\})$ always holds.
$\mathcal{I}, \Gamma \models_{\mathrm{n+1}} \mathrm{G.m} \mapsto (\overline{\mathrm{x}},\{\mathrm{P}\}\_\{\mathrm{Q}\})$ iff
$\mathcal{I} \models_{\mathrm{n}} \Gamma \Rightarrow \mathcal{I} \models_{\mathrm{n+1}} \{\mathrm{P} * \mathbf{Current} : \mathrm{G}\}\mathrm{s}\{\mathrm{Q}\}$
if G is non-abstract and true otherwise.

$\mathcal{I}, \Gamma \models_0 \mathrm{G::m} \mapsto (\overline{\mathrm{x}},\{\mathrm{S}\}\_\{\mathrm{T}\})$ always holds.
$\mathcal{I}, \Gamma \models_{\mathrm{n+1}} \mathrm{G::m} \mapsto (\overline{\mathrm{x}},\{\mathrm{S}\}\_\{\mathrm{T}\})$ iff $\mathcal{I} \models_{\mathrm{n}} \Gamma \Rightarrow \mathcal{I} \models_{\mathrm{n+1}} \{\mathrm{S}\}\mathrm{s}\{\mathrm{T}\}$

$\mathcal{I} \models_{\mathrm{n}} \Gamma$ iff $\forall \mathrm{methodspec} \in \Gamma\cdot\ \mathcal{I}, \Gamma \models_{\mathrm{n}} \mathrm{methodspec}$

We next define the semantics of the statement judgement.

Definition A.4. $\Delta; \Gamma \models \{\mathrm{P}\}\mathrm{s}\{\mathrm{Q}\}$ iff for all $\mathcal{I}$ and n, if $\mathcal{I} \models \Delta$ and $\mathcal{I} \models_{\mathrm{n}} \Gamma$, then $\mathcal{I} \models_{\mathrm{n+1}} \{\mathrm{P}\}\mathrm{s}\{\mathrm{Q}\}$

In other words, for all interpretations which satisfy the assumptions Δ, if all methods in Γ meet their specifications for up to n steps, then s meets its specification for up to n + 1 steps.

The judgements are sound with respect to their semantics.

Lemma A.2.

1. If $\Delta; \Gamma \vdash_m \ldots \mathrm{m} \ldots$, then $\forall\mathcal{I}\cdot$ if $\mathcal{I} \models \Delta$ then for all n and every spec of m we have $\mathcal{I}, \Gamma \models_{\mathrm{n}} \mathrm{spec}$
2. If $\Delta; \Gamma \vdash_s \{\mathrm{P}\}\mathrm{s}\{\mathrm{Q}\}$ then $\Delta; \Gamma \models \{\mathrm{P}\}\mathrm{s}\{\mathrm{Q}\}$

Whenever a judgement is derivable under weak assumptions, it can also be derived under stronger ones.

Lemma A.3.

1. If $\Delta; \Gamma \vdash_m \ldots \mathrm{m} \ldots$ and $\Delta' \Rightarrow \Delta$, then $\Delta'; \Gamma \vdash_m \ldots \mathrm{m} \ldots$
2. If $\Delta; \Gamma \vdash_s \{\mathrm{P}\}\mathrm{s}\{\mathrm{Q}\}$ and $\Delta' \Rightarrow \Delta$, then $\Delta'; \Gamma \vdash_s \{\mathrm{P}\}\mathrm{s}\{\mathrm{Q}\}$
3. If $\Delta_{APF}, \Delta_E, \Delta_A; \Gamma \vdash_c \mathrm{L}$ and $\Delta' \Rightarrow \Delta_{APF}$, then $\Delta', \Delta_E, \Delta_A; \Gamma \vdash_c \mathrm{L}$

Finally, here is the soundness statement and detailed proof sketch of the program verification rule.

Theorem A.4. If a program and its main body s can be proved with the program verification rule, then $\forall \mathcal{I}, \mathrm{n}\cdot\ \mathcal{I} \models_{\mathrm{n}} \{\mathrm{true}\}\mathrm{s}\{\mathrm{true}\}$.
Proof.

1. *The goal.* We have to prove $\forall \mathcal{I}, \mathrm{n}\cdot\ \mathcal{I} \models_{\mathrm{n}} \{\mathrm{true}\}\mathrm{s}\{\mathrm{true}\}$, which abbreviates $\forall \mathcal{I}, \mathrm{n}\cdot$ whenever σ, $\mathcal{I}$, $\mathcal{L} \models \mathrm{true}$, then $\forall \mathrm{m} \leq \mathrm{n}\cdot$ 1) $\sigma, \mathrm{s} \longrightarrow^{\mathrm{m}} \mathsf{fault}$ does not hold, and 2) if $\sigma, \mathrm{s} \longrightarrow^{\mathrm{m}} \sigma', \mathbf{skip}$ then σ', $\mathcal{I}$, $\mathcal{L} \models \mathrm{true}$. This can be simplified to $\forall \mathrm{n}\cdot\ \sigma, \mathrm{s} \longrightarrow^{\mathrm{n}} \mathsf{fault}$ does not hold.

2. *Strengthened assumptions.* Let $\Delta_T \stackrel{\mathrm{def}}{=} \bigwedge_{\mathrm{i} \in 1..\mathrm{t}} apf(\mathrm{L_i})$, where $\mathrm{L_1} \ldots \mathrm{L_t}$ are all classes in the program. By Lemma A.3, we can strengthen the assumptions under which all classes and the main body have been verified. For every class $\mathrm{L_i}$, we have Δ_T, Δ_E, Δ_A; $\Gamma \vdash_c \mathrm{L_i}$, and $\Delta_T \wedge \Delta_E \wedge \Delta_A$; $\Gamma \vdash_s \{\mathrm{true}\}\mathrm{s}\{\mathrm{true}\}$ also holds for the main body s.

3. *The interpretation $\mathcal{I}'$.* Since $\Delta_T \wedge \Delta_E \wedge \Delta_A$; $\Gamma \vdash_s \{\mathrm{true}\}\mathrm{s}\{\mathrm{true}\}$, Lemma A.2 guarantees $\Delta_T \wedge \Delta_E \wedge \Delta_A$; $\Gamma \models \{\mathrm{true}\}\mathrm{s}\{\mathrm{true}\}$. This abbreviates $\forall \mathcal{I}, \mathrm{n}\cdot$ if $\mathcal{I} \models \Delta_T \wedge \Delta_E \wedge \Delta_A$ and $\mathcal{I} \models_{\mathrm{n}} \Gamma$, then $\mathcal{I} \models_{\mathrm{n+1}} \{\mathrm{true}\}\mathrm{s}\{\mathrm{true}\}$, which can be simplified to $\forall \mathcal{I}, \mathrm{n}\cdot$ if $\mathcal{I} \models \Delta_T \wedge \Delta_E \wedge \Delta_A$ and $\mathcal{I} \models_{\mathrm{n}} \Gamma$, then $\forall \mathrm{m} \leq \mathrm{n}+1\cdot\ \sigma, \mathrm{s} \longrightarrow^{\mathrm{m}} \mathsf{fault}$ does not hold. Now if we can find an $\mathcal{I}'$ such that $\mathcal{I}' \models \Delta_T \wedge \Delta_E \wedge \Delta_A$ and $\forall \mathrm{n}\cdot\ \mathcal{I}' \models_{\mathrm{n}} \Gamma$, then we can instantiate $\mathcal{I}$ to $\mathcal{I}'$ in the formula and simplify to obtain $\forall \mathrm{n}\cdot\ \sigma, \mathrm{s} \longrightarrow^{\mathrm{n}} \mathsf{fault}$ does not hold. Therefore $\mathcal{I}'$ serves as a witness that s will never fault, which is exactly our goal.

   Let $\mathcal{I}'$ be the interpretation whose existence is guaranteed by Lemma A.1 for all the where and define clauses in the program. Clearly $\mathcal{I}' \models \Delta_T$. We next prove $\mathcal{I}' \models \Delta_E$ and then $\mathcal{I}' \models \Delta_A$.

4. *Satisfiability of* Δ_E. Consider an arbitrary export clause $\mathrm{E} = \mathrm{P}$ **where** $\{\mathrm{w}_1(\overline{\mathrm{x}_1}) = \mathrm{Q}_1;\ \ldots;\ \mathrm{w}_\mathrm{n}(\overline{\mathrm{x}_\mathrm{n}}) = \mathrm{Q}_\mathrm{n}\}$ in class L. Since $apf(\mathrm{L}) \vdash_e \mathrm{E}$, we know $[apf(\mathrm{L}) \wedge (\bigwedge_{\mathrm{i}\in 1..\mathrm{n}} \forall \overline{\mathrm{x}_\mathrm{i}} \cdot \mathrm{w}_\mathrm{i}(\overline{\mathrm{x}_\mathrm{i}}) \Leftrightarrow \mathrm{Q}_\mathrm{i})] \Rightarrow \mathrm{P}$. The interpretation $\mathcal{I}'$ satisfies the antecedent, so we also have $\mathcal{I}' \models \mathrm{P}$. Therefore $\mathcal{I}' \models \Delta_E$, and $\mathcal{I}' \models \Delta_T \wedge \Delta_E$.

5. *Satisfiability of* Δ_A. We prove this by induction. If class G has children $\mathrm{H}_1 \ldots \mathrm{H}_\mathrm{k}$, let $level(\mathrm{G}) \stackrel{\text{def}}{=} 1 + \max(0, level(\mathrm{H}_1), \ldots, level(\mathrm{H}_\mathrm{k}))$. Furthermore, $P(\mathrm{n}) \stackrel{\text{def}}{=} \forall$ G in the program such that $level(\mathrm{G}) \leq \mathrm{n}$ and for all axiom clauses a: P in the listing of G, $(\Delta_T \wedge \Delta_E) \Rightarrow axiominfo(\mathrm{G}, \mathrm{a}: \mathrm{P})$.

   - Base case. Consider an arbitrary class G with $level(\mathrm{G}) \leq 1$ and an axiom clause a: P appearing in it. G has no subclasses, and
     (a) If G is abstract, there are no objects with dynamic type G or a subtype thereof, thus $axiominfo(\mathrm{G}, \mathrm{a}: \mathrm{P})$ holds vacuously and $\Delta_T \wedge \Delta_E$ implies it.
     (b) If G is non-abstract, then the only objects whose dynamic type is a subtype of G are direct instances of G. Since $(\Delta_T \wedge \Delta_E \wedge \mathbf{Current} : \mathrm{G}) \Rightarrow \mathrm{P}$ by the Implication premise, we therefore also know $(\Delta_T \wedge \Delta_E) \Rightarrow axiominfo(\mathrm{G}, \mathrm{a}: \mathrm{P})$.

     Thus $P(1)$ holds.
   - Step case. Suppose $P(\mathrm{n})$ holds. Now consider a class G at level n+1 with axiom clause a: P. Every child H of G must list a, say a: Q. By the induction hypothesis we know $(\Delta_T \wedge \Delta_E) \Rightarrow axiominfo(\mathrm{H}, \mathrm{a}: \mathrm{Q})$, and by the Parent Consistency premise of a: Q we know $(\Delta_T \wedge \Delta_E \wedge \mathrm{Q}) \Rightarrow \mathrm{P}$. Therefore $(\Delta_T \wedge \Delta_E) \Rightarrow axiominfo(\mathrm{H}, \mathrm{a}: \mathrm{P})$. We have $(\Delta_T \wedge \Delta_E) \Rightarrow axiominfo(\mathrm{G}, \mathrm{a}: \mathrm{P})$ if G is abstract, and the same holds if G is non-abstract since the Implication premise of a: P guarantees $(\Delta_T \wedge \Delta_E \wedge \mathbf{Current} : \mathrm{G}) \Rightarrow \mathrm{P}$. Thus $P(\mathrm{n}+1)$ holds.

   So $\mathcal{I}' \models \Delta_T \wedge \Delta_E \wedge \Delta_A$.

6. *Wrapping up.* We still have to prove $\forall \mathrm{n} \cdot \mathcal{I}' \models_\mathrm{n} \Gamma$. Let m be an arbitrary method in the program. Since $\Delta_T \wedge \Delta_E \wedge \Delta_A; \Gamma \vdash_m \mathrm{m}$, by Lemma A.2 we know for all n and every spec of m that $\mathcal{I}'$, $\Gamma \models_\mathrm{n} \mathrm{spec}$. Thus $\forall \mathrm{n} \cdot \forall \mathrm{methodspec} \in \Gamma \cdot \mathcal{I}'$, $\Gamma \models_\mathrm{n} \mathrm{methodspec}$, in other words $\forall \mathrm{n} \cdot \mathcal{I}' \models_\mathrm{n} \Gamma$. □

Appendix B

Antitone Galois connections

Lemma 5.2 established that an antitone Galois connection exists between the functions Specs and Terms:

$$X \subseteq \text{Terms}(Y) \Leftrightarrow Y \subseteq \text{Specs}(X) \qquad (*)$$

Theorems derived from this equivalence come in pairs because of the symmetry between Specs and Terms. Here are a few well-known ones together with their proofs:

Corollary B.1.

B.1.1. $X \subseteq \text{Terms}(\text{Specs}(X))$

B.1.2. $Y \subseteq \text{Specs}(\text{Terms}(Y))$

B.1.3. $X \subseteq X' \Rightarrow \text{Specs}(X) \supseteq \text{Specs}(X')$

B.1.4. $Y \subseteq Y' \Rightarrow \text{Terms}(Y) \supseteq \text{Terms}(Y')$

B.1.5. $X \subseteq X' \Rightarrow \text{Terms}(\text{Specs}(X)) \subseteq \text{Terms}(\text{Specs}(X'))$

B.1.6. $Y \subseteq Y' \Rightarrow \text{Specs}(\text{Terms}(Y)) \subseteq \text{Specs}(\text{Terms}(Y'))$

B.1.7. $\text{Specs}(\text{Terms}(\text{Specs}(X))) = \text{Specs}(X)$

B.1.8. $\text{Terms}(\text{Specs}(\text{Terms}(Y))) = \text{Terms}(Y)$

B.1.9. $\text{Specs}(X) \subseteq \text{Specs}(X') \Leftrightarrow \text{Terms}(\text{Specs}(X)) \supseteq \text{Terms}(\text{Specs}(X'))$

B.1.10. $\text{Terms}(Y) \subseteq \text{Terms}(Y') \Leftrightarrow \text{Specs}(\text{Terms}(Y)) \supseteq \text{Specs}(\text{Terms}(Y'))$

B.1.11. $\mathrm{Specs}(X \cup X') = \mathrm{Specs}(X) \cap \mathrm{Specs}(X')$

B.1.12. $\mathrm{Terms}(Y \cup Y') = \mathrm{Terms}(Y) \cap \mathrm{Terms}(Y')$

Proof.

B.1.1. In $(*)$, instantiate Y with $\mathrm{Specs}(X)$.

B.1.3. $X \subseteq$ "Assumption" $X' \subseteq$ "B.1.1" $\mathrm{Terms}(\mathrm{Specs}(X'))$. In $(*)$, instantiate Y with $\mathrm{Specs}(X')$.

B.1.5. If $X \subseteq X'$, then $\mathrm{Specs}(X) \supseteq \mathrm{Specs}(X')$ holds by B.1.3. The result follows from B.1.4.

B.1.7. From B.1.1 and B.1.3 follows $\mathrm{Specs}(X) \supseteq \mathrm{Specs}(\mathrm{Terms}(\mathrm{Specs}(X)))$. Instantiating Y with $\mathrm{Specs}(X)$ in B.1.2 yields $\mathrm{Specs}(X) \subseteq \mathrm{Specs}(\mathrm{Terms}(\mathrm{Specs}(X)))$.

B.1.9. $\Rightarrow$ holds by B.1.4. From $\mathrm{Terms}(\mathrm{Specs}(X)) \supseteq \mathrm{Terms}(\mathrm{Specs}(X'))$ and B.1.3, $\mathrm{Specs}(\mathrm{Terms}(\mathrm{Specs}(X))) \subseteq \mathrm{Specs}(\mathrm{Terms}(\mathrm{Specs}(X')))$. $\mathrm{Specs}(X) \subseteq \mathrm{Specs}(X')$ by B.1.7.

B.1.11. Proof by indirect equality. For arbitrary Y:

$$
\begin{array}{ll}
 & Y \subseteq \mathrm{Specs}(X \cup X') \\
\Leftrightarrow & \{\text{By } (*)\} \\
 & X \cup X' \subseteq \mathrm{Terms}(Y) \\
\Leftrightarrow & \{\text{Set theory}\} \\
 & X \subseteq \mathrm{Terms}(Y) \ \wedge \ X' \subseteq \mathrm{Terms}(Y) \\
\Leftrightarrow & \{\text{By } (*)\} \\
 & Y \subseteq \mathrm{Specs}(X) \ \wedge \ Y \subseteq \mathrm{Specs}(X') \\
\Leftrightarrow & \{\text{Set theory}\} \\
 & Y \subseteq \mathrm{Specs}(X) \cap \mathrm{Specs}(X')
\end{array}
$$

□

Bibliography

[1] Ralph-Johan Back. Correctness preserving program refinements: Proof theory and applications. *Mathematical Centre Tracts*, 131, 1980.

[2] Ralph-Johan Back and Joakim von Wright. *Refinement Calculus: A Systematic Introduction*. Springer-Verlag, 1998. Graduate Texts in Computer Science.

[3] Mike Barnett, K. Rustan M. Leino, and Wolfram Schulte. The Spec# programming system: An overview. In *CASSIS '05*, volume 3362 of *LNCS*, pages 49–69. Springer, 2005.

[4] Paul L. Bergstein. Object-preserving class transformations. In *Conference proceedings on Object-oriented programming systems, languages, and applications*, OOPSLA '91, pages 299–313, New York, NY, USA, 1991. ACM.

[5] Paulo Borba, Augusto Sampaio, Ana Cavalcanti, and Márcio Cornélio. Algebraic reasoning for object-oriented programming. *Science of Computer Programming*, 52(1-3):53–100, 2004.

[6] Paulo Borba, Augusto Sampaio, and Márcio Cornélio. A refinement algebra for object-oriented programming. In Luca Cardelli, editor, *ECOOP '03: Object-Oriented Programming*, volume 2743 of *Lecture Notes in Computer Science*, pages 457–482. Springer Berlin Heidelberg, 2003.

[7] Stephen Brookes. A semantics for concurrent separation logic. *Theor. Comput. Sci.*, 375:227–270, April 2007.

[8] James Brotherston, Richard Bornat, and Cristiano Calcagno. Cyclic proofs of program termination in separation logic. In *POPL '08: Proceedings of the 35th annual ACM SIGPLAN-SIGACT symposium on Principles of programming languages*, pages 101–112, New York, NY, USA, 2008. ACM.

[9] Lilian Burdy, Yoonsik Cheon, David R. Cok, Michael D. Ernst, Joseph R. Kiniry, Gary T. Leavens, K. Rustan M. Leino, and Erik Poll. An overview of JML tools and applications. *International Journal on Software Tools for Technology Transfer (STTT)*, 7(3):212–232, 2005.

[10] Cristiano Calcagno, Dino Distefano, Peter W. O'Hearn, and Hongseok Yang. Compositional shape analysis by means of bi-abduction. In *POPL '09: Proceedings of the 36th annual ACM SIGPLAN-SIGACT symposium on Principles of programming languages*, pages 289–300, New York, NY, USA, 2009. ACM.

[11] Luca Cardelli. A semantics of multiple inheritance. *Inf. Comput.*, 76(2-3):138–164, 1988.

[12] Yoonsik Cheon, Gary Leavens, Murali Sitaraman, and Stephen Edwards. Model variables: cleanly supporting abstraction in design by contract: Research articles. *Softw. Pract. Exper.*, 35(6):583–599, 2005.

[13] Yoonsik Cheon and Gary T. Leavens. A runtime assertion checker for the Java modeling language (JML). In *Proceedings of the international conference on Software engineering research and practice (SERP '02), Las Vegas*, pages 322–328. CSREA Press, 2002.

[14] Wei-Ngan Chin, Cristina David, Huu Hai Nguyen, and Shengchao Qin. Enhancing modular OO verification with separation logic. In *POPL '08: Proceedings of the 35th annual ACM SIGPLAN-SIGACT symposium on Principles of programming languages*, pages 87–99, New York, NY, USA, 2008. ACM.

[15] Ilinca Ciupa, Andreas Leitner, Manuel Oriol, and Bertrand Meyer. Experimental assessment of random testing for object-oriented software. In *ISSTA '07: Proceedings of the 2007 international symposium on Software testing and analysis*, pages 84–94, New York, NY, USA, 2007. ACM.

[16] D. Clarke, S. Drossopoulou, P. Müller, J. Noble, and T. Wrigstad. Aliasing, confinement, and ownership in object-oriented programming (IWACO). In *Object-Oriented Technology. ECOOP 2008 Workshop Reader*, volume 5475 of *LNCS*, pages 30–41. Springer, 2008.

[17] Á. Darvas and K. Rustan M. Leino. Practical reasoning about invocations and implementations of pure methods. In *FASE '07*, volume 4422 of *LNCS*, pages 336–351. Springer, 2007.

[18] Á. Darvas and P. Müller. Reasoning about method calls in interface specifications. *Journal of Object Technology (JOT)*, 5(5):59–85, June 2006.

[19] Ádám Darvas and Peter Müller. Faithful mapping of model classes to mathematical structures. *IET Software*, 2(6):477–499, 2008.

[20] Ewen Denney. Simply-typed underdeterminism. *Journal of Computer Science and Technology*, 13:491–508, 1998.

[21] Ewen Denney. A theory of program refinement. Technical Report ECS-LFCS-99-412, University of Edinburgh, 1999.

[22] Edsger W. Dijkstra. The humble programmer. *Commun. ACM*, 15(10):859–866, October 1972.

[23] Dino Distefano and Matthew J. Parkinson J. jStar: towards practical verification for Java. In *OOPSLA '08: Proceedings of the 23rd ACM SIGPLAN conference on Object-oriented programming systems languages and applications*, pages 213–226, New York, NY, USA, 2008. ACM.

[24] Johan Dovland, Einar Broch Johnsen, Olaf Owe, and Martin Steffen. Incremental reasoning for multiple inheritance. In *IFM '09*, pages 215–230, Berlin, Heidelberg, 2009. Springer.

[25] Sophia Drossopoulou, Adrian Francalanza, Peter Müller, and Alexander J. Summers. A unified framework for verification techniques for object invariants. In Jan Vitek, editor, *ECOOP*, volume 5142 of *LNCS*, pages 412–437. Springer, 2008.

[26] ECMA International. *Standard ECMA-367. Eiffel: Analysis, Design and Programming Language*. 2nd edition, June 2006.

[27] Margaret A. Ellis and Bjarne Stroustrup. *The annotated C++ reference manual*. Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA, 1990.

[28] EVE. The Eiffel Verification Environment. `http://se.inf.ethz.ch/research/eve/`.

[29] Gobosoft. The Gobo Eiffel Structure Library. `http://www.gobosoft.com/eiffel/gobo/structure/index.html`.

[30] Bhargav S. Gulavani, Supratik Chakraborty, Ganesan Ramalingam, and Aditya V. Nori. Bottom-up shape analysis. In *SAS '09*, volume 5673 of *LNCS*, pages 188–204. Springer, 2009.

[31] Eric C. R. Hehner. *A practical theory of programming*. Springer-Verlag New York, Inc., New York, NY, USA, 1993.

[32] C. A. R. Hoare. An axiomatic basis for computer programming. *Commun. ACM*, 12:576–580, October 1969.

[33] Bart Jacobs and Frank Piessens. Inspector methods for state abstraction. *Journal of Object Technology*, 6(5):55–75, June 2007.

[34] Bart Jacobs and Frank Piessens. The VeriFast program verifier. Technical Report CW-520, Katholieke Universiteit Leuven, August 2008.

[35] Bart Jacobs, Jan Smans, and Frank Piessens. Verifying the composite pattern using separation logic. *SAVCBS Composite pattern challenge track*, 2008.

[36] I. T. Kassios. The dynamic frames theory. *Formal Aspects of Computing*, 2010. To appear.

[37] Ioannis T. Kassios. Dynamic frames: Support for framing, dependencies and sharing without restrictions. In Jayadev Misra, Tobias Nipkow, and Emil Sekerinski, editors, *FM*, volume 4085 of *LNCS*, pages 268–283. Springer, 2006.

[38] Neelakantan R. Krishnaswami, Lars Birkedal, Jonathan Aldrich, and John C. Reynolds. Idealized ML and Its Separation Logic. Draft at `http://www.cs.cmu.edu/~neelk/idealized-ml-draft.pdf`, 2006.

[39] G. T. Leavens, K. R. M. Leino, and P. Müller. Specification and verification challenges for sequential object-oriented programs. *Formal Aspects of Computing*, 19(2):159–189, 2007.

[40] Gary T. Leavens, Albert L. Baker, and Clyde Ruby. Preliminary design of JML: a behavioral interface specification language for Java. *SIGSOFT Softw. Eng. Notes*, 31(3):1–38, 2006.

[41] Gary T. Leavens and Peter Müller. Information hiding and visibility in interface specifications. In *ICSE*, pages 385–395. IEEE Computer Society, 2007.

[42] K. R. M. Leino and P. Müller. Verification of equivalent-results methods. In *ESOP '08*, volume 4960 of *LNCS*, pages 307–321. Springer, 2008.

[43] K. Rustan M. Leino and Rajit Manohar. Joining specification statements. *Theor. Comput. Sci.*, 216(1-2):375–394, 1999.

[44] K. Rustan M. Leino and Peter Müller. Object invariants in dynamic contexts. In Martin Odersky, editor, *ECOOP*, volume 3086 of *LNCS*, pages 491–516. Springer, 2004.

[45] K Rustan M Leino and Wolfram Schulte. A verifying compiler for a multi-threaded object-oriented language. *Software System Reliability and Security*, 9:351–416, 2007.

[46] Chenguang Luo and Shengchao Qin. Separation logic for multiple inheritance. *Electr. Notes Theor. Comput. Sci.*, 212:27–40, 2008.

[47] Bertrand Meyer, Ilinca Ciupa, Andreas Leitner, and Lisa Ling Liu. Automatic testing of object-oriented software. In *SOFSEM '07*, volume 4362 of *LNCS*, pages 114–129. Springer, 2007.

[48] Anna Mikhajlova and Emil Sekerinski. Class refinement and interface refinement in object-oriented programs. In John Fitzgerald, Cliff B. Jones, and Peter Lucas, editors, *FME '97: Industrial Applications and Strengthened Foundations of Formal Methods*, volume 1313 of *Lecture Notes in Computer Science*, pages 82–101. Springer Berlin Heidelberg, 1997.

[49] Carroll Morgan. The specification statement. *ACM Trans. Program. Lang. Syst.*, 10:403–419, July 1988.

[50] Carroll Morgan. *Programming from specifications (2nd ed.)*. Prentice Hall International (UK) Ltd., Hertfordshire, UK, 1994.

[51] Joseph M. Morris. A theoretical basis for stepwise refinement and the programming calculus. *Sci. Comput. Program.*, 9:287–306, December 1987.

[52] P. Müller, A. Poetzsch-Heffter, and G. T. Leavens. Modular invariants for layered object structures. *Science of Computer Programming*, 62(3):253–286, 2006.

[53] David A. Naumann. Observational purity and encapsulation. *Theor. Comput. Sci.*, 376(3):205–224, 2007.

[54] Peter W. O'Hearn, John C. Reynolds, and Hongseok Yang. Local reasoning about programs that alter data structures. In *CSL '01*, volume 2142 of *LNCS*, pages 1–19. Springer, 2001.

[55] Richard F. Paige and Jonathan S. Ostroff. ERC – an object-oriented refinement calculus for Eiffel. *Form. Asp. Comput.*, 16(1):51–79, April 2004.

[56] Matthew Parkinson and Gavin Bierman. Separation logic and abstraction. In *POPL '05: Proceedings of the 32nd annual ACM SIGPLAN-SIGACT symposium on Principles of programming languages*, pages 247–258, New York, NY, USA, 2005. ACM.

[57] Matthew J. Parkinson. Local reasoning for Java. PhD thesis. Technical Report UCAM-CL-TR-654, University of Cambridge, Computer Laboratory, November 2005.

[58] Matthew J. Parkinson and Gavin M. Bierman. Separation logic, abstraction and inheritance. In *POPL '08: Proceedings of the 35th annual ACM SIGPLAN-SIGACT symposium on Principles of programming languages*, pages 75–86, New York, NY, USA, 2008. ACM.

[59] Mohammad Raza and Philippa Gardner. Footprints in local reasoning. *Logical Methods in Computer Science*, 5(2:4):1–27, 2009.

[60] John C. Reynolds. Separation logic: A logic for shared mutable data structures. In *LICS '02: Proceedings of the 17th Annual IEEE Symposium on Logic in Computer Science*, pages 55–74, Washington, DC, USA, 2002. IEEE Computer Society.

[61] A. Rudich, Á. Darvas, and P. Müller. Checking well-formedness of pure-method specifications. In *FM '08*, volume 5014 of *LNCS*, pages 68–83. Springer, 2008.

[62] Alexandru Salcianu and Martin C. Rinard. Purity and side effect analysis for Java programs. In *VMCAI '05*, volume 3385 of *LNCS*, pages 199–215, 2005.

[63] Bernd Schoeller, Tobias Widmer, and Bertrand Meyer. Making specifications complete through models. In *Architecting Systems with Trustworthy Components*, volume 3938 of *LNCS*, pages 48–70. Springer, 2006.

[64] A. J. Summers, S. Drossopoulou, and P. Müller. The need for flexible object invariants. In *IWACO '09: International Workshop on Aliasing,*

Confinement and Ownership in Object-Oriented Programming, pages 1–9, New York, NY, USA, 2009. ACM.

[65] Stephan van Staden and Cristiano Calcagno. Reasoning about multiple related abstractions with MultiStar. In *Proceedings of the ACM international conference on Object oriented programming systems languages and applications*, OOPSLA '10, pages 504–519, New York, NY, USA, 2010. ACM.

[66] Stephan van Staden, Cristiano Calcagno, and Bertrand Meyer. Verifying executable object-oriented specifications with separation logic. In Theo D'Hondt, editor, *ECOOP*, volume 6183 of *Lecture Notes in Computer Science*, pages 151–174. Springer, 2010.

[67] Stephan van Staden, Cristiano Calcagno, and Bertrand Meyer. Freefinement. In *Proceedings of the 39th annual ACM SIGPLAN-SIGACT symposium on Principles of programming languages*, POPL '12, pages 7–18, New York, NY, USA, 2012. ACM.

Publications arising from the work of this thesis are [65], [66], and [67].