Declarative Theorem Proving for Operational Semantics

Don Syme

April 1, 1999

Contents

List of Figures

List of Tables

Abstract

This dissertation is concerned with techniques for formally checking properties of systems that are described by operational semantics. We describe innovations and tools for tackling this problem, and a large case study in the application of these tools. The innovations centre on the notion of "declarative theorem proving", and in particular techniques for declarative proof description. We define what we mean by this, assess its costs and benefits, and describe the impact of this approach with respect to four fundamental areas of theorem prover design: specification, proof description, automated reasoning and interaction. We have implemented our techniques as the Declare system, which we use to demonstrate how the ideas translate into practice.

The case study is a formally checked proof of the type soundness of a subset of the Java language, and is an interesting result in its own right. We argue why declarative techniques substantially improved the quality of the results achieved, particularly with respect to maintainability and readability.

Declaration

This dissertation is the result of my own work and includes nothing which is the outcome of work done in collaboration.

Acknowledgements

In the last four years I have met so many wonderful individuals that it is hard to know who to thank first, but through it all have been my office mates Mark and Michael, who have provided me with both warm friendship and an invigorating if sometimes exacting level of intellectual camaraderie. Similarly, my supervisor, Mike Gordon has consistently given of his time, and I thank him for for the example he has set, particularly in striking a balance between pragmatic and theoretical computer science. I particularly also thank John Harrison, who has contributed so greatly to the intellectual environment in which I have had the privilege to work in, as have Tom Melham, Larry Paulson, Andy Pitts and Andy Gordon. Many others have developed the academic fields which form the background to this thesis, and in particular I thank those who have worked on the HOL, Isabelle, PVS, ACL2, Mizar, O'Caml and Mercury systems, and especially Sophia Drossopoulou and Susan Eisenbach at Imperial for their work on Java.

During my studies I have had the unbelievable good fortune to receive two internships in the United States, at SRI International in 1996, and at Intel in 1998. On both occasions I was privileged to work with people of great intelligence, skill and energy. In particular, I thank John Rushby, Natarajan Shankar, Carl Seger and John O'Leary for the many discussions we had about declarative proof and related topics.

Michael Norrish, Katherine Eastaughffe and Mike Gordon assisted with the final preparation of this document, for which I am very grateful. The Commonwealth Scholarship Commission provided the funds for my studies, without which I would, no doubt, have been homeless and destitute for these four years — many thanks!

My time at Cambridge and in America has, in many ways, been the richest and most fulfilled of my life, and for that I am wholly indebted to my friends. They have brought me great happiness, both intellectual and emotional, and I think particularly of Kona, Florian, Beth, Jill, Darren, Daryl, Sue, Carlos, Byron, Jess, Maria, John Matthews, John Wentworth, Peter, Phil, and my housemates Juliet, Jonathan, Kate, Sam, Søren, Kieran and Saskia, as well as those mentioned above. These people are dear to me: they are my family, my friends. Finally, I thank Julie: may your life always be filled with as much joy as you brought into mine.

Chapter 1

Introduction

1.1 Overview

This dissertation is concerned with techniques for formally checking properties of systems that are described by *operational semantics*. Roughly speaking, this means systems specified by a naive, high level interpreter (or in such a way that the production of such an interpreter is a simple task). Such formalizations are extremely common in computer science, and are used to provide specifications of:

- The dynamic execution of programs;
- Static checks on programs such as type checking and inference;
- Statics and dynamics for highly non-deterministic systems such as process calculi;
- Security protocols [Pau97].

Real machines such as hardware devices can also be described operationally, presuming an appropriate level of abstraction is chosen.

"Formal checking" means *proving properties to a sufficient degree that our formalization may be checked by a relatively simple computer program.* We describe innovations and tools for tackling this problem in the context of operational semantics, and a large case study in the application of these tools.

A computer program used to develop and check such formalizations is called a *theorem prover*. Our primary contribution is the application and further development of a particular style of specification and proof called *declarative theorem proving*. We have produced an implementation of these techniques in the form of a theorem prover called Declare, and this system will be the focus of discussion for much of this dissertation.

The remainder of this chapter considers the application of a particular kind of operational description known as Structured Operational Semantics (SOS). We argue why formal checking is interesting for this problem domain, and describe previous

examples of formal checking for SOS. We then give a tutorial-style introduction to Declare, using a small example that is similar in flavour to our later case study, and define what we mean by "declarative" theorem proving.

In Chapters 2 to 5 we discuss the impact of a declarative approach on four aspects of mechanized theorem proving systems, and describe the techniques we have adopted in Declare:

- *Specification and Validation* i.e. methods for describing operational systems in a fashion acceptable to both mathematician and machine, and for informally validating that these specifications meet our informal requirements. We describe a range of specification constructs, their realisation in Declare, a new labelling system for extracting results that follow easily from specifications, and a new validation method based on translation to the Mercury [SHC96] system.

- *Proof Description* i.e. methods for describing the proofs of problems that may not be solved immediately by automated reasoning. We describe what constitutes a declarative proof language, the pros and cons of a declarative approach to proof and the particular proof language implemented in Declare. We then contrast declarative proof with existing proof description techniques.

- *Automated Reasoning* i.e. algorithms for automatically determining the validity of formulae that arise in our problem domain. We define our requirements with regard to automated reasoning and describe the particular techniques used in Declare (many are derivative, but some are new). We then assess how our automated prover does and does not meet our requirements.

- *The Interactive Development Environment* i.e. the system used to construct declarative specifications and proofs interactively. We consider how we can determine if an interactive development environment is a success, describe the principles behind our environment IDeclare, and assess it via an informal task-analysis.

Where our techniques depart from "best known practice" we describe how they represent an improvement. By "best known practice" we mean the state-of-the-art in the domain as embodied in existing interactive theorem provers, such as Isabelle, ACL2, HOL and PVS[Pau90, GM93, COR+95, KM96a, Har96a]. We use Declare as a means to demonstrate our ideas, though the ideas themselves are independent of the actual theorem proving system used.

In Chapters 6 and 7 we turn our attention to our major case study, where we formally check the type soundness of a major subset of the Java language. This case study is one of the more complex formally checked proofs about operational semantics in existence, and is an interesting result in its own right. We argue that declarative techniques played a positive role throughout the case study, and substantially improved the quality of the results achieved, particularly with respect to maintainability and readability.

Finally in Chapter 8 we reiterate the major themes we have addressed, summarize our results and discuss possible avenues for future research.

1.2 Structured Operational Semantics and its Uses

This work shall focus on systems described by "Structured Operational Semantics," a kind of operational description first developed systematically by Plotkin [Plo91] and which has subsequently become the standard technique for describing the formal semantics of programming languages, type systems and process calculi. Classic examples of its use include the formal definition of Standard ML [MTHM97] and the definition and theory of CCS [Mil80].

The primary features of an SOS description are:

- Terms that represent the abstract syntax of the program being executed by an abstract machine;
- Terms that represent a configuration of an abstract machine, usually combining a fragment of the abstract syntax that represents the remainder of the program to be executed, with extra terms to represent state and input/output;
- Inductively defined relations that describe the execution of the machine. These are either *big step* (if we relate configurations with values that represent the complete effects of their execution); or *small step* (if we relate configurations to new configurations);
- Inductively defined relations that describe the type system for the language.

In practice, SOS is more than a style of mathematics: it is a methodology. SOS is sufficiently well developed that it may be used as a method of systematic analysis during the development of a programming language. A striking example of its utility in this role is recent work by Drossopoulou and Eisenbach (whose work we shall consider in a case study in Chapters 6 and 6.6.2). They have used operational semantics to analyse the semantics of "binary compatibility" in the Java language [DE98], and have consequently discovered a serious flaw in the type system of the language. Considering the importance of the language and the subtle nature of the problem they detected this is a remarkable result. All that was required here was a systematic means for analysing the language: operational semantics can provide this. Thus, the role of SOS and our subsequent contribution can be summarized as:

> *Structured operational semantics is a formal methodology for defining and analysing abstract machines. We seek tools to support this methodology.*

Of course, nearly any discrete system in computing may be described operationally. Such descriptions are not always mathematically satisfying (being insufficiently abstract or modular); and yet are sometimes *too* abstract for system implementors (because they may abstract away crucial details such as the allowable

interactions with the outside world). We are not trying to demonstrate that operational reasoning is the "correct" approach to proving properties of languages. After all, if more abstract (e.g. categorical) models of a language are available then they will be more appropriate for many purposes. On the other side of the coin, we accept that most operational descriptions are indeed quite distant from real implementations of languages. However the techniques we present should scale well as more complex systems are considered, and the fact that our case studies already deal with quite large systems and yet remain tractable indicates this.

1.3 Formal Checking for Operational Semantics

Why are we interested in formally checking results based on operational semantics? It is useful to answer this in the context of our major case study: a type soundness result for a subset of Java.

A type soundness result states that if a program typechecks then certain problems won't occur during the execution of the program on a certain abstract machine. Thus, proving type soundness is verifying a property of the abstract machine described by the semantics. By doing this we give a proof of the feasibility of a sound implementation of the language. In addition, we can see the abstract machine as a primitive implementation, and when we prove type soundness we get a handle on how we might prove the result for a more realistic implementation. However, verifying type soundness for such an implementation would take considerably more work.

This justifies why we are interested in such results, but why formally check them? Formal checking is primarily *a tool for maintaining certainty in the presence of complexity*. Our case study in Chapter 6 describes a large operational system that is still undergoing rapid development by language researchers [DE98]. It is difficult to maintain the integrity of paper proofs of properties as such systems develop: the number of cases to analyse is high and there is always the concern that some unexpected combination of language features will lead to a soundness problem. Thus we turn to formal machine checking. Our Java case study demonstrates its value: it has been developed in parallel with the written formalization by Drossopoulou and Eisenbach, and has provided the researchers with valued feedback.

1.3.1 The Method and its Challenges

In principle, the formal checking of results about a system described by operational semantics is a relatively simple task. We must:

- Compose a *formal description* of the system that is correct with respect to the informal semantics (or existing implementations).
- Translate this description to create a *model* of the system in the framework of

the formal checking tool;[1]

- Formulate a *specification* of the the properties we are interested in proving.
- Formulate the proofs of these properties such that the proofs are tractable for a machine to check.

Things are, of course, never as straightforward as this. The primary difficulty is complexity: formal checking may "maintain certainty in the presence of complexity," but the very use of formal checking is a difficult thing in its own right, and can turn easy problems into hard ones (consider, for example, the headaches caused by simple arithmetic in generations of theorem proving systems: many arithmetic proofs that humans consider trivial may take considerable effort in a theorem proving system). Nearly all the devices we present in this work can be seen as mechanisms for managing the complexity of the theorem proving process. Hopefully by doing so we free the user to focus on the challenges inherent in the properties they are checking.

Two particular source of difficulty in the process of formal checking are *getting the details right* and *maintaining the formalization*. Our case study represents the application of formal checking to a problem where no 100% correct formalization was previously known: a written formalization existed but it was found to be deficient in many ways. In addition formalizations must be modified, extended, revised and reused. This can contribute substantially to the overall complexity of formal checking, if not well supported. In applied verification, we can assume neither that the problem of interest is stable, nor that the formulation we begin with is correct.

The techniques we present in this dissertation have been greatly influenced by these factors. While we have not solved the problems completely, we have certainly made progress, and summarize why in Chapter 8.

1.3.2 Related Work

Many attempts have been made to reason about the operational semantics of programming languages in theorem proving systems:

- Melham and Camilleri pioneered representational techniques for inductive relations in the HOL system and studied the operational semantics of some small imperative languages [CM92]. This culminated in the proof of the Church-Rosser theorem for combinatorial logic. The proof has since been reworked and improved in Isabelle [Ras95, Nip96].
- Nipkow, Naraschewski and Nazareth have proved the correctness of the W algorithm for type inference for a small functional language, using Isabelle-HOL [NN96].

[1] We do not use "model" in its proof theoretic sense, but rather to distinguish the "abstract" formal system (as expressed in the written mathematical vernacular) from the "embedded" formal system (realised in a formal checking tool).

- Syme and Hutchins have embedded the dynamic semantics of the core language of Standard ML in the HOL system [Hut90, Sym93]. They proved some simple meta-level results, including the determinacy of the semantics, and developed a symbolic evaluator for proving results about particular programs. Gunter, Maharaj and Van Inwegen [ME93, GM95], constructed a model for the dynamic semantics of the entire Standard ML language. Van Inwegen has tackled the considerably more difficult task of proving type soundness for the core language [Inw96], though the proof itself was beset with difficulties.

- Norrish [Nor98] has developed a model of the C language in HOL based on the (informal) ANSI standard. The main difficulty here was to even find a model for the language, and to derive results that avoid the complexities of the language when only simple constructs are used, e.g. Norrish has proved that in some situations the side-effecting nature of expressions may be safely ignored. We use this work to statistically contrast declarative and procedural styles of proof in Chapter 3.

- Nipkow and von Oheimb [Nv98] have developed a proof of the type soundness of a subset of Java that closely resembles our own case study (see Chapter 6).

There are many other similar works on a smaller scale, for example those by Frost, Nesi and Melham [Fro93, Nes92, Mel91].

As indicated by the above list, researchers have applied a range of theorem proving tools to assist with the formal checking of proofs related to operational semantics. Furthermore, as shall be clear in the following chapters, it is possible to draw on work from across the spectrum of theorem proving tools in order to provide this support. Some of the systems that have most influenced our work are:

- *HOL* [GM93]. This is an implementation of polymorphic higher order logic implemented in an "LCF-style" [GMW77]. That is the logic is mechanized starting with a simple set of rules and axioms, and HOL relies heavily on user-programmed rules of inference written in a dialect of ML. HOL supports a wide range of specification constructs and automated reasoning routines. Proofs are described using *tactics*, a topic we shall return to in Section 3.6.1.

- *Isabelle* [Pau90]. This is also an LCF-style system, but is generic and may be instantiated to a number of different "object logics," including polymorphic higher order logic and set theory. Specification is succinct and a wide range of notational conventions are supported. Proofs are again described using tactics, and a number of powerful generic proof routines including first order provers and simplification engines are available.

- *PVS* [COR+95]. This is an implementation of a rich higher order logic, including "predicate subtypes", notable for its excellent interactive environment, powerful integrated decision procedures and pragmatic approach to integrating model checking. It has not been widely applied to operational semantics.

- *ACL2* [KM96b]. ACL2 implements an "integrated collection of rules for defining (or axiomatizing) recursive functions, stating properties of those functions, and rigorously establishing those properties." It is notable for its use of decision procedures, its pioneering use of rewriting, its underlying computational model and induction heuristics. We make heavy use of techniques from ACL2 and its predecessors in Chapter 4.

- *Mizar* [Rud92]. This is a system for formalizing general mathematics, designed and used by mathematicians, and a phenomenal amount of the mathematical corpus has been covered. The foundation is set theory, which pervades the system, and proofs are expressed as detailed proof outlines, leaving the machine to fill in the gaps. We discuss this system in more detail in Chapter 3. It has not been applied to operational semantics.

Many of the techniques we utilise in this thesis are derived from ideas found in the above systems, though the ones we describe in detail are novel or significant extensions to existing techniques. Our most notable point of departure is with regard to proof description. Our contention is that none of the above systems, with the possible exception of Mizar, have addressed the question of "how proof outlines should be expressed" in sufficient depth. We claim that, in many ways, "declarative" techniques form a better method of proof description when proving properties of operational systems. We define what we mean by this in the following section and chapters, and will frequently compare and contrast our work with the related features available in the above systems.

1.4 Declarative Theorem Proving and Declare

The following chapters are concerned with techniques that improve the state of the art of theorem proving as applied to operational semantics. We have implemented these as the system Declare [Sym97a]. We use this system to demonstrate the principles underlying our techniques and how they may be implemented. We have also used this system for the case study described in Chapters 6 to 7.

Declare is not a fully polished system, and its aim is not to supplant existing interactive theorem provers or to needlessly duplicate hard work. Rather we seek to explore mechanisms of specification, proof and interaction that may eventually be incorporated into those systems, and thus complement them. We encourage developers and users of other theorem provers to consider the ideas contained in Declare with a view to incorporating them in other systems.

Later in this chapter we introduce the techniques we propose via a short Declare tutorial. However, we first discuss the general principles of declarative theorem proving and analyse some of the potential benefits of a declarative approach.

1.4.1 What is "Declarative" Theorem Proving?

In the general setting, a construct is considered *declarative* if it states *what* effect is to be achieved, and not *how*. "Declarative" is inevitably a relative notion: one construct is more declarative than another if it gives fewer operational details.

Declarative ideas are common in computing: Prolog and LaTeX are examples of languages that aspire to high declarative content. In Prolog, programs are independent of many of the operational details found in procedural languages and LaTeX documents are relatively independent of physical layout information. The term *procedural* is often used to describe systems that are non-declarative.

What, then, is declarative theorem proving? In an ideally declarative system we would, of course, simply state a property without describing how it is to be proved. For complex properties this is, unfortunately, impossible, so we set our sights a good deal lower:

> One theorem proving style is more declarative than another if it reduces the amount of "procedural information" and the number of "procedural dependencies" required to specify properties and their proofs. Such dependencies include: reliance on the orderings of cases, variables, facts, goals and subgoals; reliance on irrelevant internal representations rather than external properties; reliance upon one of a number of logically equivalent forms (e.g. $n > 1$ versus $n \geq 2$); and reliance on the under-specified behaviour of proof procedures (e.g. how names are chosen).

To take a simple concrete example, proofs in interactive theorem provers (e.g. HOL, PVS and Isabelle) are typically sensitive to the order in which subgoals are produced by an induction utility. That is, if the $\mathbb{N}$-induction utility suddenly produced the step case before the base case, then most proofs would break. There are many similar examples from existing theorem proving system, enough that proofs in these systems can be extremely fragile, or reliant on a lot of hidden, assumed detail. The aim of declarative proof is to eliminate such dependencies where possible. In the next two chapters (particularly Chapter 3) we discuss the exact techniques we have implemented in Declare, and assess them relative to this definition of "declarative."

1.4.2 Costs and Benefits

There are costs and benefits to taking a declarative approach. The possible benefits in the general setting are:

- *Brevity.* The elimination of procedural content may reduce the overall size of a development. For example, Prolog programs are usually shorter than equivalent C programs.

- *Relative Simplicity.* Eliminating procedural content reduces the complexity of an artifact. For example, most Prolog programs are certainly simpler than equivalent C programs (as well as being shorter).

- *Readability.* Procedural content often obscures the structure and intent of a development, and eliminating it will thus clearly improve readability.

- *Re-usability.* Declarative content can often be reused in a similar setting, e.g. LaTeX source can typically be re-typeset for any number of output arrangements, and Prolog code can easily be transferred from system to system. Procedural code is more difficult to reuse precisely because it is often dependent on aspects of the environment that are subject to change.

- *Robustness.* Declarative content is often robust under changes to the way in which the information is interpreted. For example, pure Prolog programs may be independent of evaluation order, at least in the sense that if a predicate has a finite number of solutions, then the set of solutions will remain identical even under the reordering of conjuncts.

Note, however, that these potential benefits are not always realised. That is, the elimination of "how" dependencies can come at some cost. One problem is when the "declarative" specification is implicit in the "procedural". For example, one declarative technique used in Declare proofs is to state some propositions, and list the facts that provide support for their deduction (the automated prover is left to figure out the details). Procedurally, one might instead describe the syntactic manipulations (modus-ponens, specialization etc.) that deduce the given facts. Eliminating the procedural specification may be advantageous, however in order to provide a declarative specification of the operation we actually have to write out the deduced facts. These were left implicit in the procedural specification, and thus the procedural approach might be more succinct.

Furthermore, the declarative approach leaves the computer to work out the syntactic manipulations required to justify the step deductively. This demonstrates the two potential drawbacks of a declarative approach:

- *Requires a Specification.* Specifying a declarative view of an operation takes text, and thus does not come for free if this is was previously left implicit.[2]

- *Complexity of Interpretation.* Eliminating detail may increase the complexity of the interpretation process.[3]

We discuss the pros and cons of declarative theorem proving further in Chapters 3 and 8. To summarize, declarative theorem proving is about the elimination of detail and dependencies that might otherwise be present. This does not come for free, but in the balance we aim to achieve benefits that can only arise from a declarative approach.

[2] Another example is the specification of a a signature to a module in a programming language (a declarative view of a module). Writing and maintaining the signature takes effort, and in small programs it may be better to leave the interface implicit.

[3] Again another example: Prolog compilers must be quite sophisticated in order to achieve reasonable efficiency.

Finally, some declarative systems like LaTeX allow access to a "procedural level" when necessary. One could certainly allow this in a declarative theorem proving system, e.g. via an LCF-like programmable interface. For the moment, however, we shall not give in to such temptations!

1.4.3 A Tutorial Introduction to Declare

We now introduce Declare in a tutorial style, with the aim of demonstrating some of the declarative techniques we propose. The tutorial is designed simply to place the discussion of the following chapters in a concrete setting, and we shall frequently refer back to the examples presented here. We shall use Declare to construct the runtime operational semantics for a toy programming language (a lazy, explicitly typed, monomorphic lambda calculus with de Bruijn indexes). We prove that execution in this language is type safe by proving a subject reduction theorem. This will demonstrate:

- The *terms* and *types* of Declare's underlying logic.
- The *specification constructs* for datatypes, simple definitions, recursive definitions and least fixed points.
- Validating the specification by compiling to executable code.
- The *proof outlining constructs* for proof by decomposition, proof by automation and proof by induction.
- The *justification language constructs* for giving theorems, case analyses and explicit instantiations as hints: this is the interface to the automated reasoning engine.

The toy programming language has the following abstract syntax:

ty	$=$	$ty \to ty$	(function type)
	$\mid$	$\mathtt{i}$	(integer type)
exp	$=$	int	(constant)
	$\mid$	nat	(de Bruijn indexed bound variable)
	$\mid$	$\lambda_{ty}.\ exp$	(abstraction)
	$\mid$	$exp\ exp$	(application)

Bound variable indices refer to lambda bindings, counting outward, thus $\lambda_{\mathtt{i}\to\mathtt{i}}.\ \lambda_{\mathtt{i}}.\ 1\ 0$ could be written $\lambda f_{\mathtt{i}}\ \lambda x_{\mathtt{i}}.\ f\ x$. One-step lazy evaluation is given by the following rules:

$$\frac{f \rightsquigarrow f'}{f\ a \rightsquigarrow f'\ a} \qquad \frac{}{(\lambda_\tau\ bod)\ a \rightsquigarrow \mathtt{subst}\ a\ bod}$$

where $\mathtt{subst}\ a\ b$ implements the replacement by a of those variables in b that have index equal to the count of their outer lambda bindings. Typing is given by:

$$\frac{}{\Gamma \vdash i : \mathtt{i}} \qquad \frac{\Gamma(n) = \tau}{\Gamma \vdash n : \tau} \qquad \frac{\Gamma \vdash f : \tau_1 \to \tau_2 \quad \Gamma \vdash x : \tau_1}{\Gamma \vdash f\ x : \tau_2} \qquad \frac{\Gamma \vdash bod : \tau_2}{\Gamma \vdash \lambda_{\tau_1}.\ bod : \tau_1 \to \tau_2}$$

The specification

The first declarations in our Declare file (called an "article") are shown below. Note that we are constructing a document: batch-mode interaction with Declare is by constructing documents and submitting them for checking. The checker is very quick, so the working environment is essentially interactive. A truly interactive environment is also available: we discuss this in Chapter 5.

We begin with a notation declaration[4], and then the abstract syntax is specified as two *recursive types*, using ML-like notation. The auxiliary recursive function `subst_aux` is defined using pattern matching.

```
infixl 11 `%`;

datatype typ = TyInt | TyFun typ typ;

datatype exp =
   Int int
 | Var nat
 | Lam typ exp
 | (%) exp exp;

let rec "subst_aux n t e =
          match t with
            Var m -> if (m = n) then e else t
          | Lam ty bod -> Lam ty (subst_aux (n+1) bod e)
          | f % a -> subst_aux n f e % subst_aux n a e
          | _ -> t";

let "subst = subst_aux 0";
```

The term defining `subst_aux` is quoted because it is a term of the underlying logic, i.e. the variant of higher order logic we describe in Section 2.1. Unquoted portions of the input are part of the *meta-language* used to manipulate terms of the logic. Declare has temporarily abandoned the traditional use of highly programmable meta-languages (such as the ML dialects in LCF-style systems [GMW77]) in order to investigate *declarative* rather than *procedural* proof specification techniques. The aim has been to find a small set of "highly declarative" commands to use for specification and proof, and we have found it useful to abandon the constraints of a strictly typed meta-language for this purpose.

The evaluation and typing relations are defined as the least fixed points (`lfp`) of sets of rules (we have used lists to model type environments, though typically we use partial functions).[5]

[4] Notation declarations are typically kept in a `.ntn` file and imported with a `notation <file>` directive.

[5] Comments are nested `(* ... *)` or to end-of-line `//...`.

```
infixr 10 `--->`;
lfp (--->) =
<app1>                "e1 ---> e1'"
        // ------------------------------------------------
                      "e1 % e2 ---> e1' % e2"

<beta>
        // ------------------------------------------------
                      "(Lam ty bod) % e2 ---> subst bod e2";


threefix `|-` hastype;
lfp hastype =
<Int> [autorw]
        // ------------------------------------------------
                    "TE |- (Int i) hastype TyInt"

<Var> [autorw]    "i < len TE ∧ ty = el(i)(TE)"
        // ------------------------------------------------
                    "TE |- (Var i) hastype ty"

<Lam> [autorw]    "(dty#TE) |- bod hastype rty"
        // ------------------------------------------------
                    "TE |- (Lam dty bod) hastype (TyFun dty rty)"

<App>               "TE |- f hastype (TyFun dty rty) ∧
                     TE |- a hastype dty"
        // ------------------------------------------------
                     "TE |- (f % a) hastype rty";
```

The `[autorw]` tag is a *pragma*: this is how we declare extra-logical information to the automated prover. Pragmas may either be declared when a theorem is declared, or may be asserted at a later stage, e.g.

```
pragma autorw <hastype.Int>;
pragma autorw <hastype.Var>;
pragma autorw <hastype.Lam>;
```

Validation by execution

Having completed a model of the toy language, it is natural to validate this model by executing it on some test examples. Declare can translate many specifications to a target language called Mercury [SHC96] by a relatively simple set of translations. We discuss validation and the translation to Mercury further in Section 2.4, and shall just give a taste of what is possible here.

Mercury is a pure Prolog-like programming language with higher order predicates and functions. It includes algorithms to statically analyse programs for *type*, *mode* and other constraints, and can generate extremely efficient code as a result. Predicates such as `hastype` become Mercury relations, and other terms become Mercury data expressions. The user is required to specify mode constraints for predicates:

```
pragma mode "inp ---> outp";
pragma mode "inp |- inp hastype outp";
```

If, for example, the `<app1>` rule above had been

```
<app1>              "e1 ---> e1''"
        // ------------------------------------------------
                    "e1 % e2 ---> e1' % e2"
```

then Mercury's mode analysis would detect that the rule fails to specify a definite output for `e1'` on the bottom line.

Test programs are specified in Declare as predicates generating values for an unknown:

```
let "id = Lam TyInt (Var 0)";
pragma test "[] |- id hastype X";
pragma test "[] |- (id % id) hastype X";
```

The first test generates all types that may be assigned to *id*. Higher order operators may be used to trace the execution of a transition relation:[6]

```
pragma test "(id % Int 1) RTC(--->) X";
pragma test "(id % (id % Int 1)) Fringe(--->) X";
```

The Mercury program produced by the Declare code generator executes these test programs:[7]

```
> ./main
-------------------------------------------------------------
Executable model generated from DECLARE specification in db.art
-------------------------------------------------------------
test on line 82: "[] |- id hastype X"
  X = TyFun(TyInt,TyInt)

test on line 83 "[] |- (id % id) hastype X"
  no solutions
```

[6] Here `RTC` is a parameterized infix operator that takes the reflexive transitive closure of a relation, and `Fringe` finds all elements in this closure that have no further transitions. Both are defined in the standard Declare basis.

[7] The actual implementation does not print output terms quite so nicely, but given the meta-programming facilities of a Prolog system this would not be difficult to implement.

```
test on line 84:
  X = Lam(TyInt,Var(0)) % Int(1)
  X = Int(1)

test on line 85:
  X = Int(1)
```

Our first proofs

We now wish to prove subject reduction, i.e. if a reduction can be made to a well-typed closed term, then it produces a term of the same type. We can formalise this with the following theorem declaration:

```
thm <small_step_lazy_safe>
if "[] |- e hastype ty"
   "e ---> e'" <step>
then "[] |- e' hastype ty";
```

Investigations quickly lead us to conclude that we must first prove that typing is "monotonic over increasing type contexts" (we might discover this midway through the outline of the subject reduction proof, which we shall come to below). A larger type environment ($\leq$ or `<<=`) is one that possibly has additional entries:[8]

```
infixl 10 "<<=" --> leq;
let "TE1 <<= TE2 ↔ ∃l. TE1@l = TE2";
```

Two consequences follow easily from the definition of `<<=`, and the statement and proof outline for each of these is shown below. The propositions are introduced as theorem declarations and are followed by proof outlines (in this case very simple ones!):

```
thm <leq_nil> [autorw] "[] <<= TE";
proof qed by <leq>; ◂Proof outline
```

```
thm <cons_leq_cons> [autorw]
   "(x#TE1) <<= (x'#TE2) ↔ (x = x') ∧ TE1 <<= TE2";
proof qed by <leq>;
```

Fresh symbols (such as `TE` in the first example) are implicitly universally quantified. Proofs are given in a declarative proof language made up of *justifications by automation*, *case splits*, and *second-order schema applications*: in each case above we have

[8] In higher order logic, "if and only if" ($\leftrightarrow$) is simply equality (`=`) over booleans, but syntactically has a lower precedence. The operators `@` and `#` are "append" and "cons" over lists as usual. In Declare the `infixl` declaration defines an infix operator and gives an alpha-numeric identifier which is used as an alternative label in, for example, theorem names.

only used justification by automation (using `qed`), adding the hint that the definition for `<<=` be used in the automated proofs. Each `qed` step generates one proof obligation. In this case the automated engine can check these proof obligations by using a combination of rewriting (utilising background rules), instantiation and arithmetic decision procedures.

An inductive proof

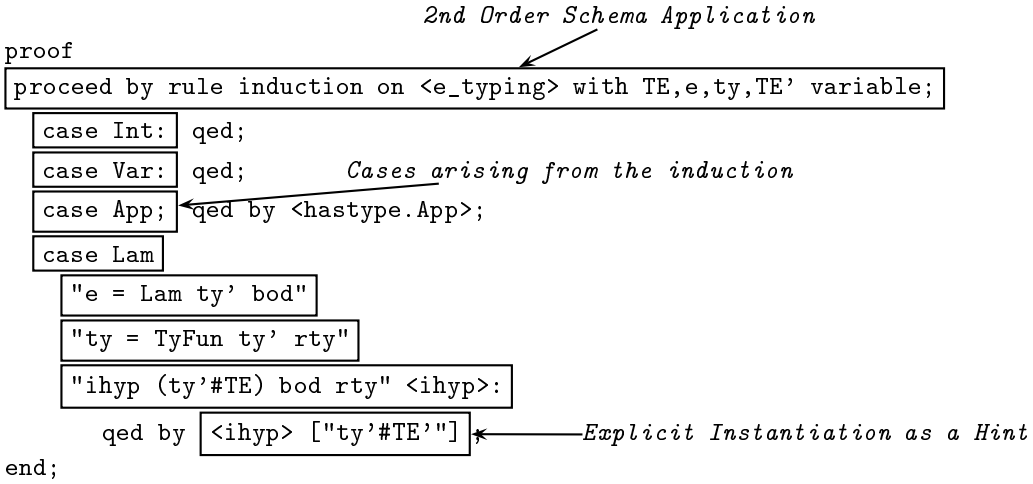
We can now state the monotonicity result. Informally, we might state and prove it as follows:

Theorem 1 Monotonicity *If* $\Gamma \vdash e : \tau$ *and* $\Gamma \leq \Gamma'$ *then* $\Gamma' \vdash e : \tau$

The proof is by induction on the derivation of the typing judgement. The interesting case is when $e = \lambda_{\tau'}.\ b$ and $\tau = \tau' \rightarrow \rho$ where the induction hypothesis gives us $\Gamma \leq \Gamma'' \rightarrow \Gamma'' \vdash b : \rho$ for all Γ''. When $\Gamma'' = \tau', \Gamma'$ the result follows by the typing rule for lambda applications. In Declare the problem is stated as:

```
thm <hastype_mono>
if "TE |- e hastype ty" <e_typing>
   "TE <<= TE'"
then "TE' |- e hastype ty";
```

Within the sequent, the label `<e_typing>` gives a name to a local fact. The corresponding Declare proof outline is:



The proof itself first utilises the induction proof language construct, described in detail in Section 3.5. The induction predicate is:

$\lambda TE\ v\ ty.\ \forall TE'.\ TE$ `<<=` $TE' \longrightarrow TE'$ `|-` v `hastype` ty

This predicate becomes the macro `ihyp` on the branches of the proof.

The induction construct itself generates no proof obligation, but rather four cases, corresponding to the four rules for the least-fixed point. The cases may be given in any order. In three of the cases the induction hypotheses are left implicit and the proof is simple. In the `Lam` case a small hint is required: the explicit instantiation of an induction hypothesis (`by <ihyp> ["dty#TE'"]`). To enforce good declarative proof style, Declare demands that we can only use facts if they are present in the text of the proof document, and so we record the induction hypothesis explicitly. These are:

- The equational constraints for the `Lam` case; and
- The induction hypotheses from the top line of the rule on page 12.

We then explicitly instantiate the fact `<ihyp>` on the justification line, which completes the proof up to the four proof obligations that must be checked by Declare. What theorem results from the successful proof on the main branch of the article? The local constants `TE`, `v`,`ty` and `TE'` are universally quantified, and the sequent becomes an implicative formula:

```
thm <hastype_mono> "∀TE TE' v ty. TE |- v hastype ty ∧ TE <<= TE'
                                 → TE' |- v hastype ty"
```

The subject reduction proof

The next fact we prove is that substitution preserves types:[9]

```
thm <subst_aux_safe>
if "[] |- v hastype vty"
   "len TE = n"
   "(TE #! vty) |- e hastype ty" <typing>
then "TE |- (subst_aux n e v) hastype ty";
```

We omit the proof: it is not interesting for our purposes as it uses only the constructs described above. Finally we prove the subject reduction theorem itself:

```
thm <small_step_lazy_safe>
if "[] |- e hastype ty" <typing>
   "e ---> e'" <step>
then "[] |- e' hastype ty";
proof
proceed by rule induction on <step> with ty variable;
  case beta
    "e = Lam xty bod % e2"
    "e' = subst bod e2";
```

[9] `[]` is the empty list, `len` is the length of a list and `#!` adds an element to the end of a list

```
    consider dty st
         "[] |- Lam xty bod hastype (TyFun dty ty)" <ty2>
         "[] |- e2 hastype dty"
        by rulecases(<typing>);
     qed by rulecases(<ty2>), <subst_aux_safe> ["[]"],
          <nil_snoc_cons>;
  case app1; qed by <hastype.App>,rulecases(<typing>);
end;
```

The long case of the proof corresponds to a beta-reduction step. The `consider` construct is an instance of the third (and final) proof outlining construct of the proof language: *case decomposition combined with constant and fact introduction*. The general form is described in Section 3.3. In the example we assert the existence of an object `dty` with the properties given by the two facts, justified by automatic proof and several theorems.

1.4.4 Checking the Article

Once written, the article may be checked as follows. For illustrative purposes, we show the output if `rulecases(<typing>)` is replaced simply by `<typing>` on the last line of the proof.

```
> decl lang.art
DECLARE v. 0.2a
parsing...done
importing and merging abstracts...done
type checking...done
...
checking proof of <small_step_safe>

File "db.art", line 192, characters 13-34:
This step could not be justified.

Simplification produced
  + <App>    ∀dty TE f a rty.
               (TE |- f hastype (TyFun dty rty)) ∧
               (TE |- a hastype dty)
               → (TE |- f % a hastype rty)
  + <ihyp>   ∀ty. ([] |- e1 hastype ty) → ([] |- e1' hastype ty)
  + <ihyp>   e1 ---> e1'
  + <typing> [] |- e1 % e2 hastype ty
  - <oblig>  [] |- e1' % e2 hastype ty

  where e' = e1' % e2
        e = e1 % e2
```

The feedback shown is from the automated prover used to discharge proof obligations: here it is easy to spot that a necessary condition in `<ihyp>` has not been discharged, and hence deduce that a rule analysis on `<typing>` will be helpful. Note that Declare has checked the rest of the proof on the assumption that the facts stated on line 192 were indeed derivable. If a proof obligation cannot be discharged by the automatic prover, a warning is given and the fact is assumed.

An article is typically written and checked within IDeclare, the interactive development environment (IDE) for Declare described in Chapter 5.

Part I

Tools and Techniques

Chapter 2

Specification and Validation

In this chapter we consider *specification* and *validation* techniques for operational semantics, i.e. methods for describing systems in a fashion acceptable to both human and machine, and for checking that our specifications correspond to our informal requirements. Specifications must be interpreted with respect to some foundational logical system. We briefly describe some such systems, and give a rationalization for the choice of *higher order logic* (h.o.l.).[1] After introducing this logic we outline the constructs we use to specify operational semantics, and give details of their realization in Declare. Finally we address the issue of executing specifications for the purposes of validation.

Specification is quite a well-understood area, so most of this chapter is background material. Our main contributions are around the edges:

- The use of a systematic labelling mechanism to easily "get a handle on" results that follow trivially from definitions.
- The use of a higher order pure Prolog (Mercury) as a target language for generating executable code. This gives us the power to perform mode, determinism, uniqueness and termination analyses on our specifications, and to execute test cases to validate the specifications in particular cases.

2.1 Foundations and Higher Order Logic

A plethora of techniques has been developed for the formal specification of systems, and the topic is a significant and complex one in its own right. Typically each technique is accompanied by a logic for use with the specification language, although sometimes the specification language is precisely the logic and sometimes no coherent and complete logic is immediately apparent. Commonly cited specification languages include: axiomatization in first order logic; the Z and VDM notations [Spi88, Jac88]; variants of higher order logic (e.g. the specification languages of HOL

[1] As distinct from the HOL or Isabelle/HOL implementations of higher order logic.

[GM93], PVS [COR+95] and Isabelle/HOL [Pau90]); set theory; temporal logics; specialised formalisms for finite state machines and hardware; and restricted subsets of logic that are highly amenable to automation (e.g. Monadic 2nd Order Logic [JJM+95]) and process calculi. This is not the place to give a detailed analysis of the merits of these methods: Rushby has written a good introductory overview [Rus93].

We choose a simple *higher order logic* as our foundational system: everything we do can be given a semantics by translation into this logic. The following issues dictated our choice of logical foundation:

- We take it as axiomatic that a certain coherency and simplicity with regard to semantics, implementation and use are all "Good Things" to look for in a framework. Difficulties with providing a simple coherent semantics or good tool support rule out approaches based on Z, VDM or object-oriented concepts.

- We are interested in modelling systems that have infinite state spaces. Thus finite state techniques, where the model is compiled to some more convenient representation, e.g. a finite state machine, are not immediately applicable.

- Similarly, we need to perform second-order reasoning such as induction arguments. Thus approaches based around purely first-order techniques such as Prolog are not sufficient. A more syntactic, explicit representation of knowledge is required.

- We are thus led to the necessity of supporting a high degree of syntactic (or deductive) reasoning, which is normally done using some variant of *higher order logic*. An excellent summary of the benefits of this approach can be found in [Rus93].

We now go on to give a brief account of higher order logic. We assume familiarity with first order logic. *Second order logic* allows quantification over predicates. For example this allows the encoding of induction schemes:

$$\forall P.\ P(0)\ \wedge\ (\forall k.\ P(k) \longrightarrow P(k+1)) \longrightarrow \forall n.\ P(n)$$

Second order logic can frequently "act as its own meta-language." That is, higher order theorems can express many effects normally achieved by proof procedures, e.g. a single higher order theorem (interpreted as an algorithm in the obvious fashion) can express the standard transformation to negation normal form.

Higher order logic allows quantification over functions of any order, as well as predicates. Apart from second order quantification like the above, the most common uses of higher order features are:

- For higher order predicates such as $\forall$, $\exists$ or reflexive transitive closure.

- For higher order functions such as "map" (over a list), or the iterated application of a function.

- To model "data" objects using functions, e.g. sets, or tables using partial functions.

To avoid logical contradictions such as Russell's paradox, higher order logics are usually typed. Many typing schemes are possible: we adopt the simple polymorphic typing scheme used in HOL and Isabelle. Other typing schemes, notably that of PVS, address issues such as predicate and structural subtyping. Melham's system allows quantification over type variables [Mel92], and one can also admit record types. We have been able to survive without such features in our case study.

The primitive terms of higher order logic are as in the λ-calculus: variables, constants, applications and functions. Types are either type variables (α) or constructed types using some type functor applied to a number of arguments (e.g. *bool*, α *list* or $\alpha \to \alpha$). Constants may be polymorphic, and the primitive constants are normally just $=_{\alpha \to \alpha \to bool}$, $\to_{bool \to bool \to bool}$ and the Hilbert-choice operator $\epsilon_{(\alpha \to bool) \to \alpha}$. Theorems are terms of type *bool* deduced from the primitive axioms and the rules of the logic. These are typically α, β and η conversion, type and term specialization, modus-ponens, the congruence properties of equality, the axiom of choice and deduction rules in a sequent style.

From the point of view of mechanization, polymorphic simple type theory seems to occupy a neat, locally optimum position in the spectrum of possible logics. Type checking is decidable and efficient, terms can be represented fairly compactly, and a fair degree of expressiveness is achieved. It is not ideal for all purposes, but is excellent for many. See Harrison's HOL-lite [Har96a] for an elegant implementation of h.o.l. from first principles.

Logic of description v. logic of implementation

Simple polymorphic higher order logic acts as the logic we use to provide a coherent semantic framework for the system we implement. We could call this the "logic of discourse". Unlike the LCF family of theorem provers, it is not *precisely* the logic we implement in Declare, in two senses:

- The mapping between the representation used for terms in the computer and terms of the logic is not entirely trivial, e.g. see the representation of pattern matching in Section 2.2.1.

- The logical system is extended with strong rules of inference, e.g. decision procedures. We rely on the soundness of these and do not perform the proofs by syntactic deduction (i.e. Declare is not "fully expansive")

Both PVS and Isabelle follow similar approaches: while in principle the core of Isabelle implements intuitionistic higher order logic, it also contains one powerful primitive inference rule (simplification) — this is naturally omitted from the description of the logic implemented. Similarly the formal description of PVS describes simpler logical rules than those actually built into the prover.

2.2 Specification Constructs for Operational Semantics

In this section we shall introduce the range of constructs we use to specify operational systems. We shall briefly describe each, and present their realisation in the Declare specification language. Most of the constructs have been presented by example in Section 1.4.3. At the end of the section we discuss the issues of partiality, "declarative" specifications and modularity.

The devices presented in the following sections are shortcuts for declarations given in a primitive language of types, constants, theorems (axioms) and annotations (*pragmas*). We will show the equivalent declarations in each case. The shortcuts are used for brevity and to greatly simplify the proof obligations that arise.

The language of pragmas is used to declare extra-logical information, normally about theorems, for the benefit of tools such as the proof language analyser, the automated reasoning engine and the code generator. Pragmas relating to each tool are discussed in the following chapters, though their intuitive meaning should be clear.

2.2.1 Pattern Matching

Pattern matching is a construct in specification and programming languages where a term may be compared to other terms, the latter possibly containing fresh (binding) variables. Just as in programming languages this is a succinct way to specify structural and other equational constraints. We replace λ terms in h.o.l. by pattern matching functions. For example consider the (equivalent) pattern match applications

```
1. (function
      0 -> 1
    | 1 -> 1
    | n -> fib(n-1) + fib(n-2)) t

2. match t with
      0 -> 1
    | 1 -> 1
    | n -> fib(n-1) + fib(n-2)
```

The informal semantics is the same as for functional languages: the first rule `0 -> 1` must fail before the second may be used, and if the first succeeds the others are ignored. If no rules remain then the term represents some arbitrary member of its type. We could decode pattern matching into simple higher order logic by using the Hilbert choice operator. The term above would become:

```
λtmp.
  Choose res.
     (tmp = 0 ∧ res = 1)
```

```
∨ (tmp <> 0 ∧ tmp = 1 ∧ res = 1)
∨ (∃n. tmp <> 0 ∧ tmp <> 1 ∧ tmp = n ∧ res = fib(n-1) + fib(n-2))
```

In our implementation we do not actually decode pattern matching, though all the manipulations we perform on such terms (e.g. see Section 4.2.3) have equivalent manipulations on the translated forms.

The patterns may be arbitrary terms, may bind an arbitrary number of variables and variables may even be repeated in each pattern. Left-to-right interpretation allows such liberal patterns because there is no obligation to prove that only one path of the match may succeed.

2.2.2 Simple Definitions and Predicates

Simple non-recursive definitions account for the majority of definitions in a model of an operational semantics, e.g.

```
let "(union) p q = (fun x -> p x ∨ q x)";
let "subst = subst_aux 0";
let "(--*>) = RTC(--->)";
```

There is no proof obligation for such specification constructs, and in Declare they give rise to a constant, an equational theorem, an "elimination" theorem and appropriate pragmas, e.g.

```
constant union ":α set → α set → α set";
thm <union> "p union q = (λx. p x ∨ q x)";
thm <union.elim> "(union) = λp q x. p x ∨ q x";
pragma defn <union>;
pragma code <union.elim>;
pragma elim <union.elim>;
```

The interpretation of the above pragmas is discussed in later sections. Definitions can be conditional in order to document constraints on their arguments: the functions will be under-specified outside this domain (see Section 2.2.6). Arguments can be any terms, just as with pattern matching.

2.2.3 Datatypes

Recursive datatypes, or *free algebras* are familiar to anyone who has programmed in an ML dialect, and are a key construct for modelling operational systems. Typically we require the construction of recursive types using (non-dependent) sums ($\_ + \_$), (non-dependent) products ($\_ \times \_$) and covariant type constructors such as $\_$ *list*, $\alpha \rightarrow \_$ and $\alpha \stackrel{table}{\longmapsto} \_$. We use datatypes to model pairs, lists, trees, records, enumerations and abstract syntax. In our case study we use them for both the abstract syntax of Java and runtime objects that get created. Some examples, using ML-like syntax, are:

```
datatype (α,β) (×) = (,) of α β;  // pairs
datatype α list = (#) of α | ([]);
datatype α option = None | Some of α;
```

Operational descriptions typically require *mutually recursive* datatypes to describe abstract syntax succinctly. A common example of the use of mutual recursion is for expressions and declarations in a functional programming language:

```
datatype exp = Dec of dec × exp | Int of int | ...
and dec = Let of string × exp | Decs of dec list | ...
```

Often we need more specific algebras, e.g. well-typed programs. Typically we use predicate constraints to do this, defined inductively over the corresponding free algebra.

Reductionist proofs of the existence of solutions (within higher order logic) for recursive type equations that include nested constructors have been automated by Gunter [Gun94], and for simpler types by Melham [Mel88]. For our purposes it suffices to use a routine that generates the necessary axioms. Many other theorem provers admit a similar range of types (e.g. PVS, LCF and Isabelle). Simple higher order logic requires that types be non-empty, and an initiality condition must be proved for each datatype. Declare does not currently check non-emptiness and initiality conditions, though in principle they can be determined automatically by a graph search.

See Section 4.2.3 for a discussion of the automated reasoning routines that deal with datatypes.

2.2.4 Fixed Point Relations

If datatypes are used to model syntax in operational descriptions of systems, then *(co)inductive* relations and *recursive functions* are the essential tools to model semantics. (Co)inductive relations are the preferred method for defining recursive judgments declaratively because they abstract away from so many details: the logical structure of the possible derivations becomes immediately evident in the formulae, and induction is over the space of all possible derivations, instead of some indexing scheme (e.g. $\mathbb{N}$) into this space.

An inductive relation is the least fixed point of a monotonic set transformer F within the context of a universal set U. Such fixed points are guaranteed to exist by the Knaster-Tarski theorem [Tar55]. Good references to the theory and its mechanisation are [Pau94, CM92, Har95, PM93]. Typically the transformer F is defined by a set of rules, e.g. the definition of one-step lazy evaluation for the simple lambda calculus:

$$\frac{f \rightsquigarrow f'}{\texttt{App}\ f\ a \rightsquigarrow \texttt{App}\ f'\ a} \qquad \frac{}{\texttt{App}\ (\texttt{Lam}\ x\ bod)\ a \rightsquigarrow \texttt{subst}(x,a)bod}$$

$$
\begin{array}{lcl}
U & = & exp \times exp \\
F(\rightsquigarrow) & = & \lambda p.\ \ (\exists f\ f'\ a.\ p = (\texttt{App}\ f\ a,\ \texttt{App}\ f'\ a)\ \wedge\ f\ \rightsquigarrow\ f') \vee \\
 & & \quad\ (\exists x\ bod\ a.\ p = (\texttt{App}\ (\texttt{Lam}\ x\ bod)\ a,\ \texttt{subst}(x,a)bod))
\end{array}
$$

Note that the implicit equational constraint from the bottom line of each rule has been explicitly quoted here. Intuitively, $a \rightsquigarrow b$ holds if some derivation exists using only the rules above.

In Declare we do not mechanise the theory of fixed points from first principles — previous authors have addressed this issue [CM92, Har95, Pau94]. Instead, we use an axiomatization of each relation. Examples of the syntax of least fixed point declarations were shown in Section 1.4.3.

In the present implementation we do not generate the associated monotonicity proof obligations, since this is well-understood, as is the automatic checking of monotonicity conditions (e.g. see the Isabelle implementation [Pau94]). Explicit proofs of monotonicity could be given in the proof language described in the next chapter, if necessary.

As with datatypes, theorems are generated that encode the logical properties of inductive relations in Declare. The example on page 10 generates the following theorems – clearly the axiomatization in h.o.l. is straightforward:

```
thm <reduce>
  "arg1 ---> arg2 ↔
       (∃e1 e1' e2. arg1 = e1 % e2 ∧ arg2 = e1' % e2 ∧ e1 ---> e1')
     ∨ (∃e1 e2 ty bod. arg1 = e1 % e2 ∧ arg2 = subst bod e2 ∧ e1 = Lam ty bod)"
pragma defn <reduce>
pragma code <reduce>
thm <reduce.app1> "e1 ---> e1' → e1 % e2 ---> e1' % e2"
thm <reduce.beta> "e1 = Lam ty bod → e1 % e2 ---> subst bod e2"
thm <reduce.induct>
  "(∀arg1 arg2.
          (∃e1 e1' e2.arg1 = e1 % e2 ∧ arg2 = e1' % e2 ∧ P e1 e1')
         → P arg1 arg2)
   ∧ (∀arg1 arg2.
          (∃e1 e2 ty bod. arg1 = e1 % e2 ∧ arg2 = subst bod e2 ∧ e1 = Lam ty bod)
         → P arg1 arg2)
   ∧ arg1 ---> arg2
   → P arg1 arg2"
pragma induct <reduce.induct> [app1,beta]
thm <reduce.cases>
  "arg1 ---> arg2 →
       (∃e1 e1' e2. arg1 = e1 % e2 ∧ arg2 = e1' % e2 ∧ e1 ---> e1')
     ∨ (∃e1 e2 ty bod. arg1 = e1 % e2 ∧ arg2 = subst bod e2 ∧ e1 = Lam ty bod)"
pragma rulecases <reduce.cases>
thm <reduce.elim>
  "(--->) = λarg1 arg2.
       lfp(λR (arg1,arg2).
         (∃e1 e1' e2. arg1 = e1 % e2 ∧ arg2 = e1' % e2 ∧ R(e1,e1'))
       ∨ (∃e1 e2 ty bod. arg1 = e1 % e2 ∧ arg2 = subst bod e2 ∧ e1 = Lam ty bod))
```

```
        arg1  arg2"
pragma elim <reduce.elim>
```

2.2.5 Recursive Functions

Recursive functions are admissible in h.o.l. if the recursion can be proven well-founded. Slind has made a comprehensive study of this topic in the context of deductive frameworks [Sli96] and has implemented his algorithms in a package called TFL, suitable for use with HOL and Isabelle. Because his work has explored the issues thoroughly, for our purposes it is adequate to simply axiomatize recursive functions. Furthermore, in practice we only tend to use primitive recursive functions, and it is easy to verify by inspection that our definitions are indeed primitive recursive. However, the mechanism we propose to implement in future versions of Declare is to axiomatize recursive functions up to the generation of a proof obligation, as in PVS. We would ensure that Declare's automatic prover could detect and prove side conditions for the primitive recursive subset automatically.

2.2.6 Partial Functions and Undefinedness

Partial functions are only fully defined on some elements of their domain. What "happens" outside this domain can vary greatly according to the logical treatment chosen. Muller and Slind's excellent overview of different treatments in a logic of total functions [MS97] demonstrates that it is essential to take an approach to partiality that is both accurate and pragmatic. The basic approaches available when using h.o.l. are:

- *Define fully.* The function is given particular values outside its domain.
- *Underspecify.* The function has various values outside its domain but they are arbitrary and otherwise uninterpreted.
- *Use relations.* That is, model the function by a subset of $\alpha \times \beta$. This is precise, but often requires additional lemmas.
- *Use the* `option` *type.* Model partial functions $\alpha \to \beta$ by total functions of type $\alpha \to \beta$ *option*. This is precise, but requires additional case splits and reasoning about datatypes.

Whichever model is chosen, it is good "declarative" practice to avoid relying on the behaviour of functions and relations outside their natural domain. For example, it is bad practice to rely on $1/0$ having a definite value (e.g. 0, as in $n \times (1/n) = 1$), since it becomes less clear what exactly has been proven, and theorems are not easily transferable (textually) to other theorem proving systems.

We return briefly to these questions in Section 6.3.2 in the context of well-formedness criteria for types and type environments in our case study.

2.2.7 Declarative Specification and Modularity

Specification is one area in which traditional theorem proving has largely achieved the declarative ideal: sufficient forms are available that most specifications can be given without resorting to "irrelevant" detail. The biggest potential source of such detail is "specification by construction", e.g. when constructing the theory of lists via a theory of partial functions from an initial segment of $\mathbb{N}$, or a theory of the real numbers by an elaborate construction. Theorem provers typically support techniques which admit wide classes of constructs declaratively (such as algebraic datatypes), and may also provide mechanisms to hide constructions once they have been completed.

As with some other theorem provers (e.g. PVS), Declare goes a little further: a theory may be specified and used independently of the proofs that demonstrate that the theory is a sound extension of the logic. That is, Declare supports a primitive notion of modularity. The interface to a theory may either be given in a separate file (called an "abstract"), or may be extracted from an existing file (an "article" — the proofs in the article need not be checked to do this). Naturally there need be no textual dependency of an abstract on its article, and an article is checked for conformance to its abstract. Additionally, every type and term constant in Declare is qualified by the name of the module in which it occurs – discrimination of constants occurs during parsing.

Unlike PVS, a Declare theory comes equipped with the pragmas that give extra-logical information about the theory. Declare also comes with traditional compiler-like tools for processing abstracts and articles, and `make` facilities can be employed in the usual fashion.[2]

Finally, Declare comes with a standard library of theories that axiomatize first order logic, pairs, lists, options, finite partial functions, first order set theory, finite sets, lists-as-vectors and some conversions between these structures. We have not yet provided proofs of the soundness of these axiomatizations, though they were originally copied from similar theories in HOL and HOL-lite.

2.3 Labelling and Theorem Extraction

Predicates in h.o.l. are functions from several arguments to type *bool*. In this section we describe a new mechanism whereby the labelling of a subterm within the definition of a predicate gives rise to a "theorem for free".[3]

For example, consider the following definition, which is an alternative way of defining `<<=` from Section 1.4.3. The labels have been emphasized, and the underlining indicates the term identified by the labels:

`let "`*E1* `<<=` *E2* `[<derive>]` ↔

[2] A prototype module system has also been designed and implemented for Declare, however this is beyond the scope of this thesis.

[3] Indeed, while this mechanism is usually used within predicate definitions, it can also be used within any fact stated anywhere in a specification or proof (proofs are discussed in Chapter 3).

$\mathsf{pos\_res}_{\epsilon}(P) = P$	$\mathsf{neg\_res}_{\epsilon}(P) = \neg P$
$\mathsf{pos\_res}_{0,p}(\underline{P} \wedge Q) = \mathsf{pos\_res}_{p}(P)$	$\mathsf{neg\_res}_{0,p}(\underline{P} \wedge Q) = \mathsf{neg\_res}_{p}(P)$
$\mathsf{pos\_res}_{1,p}(P \wedge \underline{Q}) = \mathsf{pos\_res}_{p}(Q)$	$\mathsf{neg\_res}_{1,p}(P \wedge \underline{Q}) = \mathsf{neg\_res}_{p}(Q)$
$\mathsf{pos\_res}_{0,p}(\neg\underline{P}) = \mathsf{neg\_res}_{p}(P)$	$\mathsf{neg\_res}_{0,p}(\neg\underline{P}) = \mathsf{pos\_res}_{p}(P)$
$\mathsf{pos\_res}_{0,p}(\forall x.\ \underline{P}) = \mathsf{pos\_res}_{p}(P)$	
$\mathsf{pos\_res}_{0,p}(\underline{P} \leftrightarrow Q) = \mathsf{pos\_res}_{p}(P)$	$\mathsf{neg\_res}_{0,p}(\underline{P} \leftrightarrow Q) = \mathsf{neg\_res}_{p}(P)$
$\mathsf{pos\_res}_{1,p}(P \leftrightarrow \underline{Q}) = \mathsf{pos\_res}_{p}(Q)$	$\mathsf{neg\_res}_{1,p}(P \leftrightarrow \underline{Q}) = \mathsf{neg\_res}_{p}(Q)$

Table 2.1: The Result at a Location.

```
len E1 <= len E2 [<length>] ∧
(∀j. j < len E1 → (el j E2 = el j E1) [<contents>])";
```

The corresponding three theorems are:

```
thm <leq.derive> "len E1 <= len E2 ∧
                  (∀j. j < len E1 → el j E2 = el j E1) →
                  E1 <<= E2"
thm <leq.length> "E1 <<= E2 → len E1 <= len E2"
thm <leq.contents> "E1 <<= E2 ∧ j < len E1 → el j E2 = el j E1"
```

Labels can be placed anywhere within a propositional structure. The resulting theorem is $C \rightarrow P$ where C is the "minimal support" at the loci (defined formally below), and P is the labelled term. P is negated if it appears in at a "negative" location, e.g. under a single negation or immediately on the left of an implication. Labels may also be placed under $\forall$ quantifiers, which generate free variables in the theorem (no two variables in the same scope may have the same name), and also under applied pattern matches.

This mechanism was used extensively in the case studies, as it gives a succinct way of "getting a handle on" the immediate consequences of a definition without needlessly restating the obvious. This can save pages of text in a large specification. The only down-side is that the term language must be syntactically extended to include constructs that rightly seem to belong in the specification language, but this is a small price to pay.

We can formalize what is going on here. Let a *path* be a list of zeros and ones, and let $\mathsf{pos\_res}_p(A)$ be the result of a path p in term A, as defined in Table 2.1. Let $\mathsf{pos\_supp}_p(A)$ be the minimal support as defined in Table 2.2. We have the following soundness theorem:

Theorem 2 ***Soundness of Theorem Extraction.*** *If A is a proposition, p is a well-formed path for A, and A and $\mathsf{pos\_supp}_p(A)$ holds in the current theory, then $\mathsf{pos\_res}_p(A)$ also holds. Similarly if $\neg A$ and $\mathsf{neg\_supp}_p(A)$ hold, then $\mathsf{neg\_res}_p(A)$ holds.*

The proof is straightforward and is by induction on the length of the path p.

$$
\begin{array}{ll|ll}
\mathsf{pos\_supp}_{\epsilon}(P) & = \mathsf{true} & \mathsf{neg\_supp}_{\epsilon}(P) & = \mathsf{true} \\
\mathsf{pos\_supp}_{0,p}(\underline{P} \wedge Q) & = \mathsf{pos\_supp}_{p}(P) & \mathsf{neg\_supp}_{0,p}(\underline{P} \wedge Q) & = \neg Q \wedge \mathsf{neg\_supp}_{p}(P) \\
\mathsf{pos\_supp}_{1,p}(P \wedge \underline{Q}) & = \mathsf{pos\_supp}_{p}(Q) & \mathsf{neg\_supp}_{1,p}(P \wedge \underline{Q}) & = \neg P \wedge \mathsf{neg\_supp}_{p}(Q) \\
\mathsf{pos\_supp}_{0,p}(\neg \underline{P}) & = \mathsf{neg\_supp}_{p}(P) & \mathsf{neg\_supp}_{0,p}(\neg \underline{P}) & = \mathsf{pos\_supp}_{p}(P) \\
\mathsf{pos\_supp}_{0,p}(\forall x.\ \underline{P}) & = \mathsf{pos\_supp}_{p}(P) & &
\end{array}
$$

$$
\begin{array}{ll}
\mathsf{pos\_supp}_{0,p}(\underline{P} \leftrightarrow Q) = Q \wedge \mathsf{pos\_supp}_{p}(P) & \mathsf{neg\_supp}_{0,p}(\underline{P} \leftrightarrow Q) = Q \wedge \mathsf{neg\_supp}_{p}(P) \\
\mathsf{pos\_supp}_{1,p}(P \leftrightarrow \underline{Q}) = P \wedge \mathsf{pos\_supp}_{p}(Q) & \mathsf{neg\_supp}_{1,p}(P \leftrightarrow \underline{Q}) = P \wedge \mathsf{neg\_supp}_{p}(Q)
\end{array}
$$

Table 2.2: The Minimal Logical Support at a Location. We omit $\vee$, $\rightarrow$ and $\leftarrow$ since they may be defined simply via $\wedge$ and $\neg$. There is no rule for neg_supp and $\forall$.

$$
\begin{array}{ll}
\mathsf{pos\_res}_{0^*,p}(\underline{P} \leftrightarrow Q) = \mathsf{neg\_res}_{p}(P) & \mathsf{pos\_supp}_{0^*,p}(\underline{P} \leftrightarrow Q) = \neg Q \wedge \mathsf{neg\_supp}_{p}(P) \\
\mathsf{pos\_res}_{1^*,p}(P \leftrightarrow \underline{Q}) = \mathsf{neg\_res}_{p}(Q) & \mathsf{pos\_supp}_{1^*,p}(P \leftrightarrow \underline{Q}) = \neg P \wedge \mathsf{neg\_supp}_{p}(Q) \\
 & \\
\mathsf{neg\_res}_{0^*,p}(\underline{P} \leftrightarrow Q) = \mathsf{pos\_res}_{p}(P) & \mathsf{neg\_supp}_{0^*,p}(\underline{P} \leftrightarrow Q) = \neg Q \wedge \mathsf{pos\_supp}_{p}(P) \\
\mathsf{neg\_res}_{1^*,p}(P \leftrightarrow \underline{Q}) = \mathsf{pos\_res}_{p}(Q) & \mathsf{neg\_supp}_{1^*,p}(P \leftrightarrow \underline{Q}) = \neg P \wedge \mathsf{pos\_supp}_{p}(Q)
\end{array}
$$

Table 2.3: Possible reversed support rules for $\leftrightarrow$.

Tables 2.1 and 2.2 also define what happens at the ambiguous connective $\leftrightarrow$. For pos_supp, this is interpreted as a left-implication ($\leftarrow$) when the path points to the left, and a right-implication ($\rightarrow$) when to the right, as we can see in the first example above.

2.3.1 Possible Extensions to the Mechanism

It could potentially be useful to allow the reversal of the interpretation of $\leftrightarrow$, interpreting it as ($\rightarrow$) when on the path points to the left, as shown in Table 2.3 (we use 0^* and 1^* to indicate this in a path). For example:

```
let "either(P,l,r) [<*rule*>] ↔ P(l) [<*left*>] ∨ P(r) [<*right*>]";
```

would give the theorems:

```
thm <either.rule> "either(P,l,r) → P(l) ∨ P(r)"
thm <either.left> "P(l) → either(P,l,r)"
thm <either.right> "P(r) → either(P,l,r)"
```

Furthermore, the whole scheme could be extended to work with non-first order operators, for example fixed points.[4] Table 2.4 shows the appropriate rules. For example

[4] Rather than using the fixed point specification syntax from Section 2.2.4 we use `lfp` to denote a general least fixed point operator.

$$\begin{array}{ll}
\mathsf{pos_supp}_0(\forall x.\ \underline{c(x)} \leftrightarrow \mathsf{lfp}(\lambda P x.\ F[P,x])(x)) & = F[c/P](x) \\
\mathsf{pos_supp}_{1,p}(\forall x.\ c(x) \leftrightarrow \underline{\mathsf{lfp}(\lambda P x.\ F[P,x])(x)}) & = c(x) \wedge \mathsf{pos_supp}_p(F[c/P](x)) \\
\mathsf{pos_supp}_{0^*}(\forall x.\ \underline{c(x)} \leftrightarrow \mathsf{lfp}(\lambda P x.\ F[P,x])(x)) & = \neg F[c/P](x) \\
\mathsf{pos_supp}_{1^*,p}(\forall x.\ c(x) \leftrightarrow \underline{\mathsf{lfp}(\lambda P x.\ F[P,x])(x)}) & = \neg c(x) \wedge \mathsf{neg_supp}_p(F[c/P](x))
\end{array}$$

Table 2.4: Possible Support Rules for Fixed Points. Here the fixed point expression is given a name c so we can succinctly unwind it once — several logically equivalent forms could be similarly detected.

```
let "all P l [<*cases*>] ↔
       lfp (λall l.
           (l = []) [<*nil*>]
         ∨ ∃h t. ((l = h#t ∧ P(h) ∧ all t) [<*cons*>])
       ) l";
```

would give the theorems:

```
thm <all.cases> "all P l →
             (l = [])
           ∨ (∃h t. l = h#t ∧ P(h) ∧ all P t)"
thm <all.nil> "l = [] → all P l"
thm <all.cons> "l = h#t ∧ P(h) ∧ all P t → all P l"
```

This could unify the existing mechanism with the current labelling mechanism for rules of fixed point relations. One could also investigate the generalisation of this mechanism in a system such a Isabelle, perhaps allowing labels within further non-first order (e.g. modal) structures if appropriate rules are present to interpret the paths to these labels. We have not implemented these mechanisms.

In principle, labels could also be placed under positive $\exists$ quantifiers, which would be systematically skolemized to generate constants. For example:

```
let "big n [<derive>] ↔ ∃m. (m < n) [<c1>] ∧ (m > 1000) [<c2>]";
```

would give one constant (`big.m`) and four theorems:

```
thm <big> "big n ↔ ∃m. m < n ∧ m > 1000"
thm <big.c1> "big n → big.m n < n"
thm <big.c2> "big n → big.m n > 1000"
thm <big.derive> "(∃m. m < n ∧ m > 1000) → big n"
```

Note the skolem constant `big.m` is parameterized by the free variable n. This mechanism was implemented, but was not used in the case studies.

2.3.2 Related Work

Mizar [Rud92] allows facts to be labelled as they are stated, taking the current context and generating an implicative theorem. This mechanism was the inspiration for the mechanism presented here, but is not as general, since labels may not appear inside arbitrary propositional structures.

2.4 Validation

If all proof obligations are discharged, the logical consistency of a specification is essentially trivial to check, simply because of the limited range of specification constructs that we admit.[5] Considerably more difficult is the *validity* of the specification, by which we mean whether the specification meets our informal expectations of the system we are describing. For example, in Chapter 6 we must argue that, in some sense, our model of the language conforms to the Java Language Specification [GJS96].

We regard the issue of validation as extremely important in the context of operational semantics. Without validation, we really have no guarantee that our theorem proving efforts have demonstrated anything useful. Whether we like it or not, specifications frequently contain errors, ranging from small syntactic mistakes to entire rules that are simply forgotten. We found examples of such mistakes even toward the end of our major case study (see Section 6.7).

Clearly complete formal validation is not possible, since this would require a formal specification at least as accurate as our own. Thus we turn to partial and informal techniques. In addition to simply eye-balling the specification, we utilise the following (semi-)automatic techniques:

1. Type checking;
2. Static mode analysis;
3. Generation of executable code;
4. Execution of test cases.

Type checking is of course decidable in our variant of higher order logic, and successful type checking at least demonstrates that the various terms within the specification lie within the correct sets.

Typechecking finds many bugs, but is well-understood, and so the remaining techniques are of more interest. As demonstrated in Section 1.4.3, we compile specifications to the programming language Mercury [SHC96] and leverage the static analysis and animation facilities of that system. In the context of operational semantics this generates an interpreter for the language based directly on our definitions.

[5] As mentioned in the previous sections, in the current implementation of Declare we must also check (by inspection) that datatypes are initial, that inductive relations are monotonic and that recursive function axiomatizations are indeed primitive recursive.

The interpreter is typically able to execute concrete programs if given a concrete environment, and suffices to test small programs.

2.4.1 Mercury

Mercury is a pure Prolog-like programming language with higher order predicates and functions (though without higher order unification). It includes algorithms to statically analyse programs for *type*, *mode*, *determinism*, *uniqueness* and *termination* constraints.

- The type system is based on polymorphic many-sorted logic and is much the same as typical functional type systems. It includes polymorphic datatypes.[6]
- Modes specify the flow of information through a computation by indicating, among other things, how predicates effect the "instantiatedness" of expressions. Typical modes are `in` and `out` for inputs and outputs respectively. Other modes include `di` and `uo` for destructive input and unique outputs: these are not yet used in Declare, though there is no real reason why the entire Mercury mode language could not be used.
- Determinism constraints indicate the potential number of solutions to a predicate and form a lattice: `nondet` indicates 0 or more solutions, `multi` is 1 or more, `semidet` is 0 or 1, `det` is 1, `failure` is 0. As yet we do not take advantage of Mercury's determinism checks. They are, unfortunately, not quite powerful enough to detect the determinism of our typical inductive relations (that is, without substantial modification to the translation process, or considerable artificiality in how the relations are formulated). We leave this as future work, and for the moment declare all translated relations as `nondet`.

As with other Prologs, Mercury also warns about such common programming errors as variables that are only used once within a particular scope.

Mercury predicates follow Prolog, though require type and mode declarations, e.g.[7]

```
:- pred append(list(T), list(T), list(T)).
:- mode append(di, di, uo) is det.
:- mode append(in, in, out) is det.
:- mode append(in, out, in) is semidet.
:- mode append(out, out, in) is multi.
```

[6] Mercury has options to infer types. Since we have already inferred types in Declare we can generate the type declarations directly.

[7] Familiarity with Prolog syntax is required to understand this section. A quick summary: '`;`' represents disjunction, '`,`' conjunction, '`=>`' implication, '`:-`' is the turnstile. Variables begin with capitals and constants with lowercase (unless quoted as in `'Var'`). Clauses have the form *pred*`(`*args*`)` `:-` *goal*`.` for predicates, and *func*`(`*args*`)` `=` *expr* `:-` *goal*`.` for expressions. Existential/universal quantification is `some/all`.

`pragma code` *thm*	The theorem should be used to generate Mercury code.
`pragma func` *name*	The given constant should be translated a a Mercury function generating boolean values, rather than as a predicate.
`pragma mode` *term*	The term specifies a Mercury mode for a relation.
`pragma test` *term*	The term specifies a test predicate.

Table 2.5: Pragmas relevant to Mercury

```
append([], Ys, Ys).
append([X | Xs], Ys, [X | Zs]) :-
        append(Xs, Ys, Zs).
```

Higher order predicates may take expressions and predicates as arguments, e.g. reflexive transitive closure:

```
:- pred rtc(pred(A, A), A, A).
:- mode rtc(pred(in,out) is nondet, in, out) is nondet.
rtc(R,X,X).
rtc(R,X,Y) :- R(X,X1), rtc(X1,Y).
```

Expressions include the standard range of terms found in a pure functional programming language, such as constructed terms, lambda expressions function applications and conditional expressions, and also unassigned variables as in Prolog.[8]

Because of its extensive static analyses, Mercury can generate extremely efficient code, often many times faster than existing Prolog systems. However, execution times were not particularly important for our case studies, since the tests we ran were small.

2.4.2 Example translations

We shall demonstrate the translation of specifications to Mercury by some examples. The type declarations from page 11 (actually, with some slight variations) translate as follows[9]

```
:- type typ ---> 'TyCon'; 'TyFun'(typ,typ).
:- type exp ---> 'Var'(int); 'Con'; 'Lam'(typ,exp); 'App'(typ,exp,exp).
```

[8] Expressions are normally deterministic, but may also be semi-deterministic or non-deterministic, and thus denote sets of values. Non-deterministic expressions are rarely, if ever, used.

[9] Mercury does not accept curried datatype constructors, so we uncurry and demand that datatype constructors are not partially instantiated in the specification.

The translation of curried functions is somewhat grotesque: Mercury's preferred syntactic form is to have uncurried functions, but to cleanly support the partial application of functions we generate curried forms. The `subst` function from page 11 becomes:

```
:- func subst = (func(exp) = (func(exp) = exp)).
:- mode subst =
      out(func(in) =
      out(func(in) = out is semidet) is semidet) is semidet.

subst = apply(subst_aux,0).
```

The first line specifies the (curried) type of the function, and the mode constraint specifies that partial application produce outputs that are functions that subsequently produce further inputs. Note that if we uncurried, the form would be the somewhat simpler:

```
:- func subst(exp,exp) = exp.
:- mode subst(in,in) = out is semidet.
subst(X,Y,Z) = subst_aux(0,X,Y,Z).
```

The axiom `<reduce>` from page 27 translates as follows[10]

```
:- pred '--->'(exp,exp).
:- mode '--->'(in,out) is nondet.

'--->'(Arg1,Arg2) :-
    (some (E1,Dty,E1_prime,E2)
       Arg1 = 'App'(Dty,E1,E2),
       Arg2 = 'App'(Dty,E1_prime, E2),
       '--->'(E1,E1_prime))
  ; (some (Ty,Dty,Bod,E2)
       Arg1 = 'App'(Dty,'Lam'(Ty,Bod),E2),
        Arg2 = apply(apply(subst,Bod),E2)).
```

It is useful to extend the range of translated constructs by detecting first order constructs that correspond to common idioms:

- $\forall j.\ m \leq j < n \rightarrow P[j]$ and related forms are translated to a call to the higher order predicate `finite_int_forall`:

```
finite_int_forall(M,N,P) :-
    if M >= N then true else (P(M), finite_int_forall(M+1,N,P)).
```

[10] We also uncurry predicates, even though Mercury supports higher order predicates. This is because Mercury has built in support for the partial application of "uncurried" predicates.

- All other bounded universal quantifications are translated with the expectation that the bound represents a call that generates a finite range of values, that is $\forall \vec{v}.P[\vec{v}] \rightarrow Q[\vec{v}]$ becomes a call to the higher order predicate `bounded_forall`:[11]

```
bounded_forall(P,Q) :-
     solutions(P,List), all [X] (member(X,List) => Q(X)).
```

- Pattern matches in the expressions become the appropriate Mercury conditionals. For example

```
function [] -> e1 | (h#t) -> e2
```

  becomes the Mercury lambda expression

```
(func(X) = Y :- if X = [] then Y = e1
                else if X = [ H | T ] then Y = e2
                else fail)
```

  That is, the result `Y` of the function with input `X` is the solution to the predicate after the turnstile `:-`.

- Pattern matches in predicates are treated similarly. For example

```
match x with [] -> true | (h # _t) -> (h = 1)
```

  becomes the Mercury predicate[12]

```
if X = [] then true
else if X = [ H | _T ] then H = 1
else fail
```

Note that boolean valued functions will normally be treated as predicates, unless the `func` pragma is used (see Table 2.5).[13] Negation is translated to Mercury's negation-as-failure.

[11] The call `solutions(P,List)` deterministically generates a solution set for the predicate `P`.

[12] Aficionados of functional programming languages may note that we have collapsed the function application hidden inside the `match` expression. Applied pattern matches in predicates (such as the above example) are not translated to corresponding higher order predicate applications because Mercury does not recognise that it can $\beta-$reduce the immediate function application. In the given example it would complain that `H` is being bound within a closure.

[13] The `func` pragma is not yet implemented in Declare, as manipulations on boolean values as data are rare in our specifications. We mention it here to show how one might declare boolean valued functions.

At the top level, Declare generates Mercury code off datatype declarations and any axiom with a `code` pragma (see Table 2.5 for all the pragmas relevant to the Mercury translation.[14] These normally define a predicate by an if-and-only-if $\leftrightarrow$ or a data value by an equation `=`. Declare implicitly generates code for all constructs introduced by `defn`, `lfp` or `gfp` unless the pragma `nocode` is given. Of course, not all h.o.l. axioms represent executable code: in these cases the process normally fails when the system tries to compile the generated Mercury code.[15]

Declare produces a Mercury module for each Declare input file. The modules are compiled together and linked against some core functionality and a `main` program that executes all `test` pragmas (see also the example on page 13).

2.4.3 Related Work

A previous version of this work generated executable code by compiling specifications to CaML-light [Mau91] and performing a modicum of mode analysis during this translation (see [Sym97b]). Although useful at the time, the translation was clumsy in comparison to the translation to Mercury. The Mercury version allows considerably more flexibility in the style in which specifications are written. Previously some rather artificial devices were needed to distinguish relations from functions, higher order relations could not be translated, bounded quantifications were clumsy, and special hacks were needed to translate relations that generated lists of outputs.

The executability of specifications has been widely discussed amongst users of the Z specification methodology. Early work by Hayes and Jones [HJ89] identified that executable specifications may be inferior to non-executable ones. Two types of reasons are cited:

- The executability mechanism may force the essentially the same specification to be written in an unnatural style, e.g. conjuncts may need to be given in a particular order.
- Executability may force a simple specification to be abandoned, e.g. because it limits the use of many constructs such as negation, disjunction and universal quantification.

Many of these criticisms are not terribly important in our problem domain, because we are trying to prove properties of systems that should certainly be executable. However, in any case,

- Mercury is very flexible in the programs it will accept. For example, it places conjuncts in a sensible execution order using mode analysis.
- Declare allows code to be generated from any *theorems*, and not just definitions. Thus a specification can be given in a natural fashion, and an equivalence or refinement with an executable version can be proved.

[14] All definitions have `code` pragmas unless the `nocode` pragma is used.

[15] One deficiency in the current system is that line numbers are not faithfully translated from Declare to Mercury.

Wahls, Leavens and Baker [WLB98] use the constraint based programming language AKL [JH94] to provide an execution apparatus for their language SPECS-C++. The apparatus is roughly as flexible as our own, though the flexibility is provided by quite a different means: for example Mercury places conjuncts in a sensible execution order using mode analysis, while AKL does this by propagating constraints. AKL supports the additional expressive power of linear inequality constraints, but does not support higher order features. Whals et al. do not consider the important issue of leveraging the static analysis algorithms available in the underlying logic programming engine — clearly Mercury is particularly strong in this regard.

Andrews [And97] translates the specification language S [JDD94] to Lambda Prolog [FGMP90], a higher order Prolog, and this work has quite a similar feel to our own. The result is convincing as far as it goes, however again static analyses are not utilised. Mercury also supports the definition of expression (function) constants, which Andrews notes as a particular obstacle for his translation.

Because of this rich range of features, Mercury appears to be very much a "natural" programming language corresponding to higher order logic (except for, perhaps, the absence of higher order unification). Indeed Mercury is so strong (though a little syntactically clumsy) that one could imagine turning the tables and using it as a specification language. Specifications could then be given a semantics in h.o.l. when theorem proving is required, and the meta-programming facilities available in Mercury would make the implementation of the theorem prover relatively easy.

Chapter 3

Declarative Proof Description

In this chapter we describe the technique we use for proof description, called *declarative proof*. We consider the principles that guided the design of the Declare proof language and detail the three primary constructs of the language. The technique represents a somewhat radical departure from standard practice in higher order logic theorem proving, and we explain the pros and cons of the approach.

3.1 The Principles of Declarative Proof

Harrison [Har97b] describes several different uses of the word "proof" in the field of automated reasoning. Three of these are of interest here:

1. A proof as found in a mathematical text book, i.e. a sketch given in a mixture of natural, symbolic and formal languages, sufficient to convince the reader.

2. A script to be presented to a machine for checking. This may be just a sketch, or a program which describes the syntactic manipulations needed to construct a formal proof.

3. A formal 'fully expansive' proof in a particular formal system, e.g. a derivation tree of inference rules and axioms.

We use the word 'proof' in the second sense, and 'proof outline' to mean proofs (again in the second sense) that are merely sketches, and that require significant reasoning to fill in gaps. Proofs in Declare are expressed as proof outlines, in a language that approximates written mathematics. This builds on work done with similar languages by the Mizar group [Rud92] and Harrison [Har96b].

One traditional form of proof description is "tactic" proof, described more fully at the end of this chapter. Although tactics are in principle a very general mechanism, in practice their use is highly "procedural": the user issues proof commands like "simplify the current goal", "do induction on the first universally quantified variable" or "do a case split on the second disjunctive formula in the assumptions". That

is, tactic proof almost invariably proceeds by *giving commands that syntactically manipulate existing facts and goals.* The primary proof description languages of HOL, Isabelle and PVS are tactic based.

In contrast, a declarative style is based on *decomposing* and *enriching* the logical environment (which is the sum of all available facts). Our proposal is that for many purposes declarative proof is a superior method of proof description.

In a declarative proof, the logical environment is *monotonically increasing* along any particular branch. That is, once a fact becomes available, it remains available.[1] The user manages the logical context by labelling facts and goals, and specifying meaningful names for local constants. This allows coherent reasoning within a complicated logical context.

Our declarative proof language separates *proof outlining* from *automated reasoning.* We adopt the principle that these are separate activities and that the proof outline should not invoke complicated routines such as simplification, except to discharge obligations. The link between the two is provided by *justifications*, and the justification language is quite different to the proof outlining language. This is as opposed to tactic based theorem provers (see Section 3.6.1) where one mechanism is typically used for both tasks.

Mechanisms for brevity are essential within declarative proofs, since a relatively large number of terms must be quoted. Declare attempts to provide mechanisms so that the user need never quote a particular term more than once within a proof. For example one difficulty is when a formula must be quoted in both a positive and a negative sense (e.g. as both a fact and an antecedent to a fact): this happens with induction hypotheses. Another is when using chained (in)equality reasoning. Later in this chapter we describe the particular mechanisms provided: local definitions; abbreviations; type checking in context; stating problems in sequent form; instantiation up to type unification; and `ihyp` macros.

In our declarative proof language, the user states "where he/she wants to go". That is, the user declares an enrichment or decomposition, giving the logical state he/she wants to reach, and only states "how to get there" in high level terms. The user does not specify the syntactic manipulations required to get there, except for some hints provided in the justification, via mechanisms we have tried to make as declarative as possible. Often the justification is simply a set of theorem names.

Existing theorem provers with strong automation effectively support a kind of declarative proof at the top level. For example, the Boyer-Moore prover [BM81] is declarative in this sense — the user conjectures a goal and the system tries to prove it. If the system fails, then the user adds more details and tries again. The process is like presenting a proof to a colleague: one starts with an outline and then provides extra detail if he/she fails to follow the argument. Declare extends this approach to allow declarative decompositions and lemmas in the internals of a proof, thus giving the benefits of scope and locality.

[1] There is one important exception to this rule: see Section 3.5

In Section 1.4.1 we defined "declarative" to mean "relatively free of operational detail", i.e. "what" not "how". Proofs in Declare are relatively independent of a number of factors that are traditional sources of dependency in tactic proofs. These include:

- The ordering of facts and goals in a problem statement (in Declare the context is a set indexed by user supplied names);
- The order in which subgoals are produced by a proof utility (in Declare the user can solve subgoals in any order, and Declare produces an obligation that justifies the user's choice of decomposition) ;
- The order of quantifiers in a formula (e.g. in Declare the difference between $\forall a\ b. \ldots$ vs. $\forall b\ a. \ldots$ is irrelevant when providing an instantiation — *cf.* the standard HOL mechanism that instantiates the outermost quantifier);
- The choice of names made by proof utilities for local constants and variables (in Declare all local names are specified by the user);
- The absence of certain kinds of facts in the statement of the problem, e.g. introducing an extra assumption may cause a rewriting proof utility to fail to terminate, or may reduce a goal further than expected by a later tactic (in Declare adding an extra fact to the context can do no harm, unless that fact is explicitly placed in the databases of the automatic tools).

For example, Isabelle, HOL and PVS proofs frequently contain references to assumption or subgoal numbers, i.e. indexes into lists of each. The proofs are sensitive to many changes in problem specification where corresponding Declare proofs will not be. In Declare such changes will alter the proof obligations generated, but often the obligations will still be discharged by the same justifications.

Much "proof independence" (i.e. declarative content) arises from (and depends on) the presence of powerful automation. For example, automating Presburger arithmetic lets the user ignore the difference between $x < 0$ and $x \leq -1$ for most purposes, and thus the user can operate on a semantic level with respect to parts of their theory. Declare utilises the successful automation of large segments of propositional first order reasoning to allow proofs that are relatively free of propositional and first order dependencies.

The advantages of using a declarative proof language in contrast to tactic proof are:

- Proofs are described using only a small number of simple constructs, and thus proofs may be interpreted without knowing the behaviour of a large number of (often *adhoc*) tactics.
- Declarative proofs are more readable.

- A declarative style allows the user to easily specify "mid-points" in a line of argument that divide the complexity of the reasoning into approximately equal chunks (we give a lengthy example of this in Section 3.3.1).
- Automation is aided by having explicit goals at each stage. For example we typically give both the left and right hand sides of an equation, and leave the automated engine to prove the equality.
- Analysing the outline of a declarative proof always terminates, because we can choose to simply generate and not discharge obligations. This means it is possible to typecheck declarative proofs in their entirety, before trying the obligations, thus catching many errors in proofs at an early stage.
- It is relatively easy to implement error-recovery.

Three additional, important benefits seem probable but are difficult to demonstrate conclusively:

- Declarative proofs are potentially more maintainable;
- Declarative proofs are potentially more portable;
- Declarative proofs may appeal to a wider class of users, helping to deliver automated reasoning and formal methods to mathematicians and others.

These, in principle, are instances of the general benefits that arise from increasing the declarative content of an artifact, as discussed in Section 1.4.1.

3.2 Three Constructs For Proof Description

In this section we shall describe the three primary constructs of the Declare proof language, which we have already introduced by example in Section 1.4.3. These are:

- Decomposition and enrichment;
- Proof by automation (with hints) using `by` clauses;
- 2nd order schema application for inductive and other arguments.

Sketching the Semantics

For each construct we shall briefly describe its semantics by using a proof system with judgments

- $\Gamma \vdash \Gamma'$, that is Γ' is a conservative extension of Γ (i.e. Γ' possesses a standard model (see [GM93]) if Γ does);[2]

[2] Actually the conservative extension relation is the reflexive transitive closure of this relation.

- $\Gamma \vdash \mathsf{F}$, that is Γ leads to a contradiction.[3]

Here Γ is a *logical environment* that contains:

- A signature of type and term constants;
- A set of axioms, each of which are closed higher order logic terms (free type variables are treated as quantified at the outer level of each axiom).

Logical environments must always be *well-formed*: i.e. all their terms must typecheck with respect to their signature. We omit well-formedness judgments in this chapter since they may always be checked syntactically. Enrichment of logical environments by new constants ($\oplus_{sig}$) and new axioms ($\oplus_{ax}$) are each defined in the obvious way, with the (normally implicit) side condition that the new constants are not already present in the environment. For simplicity, logical environments are always assumed to contain all the standard propositional and first order connectives. In the implementation of the logic, axioms in logical environments are named and are tagged with "usage directives" as described in Chapter 4.

In this setting, each specification construct of the previous chapter corresponds to a $\Gamma \vdash \Gamma'$ inference rule, e.g. for simple definitions:

$$\frac{\begin{array}{c} c \text{ is fresh in } \Gamma \\ c \text{ not free in } t \end{array}}{\Gamma \vdash (\Gamma \oplus_{sig} c) \oplus_{ax} (c = t)}$$

and for datatypes

$$\frac{\begin{array}{l} D \text{ is a description of a free algebra} \\ D \text{ is initial (given the types in } \Gamma) \\ \vec{c} \text{ are the type and term constants defined by } D \\ \text{All } \vec{c} \text{ are fresh in } \Gamma \\ ax \text{ is the algebraic axiom characterizing } D \end{array}}{\Gamma \vdash (\Gamma \oplus_{sig} \vec{c}) \oplus_{ax} ax}$$

It is possible to combine such rules into just one specification rule for type and term constants: see Harrison's rule in HOL-lite for example [Har96a]. Also see HOL [GM93] for proofs that such constructs do indeed form conservative extensions to higher order logic.

$\Gamma \vdash \mathsf{F}$ judgments are used when interpreting proofs. The two kinds of judgments are linked by the problem-introduction rule:

$$\frac{\Gamma \oplus_{sig} \vec{v} \oplus_{ax} p_1, \ldots, p_m, \neg q_1, \ldots, \neg q_n \vdash \mathsf{F}}{\Gamma \vdash \Gamma, \forall \vec{v}.\ p_1 \ \wedge \ \ldots \ \wedge \ p_m \rightarrow q_1 \vee \ldots \vee q_n}$$

That is, if we can prove a contradiction after assuming all our facts and the negation of each of our goals, then we have proved the corresponding implicative theorem, and can add it to the environment.

[3] We prefer the simpler one-sided judgments $\Gamma \vdash \mathsf{F}$, as compared to the traditional two-sided sequent judgments of a sequent calculus because, when using classical higher order logic, goals correspond precisely to negated facts, and the given presentation corresponds very closely to the implementation.

3.3 Decomposition and Enrichment

Enrichment is the process of adding facts, goals and local constants to a logical environment in a logically sound fashion. Most steps in vernacular proofs are enrichment steps, e.g. "now we know a is even because $2 * b = a$" or "consider d and r such that $n = d * m + r$ and $0 < r < m$." An enrichment step has a corresponding proof obligation that constants "exist" with the given properties, i.e. have witnesses. In Declare, the above examples would translate to, approximately,

```
(a)  have "2*b = a" by ...;
     have "even(a)" by ...;

(b)  consider d, r such that
        "n = d*m + r"
        "0 < r < m"
       by ...;
```

The above are examples of *forward reasoning*. When goals are treated as negated facts, *backward reasoning* also corresponds to enrichment. For example if our goal is $\forall x.(\exists b.x = 4b) \rightarrow \mathsf{even}(x)$ then the vernacular "given b and x such that $x = 4b$ then by the definition of even it suffices to show $\exists c.2 * c = x$" is an enrichment step: based on an existing goal, we add two new local constants (b, x), a new goal $(\exists c.2 * c = x)$ and a new fact $(x = 4b)$. In Declare this would translate to:

```
consider b,x such that
   + "x = 4*b"
   - "∃c. 2*c = x"
  by <even>,<goal>;
```

Decomposition is the process of splitting a proof into several cases. The Declare proof language combines decomposition and enrichment in one construct. The general form is:

```
cases justification
  case label_1
    consider c_{1,1}, ..., c_{1,k_1} such that
       p_{1,1}
       ...
       p_{1,m_1} :
          proof_1
  ...
  case label_n
    consider c_{n,1}, ..., c_{n,k_n} such that
       p_{n,1}
       ...
       p_{n,m_n} :
          proof_n
```

External Form	Internal Form
`have` *facts justification*; *rest of proof*	`cases` *justification* `case` *facts* : *rest of proof* `end;`

`consider` *vars* `st` *facts justification*;
rest of proof
	`cases` *justification*
	  `case`
	  `consider` *vars* `st` *facts* :
	  rest of proof
	`end;`

`let` *id* = *term*;
rest of proof
	`cases`
	  `case "`*id* = *term*`"` : *rest of proof*
	`end;`

`sts` *goal justification*; *rest of proof*	`cases` *justification* `case -` *goal* : *rest of proof* `end;`

Table 3.1: Syntactic variations on enrichment/decomposition with equivalent primitive forms.

The identifiers $c_{1,1}, \ldots, c_{n,m_n}$ are the new local constants and the $p_{i,j}$ are the new facts on each branch. New goals are simply negated facts, syntactically marked either by the word `goal` or "`-`". The proof obligation is that one of the cases always holds, or, equivalently, if we assume the negation of each case we can prove a contradiction:

$$\frac{\begin{array}{l}\Gamma \oplus_{sig} c_{1,1} \ldots c_{1,k_1} \oplus_{ax} p_{1,1}, \ldots, p_{1,m_1} \vdash \mathsf{F} \\ \ldots \\ \Gamma \oplus_{sig} c_{n,1} \ldots c_{n,k_n} \oplus_{ax} p_{n,1}, \ldots, p_{n,m_n} \vdash \mathsf{F} \\ \Gamma \oplus_{ax} \neg(\exists c_{1,1} \ldots c_{1,k_1}.\ \bigwedge p_{1,i}), \ldots, \neg(\exists c_{n,1} \ldots c_{n,k_n}.\ \bigwedge p_{n,i}) \vdash \mathsf{F}\end{array}}{\Gamma \vdash \mathsf{F}}$$

The last proof obligation corresponds to the "default" case of the split, where we may assume each other case does not apply. The case labels are used to refer to these assumptions.[4] The obligation is normally justified by an appeal to automated reasoning, but a nested proof outline can also be given. Syntactically, case labels and the `consider` line can normally be omitted (new symbols are assumed to be new local constants); and we can shorten `such that` to `st`. The special derived forms for the "linear" case $n = 1$ are shown in Table 3.1.

In principle all constant specification constructs could also be admitted within the language, e.g. to define local constants by fixed points, with the implicit support provided by the device in Section 2.2.4. Declare does not implement these within

[4] As it is, the cases are free-standing. They could be interpreted top-to-bottom, left-to-right, so you could assume that previous cases have not held in the proof of a particular case. We have not found this form useful in case studies.

proofs.[5]

3.3.1 A longer example

We shall now look at a longer example of the use of enrichment/decomposition. The example is similar to one that arose in practice in our case study, but is modified to demonstrate several points. The scenario is this:

- We are trying to prove $G(c, c')$ where c and c' are configurations of an abstract machine and we know $c \rightsquigarrow c'$.
- We know we must do a case analysis on all possible ways this transition has occurred.
- $\rightsquigarrow$ is defined by many rules (say 50).
- c takes a particular form $(\mathsf{A}(a, b), s)$ (that is, configurations are pairs of constructed terms and a state)
- Only 8 of the rules apply when c is of this form.
- Out of these 8, 5 represent "exceptional transitions", that is, the machine throws an exception and c' has the form $(\mathsf{E}(val), s)$, i.e. the state doesn't change. For these cases, the goal $G((t_1, s), (\mathsf{E}(val), s))$ happens to be trivial, in the sense that it follows easily from some previous results `<L1>` and `<L2>`
- The last 3 possible transitions arise from the following rules (note the rules do not represent any particular transition system):

$$\frac{(a,s) \rightsquigarrow (v,s') \ \vee \ (b,s) \rightsquigarrow (v,s')}{(\mathsf{A}(a,b),s) \rightsquigarrow (v,s')} \qquad \frac{}{(\mathsf{A}(a,b),s) \rightsquigarrow (a,s)} \qquad \frac{}{(\mathsf{A}(a,b),s) \rightsquigarrow (b,s)}$$

So, how would we formulate the case split in the proof at this point? Consider the following:

```
// The environment contains:
//   "c ---> c'"      <trans>
//   "c = (A(a,b),s)"
// and <L1>, <L2>
cases by rulecases(<trans>), <L1>, <L2>, <goal>
  case "c' = (v', s')"
       "(t,s) ---> (v',s')"
```

[5] Specification constructs that generate new monomorphic types within proofs would require quantification over type variables in the underlying logic [Mel92], and admitting polymorphic types would require quantification over type functions. However, there is little need for the definition of types mid-proof.

```
      "t = a ∨ t = b" :
    rest of proof;
  case "c' = (t, s)"
      "t = a ∨ t = b" :
    rest of proof;
end;
```

The key point is that the structure of the decomposition does not have to match the structure inherent in the theorems used to justify it (i.e. the structure of the rules). There must, of course, be a logical match (one that can be discovered by the automated engine), but the user is given a substantial amount of flexibility in how the cases are arranged. He/she can:

- *Implicitly discharge trivial cases.* This is done by including the facts that support the proof for the 5 "exceptional" cases in justification of the split.
- *Maintain disjunctive cases.* Many tactic based splitting tools such as `STRIP_TAC` in HOL would have generated two cases for the first case listed above, by automatically splitting the disjunct. However, the proof may be basically identical for these cases, up to the choice of t.
- *Subsume similar cases.* That is, two structurally similar cases may be subsumed into one branch of the proof using disjuncts (as in the second case), even if the case splitting theorem generated them separately.[6]

The user can use such techniques to split the proof into chunks that are of approximately equal difficulty, or to dispose of many branches of the proof at one stroke. This is much as in written mathematics, where much trivial reasoning is left to "come out in the wash."

3.4 Justifications, Hints and Automation

At the tips of a problem decomposition we find appeals to automated reasoning to "fill in the gaps" of an argument. We shall discuss the composition of automated reasoning engines for declarative proof in detail in the next chapter: here we shall concentrate on issues related to the proof language.

The automated reasoning engine is treated as an oracle, though of course the intention is that it is sound with respect to the axioms of higher order logic. A set of "hints" (also called a *justification*) is provided to the engine:

$$\frac{\mathsf{prover}(\Gamma, \mathit{hints}(\Gamma)) \text{ returns "yes"}}{\Gamma \vdash \mathsf{F}}$$

The significant issues here are the language used to describe justifications, and the extra information we are allowed to add to Γ to assist the automated reasoner. While

[6] This is, in a sense, a form of "first order factorization." As in arithmetic, helpful factorizations are hard to predict, but easy to justify (e.g. by distribution) once given.

the decomposition construct described in the previous section is clearly quite general and not system-specific, a wide spectrum of justification languages is possible. For example, we might have no language at all (which would assume the automated engine can draw useful logical conclusions efficiently when given nothing but the entire logical environment). Alternatively we might have a language that spells out the syntactic proof in great detail (e.g. the forward inference rules of an LCF-like theorem prover). In some domains it may be useful to have many domain specific constructs.

We have concentrated on finding a minimal set of general justification constructs that are adequate for our case studies. These have indicated that it is extremely useful for the justification language to allow the user to:

- Highlight facts from the logical environment that are particularly relevant;
- Offer explicit instantiations and resolutions as hints;
- Offer case-splits as hints;
- Indicate how various facts can be utilised by the prover, using pragmas;

The first three constructs are quite declarative and correspond to constructs found in vernacular proofs, and we describe them below. We discuss the last mechanism in Chapter 4.

3.4.1 Highlighting Relevant Facts

Facts are *highlighted* in two ways:

- By simply quoting their label, as in "`by <subst_aux_safe>`"
- By never giving them a label in the first place, as all unlabelled facts within proofs are treated as if they were highlighted in every subsequent proof step.

The exact interpretation of the effect highlighting is determined by the automated engine and is described in Section 4.3.1, but the general idea is that highlighted facts must be used by the automated engine for the purposes of rewriting, decision procedures, first order search and so on.

3.4.2 Explicit Instantiations

Our case studies have indicated that a "difficult" proof often becomes quite tractable by simple techniques (e.g. rewriting and first order search) by just providing a few simple instantiations. Furthermore, explicit instantiations are an essential debugging technique when problems are not immediately solvable: providing them usually simplifies the feedback provided by the automated reasoning engine. In a declarative proof language the instantiations are usually easy to write, because terms are parsed in-context and convenient abbreviations are often available. Instantiations can be given by two methods:

- *Type directed.* A fact and a term are given, and we search for "instantiable slots" (that is outer quantifiers of universal strength) that are type-compatible up to the unification of type variables.
- *Explicitly named.* A fact, a term and the name of the variable at an instantiable slot are given.

The mechanism is pleasingly declarative: instantiations can be given in any order, and do not depend on the ordering of instantiable slots in the target fact. For example, consider the explicit instantiation of the theorem `<subst_aux_safe>` from the example in Section 1.4.3. The fact being instantiated is:

```
<subst_aux_safe> ⊢
   ∀e v xty TE n ty.
     [] |- v hastype xty ∧
     len TE = n ∧
     (TE #! xty) |- e hastype ty
     → TE |- (subst_aux n e v) hastype ty
```

and the instantiation directive is:

```
qed by ..., <subst_aux_safe> ["[]", "0", "xty"/xty], ...
```

We have one named and two type-directed instantiations. After processing the named instantiation five instantiable slots remain: e,v,TE,n and ty. Unifying types gives the instantiations $TE \rightarrow$ `[]` and $n \rightarrow$ `0` and the final fact:

```
⊢ ∀e v ty.
     [] |- v hastype xty ∧
     len [] = 0 ∧
     ([] #! xty) |- e hastype ty
     → [] |- (subst_aux 0 e v) hastype ty
```

3.4.3 Explicit Resolutions

Explicit resolution is a mechanism similar in spirit to explicit instantiation. It combines instantiation and resolution by allowing a fact to be used to eliminate a unifying instance in another fact. Continuing the example above:

```
  ...
  have "[] |- e2 hastype xty" <e2_types>;
  ...
  qed by ..., <subst_aux_safe> ["0", <e2_types>], ...
```

The explicit resolution on the justification line gives rise to the hint:

```
⊢ ∀e v ty.
     true ∧
```

```
len [] = 0 ∧
([] #! xty) |- e hastype ty
→ [] |- (subst_aux 0 e v) hastype ty
```

Note only one literal in `<subst_aux_safe>` unified with `<e2_types>`: resolutions must be unique, in order to provide better feedback to the user. Literals do not have to be resolved against antecedents: for example goals (or any negated literal) can be used to resolve against consequents.

Note also that we have not destroyed the first-order structure in the process of resolution. This aids debugging, since all hints are printed out before being passed to the automated engine, and indeed supplying additional explicit resolutions was the primary mechanism for debugging problematic proof steps in the case studies.

One problem with this mechanism is that, as it stands in Declare, unification takes no account of ground equations available in the logical context, and thus some resolutions do not succeed where we would expect them to. For example,

```
...
let "TE' = [] #! xty";
have "TE' |- bod hastype (dty --> ty)" <bod_types>;
...
qed by ..., <subst_aux_safe> [<bod_types>], ...
```

fails because the term constant `TE'` does not unify with (*TE* `#!` *xty*) without considering the equation introduced by the `let`. Such equations are used during automatic proof, but not when interpreting the justification language (we discuss ground equational reasoning and its integration during automatic proof further in Sections 4.2.1 and 4.3). However, this can open a can of worms regarding the treatment of equational reasoning during unification. For example, if we had used the definition

```
let "TE' = [xty]";
```

then should the unification succeed? This would need some special knowledge within the unification algorithm of the equation

$\vdash \forall x$ `[]#!`x `=` $[x]$?

It is also tempting to allow this mechanism to abandon first-order unification and instead generate equational constraints from some "obvious, intended unification". We could require, for example, that only one literal has a matching head constant. However, note that this would not be sufficient to disambiguate the resolution above, as there would now be two potential target literals. Thus we have chosen to live with the syntactic constraints imposed by simple first order unification. If nothing else this is easy for the user to predict and understand.

In this work we only consider explicit resolutions where one fact is a literal: it may be useful to admit more general resolutions but we leave this for future research.

`pragma induct` *thm names*	The theorem specifies an induction scheme, and *names* gives names the subgoals that arise from the application of the schema.
`pragma rulecases` *thm*	The theorem specifies a default rule case analysis technique, suitable for use with the `rulecases` or `structcases` mechanisms.

Table 3.2: Pragmas relevant to induction and justifications

3.4.4 Explicit Case Splits

Explicit case splits can be provided by *instantiating a disjunctive theorem*, *rule case analysis*, or *structural case analysis*. Rule case analysis (`rulecases`) accepts a fact indicating membership of an inductive relation, and generates a theorem that specifies the possible rules that might have been used to derive this fact. Structural case analysis (`structcases`) acts on a term belonging to a free algebra (i.e. any type with an abstract datatype axiom): we generate a disjunctive theorem corresponding to case analysis on the construction of the term.

Other case analysis theorems may be specified using the `rulecases` pragma (see Table 3.2). The theorems must have a similar form to `<reduce.cases>` on page 27. Case analyses could also be achieved by explicitly instantiating these theorems, however building default tables allows the machine to automatically infer the relevant theorem to use.

3.5 Second order Schema Application

In principle, decomposition/enriching and automated proof with justifications are sufficient to describe any proof in higher order logic, assuming a modicum of power from the automated engine (e.g. that it implements the 8 primitive rules of higher order logic described by Gordon and Melham [GM93]). However, we have found it very useful to add one further construct for inductive arguments. The general form we have adopted is *second-order schema application*, which includes structural, rule and well-founded induction, and potentially a range of other proof strategies.

Why is this construct needed? Consider the proof of the theorem `<subst_aux_safe>` from page 16:

```
thm <subst_aux_safe>
if "[] |- v hastype xty"  <v_hastype>
   "len TE = n"           <n>
   "(TE #! xty) |- e hastype ty" <typing>
then "TE |- (subst_aux n e v) hastype ty";
```

We wish to induct over the derivation of the fact `<typing>`, that is over the structure of this inductive set. The induction predicate that we desire is:

```
λTE e ty.
  ∀n. len TE = n →
      TE |- (subst_aux n e v) hastype ty
```

It is essential that n be universally quantified, because it "varies" during the induction, in the sense that it is necessary to instantiate it with different values in different cases of the induction. Likewise TE, e and ty also vary. Furthermore, because v and xty do not vary, it is better to leave `<v_hastype>` out of the induction predicate to avoid unnecessary extra antecedents to the induction hypothesis.

We now contrast how the induction step of the proof is described with a typical tactic proof language, and in Declare with and without a special construct for this purpose.

3.5.1 Induction in Typical Tactic Proof Languages

In a typical tactic language we must state the goal in such a way that the application of the induction schema can be achieved by matching or rewriting, or a tactic program. Thus the problem would be stated as follows (using a sequent form where goals are marked `-` and facts `+`):

```
---------------------------------
- ∀v xty.
   [] |- v hastype xty →
   ∀TE e ty.
     (TE #! xty) |- e hastype ty →
     ∀n. len TE = n →
         TE |- (subst_aux n e v) hastype ty
```

A HOL tactic program to perform an inductive step runs something like `REPEAT GEN_TAC THEN DISCH_TAC THEN RULE_INDUCT_TAC` *schema*, meaning "repeatedly replace universal quantifiers by local constants, then eliminate one implication by placing the antecedent in the assumption list, then apply induction where the inductive set is the antecedent of the current goal and the induction predicate implicit in the consequent". After the `REPEAT GEN_TAC THEN DISCH_TAC` steps the sequent is:

```
+ [] |- v hastype xty
---------------------------------
- ∀TE e ty.
     (TE #! xty) |- e hastype ty →
     ∀n. len TE = n →
         TE |- (subst_aux n e v) hastype ty
```

The user has syntactically isolated the facts that are unchanging from the "dependent" facts that make up the induction predicate. This is not only artificial: we lose the opportunity to mark dependent facts with meta-level information such as names and usage directives when they are introduced.

Perhaps most problematically, the `RULE_INDUCT_TAC` step typically chooses names for local constants, and automatically place induction hypotheses in the assumption lists. Choosing names automatically tends to make proofs fragile (in the sense they become dependent on the rather arbitrary behaviour of the choice mechanism). Also, further tactic steps would be required to attach meaningful names and usage directives to induction facts. Furthermore, proofs of the cases must be listed in a particular order, and the interpreter for the tactic language can't make sense of the proof if cases are omitted.

3.5.2 Induction in Declare without a special construct

It is naturally possible to use Declare's decomposition construct, combined with an explicit instantiation of the induction theorem, to express the desired decomposition:[7]

```
thm <subst_aux_safe>
if "[] |- v hastype xty"  <v_hastype>
then "(TE #! xty) |- e hastype ty ∧
      len TE = n →
        TE |- (subst_aux n e v) hastype ty"
proof
  let "ihyp TE e ty =
        ∀n. len TE = n →
          TE |- (subst_aux n e v) hastype ty";
  cases by <hastype.induct> ["ihyp"], <goal>, ...
    case + "e = Lam dty bod"
         + "ty = dty --> rty"
         + "ihyp (dty#(TE#!xty)) bod rty" <ihyp>
         + "len TE = n"
         - "TE |- (subst_aux n e v) hastype ty" :
     ...

    case + "e = f % a"
         + "ihyp (TE#!xty) f (dty --> ty)" <ihyp1>
         + "ihyp (TE#!xty) a dty" <ihyp2> :
         + "len TE = n"
         - "TE |- (subst_aux n e v) hastype ty" :
     ...

  end;
end;
```

We have given one case for each rule of the induction relation where the proof is not simple: the other cases can be "consumed" in the decomposition by merging their justifications with that for the case split. For the non-trivial cases we have listed the

[7] In principle the automated engine might be able to find the higher order instantiation of the induction theorem, but this is, in general, difficult to rely upon.

available induction hypotheses in a systematic fashion. This approach is, in some ways, acceptable. Its advantages include:

- Flexibility: if any induction cases are simple then they may be omitted, leaving the automated checker to discharge the corresponding decomposition obligation. In the above example we could omit two of the cases. In addition, the cases may be presented in any order, since the automated engine will still be able to discharge the obligation.
- Control: we name the local constants introduced on each branch of the induction, and can tag induction facts with names and usage directives.
- Clarity: the logic of the decomposition is made explicit.

Its disadvantages are:

- Verbosity: not only do we have to quote all induction hypotheses for non-trivial cases, but we are obliged to restate

  ```
  + "len TE = n"
  - "TE |- (subst_aux n e v) hastype ty" :
  ```

  in each case also. This is because we had to state the original goal in quantified form, as TE, e and ty must all be universal if the decomposition obligation is to be provable. As with the programmed approach above, we do not have the opportunity to mark dependent facts with names and usage directives once and for all, and would have to repeat these on each branch of the proof. Furthermore, we must explicitly instantiate the induction theorem, and explicitly define the induction predicate.
- Complex Proof Obligations: for induction decompositions involving many cases, the decomposition obligation gets very large.
- Inaccuracy: it is fairly likely the user will make mistakes when recording induction hypotheses.
- Debugging: it is non-trivial to provide good feedback to the user if they make such a mistake.

Although sometimes the above form might be preferred, the majority of inductive arguments follow a very standard pattern of highly syntactic reasoning. With a dedicated induction construct we can improve the feedback provided to the user; eliminate a source of particularly complex proof obligations; and make our proofs far more succinct. We do, however, lose some flexibility, because an explicit proof must be given for each case of the induction. That is, the reasoning is syntactic and does not produce a proof obligation, and so the automated engine cannot be used to subsume trivial proof steps.

3.5.3 The Induction Construct in Declare

We now consider the corresponding Declare proof:

```
thm <subst_aux_safe>
if "[] |- v hastype xty"
   "len TE = n"
    "(TE #! xty) |- e hastype ty" <typing>
then "TE |- (subst_aux n e v) hastype ty";
proof
  proceed by rule induction on <typing> with n,TE,ty,e variable;
    case Con: ...
    case Var: ...
    case Lam
      "e = Lam dty bod"
      "ty = dty --> rty"
      "ihyp (dty#(TE#!xty)) bod rty"  <ihyp> :
      ...
    case App
      "e = f % a"
      "ihyp (TE#!xty) f (dty --> ty)" <f_ihyp>
      "ihyp (TE#!xty) a dty"          <a_ihyp> :
      ...
  end;
end;
```

We explain the details of the construct below, but the approach we have taken is clear: provide one very general construct for decomposing problems along syntactic lines based on second-order arguments.[8] The scope of the induction predicate is determined automatically by indicating those local constants (i.e. variables universally quantified at the outer of the current proof) which vary during the induction. The basic form of the construct is:

```
  proceed by schema on fact with constants variable
    case name_1
      facts_1 : proof_1;
    ...
    case name_n
      facts_n : proof_n;
  end;
```

where, as with the decomposition construct, each $facts_i$ has the form

```
      consider c_{i,1}, ..., c_{i,k_i} such that
         p_{i,1}
```

[8] The argument is second-order because it involves instantiating a theorem with a predicate.

$\ldots$

p_{i,m_i}

The *fact* must be an instance of the inductive set required by the given schema — it is negated (e.g. a goal) for co-inductive schemas (see below) and it may be replaced by a term if the inductive set is universal for some type (i.e. for structural induction).

The *schema* must be a fact in the logical environment of the form:

$$(\forall \vec{v}.\ ihyps_1 \rightarrow P\vec{v}) \ \wedge \ \ldots \ \wedge \ (\forall \vec{v}.\ ihyps_n \rightarrow P\vec{v}) \rightarrow (\forall \vec{v}.\ Q[\vec{v}] \rightarrow P\vec{v})$$

where the inductive set given in the support fact is an instance of $Q[\vec{v}]$ (this is the form schemas take in HOL and Isabelle, except equational constraints must always be encoded in the induction hypotheses). The schema fact must have a pragma giving names to the subgoals of the induction (see Table 3.2). The production of schemas is automated for inductive relations and datatypes, so the user rarely needs to know the form that schemas take internally. However, the general mechanism is provided to allow the declaration of the inductive structure of constructs that were not defined via these mechanisms, and to allow several proof principles to be declared for the same inductive set.

$Q[\vec{v}]$ takes the form $R(\vec{v})$ for an inductive relation R, and $\neg R(\vec{v})$ for a co-inductive relation. The condition $Q(\vec{v})$ is optional — without it Q is assumed to be universal. Induction over the natural numbers is thus written as:

$(\forall n.\ n{=}0 \rightarrow P\ n) \wedge$
$(\forall n.\ (\exists k.\ n{=}k{+}1 \wedge P\ k) \rightarrow P\ n)$
$\rightarrow (\forall n.\ P\ n)$

If the naturals are considered an inductive subset of the integers, then the schema is:

$(\forall i.\ i{=}0 \rightarrow P\ i) \wedge$
$(\forall i.\ (\exists k.\ i{=}k{+}1) \wedge P\ k) \rightarrow P\ i)$
$\rightarrow (\forall i.\ \texttt{is\_nat}(i) \rightarrow P\ i)$

We also admit forms where the schema is implicit from the induction fact: for inductive relations, the schema can be determined from the outermost construct, so "`rule induction on` *fact*," suffices, and similarly "`structural induction on` *term*" for inductive datatypes.

3.5.4 The Cases

Each antecedent of the inductive schema generates one new branch of the proof. At each branch of the proof the user must specify either a proof, or a set of purported hypotheses and a proof.

- If no purported hypotheses are given, then the actual hypotheses (i.e. those specified in the schema) are made available implicitly, but may not be referred to by name: they become "automatic" unlabelled facts.

- If purported hypotheses are given, then a syntactic check is made to ensure they correspond to the actual hypotheses. This check is quite liberal: both the purported and actual hypotheses are normalized with respect to various equations (including beta reduction, local constant elimination and NNF), and then must be equal up to alpha conversion. Thus the user gains control over the naming of introduced constants and facts, and may also simultaneously introduce local abbreviations: these may be convenient in the remainder of that branch of the proof. Hypotheses must be listed in the order they appear in the schema, but this is generally the most appropriate order in any case.

The semantics for the construct can be characterized as follows:

$$\frac{\begin{array}{l}\forall P.\ (\forall \vec{v}.ihyps_1 \to P(\vec{v})) \\ \quad \ldots \\ \quad (\forall \vec{v}.ihyps_n \to P(\vec{v})) \qquad \in_{ax} \Gamma \\ \quad \to (\forall \vec{v}.R(\vec{v}) \to P(\vec{v})) \\ R(\vec{t}) \in_{ax} \Gamma \\ P = \lambda \vec{v}.\forall \vec{V}.\ \bigwedge(\vec{v} = \vec{t}) \to \Gamma/\vec{V} \\ \bigwedge(\vec{v} = \vec{t}) \ \wedge\ ihyps_i \to \exists c_{i,1} \ldots c_{i,k_i}.p_{i,1} \ \wedge\ \ldots\ \wedge\ p_{i,m_i} \quad (\forall i.1 \le i \le n) \\ \Gamma \oplus_{sig} c_{1,1} \ldots c_{1,k_1} \oplus_{ax} p_{1,1}, \ldots, p_{1,m_1} \vdash \mathsf{F} \\ \ldots \\ \Gamma \oplus_{sig} c_{n,1} \ldots c_{n,k_n} \oplus_{ax} p_{n,1}, \ldots, p_{n,m_n} \vdash \mathsf{F}\end{array}}{\Gamma \vdash \mathsf{F}} \qquad (3.1)$$

The conditions specify that:

- The schema is indeed a fact in the current logical context;
- The inductive relation is satisfied for some terms $\vec{t}$;
- In the generated hypotheses, P is replaced by the induction hypotheses;
- The matching criteria: the generated hypotheses must (as a minimum) imply the purported hypotheses. In the generated hypotheses equational constraints between $\vec{v}$ and $\vec{t}$ are also available: these are always eliminated in the normalization that precedes matching.
- Each actual subcase must be provable.

Here $c_{1,1}, \ldots, c_{n,m_n}$ are the new local constants and the $p_{i,j}$ are the new purported hypotheses for each case. $\vec{V}$ is the variance specified in the induction step, and $\Gamma/\vec{V}$ represents the conjunction of all axioms in Γ involving any of the local constants in $\vec{V}$.[9]

[9]Note type constants may not vary: this could be supported if we admitted quantification over type variables [Mel92] in the underlying logic.

3.5.5 Strong Induction

Strong induction is a simple modification to the above mechanism where the induction predicate is automatically augmented with membership of the inductive set:

$$\begin{array}{l}\ldots\\ R(\vec{t}) \in_{ax} \Gamma\\ P = \lambda\vec{v}.\, \forall\vec{V}.\, (\bigwedge(\vec{v} = \vec{t})) \rightarrow \Gamma/\vec{V} \ \wedge\ R(\vec{v})\\ \ldots\end{array}$$

Strong induction is the default in the Declare proof language: weak induction must be specified using the keyword `weak` and is useful only when the added information is useless and confusing for the automated proof engine.

3.5.6 Co-induction and Strengthening

To illustrate the use of co-induction in the proof language, consider the definition of divergence for a transition relation:

```
gfp Divergent =
<Step>          ∃b. R a b ∧ Divergent R b
               ------------------------------
                    Divergent R a
```

Assume that $R = \rightsquigarrow$ and $W_1 \rightsquigarrow W_2$ and $W_2 \rightsquigarrow W_1$. If we want to prove that W_1 is divergent, we use co-induction over an appropriately strengthened goal:[10]

```
thm <example>
if  "W1 --> W2"
    "W2 --> W1"
then "Divergent(-->)(W1)" <g1>
proof
  // Strengthen the goal a little...
  consider x st
      + "x = W1 ∨ x = W2"
      - "Divergent(-->)(x)" <g2>
  // Co-induct and the rest is easy...
  proceed by rule induction on <g2> with x variable discarding <g1>
  case Step
     - "∃b. x --> b ∧ (b = W1 ∨ b = W2)";  // (could be left implict)
     qed;
  end
end
```

[10] Note we use co-induction to demonstrate membership of the set, and rules to prove non-membership. This is the opposite way around to the inductive case, as expected!

We explain the `discarding` construct below. Not surprisingly, the proof is very similar to an inductive proof, though we quote a goal rather than a fact as the support for the co-induction.

3.5.7 `ihyp` macros

Writing out induction hypotheses in detail can be informative, but also time-consuming and error-prone. Two mechanisms are available to help with this. First, the cases of the induction can be generated automatically by Declare, though typically the user still copies the hypotheses in order to record choices for new constants and names for facts. We consider this in detail in Chapter 5. Secondly, the shorthand `ihyp(...)` can be used within the scope of a `proceed by ...` construct as a macro for the implicit induction predicate. Without this mechanism our example would have been:

```
thm <subst_aux_safe>
...
proceed by weak rule induction on <typing> with n,TE,ty,e variable;
  case Con; qed by ...
  case Var; qed by ...
  case Lam
    "e = Lam dty bod"
    "ty = dty --> rty"
    "∀TE'. TE'#!xty = dty#(TE#!xty) ∧
           [] |- v hastype xty ∧
           len TE'  = n
           → (TE' #! xty) |- e hastype ty"  <ihyp>;
    qed by ...
  case App ...
end;
```

`ihyp` provides a robust and succinct mechanism for quoting induction conditions, at the risk of some obscurity. The expanded version could be recorded in the proof script, but with a little practice it is easy to syntactically predict the available hypotheses, just as in hand proofs. The successful use of `ihyp` clearly relies on the user having a strong intuitive understanding of induction (i.e. there is no substitute for mathematical training!) Note the interactive environment for Declare displays the induction predicate as it is generated, and also unwinds the use of `ihyp` when displaying formulae (i.e. `ihyp` is regarded as a macro rather than a local constant).

It may be possible to extend the labelling mechanism of Section 2.3 to enable labels on the top lines of rules (i.e. in inductive schemas) to be used to access elements of the inductive hypotheses without re-quoting the terms involved — we leave this for future research.

3.5.8 Discarding Facts

The coinductive example above demonstrates a common pattern in inductive proofs: the goal must be strengthened, or the assumptions weakened, before an induction is commenced. This is done in two steps: we prove that a stronger goal is sufficient (this is usually trivial), and before we perform an induction we purge the environment of all irrelevant facts, to avoid unnecessary antecedents being added to the induction hypothesis. Unless we force the user to resort to a lemma, this last step requires a "discarding" construct to be added to the proof language. Discarding facts destroys the monotonicity property of the proof language, so to minimize its use we have chosen to make it part of the induction construct. Our case studies suggest it is only required when significant reasoning is performed before the induction step of a proof, which is rare.

The semantics of the operator is trivial:

$$\frac{\Gamma \setminus ax \vdash \mathsf{F}}{\Gamma \vdash \mathsf{F}}$$

3.5.9 Mutually Recursive Inductive Proofs

The final twist on the schema-application construct comes when we consider *mutually recursive inductive proofs*. This occurs in operational semantics when, for example, we are proving facts about a functional language containing both expressions and declarations. We can use odd and even numbers as a prototypical instance of this problem, when characterized inductively by

$$\frac{}{\mathsf{even}(0)} \qquad \frac{\mathsf{even}(m)}{\mathsf{odd}(m+1)} \qquad \frac{\mathsf{odd}(m)}{\mathsf{even}(m+1)}$$

Typically we want to prove two facts "simultaneously" over the two inductive sets, using the induction theorem for the inductive relation

$$\begin{array}{l}
\forall P_{even}\ P_{odd}. \\
\quad (\forall n.\ n\texttt{=0} \rightarrow P_{even}\ n) \wedge \\
\quad (\forall n.\ (\exists m.\ n\texttt{=}m\texttt{+1} \wedge P_{even}\ m) \rightarrow P_{odd}\ n) \wedge \\
\quad (\forall n.\ (\exists m.\ n\texttt{=}m\texttt{+1} \wedge P_{odd}\ m) \rightarrow P_{even}\ n) \\
\quad \rightarrow (\forall n.\ \texttt{even(n)} \rightarrow P_{even}\ n) \wedge \\
\qquad (\forall n.\ \texttt{odd(n)} \rightarrow P_{odd}\ n)
\end{array}$$

For example, assume we are trying to show that if a number is odd, then it isn't even, and vice-versa. Unfortunately if we are to maintain the style of the proof language, it is inevitable that the proof system be adapted to accommodate multiple (conjoined) goals. In Declare we show that a number can't be both odd and even as follows:

```
thms
```

```
  <odd_implies_not_even> if "odd n" <a> then "¬even(n)"
  <even_implies_not_odd> if "even n" <b> then "¬odd(n)";
proof
  proceed by rule induction on <a>, <b> with n variable;
    case zero: ...
    case even: ...
    case odd: ...
  end;
end;
```

This is does not present any great logical problems, since logically sequents in higher order logic correspond to implicative formulae.[11] In a sense it just demonstrates how formal proof systems must adapt when being used to assign meaning to more declarative proof description styles. Formally, primitive judgments become

$$\Gamma_1 \vdash \mathsf{F}, \ldots, \Gamma_n \vdash \mathsf{F}$$

where n is the number of mutually recursive goals. Multiple goals are only useful for solving mutual recursion, and so the only proof rule we admit for the case $n > 1$ is schema application. Modifying the semantic rule 3.1 for this case is straightforward.

3.6 Related Work

3.6.1 Tactics

Tactics, first used in LCF[GMW77], are the traditional mechanism for proof description in LCF-style systems. In principle tactics simply decompose a problem and return a justification which proves that the decomposition is logically sound:

```
type tactic = sequent → sequent list × justification
type justification = thm list → thm
```

Isabelle tactics return not just one but a stream of possible decompositions and backtracking may be used over this search space.

In practice tactic collections embody an interactive style of proof that proceeds by syntactic manipulation of the sequent and existing top level theorems, and tactic proofs are often examples of arcane *adhoc* programming in the extreme. The advantage of tactic based proving is the programmability it affords, and common patterns of manipulation can in theory be automated. A major disadvantage is that the sequent quickly becomes unwieldy, and the style discourages the use of abbreviations and complex case decompositions.

We give one example from each of HOL and Isabelle. We make no attempt to explain the proofs, precisely because it is so just hard to know what's going on. The following is the proof of a lemma taken from Norrish's analysis of the semantics of C:

[11] Again, the question of where type variables are quantified must be considered: in this context they are considered global to the proof, not to each sequent.

```
val wf_type_offset = store_thm(
  "wf_type_offset",
  ``!smap sn. well_formed_type smap (Struct sn) ==>
              !fld t. lookup_field_info (smap sn) fld t ==>
                      ?n. offset smap sn fld n``,
  SIMP_TAC (hol_ss ++ impnorm_set) [offset,
    definition "choltype" "lookup_field_info",
    definition "choltype" "struct_info"] THEN
  REPEAT STRIP_TAC THEN
  IMP_RES_TAC (theorem "choltype" "well_formed_structs") THEN
  FULL_SIMP_TAC hol_ss [well_formed_type_THM] THEN
  FIRST_X_ASSUM SUBST_ALL_TAC THEN
  FULL_SIMP_TAC hol_ss [definition "choltype" "struct_info"] THEN
  POP_ASSUM_LIST (MAP_EVERY (fn th =>
    if (free_in ``nodup_flds`` (concl th) orelse
        free_in ``[]:(string # CType) list`` (concl th)) then
       ALL_TAC
    else MP_TAC th)) THEN
  SPEC_TAC (dub ``l:(string # CType) list``) THEN
  INDUCT_THEN list_INDUCT ASSUME_TAC THEN SIMP_TAC hol_ss THEN
  GEN_TAC THEN
  STRUCT_CASES_TAC (ISPEC ``h:string # CType`` pair_CASES) THEN
  SIMP_TAC hol_ss [DISJ_IMP_THM, Theorems.RIGHT_IMP_FORALL_THM,
                   FORALL_AND_THM] THEN
  ONCE_REWRITE_TAC [offset'_rewrites] THEN REPEAT STRIP_TAC THEN
  ASM_MESON_TAC [well_formed_type_sizeof]);
```

Even given all the appropriate definitions, we would challenge even an experienced HOL user to accurately predict the logical context at a given point late in the proof. Note how terms are quoted, but we don't know how they relate to the problem — where did "`l`" or "`h`" come from?

Isabelle proofs are usually substantially better, in the sense that they utilise fewer programmed proof procedures, and make less use of "assumption hacking" devices (e.g. `POP_ASSUM_LIST` above, which forces all assumptions through a function). For example, the proof of the correctness of the W type inference algorithm in Nipkow and Nazareth's formulation [NN96] begins:

```
(* correctness of W with respect to has_type *)
goal W.thy
        "!A S t m n . new_tv n A --> Some (S,t,m) = W e A n --> $S A |- e :: t";
by (expr.induct_tac "e" 1);
(* case Var n *)
by (asm_full_simp_tac (simpset() addsplits [expand_if]) 1);
by (strip_tac 1);
by (rtac has_type.VarI 1);
by (Simp_tac 1);
by (simp_tac (simpset() addsimps [is_bound_typ_instance]) 1);
by (rtac exI 1);
by (rtac refl 1);
(* case Abs e *)
by (asm_full_simp_tac (simpset() addsimps [app_subst_list]
                       addsplits [split_option_bind]) 1);
by (strip_tac 1);
by (eres_inst_tac [("x","(mk_scheme (TVar n)) # A")] allE 1);
```

...

and continues in the same vein for 200 lines. The same questions can be posed about this proof: what is the logical environment at each point? What is the result of the *adhoc* hacking on the sequent using simplification and resolution? What would happen if I stated the problem in a different (but logically equivalent) way? (e.g. using just one implication symbol).

While neither style is declarative, it is worth noting that for experienced users, both are effective for "getting the job done" (both verifications mentioned above are certainly impressive pieces of work.) Ultimately different proof styles may be applicable in different contexts, depending on the constraints of the project.

Some experienced users of tactic collections have successfully adopted a limited style of proof which allows long arguments to be expressed with some accuracy (e.g. see Harrison's construction of the real numbers in HOL-lite). These styles have not been systematized, and in often resemble aspects of our declarative proof language (e.g. Harrison is often careful to give sensible names when introducing local constants). In addition, Bailey [Bai98] has looked closely at the role of literate programming in supporting readable proofs for the LEGO [LP92] proof assistant, in the context of a major proof in algebra. He adopted a limited style of proof in places in order to maximize readability. His source texts are not themselves particularly readable — they first require extensive translation to LaTeX. However, the end result is certainly of high quality.

3.6.2 A short statistical comparison

Source level statistical analysis of different proof styles can give some indication of their differences between them. Table 3.3 presents statistics from three developments: the Java case study using Declare described in Chapters 6 and 7, a similar work by von Oheimb in Isabelle [Nv98] (see also Section 7.5.1), and Norrish's study of the operational semantics of C [Nor98]).

> *A caveat: The studies are substantially different and the statistics are only meant to give a rough impression of the nature of the style of proof and specification used!*

Controlled experiments are possible in such a domain, but require a significant resources. Aitken *et al.* [AGMT98] have used controlled quantitative experiments to investigate interaction (but not proof style) in HOL and PVS. Similar experiments investigating proof style would be interesting but are beyond the scope of this thesis. Truly controlled experiments would be very difficult, after all, these developments take years to construct (e.g. Norrish's C development took 3 man-years).

With these caveats in mind, we can turn to the figures. Certainly, for example, one can see that both Isabelle and Declare have solved a chronic problem in the HOL system: the need to add type annotations to terms quoted in proofs. This impediment alone is sufficient to deter most HOL users from a "declarative" style, because

	Java 1 (Declare)	Java 2 (Isabelle)	C (HOL)
Lines	7900	3000	16500
Text Size (Kb, no comments)	200	102	702
specification[a]	17%	32%	13%
theorem statements	20%	35%	27%
proofs	63%	33%	60%
outlines	26%		
justifications	36%		
Top level theorems[b]	93/280 [c]	207/830	700/700
Term quotations in proofs[d]	1050/3150	67/270	300/300
Adhoc proof procedures[e]	0/0	58/230	700/700
First order symbols[f]	260/780	700/2800	2200/2200
Proof steps	1100[g]/3300	670[h]/2700	7000[i]/7000
Explicit type annotations (%)[j]	$\sim 1\%$	$\sim 1\%$	$\sim 70\%$
Proof description devices [k]	~ 15	~ 60	700+

Table 3.3: Source Level Statistics for Three Operational Developments. See caveats in the text!

[a] The different components of the texts were separated using a combination of Unix tools and manual editing. For Declare, the split was between `thm` declarations, their proofs, and the remainder. For Isabelle, the specification included all `.thy` files, the proof statements all proofs up until the first proof command, and proofs were everything else. For HOL, 5 files were chosen at random from the 20 that make up the development, and manual editing was used to select the three types of text. For both Isabelle and HOL "background theories" such as those for partial functions were ignored. It is possible to quibble over what parts of the texts should be included in each category, thus these figures should only be taken as accurate to within a few percentage points.

[b] Includes top level lemmas and theorems.

[c] The first figure is the approximate total. The second figure is the first adjusted based on a rough estimate of the overall logical complexity of the development. This is clearly difficult to measure, so we accept that these must be taken with a grain of salt. My assessment is based on the total text size, but adjusting the Isabelle development because it uses extremely compact syntax (thus is more complex than the text size reveals, and it would be unfair to penalise it on this basis!). The factors we use are 3.0 for the Declare development, 4.0 for Isabelleand 1.0 for HOL (thus I have estimated Norrish's C development to be the most complex). This correlates with my personal estimate of the logical complexity, having viewed the specifications and proofs. Note that "complexity" is sometimes self-induced, e.g. some representation choices I made in the Java case study made things logically more difficult than they might have been, and thus complexity is not a direct measure of merit!

[d] Where two or more quotations appeared on the same line of text, they were counted as one.

[e] Approximate number of `val` or `fun` declarations that do not define top level theorems, or simple handles fetching top level theorems from a database. We have, perhaps, been overly generous toward Isabelle and HOL here — many more *adhoc* combinations of tactics and tactic-functionals are created mid-proof and are not bound to top level identifiers.

[f] Approximate number of explicit $\wedge$, $\vee$, implications, iterated $\exists$ or $\forall$ or iterated conjunction (`[|...|]` in Isabelle) symbols occurring within terms, apart from the specification. For HOL the figure includes the specification but is divided by 1.5 to adjust.

[g] Number of `let`, `have`, `consider`, `induct`, `qed` or `sts` steps.

[h] Number of `b y` or `K` steps, each representing a tactic application.

[i] Approximate number of `THEN`, `THENL` or `REPEAT` steps.

[j] Percentage of terms in proofs that have at least one explicit type annotation

[k] Approximate number of different proof description devices used, e.g. each different tactic counts as 1. Again we may have been over-generous — see the note on *adhoc* proof procedures above.

a declarative style will inevitably require more term quotation. We can see that this is indeed the case: Declare proof contain many more terms than corresponding Isabelle and HOL proofs.

The figures give some support for arguing that Declare proofs have better *locality*. That is, more lemmas are stated and proved in the middle of a proof, rather than being lifted to the top level. This is not surprising, as this is exactly the kind of reasoning Declare is designed to support.

Similarly, the figures support the view that HOL developments are massively overburdened with *adhoc* proof procedures, nearly all of which can be subsumed by techniques used in Declare and Isabelle. In addition, Declare developments are "simpler," if simplicity is measured by the number of proof devices and/or *adhoc* proof procedures used. However, the cost of a declarative style is also evident: Declare proofs contain many more term quotations than Isabelle and HOL proofs. This is precisely because terms are needed to "declare" the result of a step in a proof.

Finally, the figures do support the view that Declare proofs are relatively free of explicit use of first order (including propositional) symbols. Traditional written mathematics makes little use of first order symbols, e.g. I was not able to find any in either *General Topology* by Willard or *Calculus* by Spivak [Spi67, Wil70]. Instead, they prefer to use first order *terminology* (not symbols), usually in the "meta-language" surrounding the terms they are manipulating (i.e. in problem statements and proof outlines). Much the same thing happens in Declare: most first order symbols and manipulations are implicit in the statement and structure of a proof. It is only in the specification that they are widely used.

3.6.3 Mizar

Mizar is a well established system for formalizing general mathematics, and a phenomenal amount of the mathematical corpus has been covered.

Declare has been inspired in part by the Mizar system and Harrison's 'Mizar Mode' work [Har96b, Rud92]. In particular:

- The concept of specifying proofs as documents in a palatable proof language is due to Mizar. The actual proof language we use is substantially different.
- The realisation that declarative proof techniques could be used within a higher order logic based system is due to Harrison.
- The use of automated first order proof to discharge obligations arising from a declarative language comes from both Mizar and Harrison.

Significantly, the realisation that declarative techniques achieve many of the requirements of practical verification in the context of operational semantics (and, in general, for large, evolving models and specifications) is our own. Prior to this work it was commonly held that a declarative style would not work for "large" specifications, but only when specifications involved the small terms found in traditional mathematics.

Mizar is a poorly documented system, so the following comments are observations based on some sample Mizar scripts and the execution of the Mizar program. First, there are large differences between the Mizar and Declare languages:

- Declare supports constructs common in operational semantics directly, whereas Mizar supports constructs common in general abstract mathematics. For example, we provide induction and case analysis constructs suited for reasoning about inductive types and relations. In Mizar specifying and reasoning about these constructs is possible, but clumsy. These differences can, more or less, be explained by considering Declare as a system in a similar spirit to Mizar, but applied to a different domain.

- Once the concrete syntax is stripped away, Mizar proofs are mostly specifications of fine-grained syntactic manipulations, e.g. generalization, instantiation, and propositional introduction/elimination. We believe the decomposition construct of the Declare language enables the user to specify logical leaps in larger steps. For example, case splits in Mizar are usually small, and facts tend to get introduced one at a time.

- The logic underlying Mizar is rich in devices but quite complex and perhaps even *adhoc*. Many of its features are designed for abstract mathematics, and are of little relevance to practical verification. The key idea (proof outlining) can easily be transferred to a simpler setting and elaborated there,as we have done in Declare.

- Specifications in Mizar are highly constructive, and it usually takes a lot of text to get from an initial definition to the axioms that practically characterize the new construct.

The differences may also stem from the automated support provided: justifications in Mizar proofs rarely contain more than five facts, but in Declare we sometimes provide 10 or 15 (and even another 10 or 20 "automatic background" facts). Mizar provides little feedback when a step could not be justified, so perhaps it is difficult to accurately formulate logical steps that are much larger.

These points are illustrated in the following Mizar proof about an operator `idseq` that produces the list $0, 1, \ldots, n$ for a given n. In Declare or Isabelle the theorem (which is `"idseq (i+1) = (idseq i) ^ <*i+1*>"`, where `"^"` is concatenation and `"<*x*>"` is a singleton list) would either arise trivially from a recursive definition, or would be proven automatically by rewriting and arithmetic from a definition such as `idseq = mk_list (λi. i)` (which is roughly the definition used in Mizar). Although the Mizar proof comes late in the development of the theory of lists (after 2600 lines), the set theoretic constructions underlying the theory still rear their ugly heads. Note also how fine-grained the reasoning is.

```
theorem Th45: idseq (i+1) = (idseq i) ^ <*i+1*>
```

```
proof
       set p = idseq (i+1);
A1:    len p = i + 1 by Th42; then
       consider q being FinSequence , a being Any such that
A2:    p = q^<*a*> by Th21;
_278_: len p = len q + 1 by A2,Th20; then
A3:    len q = i by A1,REAL_1:10;
       i+1 ∈ Seg(i + 1) by FINSEQ_1:6;
       then p.(i+1) = i+1 by Th43; then
A4:    a = i+1 by A2,_278_,A1,FINSEQ_1:59;
A5:    dom q = Seg len q by FINSEQ_1:def 3;
       for a st a ∈ Seg i holds q.a = a
        proof let a; assume
B1:      a ∈ Seg i; then
          reconsider j = a as Nat;
          i < i+1 by NAT_1:29;
          then Seg i c= Seg (i+1) by FINSEQ_1:7;
          then j ∈ Seg(i+1) & p.j = q.j by B1,A2,A3,A5,Th18;
          hence thesis by Th43;
       end;
  then q = id Seg i by FUNCT_1:34,A3,A5;
  hence thesis by A2,A4,ID;
 end;
```

On the plus side, Mizar does run extremely quickly (much faster than Declare), and we must not forget that the system has been used to develop the most impressive corpus of fully formalized mathematics ever.

Chapter 4

Automated Reasoning for Declarative Proof

When writing a declarative proof, we leave gaps in the reasoning that we believe are "obvious", resulting in a proof obligation. We expect an *automated reasoning engine* to discharge these obligations.

Automated reasoning is the most fundamental technique available to eliminate procedural dependencies in proofs. Naturally we do not seek to solve the problem of automated reasoning once and for all. Rather we focus on the problem we are faced with: automated reasoning for declarative proof in the context of operational semantics. We first set the scene by outlining the functionality we require of the automated engine. We then describe the techniques that are used in Declare's automated engine, how they are integrated, and discuss how these do and don't meet our requirements. Few of these techniques are novel, rather the challenge is to draw on the wide range of techniques available in automated reasoning, and to compose them in a suitable fashion.

4.1 Requirements

Naturally, we require the engine to be *sound*: it should only discharge an obligation if a logically valid proof exists. We also require *relative completeness*: ideally we would like the engine to successfully discharge all obligations for which a proof exists, and to fail otherwise. Realistically, however, completeness will be relative to some class of problems. At the extreme, when developing proofs interactively it is normal to impose a time constraint whereby the prover must return a result within, say, 10 seconds, and so incompleteness is inevitable. Despite this, some notion of relative completeness is clearly desirable. If the problem lies outside this class, then the user must provide a more detailed outline of its proof.[1]

[1]The combination of the declarative proof language and the automated engine is complete (though perhaps tedious to use) if the automated engine at least implements all the basic infer-

Section 1.4.1 defines declarative proof as a relative absence of procedural detail or procedural dependencies. In Section 3.1 we expanded on this and explained how automation could help achieve this. For example, automation may effectively make $a < 2b$ and $2a \leq 4b - 1$ indistinguishable as far as the user is concerned (for $a, b \in \mathbb{Z}$). By doing so, we may have eliminated the procedural detail required to prove this to the machine, which could be an advantage. Thus, one requirement of the automated engine is to return *equivalent results on some classes of equivalent problems.* Consider a first order example: if the automated engine can prove obligations like

$$P \wedge (\forall p.\ Q_1(p) \wedge Q_2(p) \leftrightarrow R(p)) \rightarrow \exists c\ d.\ R(c, d)$$

(regardless of P, R, Q_1 and Q_2) then we expect it to be able to prove ones like

$$(\forall y_1 y_2.\ R(y_1, y_2) \leftrightarrow Q_2(y_1, y_2) \wedge Q_1(y_1, y_2)) \rightarrow (\exists p.\ R(p) \vee \neg P)$$

If it can, modifications that generate mildly different proof obligations will not break proofs.

Perhaps unexpectedly, failure of the automated proof engine is the norm, in the sense that when interactively developing complex proofs we spend most of our time on obligations that are "almost" provable. Thus we would like the prover to give us *excellent feedback* as to why obligations could not be discharged.

Ideally, declarative proof would be best served by *black box* automated reasoning, where the user does not have to understand the operation of the prover to any great depth. For example, propositional logic is decidable, and although NP-complete, "broad-spectrum" algorithms (e.g. BDD based) exist that give acceptable performance on most problems that arise in our domain, and furthermore counter examples can be generated that can be interpreted without a knowledge of the algorithm.

> *The essential challenge of automated reasoning in the declarative setting is to come up with a single general purpose prover that is sound, complete for some classes of problems, produces equivalent results on some classes of equivalent problems, reliable, simple to understand, simple to use and provides good feedback, yet still works efficiently on a sufficiently large problem domain.*

This is, needless to say, an extremely difficult task!

Unfortunately most existing work in automating non-propositional reasoning has produced provers that are far from "black box". To take one example, the first order prover Otter [MW97] has over 100 different switches and endless potential configurations and, although each has its purpose, there has been no real attempt to characterize which switches are appropriate for which classes of problems, making the

---

ence rules of higher order logic e.g. the eight primitive rules of the HOL system, in the sense that any proof that can be carried out in the HOL deductive system can be carried out in the combined system.

use of such provers somewhat of a black art.[2] Proof techniques that require arcane switch settings (e.g. weightings) within justifications negate the advantages we have achieved by using declarative proof, e.g. readability of proofs and their robustness under changes to the automated prover.

MacAllester has considered the question of "obviousness" with regard to automated deduction, and he has implemented some of his ideas in the Ontic system [McA89]. We have not tried to develop "obviousness" as an absolute concept, and are really more concerned with an automated prover that allows us to specify proof outlines that are, in some limited sense, "natural". In particular, it seems "unnatural" to specify proofs that involve manipulations of propositional or first order connectives, or tedious equality reasoning, or tedious arithmetic steps, and so our proof techniques focus on automating these domains.

4.1.1 An Example Problem

Figure 4.1 shows a typical proof obligation that arises within the context of operational semantics and which is amenable to automated reasoning. The details of the problem need not concern us (in particular it is not necessary to understand the meaning of the constructs involved, since we have listed all relevant axioms here), but the *style* of the problem does. We are trying to prove a property `heap_conforms` about `heap1` and `ht1`. These objects are modified versions of `heap0` and `ht0`: we have allocated a new location in `heap0`, and adjusted `ht0` to `ht1` to compensate. The structural modifications boil down to operations on finite partial functions (tables). We know that `heap_conforms` holds for `heap0` and `ht0`. To prove the goal we must prove the domains of `heap1` and `ht1` are equal, and that `heapobj_conforms` holds for every object `heap1`. The latter step is the harder: this requires the use of a monotonicity result for `heapobj_conforms` and a case analysis between whether the heapobj is a newly allocated object, or if it was an object already present.

We have shown only the relevant axioms and definitions here, though in general the prover must also perform adequately in the presence of irrelevant information. Most of the axioms were selected by the user as part of the justification line in the declarative proof script, however some (marked `!!`) are dragged in automatically by reasoning tools from libraries. The automated reasoning engine is essentially free to make use of any facts available in the current logical context, though it may require guidance as to how different facts may be used (we discuss this later in this chapter).

The problem is shown approximately in first-order form, though some features might need to be translated away before the problem would be acceptable to a first order prover, in particular polymorphic equality; the use of conditionals at the level of terms; and the `let ... = ... in ...` construct.

Although the problem shown is not large by the standards of some first order automated reasoning tools, it is, perhaps surprisingly, at the upper end of the size of

[2] Recently, an "autonomous" mode has been added to Otter for the purposes of the CADE provers competition [SS97]. Clearly it would be desirable to harness the work that has gone into such provers within practical verification systems, and we consider this a good avenue for future research.

```
 // The goal:
goal "heap_conforms(te,heap1,ht1)"

 // Datatypes:
datatype 'a option = None | Some('a);
datatype vt = VT(simp,nat);
datatype simp = A(string) | B(string) | C;
datatype heapobj = Object(...) | Array(...)

 // Facts about local constants:
defn "ht1 = fupdate(ht0,addr1,VT(C(c),0))";
defn "heapobj = Object(fldvals,c)"
defn "(heap1,addr1) = alloc(heap0,heapobj)"

fact "ht_leq (te,ht0,ht1)"
fact "heap_conforms (te,heap1,ht1)"
fact "heapobj_conforms(te,ht0,heapobj,VT(C(c),0)";

 // From the specification of the operational system:
defn "∀heap heapobj.
      alloc(heap,heapobj) =
        let addr = fresh(fdomain(heap))
        in (fupdate(heap,addr,heapobj),addr)";

defn "∀te heap ht.
     heap_conforms (te,heap,ht) ↔
      (fdomain heap = fdomain ht)  ∧
      (∀x y. flookup(heap)(x) = Some(y) →
             (∃z. flookup(ht)(x) = Some(z)  ∧  heapobj_conforms (te,ht,z,y)))"

fact "∀heapobj te ht0 ht1.
        ht_leq(te,ht0,ht1) ∧
        heapobj_conforms (te,ht0,heapobj,ty) →
        heapobj_conforms (te,ht1,heapobj,ty)";

 // From the theory of finite sets:
rewrite "∀fset1 fset2. (fset1 = fset2) ↔ (∀x. x ∈ fset1 ↔ x ∈ fset2)"

rewrite "∀fset x y. x ∈ finsert(y,fset) ↔ (x = y) ∨ x ∈ fset";  !!

 // From the theory of finite partial functions:
rewrite "∀f x y z. flookup(fupdate(f,x,y),z) =
           if (x = z) then y else flookup(f,z)";    !!
rewrite "∀f x y. fdomain(fupdate(f,x,y)) = finsert(x,fdomain(x))";      !!
fact "∀fset. ~(fresh(fset) ∈ fset)";
```

Figure 4.1: A typical obligation to be discharged by automated reasoning. The names `defn`, `fact` and `rewrite` refer to different categorizations of the available axioms given by pragmas.

problems that tend to occur in practice in our domain. This is because it is difficult to accurately formulate proof steps that embody larger logical leaps, at least when working on developing problems. Attempting to do so typically results in little payoff, and it tends to be quicker to simply split the proof into two or three steps rather than try to force things too far with the automated prover.

Now, consider the characteristics of this obligation:

- The problem involves a mixture of structural and logical reasoning, i.e. equational reasoning about constants and functions, and first order reasoning about various predicates.
- The structural reasoning involves a significant amount of fairly naive equational reasoning, best attacked by some kind of *rewriting*: definitions must be unfolded, and obvious reduction must be made.
- A degree of first order reasoning is clearly required: we must search for the key instantiations of facts such as the monotonicity lemma.
- Some reasoning about the datatypes (pairs, `option` and `vt`) is required, e.g. $\vdash$ `Some(`x`) = Some(`y`)` $\rightarrow$ x `=` y.
- If the complexity of the problem is to be controlled, then several functions and predicates must be treated as uninterpreted, e.g. unwinding the definition (not shown here) of `heapobj_conforms_to` substantially complicates the first order search.
- Similarly, some types are better treated as uninterpreted, e.g. we should not speculatively case split on objects of type `heapobj`, since the structure of these objects is irrelevant to the proof.

The obligation is atypical in the following ways:

- The predicates or functions are not recursively defined: typically some are.
- Once definitions have been expanded, the first order component of the problem is essentially in Horn-clause form: sometimes this is not the case.
- Each first order formula need only be instantiated once, presuming the proof search is organised well. Sometimes a formula needs 2 or 3 different instantiations, though rarely more.
- No reasoning about arithmetic is required. Often small arithmetic reasoning steps are required, e.g. proving `1 <= n` in a logical context where `n > 0` has been asserted.

Nearly all problems in our case studies (and, perhaps, in the majority of applications of theorem proving) require a mix of structural and first order reasoning. This has long been recognised, though it is rarely made explicit: here we have just tried to make this clear by an example, and to motivate the choices made in Declare.

4.2 Techniques used in Declare

We now describe the techniques used for automated reasoning in Declare, and the method by which they are integrated to form a single prover. We have tried to restrict ourselves to techniques that are *predictable*, *complete for certain classes of problems*, *simple*, *incremental* and which *offer good feedback for unsolvable problems.* However, for general first order problems good feedback will always be difficult, especially when the search space is wide.

Some notes on the implementation: the reasoning tools in Declare use databases derived from and stored alongside the current *logical environment* (that is, the collection of all available facts). Facts are pushed into these databases incrementally as they become known, e.g. by case splitting or by producing simplified versions of formulae. The logical environment and databases are stored as applicative data structures (i.e. with no references), so backtracking is trivial to implement.

Each technique described here can be made more powerful and more efficient, at the expense of greater complexity in behaviour, interface and implementation. The Declare implementation is structured so it is relatively easy to replace a reasoning component with one that uses more efficient data structures, by replacing the corresponding database in the logical environment. Databases can be computed on demand (lazily), to prevent their creation in instances where they are not used.

4.2.1 Ground Reasoning in Equational Theories

Decision procedures exist to determine the validity of formulae within various *ground equational theories.* Some basic decidable theories are:

- Propositional logic, i.e. first order formulae containing only propositional connectives and universal quantifiers at the outermost level, and finite types;
- Linear arithmetic, i.e. propositional logic extended with linear formulae over a real closed field (e.g. $\mathbb{R}$, using only $<$, $>$, $\leq$, $\geq$, $=$, $<>$, $+$, $-$);[3]
- Equational logic in the presence of uninterpreted function symbols, e.g. $a = g(b) \longrightarrow f(a, g(b)) = f(a, a)$, normally implemented by a congruence closure algorithm [NO80];

Frameworks exist for combining decision procedures for various theories. Nelson and Oppen have a quite general scheme that has been successfully re-implemented in theorem proving systems [NO79, Bou95]. Shostak has an alternative scheme that is less general but reputedly faster, and this is used in the STeP and PVS theorem provers [Sho84, MBBC95, COR+95].

[3] This subset (Presburger arithmetic) is usually expanded to include rational constants and functions that can be encoded in a linear/propositional framework, e.g. *abs*, *max*, *min* etc. An incomplete but effective procedure for $\mathbb{Z}$ and $\mathbb{N}$ can be achieved by translating to a more general problem over $\mathbb{R}$.

Declare implements a Shostak-style integration of decision procedures for uninterpreted equality and arithmetic. The implementation is very naive (for example no term graph is used, but rather we explicitly substitute), but sufficient for our case studies. The central database is a sequence of convex sub-databases [NO80], each corresponding to one case of the disjunctive normal form of the propositional structure of the logical environment.

Each convex database supports `assert` and `satisfy` functions. The former is used to add available ground equalities, inequalities and propositional formulae. The function `satisfy` is used to generate a satisfying instance, i.e. an assignment that satisfies the various constraints. This can in turn be used as a counter example in the context of refutation.

We discuss how we integrate the use of ground decision procedures with other techniques below.

4.2.2 Rewriting

Rewriting is the process of repeatedly transforming a term by replacing subterms with equivalent subterms, as justified by a set of equational rules. For example given the axioms $a = 1$ and $\forall x\ y\ z\ .\ x + (y + z) = (x + y) + z$ we may rewrite as follows:

$$(w + x) + (a + z) \leadsto ((w + x) + a) + z \leadsto ((w + x) + 1) + z$$

The axioms available (the *rewrite set*) and the order and locality of their application (the *rewrite strategy*) together form the *rewrite system*. Rewrite systems are often used for both systematic and *adhoc* proof in mechanized reasoning. Properties we look for in rewrite systems are *termination* (by ensuring that each rewrite axiom reduces some global metric); confluence (that is, the order of application of rewrites should not influence the final result); normalization (does the rewrite system reduce terms to a normal form?); and completeness (does the rewrite systems fully solve a class of problems?) An excellent introduction to the theory of rewriting can be found in [BN98] and implementations in HOL and Isabelle are documented in [Bou92] and [Nip89]. Some typical enhancements to basic rewriting described above are:

- Conditional rewriting, e.g. $i > 0 \wedge i \leq \mathit{length}(t) \longrightarrow \mathit{el}(i)(h :: t) \leadsto \mathit{el}(i-1)(t)$ perhaps using decision procedures to solve conditions.
- 2nd order and higher order matching. Equational axioms like $a = 1$ are generally interpreted as left-to-right rewrites, the left being the *pattern*. Patterns that contain free higher order variables can be interpreted as specifying families of rewrites, e.g. $(\neg\forall x.\ P\ x) \leadsto (\exists x.\ \neg P\ x)$. Other matching enhancements are also possible, though those guaranteed to produce at most one match are preferred.
- Contextual rewriting, e.g. the fact P is added to the logical environment when rewriting Q in $P \rightarrow Q$. Second order *congruence rules* may be specified for derived constructs, as in the Isabelle theorem prover (see Table 4.1).

- Infinite sets of rewrite axioms, provided by programmed procedures.[4]

Rewriting in Declare implements all of the above features. As Declare is not directly programmable the only rewriting procedures used are built in ones, which are described below. In a fully developed system it would be appropriate to support user programmable rewriting procedures, as many useful rewriting strategies can only be specified with an infinite number of axioms.

From Facts to Rewrites

Rewrites are specified in Declare by pragmas, usually when a fact is declared — this is discussed in the next section. Many facts (those not specified as left-to-right rewrites, including contextual assumptions generated by congruence rules) are used as "safe" rewrites of the form $proposition \leadsto true$. This ensures that rewriting always terminates, presuming the user has specified other pragmas sensibly.

Like most theorem provers, Declare comes with theories of important constructs such as partial functions, sets, finite partial functions, finite sets, first order logic and lists. Rewriting gives effective (though incomplete) proof procedures in many of these domains .

Declare does not implement Knuth-Bendix completion [KB70] on its rewrite set. It would be desirable to investigate the costs and benefits of this routine in the context of this problem domain, since occasionally the user must artificially modify the statement of theorems and proofs to ensure a confluent and complete rewrite strategy. For example, the user must sometimes ensure that all left-hand-side patterns of rewrites are in normal form: completion could alleviate such problems, and might further increase the declarative nature of proofs. However, most problematic examples involve ground terms, and perhaps simply further integrating ground decision procedures with rewriting (thus using congruence closure as a form of ground completion) would be sufficient. Also, full Knuth-Bendix completion requires the specification of a lexicographic term ordering. This is clearly non-declarative (in the sense of Section 1.4.1) but perhaps a sufficiently general default ordering could be specified.

4.2.3 Inbuilt Rewrite Procedures

Generalized Beta Conversion

Simple beta-conversion $(\lambda x.t)s \leadsto t[s/x]$ can be generalized to a procedure that can evaluate most pattern matches against ground values, as in functional programming languages[5], e.g.

[4] This technique was used by the author in his implementation of the `hol90` rewriter system, and has been adopted in other systems.

[5] This can in turn be generalized whenever patterns are specified by injective functions. Declare currently supports the former but not the latter, though there is no real reason (except implementation complexity) not to support both in the context of a theorem proving environment.

```
    match (Some(3),0) with (Some(x),0) -> t
 ⇝ t[3/x]
```

Matches can also be resolved in the negative:

```
    match Some(u) with None -> t | x -> s
 ⇝ match Some(u) with x -> s
 ⇝ s[Some(u)/x]
```

Generalized beta-conversion can, of course, only resolve matches in certain circumstances, e.g. when both patterns and arguments are specified by concrete constructions (datatype constructors, $\mathbb{Z}$, $\mathbb{N}$, and strings).

Resolving Matches by Throwing Side Conditions

Generalized beta-conversion may not be sufficient to make use of all the facts known in the current logical environment. For example, the reduction

```
    match a with 0 -> 0 | x -> x+1
 ⇝ a+1
```

is logically valid when $a \in \mathbb{N}$ and $a > 0$ is available in the logical context, because in these circumstances we know the first rule does not apply. One solution to such a problem is to throw off a side condition which can be solved by other, cooperating tools, in particular the ground decision procedures.

We attempt to resolve matches in the positive whenever a ground pattern is used, that is for an expression `match t with` p `->` ... where p contains no variables we generate the condition $p =$ `t`. We always attempt to resolve them in the negative presuming no other resolution is possible, so the condition for p containing variables $\vec{v}$ is $\forall \vec{v}.p \neq t$, presuming $\vec{v}$ are fresh names.

Solving for Unknowns

Declare incorporates quite powerful procedures for solving for unknowns in the middle of rewriting. Solving eliminates universal or existential quantifiers when a definite value can be determined for the quantified variable. For example:

```
    ∃a. a=b+c ∧ p(a,b) ∧ q(b,a+2)
 ⇝  p(b+c,b) ∧ q(b,(b+c)+2)
```

Some of this effect can be achieved by higher order rewriting with theorems such as

$$(\exists a.\ a{=}t \wedge P\ a) \equiv P\ t$$

but this technique is not sufficiently general when quantifiers appear in the wrong order or location. For example, the Declare automatic routine solves for a in the following situation:

$$\begin{array}{l} \quad \exists a\ b.\ (\mathtt{p}(a) \wedge a{=}t \wedge \mathtt{q}(b)) \vee \mathtt{r}(b) \\ \rightsquigarrow\ \exists b.\ (\mathtt{p}(t) \wedge \mathtt{q}(b)) \vee \mathtt{r}(b) \end{array}$$

because a does not occur on the right of the disjunct. The routine also solves for $\forall$-bound quantifiers:

$$\begin{array}{l} \quad \forall a.\ a{=}\mathtt{b{+}c} \wedge \mathtt{p}(a,\mathtt{b}) \rightarrow \mathtt{q}(\mathtt{b},a{+}\mathtt{2}) \\ \rightsquigarrow\ \ \mathtt{p(b{+}c,b)} \rightarrow \mathtt{q(b,(b{+}c){+}2)} \end{array}$$

In principle the routine could be extended to solve arithmetic equations and other equational theories, but this has not yet been done.

Implementation Issues

Implementations of rewriting systems can vary greatly in efficiency and complexity. Important issues to consider include:

- Dynamic v. Static? Can new rewrites and congruence rules be specified as part of the input? Can rewrites arise dynamically, e.g. from contextual assumptions?
- Compiled? Are rewrite sets compiled to some more efficient representation?
- Term or graph based? Graph-based rewriting algorithms can lead to far better time and space complexity, at the expense of greater implementation complexity, especially in the implementation of backtracking.

The Declare system is dynamic, uses minimal compilation in the form of term-nets and for simplicity is implemented based on terms (i.e. the implementation is along the same lines as rewriting in LCF, HOL-lite and Isabelle).

4.2.4 Grinding

Grinding (the terminology is borrowed from PVS) is essentially the repeated application of rewriting, "safe" first order and splitting steps until no goals remain or no further progress can be made. Grinding operates on a sequent (i.e. a list of conjoined facts and disjoined goals) and results in several residue sequents each of which must be solved by other techniques. The generation of an initial sequent is described in the discussion of integration issues below: essentially it is made up of a set of facts that have been selected as "primary." (In Declare this is done by quoting them on a justification line in the proof language). Grinding in Declare is fully contextual, in the sense that when a fact is being reduced, all surrounding facts (and the negation of all goals) are pushed into the logical context.[6]

[6] The facts may already appear in the context, but will normally have different pragmas, as discussed in the next section. In addition, their pragmas are maintained even though they dynamically change during rewriting, which is a simple way to cross-normalise rewrites.

Declare uses two-way, repeated grinding, in the sense that we iterate back and forth across the fact list looking for reductions, and all surrounding facts are available for use. Both the Isabelle and HOL simplifiers start at one end of the assumption list and only iterate in one direction. There is no particularly good reason for this restriction, and it can make some proofs fail (e.g. `(b*a)/a = c` $\wedge$ `a > 0` $\rightarrow$ `b = c`, where the side condition `a <> 0` to a cancellation rewrite is only provable if `a > 0` is available.)

Safe Steps

Safe first order steps include the introduction of witnesses for $\exists$ ($\forall$) quantifiers in facts (goals), splitting conjuncts (disjuncts) in facts (goals). Grinding also eliminates local constants defined by an equality, so if `a =` t is a fact then `a` can be eliminated in favour of the term t. The set of safe rules could be made extensible by using methods from the Isabelle theorem prover [Pau90].

Pattern Based Splitting and Weakening

Splitting follows fairly conventional lines, splitting on disjunctive formulae as in PVS, Isabelle and indeed most automated provers. Additional pattern based splitting rules may be specified, for example:

$$\vdash \texttt{(}b \rightarrow P\texttt{(}t\texttt{))} \wedge \texttt{(}\neg\ b \rightarrow P\texttt{(}e\texttt{))} \rightarrow P\ \texttt{(if}\ b\ \texttt{then}\ t\ \texttt{else}\ e\texttt{)}$$

Such a rule is interpreted by a procedure that searches for a free subterm that matches the pattern in the conclusion. New subgoals are then produced from the appropriately instantiated antecedents of the splitting theorem. The code is an improved version of a similar procedure found in Harrison's HOL-lite, in particular the theorem does not have to be an equality, which allows us to automatically "weaken" the sequent in cases where a certain side condition should always be provable.

For example, subtraction over $\mathbb{N}$ is often problematic in theorem provers: how should subtraction be defined outside the standard range? Our methodology is to avoid relying on the behaviour of subtraction outside its domain (indeed we do not even specify it in the definition of subtraction). Thus we require that the appropriate bound is always provable in the context in which subtraction is used. We can use pattern based weakening to generate this obligation and eliminate uses of subtraction in favour of an addition over a fresh "difference" variable. The weakening rule is:

$$\vdash a \texttt{ >= } b \wedge \texttt{(}\forall d.\ a\texttt{=}b\texttt{+}d \rightarrow P\texttt{(}d\texttt{))} \rightarrow P\ \texttt{(}a\texttt{-}b\texttt{)}$$

The side condition a `>=` b can be regarded as a condition arising out of an implicit dependent typing scheme for the subtraction operator.

Extensible splitting of a similar kind is available in the Isabelle simplifier, though it is not clear if it has been utilised to eliminate dependently typed operators as above.

4.2.5 First Order Reasoning

First order reasoning has been the primary problem of interest in automated reasoning communities. Although we work in higher order logic, the vast majority of proof obligations lie within equational first-order logic. The subset is semi-decidable, but almost invariably requires heuristic proof search.

First order reasoning has been largely neglected in interactive higher order logic based theorem provers (e.g. PVS has no support for unification and HOL existed for years without it), with Isabelle being the major exception. In his summary of first order proof in practice [Har97a] , Harrison describes the situation of interactive theorem proving with respect to first order techniques as follows:

> There is a trend away from monolithic automated theorem provers towards using automation as a tool in support of interactive proof. ... It raises a number of issues that are often neglected... Is first order automation actually useful, and if so, why? How can it be used for richer logics? What are the characteristic examples that require solution in practice? How do the traditional algorithms perform on these 'practical' examples — are they deficient or are they already too powerful?

First order systems attempt to find a contradiction (refutation) given a set of axioms. Routines often assume the axioms are in some normal form, e.g. clauses and/or prenex. The main task of algorithm is to find necessary instantiations (using unification) and to organise the search for these. Combining first order proof with equational reasoning is particularly challenging: although equality may be axiomatized, this is not terribly effective, and special heuristic rules for equality are often used.

The first order technique we use is *model elimination* [Lov68], which is essentially the natural completion of Prolog as a proof technique when negation-as-failure is excluded. The Horn clause restriction is also lifted by using the "contrapositives" of a set of formulae as the rule set. Model elimination is a simple and effective way to perform goal directed search, and as Harrison has reported [Har97a] in some cases it can even work effectively when the equality axioms are used directly.

Because we require quick feedback, and only use first order proof as a work horse to find relatively simple instantiations, we time-limit the proof search (which is based on iterative deepening) to 6 seconds in the interactive environment. This can, naturally, be specified by the user.

What feedback can be provided by the first order engine when the problem is not solvable? This is a very difficult issue: first order search spaces are large and it is very hard to distinguish promising paths from unpromising. It may be possible to employ some model generation procedure to give a counter example, but the only simple solution appears to allow the user to inspect the internal actions of the prover. Declare provides a trace of the search, though better would be an interactive method to examine paths in the search, like that provided by Isabelle.

4.3 Interface and Integration

4.3.1 Quoting and Pragmas

In the previous chapter we delayed discussing certain aspects of the proof language, because their semantics are interpreted by the automated proof engine. These constructs effectively form the interface to the automated engine. We are now in a position to complete these details.

The first question is the semantics of "quoting a theorem in a justification", or, equivalently, leaving a local fact inside a proof unnamed (these are implicitly included in all future justifications).

The second is related: we must describe the pragmas ("hints") that the automated engine understands. Table 4.1 defines the relevant pragmas and defines their meaning in terms of the proof procedures from the previous section.

Now, quoting a fact has the following effects. Most importantly, the fact is added to a set of "primary" facts that will form the initial sequent for grinding (see below). Before this is done, the pragmas of the fact may be slightly modified:

- If the fact has a non-`auto` pragma such as `defn` or `rw`, then this is promoted to the corresponding `auto`-pragma. The fact will be added to the appropriate databases during grinding. Thus quotation means "use it like I said it could be used."

- If the fact already had an `auto` pragma, the pragma is stripped from the copy of the fact that is added to the "primary" set (the fact remains in the automatic database). The assumption is that quoting the fact means the user is providing it for some special purpose (e.g. is instantiating it). Thus the quotation means "in addition to using it like I said, use it as an ordinary primary fact."

This combination has been sufficient for the case studies, and in combination with local pragmas allows any combination of pragmas to be specified. However, note that once facts are placed in a database using an `auto` pragma they may not be removed.

All facts implicitly have the `saferw` and `meson` pragmas, so quoting any fact promotes these to `auto`, and thus all quoted facts get used as safe rewrites and for first order proof.

4.3.2 Integration

The Declare automated prover uses grinding as the initial phase of the proof, before calling the decision procedures and model elimination. The starting sequent is the set of "primary" facts as defined above. This is thoroughly reduced and then model elimination is applied on the residue sequents.

Measures must be taken to ensure the use of rewriting is not problematic: without care rewriting can turn a problem otherwise be solvable by first order reasoning into one that isn't. Two typical problems arise:

`pragma defn` *thm* `pragma autodefn` *thm*	The fact specifies a set of (possibly recursive) definitions that should be used as left-to-right rewrites. Recursive definitions will be acceptable because rewriting does not occur inside λ terms, the branches of conditionals or pattern matches. The definitions may, in principle, be used by other proof procedures such as congruence closure or first order provers, but this does not occur in the current implementation. `autodefn` adds the definitions to a database of automatically applied definitions.
`pragma elim` *thm* `pragma autoelim` *thm*	The fact specifies a set of non-recursive equations that completely eliminate constants in favour of their representations. Again these may, in principle, be used by other proof procedures. `autoelim` adds the equations to an automatic database.
`pragma rw` *thm* `pragma autorw` *thm*	The fact specifies a set of (conditional) equations that should be used as a left-to-right (conditional) rewrite rules. `autorw` adds these to the database of automatic rewrite rules.
`pragma saferw` *thm* `pragma autosaferw` *thm*	The fact specifies a set of safe "boolean" rewrites (see Section 4.2.2). All facts are implicitly tagged with `saferw`.
`pragma cong` *thm* `pragma autocong` *thm*	The fact specifies a *congruence rule* in the style of the Isabelle simplifier (see Section 4.2.2), e.g. $\vdash P{=}P' \wedge (P' \rightarrow Q{=}Q') \rightarrow (P \wedge Q) = (P' \wedge Q')$
`pragma split` *thm* `pragma autosplit` *thm*	The fact specifies a *pattern based splitting rule* (see Section 4.2.4).
`pragma ground` *thm* `pragma autoground` *thm*	The fact specifies a ground fact and can be added to the ground equational database. All facts without outermost universal quantifiers are implicitly tagged with `ground`.
`pragma meson` *thm* `pragma automeson` *thm*	The fact specifies a set of first order reasoning rules, to be used by the model elimination procedure. All facts are implicitly tagged with `meson`.

Table 4.1: Pragmas recognised by the automated reasoning engine

- Rewriting can simplify away an instantiation of a fact that has been especially provided as a "hint" to help the first order prover.

  This is prevented by not applying rules that rewrite to "true"/"false" when in a positive/negative logical polarity.[7] This is rather *adhoc*, but it works well enough.

- Rewriting normalizes expressions so that unification is no longer possible, e.g. $c+b$ might be AC-normalized to $b+c$ which no longer unifies with $c+x$ without increasing the power of the unification algorithm.

  We treat this problem by avoiding automatic rewrites that disturb the structure of terms in this way.

The lesson is that when using rewriting as a preprocessor, the rewrite system must "respect" the behaviour of other automated routines.

We use the ground decision procedures to:

- Attempt to decide side conditions to conditional rewrites, after recursively grinding the condition, as in the Boyer-Moore prover [BM81];

- Attempt to decide the problem itself, again after grinding, but before model elimination.

Other theorem provers do better: ideally, asserting an equality between ground terms into should make those terms indistinguishable for nearly all purposes (this appears to be an unstated aim of the PVS prover). For example, the unification algorithm should be able to unify `f(a,`x`)` and `f(g(b),c)` when the equality `a=g(b)` is present in the logical environment (i.e. ground E-unification). We do not take things so far, though instances did arise in our cases studies where this would have led to shorter proofs.

4.3.3 Feedback

The following feedback is available from a failed proof attempt:

- The sequent as it appeared before grinding began.

- The sequent of the first case not solved by grinding, decision procedures or model elimination. The sequent is shown as it appears after grinding.

- Constraints that indicate how eliminated constants relate to constants in the sequents, e.g. `"p = (p.1, p.2)"`.

[7] As in Section 2.2.2, the logical polarity is positive/negative when reducing a redex within the propositional structure of a fact that is effectively under an even/odd number of negations. So we are ensuring, in limited circumstances, that rewriting progresses in the correct direction through the boolean lattice.

- A counter example for the unsolvable case as generated by the ground decision procedures.
- A list of the unsolved side conditions to rewrite rules, along with a counter example for each.

In the interactive environment, much of this information is elided (in particular the counter examples) and computed on demand, so the information is presented quickly and compactly. Also, tracing may be applied to show the internal actions of grinding and the model elimination prover.

4.4 Appraisal

The last three sections have described the requirements for an automated prover in our problem domain, and the actual prover we have used in our case studies. This begs the question: does the prover meet the requirements? To recap, the requirements were:

- Relative completeness;
- Equivalent results on equivalent problems;
- Simple to understand;
- Simple to use;
- One top level prover;
- Excellent feedback;
- Works efficiently on a sufficiently large problem domain.

Certainly our prover provides a degree of relative completeness: one could identify many sets of problems that it will accurately and consistently check (e.g. propositional logic, ground arithmetic, first order problems that require no more than 5-10 instantiations).

The second requirement is harder to meet: there are a substantial number of "equivalent forms" for solvable problems that will not be solved by our prover. However, different but equivalent forms of

- Propositional structure;
- Ground terms in decidable theories;
- Local solvable constants (i.e. $\exists x.\ x = t \wedge P[x]$ v. $P[t]$);
- First order structure (e.g. $\forall xy.\ Px \wedge Qy$ v. $(\forall x.\ Px) \wedge (\forall y.\ Qy)$

- Product-based structure (e.g. $Q(a) \wedge R(b) \rightarrow P(a,b)$ v. $Q(\mathsf{fst}\ p) \wedge R(\mathsf{snd}\ p) \rightarrow P(p)$

nearly always produce identical results. However, differences in instantiatedness, pragmas or specifications of rewrite axioms often produce different results.

The prover is simple to use, once its powers are understood. Understanding the prover would require a course in rewriting and first order proof, and training on a selection of appropriate problems. This is similar to provers such as Isabelle.

The feedback provided is good when simplification is the main proof technique being used, but is poor for first order proof. We have discussed this issue in Section 4.2.5. The scope of the prover was adequate for our case study, but any improvement in scope could dramatically simplify many proofs.

4.5 Related Work

This chapter builds on many techniques developed in other theorem proving systems. Most notably, the Boyer-Moore prover [BM81] pioneered the use of rewriting, the decision procedures to solve conditions, and a tagging/pragma mechanism to identify suitable rewrites. We have chosen not to adopt many of the heuristic aspects of Boyer and Moore's techniques in this work: for example we do not speculatively generate instantiations of first order formulae within decision procedures, or speculatively perform inductions. In the context of declarative proof the need for heuristics is not so great: the user can either specify the hints when required, or decompose the problem further. Indeed, heuristics go against the grain of many of our requirements.

Rewriting and grinding are used extensively for proof in PVS, again based mainly on techniques from Boyer-Moore. PVS, STeP and other systems implement various mixtures of ground decision procedures, and integrate them into the rewriting process.

The elimination of existential and universal quantifiers by automatic solving is a generalisation of the manual "unwinding" techniques from HOL [MTDR88], and relates to many *adhoc* (and often manual) techniques developed in other theorem provers. To the author's knowledge, no other interactive prover uses automatic solving techniques during rewriting to the same extent as Declare: searching for such solutions is quite computationally expensive but exceptionally useful.

Model elimination was first used in interactive higher order logic based theorem proving by Paulson and then Harrison [Har97a], and in general we owe much to Harrison's excellent implementations of model elimination and other procedures in HOL-lite.

Chapter 5

Interaction for Declarative Proof

In the previous three chapters we have considered the central logical issues relating to theorem proving for operational semantics: specification, proof description and automated reasoning. The solution we have adopted for proof description is declarative proof, as realised in the Declare system. In this chapter we turn to an issue that is considerably different in nature: the design of an *interactive development environment* (IDE) for Declare. This rounds out our treatment of tools for declarative proof, and the principles should be applicable to declarative proof systems in general.

"IDE" is jargon borrowed from the world of programming language implementations, particularly PC development suites such as Visual or Borland C++. IDEs are essentially document preparation systems combining powerful text editing facilities with tools to interpret and debug the programs developed (the documents being program texts in a range of languages).

The topic of interactive environments is different in nature from the preceding chapters because it is far more intimately concerned with human requirements, rather than machine or mathematical limitations. Human requirements are, of course, difficult to pin down precisely, but we endeavour to follow a fairly analytical approach in this chapter nevertheless, concentrating first on measurements of success for interactive systems.

5.1 Metrics for Interactive Systems

Before discussing IDEs for declarative proof, we consider the following question: what metrics should be applied to determine if an interactive system is a success? Firstly, let's make sure of our terminology: we call systems S_1 and S_2 *interfaces* if they support roughly the same fundamental task, though the means by which they support it may be different. Thus the Microsoft Windows File Manager and a subset of the Unix command line tools both support the tasks of moving, copying and searching file structures. Emacs and `vi` both support the task of editing text documents (amongst other things). A system is *interactive* if it has been designed

primarily for use by humans.[1] Thus an IDE like Visual C++ is an interactive interface to the underlying compilers.

One rather fundamental metric of success we can apply when comparing interactive interfaces is *productivity*, which is approximated by *mean time to task completion*.[2] So, all else being equal, *one interface to a theorem prover is better than another if it lets us get the same work done faster*.[3]

This, of course, begs the obvious question: how do we measure mean time to completion? Controlled experiments to determine task completion times for complex tasks such as theorem proving are expensive and difficult (*cf.* Section 3.6.2). However, this does not make the metric useless: it is possible to assess relative task completion times using informal analyses of possible scenarios. Even better, many interface devices are "clear winners" on this score: for example, a device that highlights errors in an original source text as they are detected is clearly going to improve productivity over a system that simply prints a line number which the user must look up manually. It is often surprisingly easy to argue the relative merits of individual interactive devices in such a way.

However, it is difficult to assess the differential merits of two quite disparate interactive methods. For example, we might like to be able to demonstrate that the interface IDeclare (presented in this chapter) always improves productivity over, say, the user's favourite text editor combined with the Declare command line tools. This is clearly difficult to demonstrate conclusively, and indeed is simply not the case: for some tasks one method is superior and for others the converse. Fortunately it is not an either-or situation, as we discuss later in this chapter.

5.2 IDeclare

Our IDE for Declare is called IDeclare, and a screen shot of the program in use is shown in Figure 5.1. It is being used to correct the error from Section 1.4.4. The principle features of IDeclare are:

- *Editing.* A standard text editor is provided for writing Declare articles in the usual fashion.

- *Logical Navigation and Debugging.* The state of the interface includes a *logical cursor*, that is a location within the logical structure of an article. The cursor acts much like the "program location" in a traditional program debugging system. The cursor may be moved by executing declarations and stepping

[1] Not all interactive systems are simply interfaces: for example Microsoft Word provides substantial functionality that can not realistically be accessed via any underlying mechanism.

[2] When an interactive system is not simply an interface, we cannot use such simple productivity metrics: we also have to measure the relative values of the different functionality provided. Comparing two systems that support the same overall tasks is considerably simpler.

[3] There are, of course, many other issues involved in overall usability. See Chapter 1 of [NL95] for an excellent informal description.

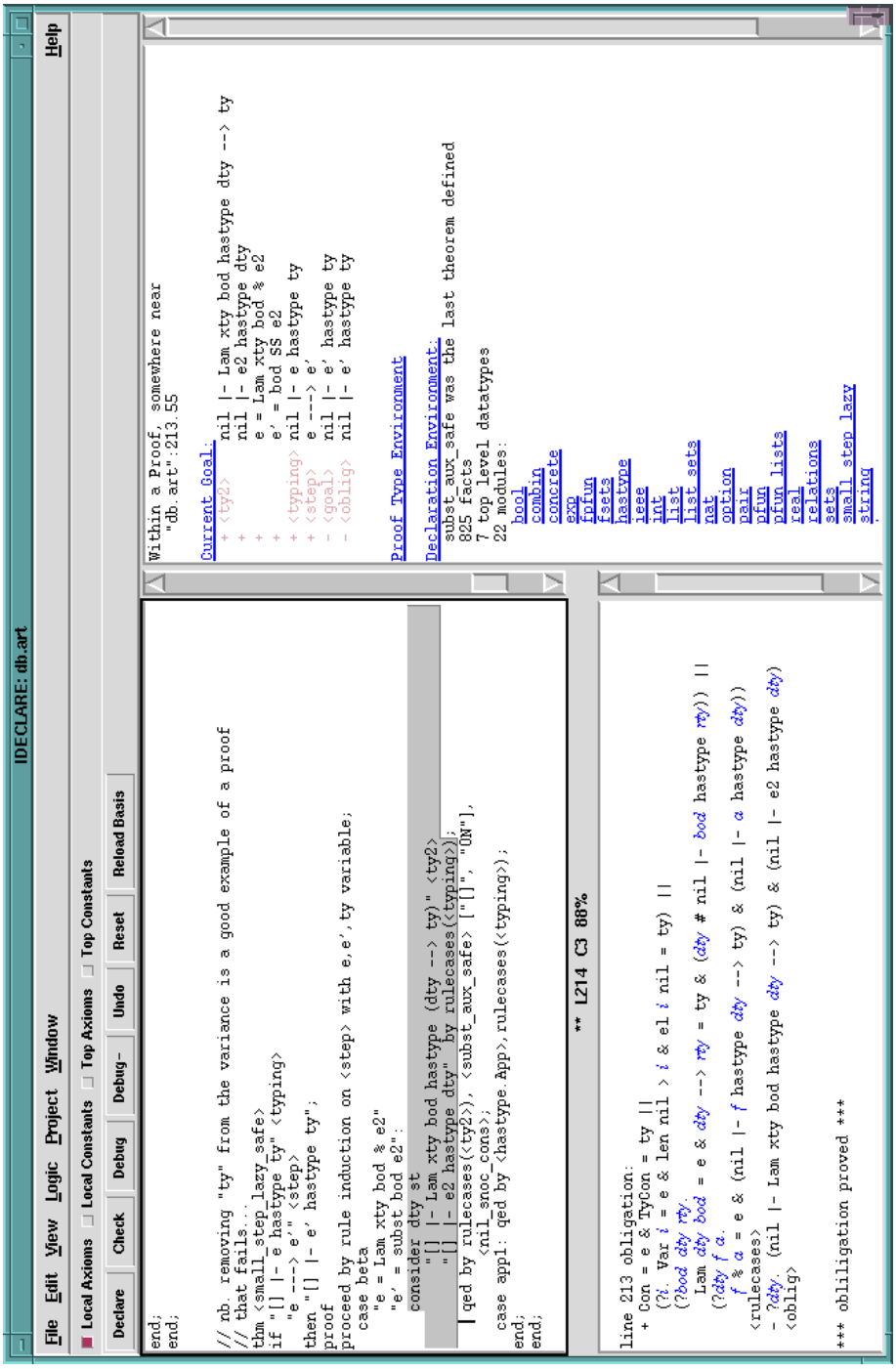


Figure 5.1: IDeclare: The Interactive Development Environment for Declare

inside constructs such as case-splits. This is discussed further in the next section.

- *Visualisation.* The location of logical cursor gives rise to a "current logical environment", in the sense of Section 3.1. The structures available in this are accessible through a window on the right of the application. They are displayed in hypertext form, i.e. with collapsible/expandable nodes for each article that has been imported.

- *Feedback.* Errors that arise during attempted manipulations of the logical cursor are displayed in the bottom left window. Some elements of the feedback are again displayed in hypertext form, e.g. counter examples can be accessed by clicking on a highlighted region. This means large amounts of feedback can be displayed quickly and compactly.

IDeclare is implemented in O'Caml-Tk [PR98] in 2000 lines of code, plus 10,000 lines shared with the batch-mode Declare implementation. Declare typically utilises 6-10 MB of memory, and IDeclare does not add significantly to this total.

5.2.1 Logical Navigation and Debugging

Logical navigation is the process of moving the logical cursor to a desired location within a Declare article. In IDeclare, the user controls the logical cursor with the following commands:

- *Declare.* That is, "step over" a construct (e.g. a `have`, `consider` or `qed` assertion, or a declaration in the specification language. Do not attempt to discharge proof obligations that arise from it.

- *Check.* That is, "step over" a construct, but attempt to discharge proof obligations.

- *Step into.* Move into a construct (e.g. a decomposition step). This will, for example, move the logical cursor into the first branch of a case split.[4]

- *Undo.* Retract the last movement made with the logical cursor.

- *Reset.* Set the logical cursor back to the "empty" environment, that is, the environment containing just the Declare standard basis.

One primary purpose of IDeclare is to allow the user to debug problematic justification steps within proofs, without repeatedly checking the entire script. This is achieved as follows:

[4] Other cases may currently be selected by "declaring" the `qed` step that ends each branch, until the desired case is reached. As in HOL IDeclare maintains a stack of pending cases. It would be fairly straightforward to allow the user to select the desired case immediately.

1. The user steps through the article, using a sequence of "Declare" and "Step into" commands, moving the logical cursor to the problematic area of the proof (for example, a `qed` step). Keyboard shortcuts are available for this, so navigation becomes quite quick for the experienced user. No justifications need be executed during this phase.

2. The user checks the justification with a "Check" command.

3. The user assesses the feedback, determines the adjustments that need to be made to the proof, and edits the document accordingly.

4. The user tries the "Check" command again, and repeats steps 2-3 until the step is accepted.

The "Declare" and "Step into" commands are only made possible by the use of declarative proof. In particular, *declarative proof allows logical navigation without having to discharge obligations or execute user-defined tactics along the way.* This is precisely because a declarative description of a proof step tells us "what" is proved, and not "how."

Interactive logical navigation and debugging for tactic proofs was first developed in TkHol [Sym95] (the author's interface for the HOL theorem prover), where it is possible to interactively move through the `THEN/THENL` structure of a HOL proof. Each such navigation step requires the execution of a tactic (and thus, in the terminology used above, we have a "Check" operation, and a "Step into" that requires tactic-execution). The user could make incremental adjustments to the proof script along the way, thus achieving a form of proof debugging. However, when navigating typical tactic proof languages, the only feasible operation is to actually execute a tactic, since we have no other way of knowing what its effects will be.

The navigation and debugging scheme we have described is primitive, but effective (we shall analyse why in the next section). Enhancements are certainly possible:

- The process of navigating to a problematic justification step could be easier than at present. Ideally, the user would place the textual cursor at a location and say "move the logical cursor here" or "show me the logical environment at this location." This would not be overly difficult to implement in the current setting.

- The current system allows only one active logical cursor. Experience with TkHol [Sym95] indicates that multiple active proofs are sometimes useful.

Finally, we emphasise that IDeclare does not maintain an exact correspondence between the logical cursor and the text of the document. For example, if a definition has been established (e.g. by a "Declare" step), and the user subsequently modifies the definition (textually), then the user must undo the old definition and reassert the new one to maintain the correspondence. That is, IDeclare is not a structure editor, and the cursor of the text editor is not the same as the the logical cursor. Like many

Task	Typical (sec.)	
Start up the interactive environment[a]	5	(5)
Navigate to the location in the proof	10-60[b]	(30)
Attempt to check the proof.	6-10[c]	(10)
Interpret the feedback	variable	(30)
Determine the appropriate fix	variable	(30)
Textually make the fix	variable	(30)
Additional repetitions of last 4 steps		(100)
Total		(235)

[a] Presuming it is not already running

[b] Depending on the size and complexity of the proof, the location of the mistake and the experience of the user

[c] The automated prover has a default timeout of 6 seconds

Table 5.1: Approximate time analysis for IDeclare. The "typical" times indicate a range of typical possibilities, and the example times in parentheses represent one hypothetical situation. Our analysis is deliberately informal: we are simply trying to indicate the order of magnitude of the major contributing delays.

Emacs based environments [Sta95], IDeclare does have some understanding of the syntactic structure of a Declare article — for example, it can syntactically detect the textual bounds on the next declaration when executing a "Declare" step. It will then move the textual cursor to the start of the next declaration, provided the previous declaration was accepted.

5.3 Appraisal

In this section we perform an informal analysis on the costs and benefits of using IDeclare for a particular task, using "mean time to task completion" as our metric of success. *The times used are not meant to be definitive, and do not represent the outcome of a controlled experiment. They are merely indicative, and based on the author's experience.* This method of analysis is adapted from the analysis techniques in Chapter 8 of Newman and Lamming's *Interactive System Design* [NL95].

The task we shall analyse is that described in the previous section: correcting an error in a justification in a proof. We have already outlined the steps required to do this in IDeclare. Without IDeclare, the user must correct the mistake in a standard text editor and then recheck the entire script. This must be repeated until the mistake is fixed. Approximate analyses of completion times for the task in the two systems are shown in Tables 5.1 and 5.2.

Although such an informal analysis does not establish conclusive results, it does support our intuitions, and certainly helps guide the design of the interface. The times for the hypothetical example indicate that, for a proof of medium complexity

Task	Typical (sec.)	
Start up the batch processor	5	(5)
Attempt to check all steps of the proof	10-180[a]	(100)
Interpret the feedback	variable	(30)
Determine the appropriate fix	variable	(30)
Textually make the fix	variable	(30)
Additional repetitions of these steps		(295)
Total		(390)

[a]Depending on the size and complexity of the entire proof script. The delay can, of course, be arbitrary large, but even long articles containing errors usually check in under 3 minutes.

Table 5.2: Approximate time analysis for batch-mode Declare. See notes for Table 5.1.

where two iterations are required to make the correction, IDeclare will indeed provide a faster solution. This is not surprising: we have eliminated the repetition of a time consuming step (the checking of other steps in a proof). The overheads required to do this (such as navigating to the location) are not overly burdensome.[5]

We have chosen the task above in order to demonstrate a situation where IDeclare is particularly useful. IDeclare is not always superior: for example, its text editor is a little clunky and the user will normally prefer his/her own for large scale text editing operations. Similarly, the visualisation tools are not always the quickest way to find information: sometimes they are, but sometimes it is preferable to look through the original source files. IDeclare is just a support tool, and its use is not mandated, and so the user is free to select the approach that will be quickest depending on the particular task. However, ideally further development could make IDeclare the preferred tool for the majority of Declare related activities.

To summarize, what results have been established by developing IDeclare? Primarily our aim has been to demonstrate that the proof debugging paradigm can carry over to declarative proof systems. Furthermore, declarative structure is precisely what is required to support certain debugging actions (in particular "Declare" and "Step into"). This indicates that a declarative proof style may allow for better theorem prover interfaces, and this appears a promising direction for future research.

[5]As an aside, the analysis also indicates why the proposal to automate the navigation process (see the end of Section 5.2.1) would provide a significant benefit, as it would cut 20-30 seconds off the task completion time.

Part II

Case Study

Chapter 6

Java$_S$

The previous chapters have described a set of tools and techniques for conducting "declarative" theorem proving in the context of operational semantics. In the following two chapters we describe a major case study in the application of these techniques: a proof of the type soundness of a subset of Java (Java$_S$) using Declare.

The case study is significant in its own right, so for the most part we concentrate on the substance of the case study rather than the role that declarative proof played in its execution, which we summarize and discuss in Chapter 8.

Drossopoulou and Eisenbach have presented a formal semantics for approximately the same subset of Java that we treat here [DE97a]. Our work is based on theirs and improves it by correcting and clarifying many details.

Our main aim has not been to find errors. However, some significant mistakes in the original formulation adopted by Drossopoulou and Eisenbach were discovered, and we were able to provide feedback and suggestions to the authors. We also independently rediscovered a significant error in the Java Language Specification [GJS96]. Our methodology and tools enabled us to find the error relatively quickly, and this demonstrates the positive role that machine checking can play when used in conjunction with existing techniques.

In this chapter we briefly introduce Java and describe our formal model of Java$_S$, including our models of type checking and execution. We also briefly describe the representation of the model in Declare, and asses the use of Declare for this purpose. The proof of type soundness itself is described in the next chapter.

6.1 Java

Java [GJS96] is a programming language developed by Sun Microsystems, and has exploded in popularity over the last 3 years. Although sometimes over-hyped as heralding a new age of computing, the language design itself is highly competent, incorporating many ideas into a framework palatable for the existing base of C++ programmers. It can be executed fairly efficiently with just-in-time compilers, and comes equipped with a well-designed set of portable basis libraries. Perhaps most

importantly, it is suitable for programming mobile code on the WWW, much more so than C++ or other procedural languages.

Java's suitability for WWW programming rests largely on its type system, which, in principle, allows for efficient execution of code in a "sand-box." Studies have uncovered flaws in the security of Java and its implementations, including its type system, and have pointed out the need for a formal semantics to complement the existing language definition [DFW96, GJS96]. A full formal treatment of many important aspects of the language (e.g. dynamic linking) has yet to be performed. Because of these things, type soundness is clearly a property we are interested in for this language.

The Java source language is compiled to a closely related bytecode format for the *Java Virtual Machine* (JVM). Although the languages are different, their type systems are quite similar. Java is defined by several standards, including those for the source language [GJS96] and the Java Virtual Machine [LY97].

6.2 Our Model of Java$_S$

The aim of a type correctness proof is to bridge the gap between:

- A model of the *static checks* performed on Java$_S$ programs; and
- A model of the *runtime execution* of the same.

The remainder of this chapter is devoted to describing these two models. We have inherited much from Drossopoulou and Eisenbach's work, so we concentrate on the areas where our model differs. The material is rather technical and there are many "building-blocks" to describe: the reader is encouraged to refer back to this section as needed.

A picture of the components of the semantics is shown in Figure 6.1. We make use of several intermediate languages along the way. The "annotated" language Java$_A$ is the result of the static checking process and the "runtime" language Java$_R$ is the code executed at runtime. We assign typing semantics to each of these components and show how these relate. We shall leave the description of the typing semantics of Java$_R$ until the next chapter as it is an artifact of the type-soundness proof.[1]

6.2.1 The Java Subset Considered

The Java subset we consider includes primitive types, classes with inheritance, instance variables and instance methods, interfaces, dynamic method binding, statically resolvable overloading of methods and instance variables, object creation, null pointers, arrays, return statements, while loops, methods for the class Object and a minimal treatment of exceptions. The subset excludes initializers, constructors,

[1] The same is true of the static semantics for Java$_A$, but it is sufficiently close to those for Java$_S$ that we describe them in this chapter.

$$\begin{array}{ccccccccc} & & & \textit{static} & & & & & \\ & & & \textit{checking} & & \textit{compilation} & & \textit{execution} & \\ \text{Java} & \supset & \text{Java}_S & \rightsquigarrow_{ann} & \text{Java}_A & \rightsquigarrow_{comp} & \text{Java}_R \times \textit{state} & \rightsquigarrow_{(\Gamma,p)} & \text{Java}_R \times \textit{state} \\ & & \downarrow & & \downarrow & & \downarrow & & \downarrow \\ & & \textit{type} & = & \textit{type} & = & \textit{type} & \geq_{wdn} & \textit{type} \end{array}$$

Figure 6.1: Components of the Semantics and their Relationships

finalizers, class variables and class methods, switch statements, local variables, class modifiers, final/abstract classes and methods, `super`, strings, numeric promotions and widening, concurrency, the handling of exceptions, dynamic linking, finalizers, packages, binary compatibility and separate compilation.

In this study we are concerned with the Java language itself, rather than the Java Virtual Machine (JVM). The two are closely related but the difference is non-trivial: for example there are JVM bytecodes that do not correspond to any Java text. Thus it remains a challenge to formalize and verify the corresponding type soundness property for the JVM (for an attempt see [Qia97]). However, unlike many high-level/low-level language combinations (e.g. C++/assembler) the type systems of Java and the JVM are closely related, and a comprehensive study of the former is a useful precursor to the study of the latter. Of course, even if we prove properties of an abstract model of Java and/or the JVM, this does not guarantee the soundness of a particular implementation.

6.2.2 Comparison with Drossopoulou and Eisenbach

Our model was originally based on that developed by Drossopoulou and Eisenbach in version 2.01 of their paper [DE97b, DE97a].[2] The differences in the subset considered are:[3]

- `Object` *has Methods.* We allow the primitive class `Object` to have methods. It was when considering this extension that one mistake in the Java Language Specification was discovered (see Section 7.4).

- *Methods for Arrays.* Arrays in Java support all methods supported by the class `Object` (e.g. `hashValue()`). We include this in our model (with non-trivial consequences). However our model of arrays is still incomplete, as Java arrays support certain array-specific methods and fields, whereas in our treatment they do not.

[2] This version was distributed only on the WWW, and is no longer directly available. If a version is needed for reference please contact the authors.

[3] Note that Drossopoulou and Eisenbach have since progressed to model other aspects of the language such as exceptions and binary compatibility [DE98].

- *Return Statements.* These were added as an exercise in extending the semantics. They are non-trivial as static checks must ensure all computation paths terminate with a return.

The main differences in the model itself are:

- *Corrections.* We correct minor mistakes, such as missing rules for null pointers, some definitions that were not well-founded, some typing mistakes and some misleading/ambiguous definitions (e.g. the definition of MethBody, and the incorrect assertion that any primitive type widens to the null type).

- *Representation.* We choose different representations for some constructs, e.g. type checking environments are represented by tables (finite partial functions) rather than lists of declarations.

- *Separate Languages.* We differentiate between the source language Java$_S$, the annotated language Java$_A$ and the 'runtime terms' Java$_R$. Java$_R$ is used to model terms arising during execution and enjoy subtly different typing rules. Drossopoulou and Eisenbach have since reported that the language Java$_A$ is useful for modelling binary compatibility [DE98], because it allows us to model precisely both compile-time and runtime analyses

- *Simpler Well-formedness.* We adopt a suggestion by von Oheimb that well-formedness for environments be specified without reference to a declaration order.

- *No Static Substitution.* We do not use substitution during typing, as it turns out to be unnecessary given our representation of environments.

- *No Dynamic Substitution.* We do not use substitution during evaluation, but use a model of stack frames instead. This seems simpler and is closer to a real implementation.

The differences in our approach to the type soundness proof are detailed in the next chapter.

6.2.3 Syntax

Figure 6.2 presents the abstract syntax of Java$_S$ programs, along with the changes for the abstract syntax of the annotated language Java$_A$.

- Variables are terms that evaluate to storage locations and play the role of lvalues in C.

- In Java$_A$ variables are annotated with the actual class referred to by the access, and method calls are annotated by the argument signature resolved by static-overloading.

prim-type	=	`bool` | `char` | `byte` | `short`	
	|	`int` | `long` | `float` | `double`	
simple-type	=	*primitive-type* | *class-id* | *interface-id*	
var-type	=	*simple-type*`[]`^{n}	($n \geq 0$)
expr-type	=	*var-type* | `void`	
literal	=	*bool* | *uchar* | *int8*	
	|	*int16* | *int32* | *int64* | *ieee32* | *ieee64*	
var	=	*id*	(local variable)
	|	*expr*.field-name	(object field)
		expr.field-name$_{\textit{class-name}}$	(annotated object field)
	|	*expr*`[`*expr*`]`	(array element)
expr	=	*literal*	(literal value)
	|	*var*	(dereferencing)
	|	`null`	(null literal)
	|	*expr*.*method-name*`(`*expr*`*)`	(method call)
		expr.*method-name*$_{\textit{var-type list}}$`(`*expr*`*)`	(annotated method call)
	|	`new` C	(object creation)
	|	`new` *comptype*`[`*expr*`]*[]*`	(array creation)
stmt	=	`if` *expr* `then` *stmt* `else` *stmt*	(conditional)
	|	`while` *expr* `do` *stmt*	(while loop)
	|	*var* `:=` *expr*	(assignment)
	|	{ *stmt*$_1$; ...; *stmt*$_n$; }	(block)
	|	*expr*	(evaluation)
	|	`return` *expr*?	(method return)
method	=	*expr-type method-name*(*var-type* x$_1$, ..., *var-type* x$_n$)	(method declaration)
		{ *stmt* }	
field	=	*var-type field-name*	(field declaration)
class	=	C `extends` C$_{sup}$ `implements` I$_1$, ..., I$_n$ {	(class declaration)
		field$_1$; ...; *field*$_n$;	
		method$_1$; ...; *method*$_m$;	
		}	
prog	=	*class*$_1$; ...; *class*$_n$;	(programs)

Figure 6.2: The Abstract Syntax of Java$_S$ and Java$_A$

prim-type	=	`void` | `bool` | `char` | `byte` | `short`
	|	`int` | `long` | `float` | `double`
ref-type	=	*type*`[]` | *class-id* | *interface-id* | `null`
type	=	*primitive-type* | *ref-type*

Figure 6.3: von Oheimb's Extended Range of Types

- Formal parameters are represented by a list of identifiers and a table assigning types to these identifiers.

The types that appear in the concrete syntax of Java$_S$ expressions are also shown in Figure 6.2. Following von Oheimb's treatment [Nv98] we extend the domain of types to include a primitive `void` type, a `null` type to assign to the `null` literal during typechecking, and syntactically differentiate between reference and plain types.[4][5] We use τ and ρ to range over types, the latter used for method return types.

6.3 Preliminaries

In the next two sections we shall present the static semantics for Java$_A$ and Java$_S$. The complicating factors common to both are:

- *Subtyping.* Java allows subtyping in a typical object-oriented fashion, which leads to a *widening* ($\leq$) relation.
- *Forward Use.* Java allows the use of classes before they are defined. Thus we define *type environments*, extracted from all the classes and interfaces that make up a program.
- *Complex Well-formedness.* The constraints on valid type environments are non-trivial, e.g. non-circular class and interface hierarchy must result, classes must implement interfaces and so on. One of the main challenges of this case study is to identify precisely the well-formedness criteria required.
- *Visibility.* Aspects of the semantics depend on name-visibility properties, e.g. to define fields and methods are visible from subclasses. Visibility is defined by relations for traversing the class and interface hierarchies.

In this section we define the preliminaries that are required to deal with these problems.

6.3.1 The Structure of Type Environments

Constructs are given typing semantics with respect to *type environments*, which contain several components (Figure 6.4). Always present are tables of class and interface declarations. When typechecking variables and expressions we add a table of local variable declarations, and for statements we also add the declared return type

[4] Unlike Standard ML, `void` is not a first-class type in Java, e.g. an array of `void`s is not possible. We treat `void` as a first-class type in our models of Java$_A$ and Java$_R$, but exclude it at the source language Java$_S$.

[5] Our original model [Sym97b] used an overly complicated model of types, with multiple widening and well-formedness relations for these. When we modified the proof to take advantage of von Oheimb's simpler formulation, the textual size of our formulation was reduced by around 15% — a useful saving.

$$
\begin{array}{lcl}
\mathit{env} & = & \mathit{class\text{-}env} \times \mathit{interface\text{-}env}\ \ (\times \mathit{variable\text{-}env?})\ \ (\times \mathit{expr\text{-}type?}) \\
\mathit{class\text{-}env} & = & \mathit{class\text{-}ids} \stackrel{table}{\longmapsto} \mathit{class\text{-}dec} \\
\mathit{interface\text{-}env} & = & \mathit{interface\text{-}ids} \stackrel{table}{\longmapsto} \mathit{interface\text{-}dec} \\
\mathit{variable\text{-}env} & = & \mathit{variable\text{-}ids} \stackrel{table}{\longmapsto} \mathit{type} \\
\mathit{class\text{-}dec} & = & \langle\ \text{superclass:}\ \ \mathit{class\text{-}id}, \\
 & & \ \ \text{interfaces:}\ \ \text{set of } \mathit{interface\text{-}ids}\ , \\
 & & \ \ \text{fields:}\ \ \mathit{field\text{-}ids} \stackrel{table}{\longmapsto} \mathit{type}, \\
 & & \ \ \text{methods:}\ \ \mathit{meth\text{-}ids} \times \mathit{arg\text{-}types} \stackrel{table}{\longmapsto} \mathit{expr\text{-}type}\rangle \\
\mathit{interface\text{-}dec} & = & \langle\ \text{superinterfaces:}\ \ \text{set of } \mathit{interface\text{-}ids}\ , \\
 & & \ \ \text{methods:}\ \ \mathit{meth\text{-}ids} \times \mathit{arg\text{-}types} \stackrel{table}{\longmapsto} \mathit{type}\rangle
\end{array}
$$

Figure 6.4: Type checking environments

of the method in order to check `return` statements. We often write environments as records ($\langle \ldots \rangle$), and omit record tag names when it is obvious which record field is being referred to.[6]

We use Γ for a composite environment, Γ^V, Γ^C and Γ^I its respective components, and $\Gamma(x)$ for the lookup of x in the appropriate table. We also use $x \in \Gamma$ to indicate that x has an entry in the relevant table in Γ.

6.3.2 Well-formed Types

Types and other simple semantic objects are said to be well-formed, (e.g. $\Gamma \vdash C \diamond_{class}$, or `TE |- C wf_class` in the Declare specification) if all classes and interfaces are in scope. For example:

$$
\begin{array}{c}
\Gamma \vdash C \diamond_{class} \equiv C \in \Gamma^C \\
\Gamma \vdash I \diamond_{intf} \equiv I \in \Gamma^I
\end{array}
$$

$$
\frac{\Gamma \vdash C \diamond_{class}}{\Gamma \vdash C \diamond_{refty}} \qquad \frac{\Gamma \vdash I \diamond_{intf}}{\Gamma \vdash I \diamond_{refty}} \qquad \frac{\Gamma \vdash \tau \diamond_{ty}}{\Gamma \vdash \tau[\,] \diamond_{refty}} \qquad \frac{\Gamma \vdash \tau \diamond_{refty}}{\Gamma \vdash \tau \diamond_{ty}} \qquad \frac{pt \in \mathit{prim\text{-}types}}{\Gamma \vdash pt \diamond_{ty}}
$$

An Aside: Well-formedness predicates can be thought of as *dependent predicate subtypes* (dependent because they are parameterized by, for example, Γ). As such they are not representable as types in simple h.o.l. (though would be in, say, PVS), but in practice we treat them much like types.

Each relation we define has implicit side conditions, i.e. that each argument satisfies the appropriate well-formedness condition. For example, the relation $\Gamma \vdash C \sqsubseteq_{class} C'$ has the implicit side conditions $\vdash \Gamma \diamond_{tyenv}$, $\Gamma \vdash C \diamond_{class}$ and $\Gamma \vdash C' \diamond_{class}$. Using the relation in Declare without being able to prove these side-conditions is a violation

[6] In the machine acceptable model we do not use such conveniences: the records are represented as tuples.

of our methodology, but correct usage is not proved automatically by typechecking. Note we may leave relations underspecified (see Section 2.2.6) where the side conditions do not hold.

This matter is of some importance: for example, a typical type soundness theorem states that for each reduction to a new configuration cfg' there exists some τ' such that the cfg' conforms to τ'. (We define these terms in Chapter 7 — what is important here is that τ is existentially quantified.) If we don't explicitly prove that τ' is well-formed, then we have hardly guaranteed the correct operation of the system. So, we explicitly add the assertion $\Gamma \vdash \tau' \diamond$ to the statement of the theorem, and with such assertions, we can see by inspection that our final theorems treat well-formedness correctly.[7]

6.3.3 The $\sqsubseteq_{class}$, $\sqsubseteq_{intf}$ and $:_{imp}$ Relations

We define the subclass ($\sqsubseteq_{class}$, or `subclass_of` in Declare), subinterface ($\sqsubseteq_{intf}$, or `subinterface_of`) and implements ($:_{imp}$, or `implements`) relations as shown below. All classes are a subclass of the special class `Object`, though we do not have to mention this explicitly as the well-formedness conditions for environments ensure it.

$$\frac{}{\Gamma \vdash C \sqsubseteq_{class} C}\ (reflC) \qquad \frac{C \text{ has super } C_{sup} \quad \Gamma \vdash C_{sup} \sqsubseteq_{class} C'}{\Gamma \vdash C \sqsubseteq_{class} C'}\ (stepC)$$

$$\frac{}{\Gamma \vdash I \sqsubseteq_{intf} I}\ (reflI) \qquad \frac{I \text{ has } I_k \text{ amongst its superinterfaces} \quad \Gamma \vdash I_k \sqsubseteq_{intf} I'}{\Gamma \vdash I \sqsubseteq_{intf} I'}\ (stepI)$$

$$\frac{C \text{ has } I_k \text{ amongst its implemented interfaces}}{\Gamma \vdash C :_{imp} I_k}\ (implements)$$

6.3.4 Widening

Subtyping in Java is the combination of the subclass, subinterface and implements relations, and is called *widening* (`widens_to` in Declare) and also for vectors of types $\leq_{vartys}$ (`tys_widen_to`). The rules for widening in Java$_A$ are:

$$\frac{\Gamma \vdash C \sqsubseteq_{class} C'}{\Gamma \vdash C \leq C'} \qquad \frac{\Gamma \vdash I \sqsubseteq_{intf} I'}{\Gamma \vdash I \leq I'} \qquad \frac{I \in \Gamma}{\Gamma \vdash I \leq \texttt{Object}} \qquad \frac{\begin{array}{l}\Gamma \vdash C \sqsubseteq_{class} C' \\ \Gamma \vdash C' \diamond_{class} \\ \Gamma \vdash C' :_{imp} I \\ \Gamma \vdash I \diamond_{intf} \\ \Gamma \vdash I \sqsubseteq_{intf} I'\end{array}}{\Gamma \vdash C \leq I'} \qquad \frac{pt \in \textit{prim-types}}{\Gamma \vdash pt \leq pt}$$

$$\frac{\Gamma \vdash \tau \diamond_{ty}}{\Gamma \vdash \tau\texttt{[]} \leq \texttt{Object}} \qquad \frac{\Gamma \vdash \tau \leq \tau'}{\Gamma \vdash \tau\texttt{[]} \leq \tau'\texttt{[]}}$$

An example graph that covers all possible connection paths is shown in Figure 6.5.

[7] Thus it is insufficient to say "we assume all types are well-formed," since well-formedness sometimes involves proof obligations.

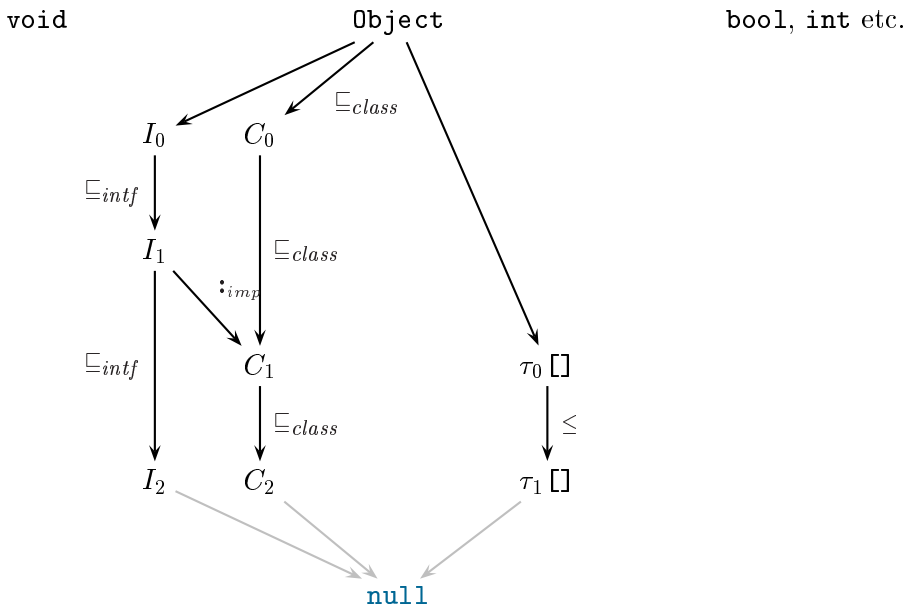


Figure 6.5: Connections in the Subtype (Widening) Graph.

6.3.5 Visibility

The relations $\lhd_{fld}$ and $\lhd_{meth}$ (`sees_field` and `sees_meth` in Declare) tell us what fields and methods are visible from a given class or interface.

- $\lhd_{meth}$: Finds the 'nearest' version of a method starting at a particular reference type (i.e. an array, interface or class type).

  For example, $\Gamma \vdash \tau_0 \lhd_{meth} (m, AT), \rho$ holds whenever the method m with argument signature AT is visible from type τ_0 (in the type environment Γ) and the 'nearest' version of the method has return type ρ.[8]

  Methods may be overloaded, so, after static resolution, method call statements are annotated with argument descriptors. Consequently we often write (m, AT) as $mdesc$ because the pair acts as a descriptor indexing into the method tables. Methods with identical argument descriptors hide methods further up the hierarchy, though their return types differ.[9]

  The relation may be used with a definite argument descriptor, when it effectively finds the return type for the visible version of that method. Constraints on well-formed type environments ensure that this result is unique.

[8] It turns out not to matter exactly "where" the method was found.

[9] This is not the case in Java, but is dealt with by Drossopoulou and Eisenbach. One must take more care with this extension than originally thought by Drossopoulou and Eisenbach— see Section 7.4.

- $\lhd_{\mathit{fld}}$: Finds the 'first visible' definition of a field starting at a particular class.

  For example, $\Gamma \vdash C_0 \lhd_{\mathit{fld}} (\mathit{fld}, C), \tau$ holds whenever the field fld is visible from C_0 at class C in the type environment Γ and is declared to be of type τ.

In Drossopoulou and Eisenbach's original formulation these definitions were given as recursive functions (FDec and MSigs). They only make sense for well-formed environments, as the search may not terminate for circular class and interface hierarchies. To avoid this problem we define the constructs as inductively defined sets. Method visibility $\lhd_{meth}$ is defined via $\lhd_C$ $\lhd_I$ and $\lhd_A$ for the the three different reference types. All methods found in the type `Object` are visible from array and interface types (see also Section 7.4.1):

$$\frac{\Gamma(C_0).\text{methods}(mdesc) = \rho}{\Gamma \vdash C_0 \lhd_C mdesc, \rho}\ (BaseC) \qquad \frac{\begin{array}{l}\Gamma(C_0).\text{methods}(mdesc) = \bot \\ C_0 \text{ has superclass } C_{sup} \\ \Gamma \vdash C_{sup} \lhd_C mdesc, \rho\end{array}}{\Gamma \vdash C_0 \lhd_C mdesc, \rho}\ (StepC)$$

$$\frac{\Gamma(I_0).\text{methods}(mdesc) = \rho}{\Gamma \vdash I_0 \lhd_I mdesc, \rho}\ (BaseI) \qquad \frac{\begin{array}{l}\Gamma(I_0).\text{methods}(mdesc) = \bot \\ I_{sup} \text{ is a superinterface of } I_0 \\ \Gamma \vdash I_{sup} \lhd_I mdesc, \rho\end{array}}{\Gamma \vdash I_0 \lhd_I mdesc, \rho}\ (StepI)$$

$$\frac{n > 0 \quad \Gamma \vdash \texttt{Object} \lhd_C mdesc, \rho}{\Gamma \vdash \tau_0\texttt{[]}^n \lhd_A mdesc, \rho}\ (Array) \qquad \frac{\begin{array}{l}\Gamma \vdash \tau_0 \lhd_C mdesc, \rho \text{ or} \\ \Gamma \vdash \tau_0 \lhd_I mdesc, \rho \text{ or} \\ \Gamma \vdash \tau_0 \lhd_A mdesc, \rho \text{ or}\end{array}}{\Gamma \vdash \tau_0 \lhd_{meth} mdesc, \rho}\ (Any)$$

Field visibility is simpler:

$$\frac{\Gamma(C_0).\text{fields}(\mathit{fld}) = \tau}{\Gamma \vdash C_0 \lhd_{\mathit{fld}} (\mathit{fld}, C_0), \tau} \qquad \frac{\begin{array}{l}\Gamma(C_0).\text{fields}(\mathit{fld}) = \bot \\ C_0 \text{ has superclass } C_{sup} \\ \Gamma \vdash C_{sup} \lhd_{\mathit{fld}} (\mathit{fld}, C), \tau\end{array}}{\Gamma \vdash C_0 \lhd_{\mathit{fld}} (\mathit{fld}, C), \tau}$$

This relation is only employed during type checking and annotation of Java$_S$, in particular to determine the class where a field is declared. Once field names are resolved we require a relation $\lhd_{allfields}$ (`has_field` in Declare) that finds all fields including hidden ones:

$$\frac{C_0 \text{ has superclass } C_{sup} \quad \Gamma \vdash C_{sup} \lhd_{allfields} \mathit{fldspec}}{\Gamma \vdash C_0 \lhd_{allfields} \mathit{fldspec}}\ (Super) \qquad \frac{\Gamma(C_0).\text{fields}(\mathit{fld}) = \tau}{\Gamma \vdash C_0 \lhd_{allfields} ((\mathit{fld}, C_0), \tau)}\ (Hit)$$

If Γ is well-formed (see the next section) then $\Gamma \vdash C_0 \lhd_{allfields} (\mathit{fld}, C), \tau$ holds for at most one τ (given all other arguments).

6.3.6 Well-formedness for Type Environments

Well-formedness for a type checking environment ($\vdash \Gamma \diamond_{tyenv} \equiv$ `wf_tyenv`) ensures crucial properties such as subclasses providing methods compatible with their superclasses, and classes providing methods that implement their declared interfaces. Drossopoulou and Eisenbach originally formulated this by an incremental process, where the environment is constructed from a sequence of definitions. We have adopted a suggestion from von Oheimb who has pointed out that this is not necessary, since the definition is independent of any ordering constraints. A finiteness constraint is needed to ensure no infinite chains of classes exist that do not terminate in 'Object'.

The criteria for each class in an environment are: [10]

- Its superclass (if it has one) and its implemented interfaces must be well-formed and no circularities can occur in the hierarchy;
- If the class has no superclass it must be the special class `Object`.
- All the methods declared for the class must have well-formed types.
- All declared fields must have well-formed types.
- Any declared method that overrides an inherited method (by having the same name and argument types) must have a narrower return type;
- All methods accessible via each implemented interface must be matched by a method in this class or some superclass. The method is allowed to have a narrower return type.

These constraints are written formally as:

$$
\begin{array}{lll}
\multicolumn{3}{l}{\text{if } \Gamma(C) = \langle C_{sup}, Is, \mathit{fields}, \mathit{methods}\rangle \text{ then}} \\
& \Gamma \vdash C_{sup} \diamond_{class} & (A.1) \\
\text{and} & \neg(\Gamma \vdash C_{sup} \sqsubseteq_{class} C) & (A.2) \\
\text{and} & \forall I \in Is.\ \Gamma \vdash I \diamond_{intf} & (A.3) \\
\text{and} & \forall \mathit{fld}, \tau.\ \text{if } \mathit{fields}(\mathit{fld}) = \tau \text{ then } \Gamma \vdash \tau \diamond_{ty} & (A.4) \\
\text{and} & \forall m, AT, \rho.\ \text{if } \mathit{methods}(m, AT) = \rho \text{ then } \Gamma \vdash AT \diamond_{tys} \text{ and } \Gamma \vdash \rho \diamond_{ty} & (A.5) \\
\text{and} & \forall \mathit{mdesc}, \rho_1, \rho_2. & \\
& \quad \text{if } \mathit{methods}(\mathit{mdesc}) = \rho_1 \text{ and } \Gamma \vdash , C_{sup} \lhd_{meth} \mathit{mdesc}, \rho_2 & \\
& \quad \text{then } \Gamma \vdash \rho_1 \leq \rho_2 & (A.6) \\
\text{and} & \forall I \in Is, \mathit{mdesc}, \rho_1, \rho_2. & \\
& \quad \text{if } \Gamma \vdash , I \lhd_I \mathit{mdesc}, \rho_1 & \\
& \quad \text{then } \exists \rho_2. \Gamma \vdash C \lhd_C \mathit{mdesc}, \rho_2 \text{ and } \Gamma \vdash \rho_2 \leq \rho_1 & (A.7)
\end{array}
$$

A similar set of constraints must hold for each interface declaration:

[10] There are other criteria that are implicit in the structures we have used for environments, e.g. that no two methods have the same method descriptor.

- Its superinterfaces must be well-formed and no circularities can occur in the hierarchy;
- All the methods declared for the class must have well-formed types.
- All declared fields must have well-formed types.
- Any declared method that overrides an inherited method (by having the same name and argument types) must have a narrower return type;
- Any declared method that overrides an `Object` method must have a narrower return type;

These constraints are written formally as:

$$
\begin{array}{lll}
\text{if } \Gamma(I) = \langle Is, methods\rangle \text{ then} & & \\
\quad & \forall I_{sup} \in Is.\ \Gamma \vdash I_{sup} \diamond_{intf} & (B.1) \\
\text{and} & \forall I_{sup} \in Is.\ \neg(\Gamma \vdash I_{sup} \sqsubseteq_{intf} I) & (B.2) \\
\text{and} & \forall m, AT, \rho.\ \text{if } methods(m, AT) = \rho \text{ then } \Gamma \vdash AT \diamond_{tys} \text{ and } \Gamma \vdash \rho \diamond_{ty} & (B.3) \\
\text{and} & \forall I_{sup} \in Is, mdesc, \rho_1, \rho_2. & \\
\quad & \quad \text{if } methods(mdesc) = \rho_1 \text{ and } \Gamma \vdash I_{sup} \lhd_{meth} mdesc, \rho_2 & \\
\quad & \quad \text{then } \Gamma \vdash \rho_1 \leq \rho_2 & (B.4) \\
\text{and} & \forall I_{sup} \in Is, mdesc, \rho_1, \rho_2. & \\
\quad & \quad \text{if } methods(mdesc) = \rho_1 \text{ and } \Gamma \vdash \texttt{Object} \lhd_{meth} mdesc, \rho_2 & \\
\quad & \quad \text{then } \Gamma \vdash \rho_1 \leq \rho_2 & (B.5)
\end{array}
$$

In addition the class `Object` must be defined and have no superclass, superinterfaces or fields.

$$\exists methods.\ \Gamma(\texttt{Object}) = \langle \mathsf{None}, \{\}, methods, \{\}\rangle$$

Well-formedness of a type environment is sufficient to guarantee many important properties including:

- Reflexivity and transitivity of $\sqsubseteq_{class}$, $\sqsubseteq_{intf}$, $\leq$.
- Monotonicity of $\lhd_{meth}$ up to $\leq$, with possibly narrower return types.
- Monotonicity of $\lhd_{allfields}$ up to $\leq$ ($\lhd_{fld}$ is not monotonic because fields may be hidden).
- Uniqueness of fields when qualified by class names.

We state these formally in the next chapter.

6.4 Static Semantics for Java$_A$

In this section we present the static semantics for the annotated language Java$_A$. We present this language first because its static semantics are considerably simpler than those for Java$_S$, and because they take us considerably closer to the heart of the soundness proof presented in the next chapter. The types assigned to Java$_A$ fragments are the same as the types that appear in the Java$_S$ source language. The rules give rise to a series of relations (`avar_hastype` through to `aprog_hastype` denoted here by $\Gamma \vdash \_ : \_$ and $\Gamma \vdash \_\checkmark$ — we use subscripts when the exact relation is ambiguous). The rules for variables are:

$$\frac{\Gamma^V(id) = \tau}{\Gamma \vdash id : \tau}\ (Var) \qquad \frac{\Gamma \vdash arr : \tau[] \quad \Gamma \vdash idx : \texttt{int}}{\Gamma \vdash arr[idx] : \tau}\ (Access) \qquad \frac{\Gamma \vdash obj : C_0 \quad \Gamma \vdash C_0 \lhd_{allfields} (fld, (C, \tau))}{\Gamma \vdash obj\,.\,fld_C : \tau}\ (Field)$$

The rules for expressions are:

$$\frac{\tau \text{ is the primitive type for } pval}{\Gamma \vdash pval : \tau}\ (Prim) \qquad \frac{}{\Gamma \vdash \texttt{null} : \texttt{null}}\ (Null)$$

$$\frac{\Gamma \vdash var :_{var} \tau}{\Gamma \vdash var :_{exp} \tau}\ (Deref) \qquad \frac{\Gamma \vdash d_i : \texttt{int} \quad (1 \le i \le n)}{\Gamma \vdash \texttt{new}\ \tau[d_1] \ldots [d_n][]^m : \tau[]^{m+n}}\ (NewArray)$$

$$\frac{\Gamma \vdash C \diamond_{class}}{\Gamma \vdash \texttt{new}\ C : C}\ (NewClass) \qquad \frac{\begin{array}{l}\Gamma \vdash \tau \diamond_{ty} \\ \Gamma \vdash obj : \tau \\ \Gamma \vdash arg_i : tys_i \quad (1 \le i \le n) \\ \Gamma \vdash \tau \lhd_{meth} (meth, AT), \rho \\ \Gamma \vdash tys \le_{vartys} AT\end{array}}{\Gamma \vdash obj\,.\,meth_{AT}(arg_1, \ \ldots, \ arg_n) : \rho}\ (Call)$$

Statements are checked against a given return type, and are not themselves assigned types.

$$\frac{\begin{array}{l}\Gamma \vdash var : \tau \\ \Gamma \vdash exp : \tau' \\ \Gamma \vdash \tau' \le \tau\end{array}}{\Gamma, \rho \vdash (var\ \texttt{:=}\ exp)\checkmark}\ (Assign) \qquad \frac{\begin{array}{l}\Gamma \vdash b : \texttt{bool} \\ \Gamma, \rho \vdash t\checkmark \\ \Gamma, \rho \vdash e\checkmark\end{array}}{\Gamma, \rho \vdash (\texttt{if}\ b\ \texttt{then}\ t\ \texttt{else}\ e)\checkmark}\ (If)$$

$$\frac{\Gamma, \rho \vdash stmt_i\checkmark \quad (1 \le i \le n)}{\Gamma, \rho \vdash \{stmt_1; \ldots; stmt_n\}\checkmark}\ (Block) \qquad \frac{\Gamma \vdash e : \tau}{\Gamma, \rho \vdash e\checkmark}\ (Expr)$$

$$\frac{\begin{array}{l}\Gamma \vdash exp : \tau \\ \rho \ne \texttt{void} \\ \Gamma \vdash \tau \le \rho\end{array}}{\Gamma, \rho \vdash (\texttt{return}\ exp)\checkmark}\ (Return) \qquad \frac{\rho = \texttt{void}}{\Gamma, \rho \vdash \texttt{return}\checkmark}\ (Return')$$

$$\frac{\begin{array}{l}\Gamma \vdash b : \texttt{bool} \\ \Gamma, \rho \vdash stmt \checkmark\end{array}}{\Gamma, \rho \vdash (\texttt{while}\ \ b\ \ \texttt{do}\ \ stmt) \checkmark}\ (While)$$

When checking statements used as method bodies with non-`void` return types, we ensure that a `return` is always executed:[11]

```
always_returns(stmt) =
    match stmt with
       Block(stmts) -> existsl always_returns stmts
     | If(e,stmt1,stmt2) -> always_returns(stmt1) & always_returns(stmt2)
     | Return(ropt) -> true
     | _ -> false
```

This leads to the following rule for methods in class C. Method bodies are written here in lambda notation, and are typechecked with reference to C as C provides the type for the `this` variable:

$$\frac{\begin{array}{l}VE = \{\texttt{this} \mapsto C\} \oplus AT \\ \Gamma \oplus VE, \rho \vdash body \checkmark \\ \text{if } rt \neq \texttt{void} \text{ then } \mathsf{always\_returns}(body)\end{array}}{\Gamma, C, \rho, AT \vdash body \checkmark}$$

Finally, the rules for classes and programs are:

$$\frac{\begin{array}{l}\Gamma(C) = \langle C_{sup}, Is, fields, methods\rangle \\ \text{for each } m, AT \\ \quad methods(m, AT) = \rho\wedge \\ \quad methbods(m, AT) = bod \rightarrow \\ \quad \Gamma, C, \rho, AT \vdash bod \checkmark\end{array}}{\Gamma \vdash \texttt{class}\ \ C\ \ \texttt{extends}\ \ C_{sup}\ \ \texttt{implements}\ \ Is\ \ \{fields\,;\ \ methbods\} \checkmark}$$

$$\frac{\Gamma \vdash class_i \checkmark \quad (1 \leq i \leq n)}{\Gamma \vdash class_1\,;\ \ \ldots\,;\ \ class_n \checkmark}$$

6.5 Static Semantics for Java$_S$

The type-checking rules for the source language Java$_S$ are close to those for Java$_A$. The additional complicating factors are:

[11] The function `existsl` checks that a predicate is satisfied for some element of a list.

- *Deterministic Algorithm.* The Java$_S$ typechecking rules must represent a practical type-checking algorithm, while the rules for Java$_A$ simply check the validity of a type-assignment that can be derived from a successful application of the Java$_S$ rules.

- *Static Resolution.* Java implementations disambiguate field and method references at compile-time. Method calls may be statically overloaded (not to be confused with the object oriented late-binding mechanism), and fields may be hidden by superclasses.

Constraints are placed on types appearing in the source to accurately reflect the Java language. We omit the typing rules for Java$_S$, though they are formalised in Declare. The rules to annotate Java$_S$ ($\rightsquigarrow_{ann}$) to produce Java$_A$ are similar to the Java$_S$ type-checking rules and again we omit them here (see also Drossopoulou and Eisenbach's work [DE97a]).

6.6 The Runtime Semantics

We follow Drossopoulou and Eisenbach and model execution by a transition semantics, i.e. a "small step" rewrite system [Plo91]. A small step system is chosen over a "big step" (evaluation semantics) because it enables us to state substantially stronger results about the runtime machine — in particular we can prove that the abstract machine does not "get stuck" (see the liveness result in Chapter 7). This cannot be done with a big step semantics. Small step systems also give meaning to non-terminating and non-deterministic programs, and clearly we would like our model to be extendable to non-deterministic Java constructs such as threads. However using a small step system does impose significant overheads in the safety portion of the type soundness proof, precisely because certain intermediary configurations arise that need not be considered in a big step system.

6.6.1 Configurations

A *configuration* (t, s) of the runtime system has a *term* t and a *state* s. The term represents both expressions yet to be evaluated and the partial results of terms evaluated so far. Because of this, the term language must be extended to include addresses, void values and incomplete method invocations. We merge variables into the term structure and deal with three kinds of terms: an expression, a list of expressions, and a statement.[12] The syntax for runtime terms is shown in Figure 6.7.

The program state $s = (\phi, \hbar)$ consists of a *frame* ϕ of local variables and a *heap* $\hbar$ containing objects and arrays. In Java, local variables are mutable, but only one frame of variables is active at any one time, hence we cannot access locations further up the stack.

[12] In principle the "top level" configuration always contains an expression since Java begins execution with the `main` method from a given class.

$$
\begin{array}{lcll}
\textit{configuration} & = & (\textit{rexp} \mid \textit{rexp list} \mid \textit{rstmt}) \times \textit{state} & \\
& \mid & (\textit{exn-name} \times \textit{state})_{\mathsf{exn!}} & \\
& \mid & (\textit{value} \times \textit{state})_{\mathsf{return!}} & \\
\textit{state} & = & \langle \text{frame: } (\textit{id} \overset{\textit{table}}{\longmapsto} \textit{val}), & \\
& & \quad \text{heap: } \textit{addr} \overset{\textit{table}}{\longmapsto} \textit{heap-object} \rangle & \\
\textit{heap-object} & = & \ll (C_1, \textit{fld}_1) \mapsto \textit{val}_1, \ldots, (C_n, \textit{fld}_n) \mapsto \textit{val}_n \gg^C & (\textit{object}) \\
& \mid & [\![\textit{val}_0, \ldots, \textit{val}_{n-1}]\!]^\tau & (\textit{array})
\end{array}
$$

Figure 6.6: The Runtime Machine: Configurations and State

rval	=	*literal*	(literal value)
	|	*addr*	(pointers)
	|	`null`	(null pointer)
rexp	=	*rval*	(simple value)
	|	*id*	(local variable lookup)
	|	$\textit{rexp}_C$`.`*fld*	(field lookup)
	|	*rexp*`[`*rexp*`]`	(array lookup)
	|	*rexp*`.`M_{AT}`(`*rexp*`*)`	(method call)
	|	`new` C	(object creation)
	|	`new` *type*`[`*rexp*`]+[]*`	(array creation)
	|	$\{\textit{rstmt}\}_{\textit{frame}}$	(active method invocations)
rstmt	=	`if` *rexp* `then` *rstmt* `else` *rstmt*	(conditional)
	|	`while` *rexp* `do` *rstmt*	(while)
	|	`return` *rexp*	(return)
	|	*id* `:=` *rexp*	(local variable assignment)
	|	*rexp*`.[`C`]`*fld* `:=` *rexp*	(field assignment)
	|	*rexp*`[`*rexp*`]` `:=` *rexp*	(array assignment)
	|	$\{\textit{rstmt}_1; \ldots; \textit{rstmt}_n;\}$	(block)
	|	*rexp*	

Figure 6.7: The syntax of runtime terms

Heap objects are annotated with types for runtime typechecking (in the case of arrays this is the type of values stored in the array). The symbol $\oplus$ denotes replacing the active frame, while $s(id)$ and $s(addr)$ are the obvious lookups.

Global parameters to the rewrite system include an environment Γ (containing the class and interface hierarchies, needed for runtime typechecking) and the program p being executed. The latter contains Java_A terms: each time a method is executed we create a Java_R term for the body of that method.

6.6.2 The Term Rewrite System

A configuration is progressively modified by making reductions. The rewrite system thus specifies an abstract machine, which is an inefficient but simple interpreter for our subset of Java. The reduction of terms $\leadsto_{(\Gamma,p)}$ is specified by three relations, one for each kind of configuration (`exp_reduces`, `exps_reduce`, `stmt_reduces` in Declare). We typically omit the parameters Γ and p.

Ground Terms

A term is *ground* if it is in normal form, i.e. when no further reduction can be made.

- An expression e is ground *iff* it is a value, which we denote by k, b or v for an integer, boolean or arbitrary value;
- A list of expressions is ground *iff* all the expressions are ground;
- A statement is ground *iff* it is an empty block of statements or a ground expression.

Transfer of Control

The right-hand-side of a reduction may be either a regular configuration, or a configuration that represents a transfer of control because of an exception or a `return` statement (marked with exn! or return!). We do not list all the rules for propagating exceptions or return statements here — examples of each are:

$$\frac{arr, s \leadsto (exn, s')_{\mathsf{exn!}}}{arr\,\texttt{[}idx\texttt{]}, s \leadsto (exn, s')_{\mathsf{exn!}}} \qquad \frac{stmt, s \leadsto (rval, s')_{\mathsf{return!}}}{\{stmt\,;stmts\}, s \leadsto (rval, s')_{\mathsf{return!}}}$$

Note that transfer of control happens in a "bigstep" fashion, i.e. it takes only one reduction to transfer control to the handling location. This is because no particularly interesting intermediary configurations arise during transfer of control.

Redex Rules

"Redex" rules serve to navigate to the location where we next reduce a term, and thus define evaluation order. For example, the redex rules for array access are:

$$\frac{arr, s \rightsquigarrow arr', s'}{arr\,\texttt{[}idx\texttt{]}, s \rightsquigarrow arr'\,\texttt{[}idx\texttt{]}, s'} \qquad \frac{idx, s \rightsquigarrow idx', s'}{v\,\texttt{[}idx\texttt{]}, s \rightsquigarrow v\,\texttt{[}idx'\texttt{]}, s'}$$

For brevity we omit redex rules from here on, except where they relate to catching a transfer of control.

Array Access

Once the component expressions of an array access have been fully reduced we resolve the access as follows:

$$\frac{}{\texttt{null[}v\texttt{]}, s \rightsquigarrow (\texttt{NullExc}, s)_{\mathsf{exn!}}} \qquad \frac{k < 0}{addr\,\texttt{[}k\texttt{]}, s \rightsquigarrow (\texttt{IndOutBndExc}, s)_{\mathsf{exn!}}}$$

$$\frac{s(addr) = \texttt{[[}val_0, \ldots, val_{n-1}\texttt{]]}^{\tau} \quad k \geq n}{addr\,\texttt{[}k\texttt{]}, s \rightsquigarrow (\texttt{IndOutBndExc}, s)_{\mathsf{exn!}}} \qquad \frac{s(addr) = \texttt{[[}val_0, \ldots, val_{n-1}\texttt{]]}^{\tau} \quad 0 \leq k < n}{addr\,\texttt{[}k\texttt{]}, s \rightsquigarrow v_k, s}$$

Field and Local Variable Access

$$\frac{}{\texttt{null}.\mathit{fld}_C, s \rightsquigarrow (\texttt{NullExc}, s)_{\mathsf{exn!}}} \qquad \frac{\begin{array}{l} s(addr) = \ll vals \gg^{C'} \\ vals(C, \mathit{fld}) = v \end{array}}{addr.\mathit{fld}_C, s \rightsquigarrow v, s} \qquad \frac{s(id) = v}{id, s \rightsquigarrow v, s}$$

Object and Array Creation

$$\frac{\begin{array}{l} addr \text{ is fresh in } s \\ \mathit{flds} = \{\mathit{fldspec} \mid \Gamma, C \lhd_{\mathit{allfields}} \mathit{fldspec}\} \\ obj = \ll \text{initial values for } \mathit{flds}^C \gg \\ s' = s \leftarrow (addr, obj) \end{array}}{\texttt{new } C, s \rightsquigarrow addr, s'} \qquad \frac{0 \leq i < \mathsf{len}(\vec{k}) \quad \vec{k}_i < 0}{\texttt{new } \tau\overrightarrow{\texttt{[}k\texttt{]}}\,\texttt{[]}^m, s \rightsquigarrow (\texttt{BadSizeExc}, s)_{\mathsf{exn!}}}$$

$$\frac{\begin{array}{l} \forall 0 \leq i < \mathsf{len}(\vec{k}).\ \vec{k}_i \geq 0 \\ (s', addr) = \mathsf{Alloc}(s, \tau, \vec{k}, m) \end{array}}{\texttt{new } \tau\overrightarrow{\texttt{[}k\texttt{]}}\,\texttt{[]}^m, s \rightsquigarrow addr, s'}$$

Here Alloc recursively allocates $k_1 \times \ldots k_{n-1}$ arrays that contain initial values appropriate for the type $\tau\,[\,]^m$. This process is described in detail in [GJS96].[13]

The heap is not garbage collected. A garbage collection rule allowing the collection of inaccessible items could be added. Note garbage collection is semantically visible in Java because of the presence of `finally` methods.

Method Call

$$\frac{}{\texttt{null}.meth_{AT}(\vec{v}), s \leadsto (\texttt{NullExc}, s)_{\mathsf{exn!}}} \qquad \frac{\begin{array}{l}\mathsf{Tag}(s, addr) = \tau \\ \mathsf{MethBody}(meth, AT, \tau, p) = \lambda\vec{x}.body \\ \phi = \{\vec{x} \mapsto \vec{v}, \texttt{this} \mapsto addr\}\end{array}}{\texttt{addr}.meth_{AT}(\vec{v}), s \leadsto \{body\}_{\phi}, s}$$

Tag finds the type tag for the array or object at the given address. MethBody($meth$, AT, τ, p) implements dynamic dispatch: it finds the method body with name $meth$ and type signature AT relative to the type τ.

The result of calling a method is a method invocation record. These may be nested, and thus the term structure effectively records the stack of invocations.

Active Method Invocations

Inside active method invocation blocks we replace the frame of local variables. Transfers of control due to a `return` are also handled here.

$$\frac{body, (\phi, \hbar) \leadsto body, (\phi', \hbar')}{\{body\}_{\phi}, (\phi_0, \hbar) \leadsto \{body'\}_{\phi'}, (\phi_0, \hbar')} \qquad \frac{body, (\phi, \hbar) \leadsto (rval, (\phi', \hbar'))_{\mathsf{return!}}}{\{body\}_{\phi}, (\phi_0, \hbar) \leadsto rval, (\phi_0, \hbar')}$$

Lists of Expressions

Vectors of expressions are reduced to values prior to method call and array creation, using just one redex rule:

$$\frac{e_i, s \leadsto e_i', s'}{(v_1, \ldots, v_{i-1}, e_i, \ldots, e_n), s \leadsto (v_1, \ldots, v_{i-1}, e_i' \ldots, e_n), s'}$$

[13] This model of array creation should be modified if threads or constructors are considered. Array creation is not atomic with respect to thread execution, may execute constructors (and thus may not even terminate), and may raise an out-of-memory exception.

Block, If, While and Return Statements

The non-redex rules are:

$$\frac{\mathsf{stmt\_ground}(stmt)}{\{stmt\,;stmts\}, s \rightsquigarrow \{stmts\}, s} \qquad \frac{\text{if } b \text{ then } stmt = stmt_1 \text{ else } stmt = stmt_2}{(\texttt{if } b \texttt{ then } stmt_1 \texttt{ else } stmt_2), s \rightsquigarrow stmt, s}$$

$$\frac{}{(\texttt{while } e \texttt{ do } stmt), s \rightsquigarrow (\texttt{if } e \texttt{ then } \{stmt\,;\texttt{while } e \texttt{ do } stmt\} \texttt{ else } \{\}), s}$$

$$\frac{}{\texttt{return } v, s \rightsquigarrow (v, s)_{\mathsf{return!}}} \qquad \frac{}{\texttt{return}, s \rightsquigarrow (\texttt{void}, s)_{\mathsf{return!}}}$$

Assign to Arrays

The rules for assigning to arrays are similar to the rules for resolving array accesses, except, of course, when the action is resolved. For brevity we omit the rules that detect null pointers and array bounds errors.

Java performs runtime typechecks at just two places: during array assignment, and when casting reference values. Runtime typechecking is needed for array assignment because the type available on the left may become arbitrarily narrower. Casts are not covered in this case study: they are a trivial extension once runtime checking for arrays is in place. The partial function $\mathsf{Typecheck}$ checks that an address value $addr$ to be stored is compatible with the type tag attached to a target array τ, i.e. that $\Gamma \vdash \mathsf{Tag}(\hbar, addr) \leq \tau$[14]

$$\frac{\begin{array}{l} s(addr) = [\![val_0, \ldots, val_{n-1}]\!]^\tau \\ 0 \leq k < n \\ \mathsf{Typecheck}(\Gamma, s, v, \tau) \\ s' = \text{“replace } val_k \text{ with } v \text{ in } s\text{”} \end{array}}{(addr[k] \texttt{ := } v), s \rightsquigarrow \texttt{void}, s'} \qquad \frac{\begin{array}{l} s(addr) = [\![val_0, \ldots, val_{n-1}]\!]^\tau \\ 0 \leq k < n \\ \neg\mathsf{Typecheck}(\Gamma, s, v, \tau) \end{array}}{(addr[k] \texttt{ := } v), s \rightsquigarrow (\texttt{ArrayStoreExc}, s)_{\mathsf{exn!}}}$$

Assign to Fields and Local Variables

No runtime typechecking is required when assigning to fields or local variables, because, as we shall prove in the next chapter, the static checks are adequate.

$$\frac{\begin{array}{l} s(addr) = \ll vals \gg^{C'} \\ s' = \text{“replace } fld_C \text{ with } v \text{ in } vals\text{”} \end{array}}{(addr\,.fld_C \texttt{ := } v), s \rightsquigarrow \texttt{void}, s'} \qquad \frac{s' = \text{“replace } s(id) \text{ with } v \text{ in } s\text{”}}{(id \texttt{ := } v), s \rightsquigarrow \texttt{void}, s'}$$

[14] This notion of runtime type checking comes from Drossopoulou and Eisenbach's original work (weak conformance) and is really a little too strong: it allows the runtime machine to check the conformance of primitive values to primitive types. No realistic implementation of Java checks at runtime that a primitive type such as `int` fits in a given array slot.

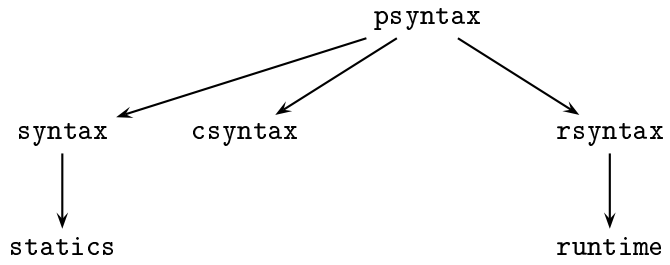


Figure 6.8: Organisation of the Model in Declare

6.7 The Model as a Declare Specification

So far we have described our model of Java$_S$ in the traditional manner — however, the model has, of course, been realised as Declare specifications. The model runs to around 2000 lines, and we have shown an extended excerpt in Appendix A. The dependency graph between files in the model is shown in Figure 6.8. The use of three similar versions of the language results in some duplication. However, the need for clarity was perceived to be greater than the need for brevity. Importantly, the Declare model could be easily read and understood by Drossopoulou and Eisenbach when shown to them.

We have discussed the use of code generation to validate the correctness of a Declare model against our informal expectations in Section 2.4. Declare produces a Mercury module for each article we have written. Test programs are expressed as higher order logic expressions.[15] Many errors were discovered by using these techniques (more than 15). The breakdown of these was roughly as follows:

- Around 5 variables that were only used once, because of some kind of typing mistake.
- Around 5 Mercury mode violations, because of typing mistakes and some logical errors.
- Around 5 logical mistakes in the typing and runtime rules, detected when actually executing expressions.

It is clear that validation of this kind plays an essential role in stress-testing the integrity of such a development. Further, the same tests can be used as the semantics

[15] Better would be the ability to parse, compile and run programs directly from concrete syntax. Such a facility could be added, perhaps by using Boulton's Claret tools [Bou97].

is extended and modified. After making some modifications to the semantics we detected several new mistakes by re-running earlier test cases.

Chapter 7

Type Soundness for Java$_S$

In this chapter we describe the type soundness properties we proved for Java$_S$ and state the major lemmas used in their proof. We also present extracts from the Declare proofs, and discuss the errors found while performing these proofs.

7.1 Conformance

Informally, type soundness states that a well-typed Java program does not "go wrong" at runtime, in the sense that it never reaches a state that violates conditions implied by the typing rules. One aspect of type soundness is captured in the following statement from the Java Language Specification [GJS96]:

> The type [of a variable or expression] limits the possible values that the variable can hold or the expression can produce at runtime. If a runtime value is a reference that is not `null`, it refers to an object or array that has a class ... that will necessarily be compatible with the compile-time type.

The task of this chapter is to define what is meant here by "limits" and "compatible," a notion we call *conformance* ($\leq$:). We then show that conformance is an invariant of the abstract runtime machine described in the previous chapter. Like all invariants, it is a two-edged sword:

- Conformance must be strong enough to ensure the machine can always make a transition from a conforming configuration;
- Conformance must be liberal enough to ensure that every such transition results in another conforming configuration.

Conformance is defined for all major artifacts of the runtime machine, beginning with Java$_R$ values, expressions and statements. The rules for conformance naturally bear a similarity to the typing rules for Java$_A$: e.g. conformance does not assign types, but rather checks conformance given a particular type. However, unlike Java$_A$:

- Conformance is also defined for frames, heaps, states and configurations, relative to type assignments for these constructs;
- We define conformance "up to widening," so that, for example, any runtime reference object conforms with the type `Object`, and runtime objects of actual class A are compatible with type B if $\Gamma \vdash A \sqsubseteq_{class} B$.
- In a few places (especially assignment), the rules for conformance must be weaker than one might think. This accounts for certain intermediary states that arise during computation but are not acceptable as inputs.

Without further ado, we proceed to the necessary definitions.

Frame, Heap and State Typings

A *frame typing* ϕ_τ is a partial function that assigns a typing upper bound to each storage location in a frame ϕ. Similarly a *heap typing* $\hbar_\tau$ assigns a type to each storage location in a heap $\hbar$.[1] A *state typing* s_τ is a frame typing and a heap typing. Well-formedness ($\Gamma \vdash \_ \diamond$) extends to frame, heap and state typings in the natural way. For heap typings we impose the constraint that all types in the assignment must be reference types.

Value Conformance

A value v conforms to a type τ with respect to a type environment Γ and heap typing $\hbar_\tau$ according to the rules:

$$\frac{}{\Gamma, \hbar_\tau \vdash \texttt{void} \leq:_{val} \texttt{void}} \qquad \frac{pt \text{ is the type for literal } pval}{\Gamma, \hbar_\tau \vdash pval \leq:_{val} pt} \qquad \frac{\tau \text{ is any reference type}}{\Gamma, \hbar_\tau \vdash \texttt{null} \leq:_{val} \tau}$$

$$\frac{\hbar_\tau(addr) = \tau}{\Gamma, \hbar_\tau \vdash addr \leq:_{val} \tau} \qquad \frac{\Gamma, \hbar_\tau \vdash v \leq:_{val} \tau' \quad \Gamma \vdash \tau' \leq \tau}{\Gamma, \hbar_\tau \vdash v \leq:_{val} \tau}$$

Note the last rule gives value conformance up to widening.

[1] As it happens the types in a heap typing are exact rather than upper bounds. Drossopoulou and Eisenbach preferred not to use a heap typing and instead recovered the heap type information from the runtime type tags stored alongside objects in the heap. We used a heap typing in later versions of our work for consistency and to preserve the separation of concerns between runtime and static type information. That heap static types can be fully reverse engineered from the runtime tags in Java is somewhat unusual, and we have chosen an approach that works when this is not possible.

Frame and Heap Conformance

An object conforms to a class type C if its type tag is C and each expected field value is present and conforms to the appropriate type. Similarly, an array conforms to a type if its type tag matches and its values all conform:

$$\frac{\begin{array}{l}\mathsf{dom}(vals) = \{fidx \mid \Gamma \vdash C \lhd_{allfields} (fidx, \tau)\} \\ \forall fidx, \tau.\ \Gamma \vdash C \lhd_{allfields} (fidx, \tau) \rightarrow \Gamma, \hbar_\tau \vdash vals\,(fidx) \leq:_{val} \tau\end{array}}{\Gamma, \hbar_\tau \vdash \ll vals \gg^C \leq:_{heapobj} C}$$

$$\frac{\forall i.\ 0 \leq i < n \rightarrow \Gamma, \hbar_\tau \vdash val_i \leq:_{val} \tau}{\Gamma, \hbar_\tau \vdash [\![val_0, \ldots, val_{n-1}]\!]^\tau \leq:_{heapobj} \tau[]}$$

A heap $\hbar$ conforms to a heap typing $\hbar_\tau$ if each has the same domain and each heap object (either an array or object) conforms. Similarly, a frame ϕ conforms to ϕ_τ (relative to a heap typing $\hbar_\tau$) if each has the same domain and their contents conform point-wise.

$$\frac{\begin{array}{l}\mathsf{dom}(\hbar) = \mathsf{dom}(\hbar_\tau) \\ \forall addr \in \hbar.\ \Gamma, \hbar_\tau \vdash \hbar(addr) \leq:_{heapobj} \hbar_\tau(addr)\end{array}}{\Gamma \vdash \hbar \leq:_{heap} \hbar_\tau} \qquad \frac{\begin{array}{l}\mathsf{dom}(\phi) = \mathsf{dom}(\phi_\tau) \\ \forall id \in \phi.\ \Gamma, \hbar_\tau \vdash \phi(id) \leq:_{val} \phi_\tau(id)\end{array}}{\Gamma, \hbar_\tau \vdash \phi \leq:_{frame} \phi_\tau}$$

Finally, a state conforms to a state typing if its components conform:

$$\frac{\begin{array}{l}\Gamma \vdash \hbar \leq:_{heap} \hbar_\tau \\ \Gamma, \hbar_\tau \vdash \phi \leq:_{frame} \phi_\tau\end{array}}{\Gamma \vdash (\phi, \hbar) \leq: (\phi_\tau, \hbar_\tau)}$$

Term Conformance

Conformance of expressions and statements is measured relative to a state typing. While most the rules ensure essentially the same typing conditions as Java$_A$, we have added rules for incomplete method invocations and to ensure the relation is monotonic up to widening. Finally the rules for assignment are subtly different, something which is essential as we shall see.

The rules for runtime expressions are:

$$\frac{\phi_\tau(id) = \tau}{\Gamma, (\phi_\tau, \hbar_\tau) \vdash id \leq:_{exp} \tau}\ (LocalAccess) \qquad \frac{\begin{array}{l}\Gamma, s_\tau \vdash arr \leq:_{exp} \tau[] \\ \Gamma, s_\tau \vdash idx \leq:_{exp} \texttt{int}\end{array}}{\Gamma, s_\tau \vdash arr[idx] \leq:_{exp} \tau}\ (ArrayAccess)$$

$$\frac{\begin{array}{l}\Gamma, s_\tau \vdash obj \leq:_{exp} C_0 \\ \Gamma \vdash C_0 \lhd_{allfields} (fld, C), \tau)\end{array}}{\Gamma, s_\tau \vdash obj\,.fld_C \leq:_{exp} \tau}\ (FieldAccess) \qquad \frac{\Gamma, \hbar_\tau \vdash v \leq:_{val} \tau}{\Gamma, (\phi_\tau, \hbar_\tau) \vdash v \leq:_{exp} \tau}\ (Value)$$

$$\frac{\begin{array}{l}\Gamma \vdash \tau \diamond_{ty} \\ \Gamma, s_\tau \vdash d_i \leq:_{exp} \texttt{int} \quad (1 \leq i \leq n)\end{array}}{\Gamma, s_\tau \vdash \texttt{new } \tau\texttt{[}d_1\texttt{]} \ldots \texttt{[}d_n\texttt{]} \leq:_{exp} \tau\texttt{[]}^n} \ (NewArray) \qquad \frac{\Gamma \vdash C \diamond_{class}}{\Gamma, s_\tau \vdash \texttt{new } C \leq:_{exp} C} \ (NewClass)$$

$$\frac{\begin{array}{l}\Gamma \vdash \tau \diamond_{ty} \\ \Gamma, s_\tau \vdash obj \leq:_{exp} \tau \\ \Gamma, s_\tau \vdash arg_i \leq:_{exp} AT_i \quad (1 \leq i \leq n) \\ \Gamma \vdash \tau \lhd_{meth} (meth, AT), \rho\end{array}}{\Gamma, s_\tau \vdash obj\texttt{.}meth\texttt{(}arg_1\texttt{, } \ldots\texttt{, } arg_n\texttt{)} \leq:_{exp} \rho} \ (Call)$$

The new rules for expressions are:

$$\frac{\begin{array}{l}\Gamma \vdash \phi'_\tau \diamond_{frame\text{-}type} \\ \Gamma, \hbar_\tau \vdash \phi' \leq:_{frame} \phi'_\tau \\ \Gamma, (\phi'_\tau, \hbar_\tau), \rho \vdash body \checkmark \\ \text{if } \rho \neq \texttt{void} \text{ then } \mathsf{always\_returns}(body)\end{array}}{\Gamma, (\phi_\tau, \hbar_\tau) \vdash \{body\}_{\phi'} \leq:_{exp} \rho} \ (ActiveCall) \qquad \frac{\begin{array}{l}\Gamma, s_\tau \vdash e \leq:_{exp} \tau' \\ \Gamma \vdash \tau' \leq \tau\end{array}}{\Gamma, s_\tau \vdash e \leq:_{exp} \tau} \ (Mono)$$

Note that checking conformance for an incomplete method call requires a ϕ'_τ for the frame ϕ'.

The rules for runtime statements are as follows. We omit the rules for `while`, `if` and block statements for brevity.

$$\frac{\begin{array}{l}\phi_\tau(id) = \tau \\ \Gamma, (\phi_\tau, \hbar_\tau) \vdash exp \leq:_{exp} \tau' \\ \Gamma \vdash \tau' \leq \tau\end{array}}{\Gamma, (\phi_\tau, \hbar_\tau), \rho \vdash (id \texttt{ := } exp) \checkmark} \ (Local) \qquad \frac{\begin{array}{l}\Gamma, s_\tau \vdash obj \leq:_{exp} C_0 \\ \Gamma \vdash C_0 \lhd_{allfields} ((fld, C), \tau) \\ \Gamma, s_\tau \vdash exp \leq:_{exp} \tau' \\ \Gamma \vdash \tau' \leq \tau\end{array}}{\Gamma, s_\tau, \rho \vdash (obj\texttt{.}fld_C \texttt{ := } exp) \checkmark} \ (Field)$$

$$\frac{\begin{array}{l}\Gamma, s_\tau \vdash arr \leq:_{exp} \tau\texttt{[]} \\ \Gamma, s_\tau \vdash idx \leq:_{exp} \texttt{int} \\ \Gamma, s_\tau \vdash exp \leq:_{exp} \tau'\end{array}}{\Gamma, s_\tau, \rho \vdash (arr\texttt{[}idx\texttt{]} \texttt{ := } exp) \checkmark} \ (Array) \qquad \frac{\Gamma, s_\tau \vdash e \leq:_{exp} \tau}{\Gamma, s_\tau, \rho \vdash e \checkmark} \ (Expr)$$

$$\frac{\Gamma, s_\tau \vdash exp \leq:_{exp} \rho}{\Gamma, s_\tau, \rho \vdash (\texttt{return } exp) \checkmark} \ (Return)$$

Configuration Conformance

A configuration of the runtime machine (e, s) conforms to a configuration typing (τ, s_τ) if and only if both the state and the expression conform. An exceptional configuration conforms if the state conforms (exceptions do not carry values in our model), and a return configuration conforms if the return value conforms to the expected return type:

$$\frac{\Gamma \vdash s \leq: s_\tau \quad \Gamma, s_\tau \vdash e \leq:_{exp} \tau}{\Gamma \vdash (e,s) \leq: (\tau, s_\tau)} \qquad \frac{\Gamma \vdash s \leq: s_\tau}{\Gamma \vdash (e,s)_{\mathsf{exn!}} \leq: (\tau, s_\tau)} \qquad \frac{\Gamma \vdash s \leq: s_\tau \quad \Gamma, s_\tau \vdash rval \leq:_{val} \rho}{\Gamma, \rho \vdash (rval, s)_{\mathsf{return!}} \leq: s_\tau}$$

We omit similar definitions for exceptional configurations and configurations where the term is a statement or a vector of expressions.

Finally, we say that $\hbar_\tau$ is smaller than $\hbar'_\tau$ ($\hbar_\tau \trianglelefteq_{heap} \hbar'_\tau$) if and only if $\hbar_\tau$ is a subfunction of $\hbar'_\tau$, i.e. its domain is no greater and within its domain the functions agree. The intuition is that $\hbar'_\tau$ is the typing assigned after we allocate new elements in the heap. Similarly one state typing $s_\tau (= (\phi_\tau, \hbar_\tau))$ is smaller than another $s'_\tau = (\phi'_\tau, \hbar'_\tau)$ ($s_\tau \trianglelefteq_{state} s'_\tau$) if $\phi_\tau = \phi'_\tau$ and ($\hbar_\tau \trianglelefteq_{heap} \hbar'_\tau$), and similarly for configuration types. Note these relations are simple concepts, unrelated to widening, and $x \trianglelefteq y$ simply mean "x has less cells allocated than y, but is otherwise identical."

7.2 Safety, Liveness and Annotation

We are now in a position to state the type soundness results. As mentioned before, we distinguish between a *safety* property (subject reduction) and a *liveness* property:

Theorem 3 Safety *For a well-formed type environment Γ, an annotated, typechecked program p and a configuration C that conforms to C_τ, then if C can make a transition to some C' there exists a larger C'_τ such that C' conforms to C'_τ. That is, if $\vdash \Gamma \diamond_{tyenv}$, $\Gamma \vdash p \diamond$, $\Gamma \vdash C_\tau \diamond_{ty}$, $\Gamma \vdash C \leq C_\tau$ and $C \rightsquigarrow_{(\Gamma,p)} C'$ then there exists C'_τ such that*

- $\Gamma \vdash C'_\tau \diamond$
- $C_\tau \trianglelefteq_{cfg} C'_\tau$.
- $\Gamma \vdash C' \leq: C'_\tau$

Note we assume a reduction is made, rather than proving that one exists. This distinguishes the safety property from the liveness property. In the presence of nondeterminism it is not sufficient to prove that a safe transition exists: we want to show that all possible transitions are safe.

Theorem 4 Liveness *For a well-formed type environment Γ, an annotated, typechecked program p and a C that conforms to C_τ, then if the term in C is not ground then C can make a transition to some C'.*

To complement the type soundness proof, we prove that the process of annotation preserves types:

Theorem 5 Annotation *For a well-formed type environment* Γ *and a typechecked Java*$_S$ *program* p *then there exists a unique* p' *such that* $\Gamma \vdash p \rightsquigarrow_{ann} p'$. *Furthermore* p' *typechecks as a Java*$_A$ *program.*

7.2.1 Key Lemmas

The following is a selective list of the lemmas that form the basis for the type soundness results. These have, naturally, been checked using Declare.

All declared classes and interfaces are well-founded

That is, if $\Gamma \vdash C \diamond_{class}$ then $\Gamma \vdash C \sqsubseteq_{class} \texttt{Object}$, and similarly each declared interface has a chain of superinterfaces that ultimately terminates with an interface with no parents.

`Object` is the least class

If $\Gamma \vdash \texttt{Object} \sqsubseteq_{class} C$ then $C = \texttt{Object}$.

Widening is transitive and reflexive

The $\sqsubseteq_{class}$, $\sqsubseteq_{intf}$ and $\leq$ relations are all transitive and reflexive for well-formed environments and types.

Narrower types have matching structure

If $\Gamma \vdash \tau' \leq \tau$ then

- If τ is an array type $\sigma\texttt{[]}^n$ then τ' is an array type $\sigma'\texttt{[]}^n$ where $\Gamma \vdash \sigma' \leq \sigma$.
- Similarly, only primitive types are narrower than themselves, and only subclasses are narrower than class types that are not `Object`.

Conforming values have matching structure

If $\Gamma, \hbar_\tau \vdash v \leq:_{val} \tau$ then

- If τ is a primitive type then v is a matching primitive value.
- If τ is an array type $\sigma\texttt{[]}^n$ then v is either `null` or an address $addr$ with $\hbar_\tau(addr) = \sigma'\texttt{[]}^n$ and $\Gamma \vdash \sigma \leq \sigma'$.
- Similarly, if τ is a non-`Object` class type C then v is either `null` or an address $addr$ with $\hbar_\tau(addr) = C'$ and $\Gamma \vdash C \sqsubseteq_{class} C'$, and so on.

Field indexes are unique

That is, the relation $\lhd_{allfields}$ finds at most one field for each field index.

Compatible fields and methods exist at subtypes

Methods and fields visible at one type must still be visible at narrower types, though with possibly narrower return types. That is, if

$\Gamma \vdash C_1 \sqsubseteq_{class} C_0$ and
$\Gamma \vdash C_0 \lhd_{allfields} (fidx, ty_{fld})$

then $\Gamma \vdash C_1 \lhd_{allfields} (fidx, ty_{fld})$.

Similarly if $\Gamma \vdash \tau \lhd_{meth} (m, AT), \rho$ and $\Gamma \vdash \tau' \leq \tau$ then there exists some ρ' with

$\Gamma \vdash \tau' \lhd_{meth} (m, AT), \rho'$ and
$\Gamma \vdash \rho' \leq \rho$.

Method lookup behaves correctly

Fetching the annotated body of a method using dynamic dispatch from τ results in a method of the type we expect, and furthermore the method was typechecked with reference to a `this`-variable type that is compatible with τ, i.e. if

$\Gamma \vdash \tau \lhd_{meth} (m, AT), \rho$ and
$\mathsf{MethBody}(m, AT, \tau, p) = meth\_body$

then there exists C' such that

$\Gamma \vdash \tau \leq C'$ and
$\Gamma, C' \vdash meth\_body \checkmark$.

Compilation behaves correctly

If $\Gamma \vdash mbody : ty_{ret}$ and $\Gamma \vdash mbody \rightsquigarrow_{comp} rmbody$ then $\Gamma \vdash rmbody : ty_{ret}$, where $\rightsquigarrow_{comp}$ is the process of turning a Java$_A$ term into a Java$_R$ term. Note compilation is an almost trivial process in the current system, so this lemma is not difficult.

Relations are preserved under narrowing of heaps.

This holds for the value, frame, expression and statement conformance relations.

Atomic state manipulations create conforming states

We prove this for all primitive state manipulations, including object and array allocation, field, array and local variable assignment. The case for array allocation involves a double induction because of the nested loop used to allocate multi-dimensional arrays.

Method call creates a conforming state

That is, the frame allocated for a method call conforms.

Runtime typechecking is adequate

That is, typecheck guarantees that a value conforms to the given type.

7.3 Example Proofs in Declare

We now outline the Declare proofs of some of the theorems from in the previous section. The reader should keep in mind that when these proofs were begun, the only guide available was the rough outline in [DE97b], and this was based on a formulation of the problem that was subsequently found to contain errors. Thus the process was one of proof discovery rather than proof transcription. For each proof we shall a short outline in vernacular mathematics, followed by the Declare proof script, to demonstrate how proof outlines are transcribed. Although a *very* powerful automated routine may be able to do away with most of our proof scripts *after the fact*, the very process of writing them typically corrected significant errors that would confound even the best prover.

7.3.1 Example 1: Inherited Fields Exist

Inherited Fields Exist

> Given well-formed Γ, C_0, C_1, τ and a field descriptor *fidx* where
>
> $$\Gamma \vdash C_1 \sqsubseteq_{class} C_0$$
> $$\Gamma \vdash C_0 \lhd_{allfields} (fidx, v_\tau)$$
>
> then $\Gamma \vdash C_1 \lhd_{allfields} (fidx, v_\tau)$. That is, field existence is preserved at subclasses.

The proof is by induction over the derivation of the $\sqsubseteq_{class}$ judgment: in each case the result follows using the rules (*Hit*) and (*Step*) from page 108.

Now, the proof in Declare is:

```
thm <inherited-fields-exist>
  if "TE wf_tyenv" <TE_wf>
     "TE |- C0 wf_class"
     "TE |- C1 wf_class"
     "TE |- C1 subclass_of C0" <subclass>
     "TE |- C0 has_field fspec"
then "TE |- C1 has_field fspec"
proof
  proceed by rule induction on <subclass> with C1 variable;
    case BaseC: qed by <has_field.Hit>;
    case StepC: qed by <has_field.Step>,
                       <wf_tyenv.class_superclass_declared> [<TE_wf>];
  end;
end;
```

In the step case of the induction we invoke a well-formedness condition (corresponding to (A.1) on page 109) in order to prove the intermediary class in the subclass relation is well-formed.

7.3.2 Example 2: Field Assignment

Field assignment preserves conformance.

Given well-formed Γ, C, C', ϕ, $\hbar_0$, $\hbar_1$, s_0, s_1, ϕ_τ, $\hbar_\tau$, s_τ, v, v_τ and a field descriptor *fidx* where

$s_0 = (\phi, \hbar_0)$ and $s_1 = (\phi, \hbar_1)$

$\Gamma \vdash s_0 \leq: s_\tau$

$\hbar_\tau(addr) = C'$

$\Gamma \vdash C' \lhd_{allfields} (fidx, v_\tau)$

$\Gamma, \hbar_\tau \vdash v \leq:_{val} v_\tau$

$\hbar_0(addr) = \ll fidx_1 \mapsto val_1, \ldots, fidx_n \mapsto val_n \gg^C$

$\hbar_1 = \hbar_0$ with $fidx \mapsto v$ at $addr$

then $\Gamma \vdash s_1 \leq: s_\tau$. That is, if s_1 is the result of a field assignment operation on s_0, then s_1 conforms to the same type bound as s_0.

Proof: Let $obj_0 = \ll (C_1, fld_1) \mapsto val_1, \ldots, (C_n, fld_n) \mapsto val_n \gg^C$ and obj_1 be the result of replacing the value of field *fidx* by v in obj_0. We have $\Gamma, \hbar_\tau \vdash obj_0 \leq:_{heapobj} C'$ because s_0 conforms at $addr$. This in turn means the type tags match, that is $C = C'$. The values inside obj_0 conform to the types as found by $\lhd_{allfields}$, as do the values inside obj_1 because the new value v conforms. Thus $\Gamma, \hbar_\tau \vdash obj_1 \leq:_{heapobj} C'$ and $\Gamma \vdash \hbar_1 \leq:_{heap} \hbar_\tau$ by straightforward application of the rules to derive these judgments and the result follows.

Now, the proof in Declare is:

```
thm <field-assign-conforms-lemma>
  if "TE wf_tyenv" <TE_wf>
     "TE |- C wf_class" <C_wf>
     "s0 = (frame0,heap0)"
     "s1 = (frame0,heap1)"
     "ST = (FT,HT)"
     "TE |- s0 state_conforms_to ST"
     "flookup HT taddr = Some(VT(ClassTy(C),0))" [autorw]
     "TE |- C has_field (fidx,vty)"
     "(TE,HT) |- sval rval_conforms_to Some(vty)"
     "flookup heap0 taddr = Some(OBJECT(fldvals0,C'))"
     "fldvals1 = fupdate fldvals0 (fidx,sval)"
     "heap1 = heap0 <?++ (taddr,OBJECT(fldvals1, C'))"
 then "TE |- s1 state_conforms_to ST";
proof
  let "obj0 = OBJECT(fldvals0,C')"
      "obj1 = OBJECT(fldvals1,C')";

  have "(TE,HT) |- obj0 heapobj_conforms_to VT(ClassTy(C),0)" <heapobj_conforms>
    by <state_conforms_to.heap>, <heap_conforms_to.rool>;
```

```
  have "C = C'"
    by <heapobj_conforms_to.object-tag-matches> [<heapobj_conforms>,"fldvals0","C'"];

  have "∀vty'. TE |- C has_field (fidx,vty') ↔ vty' = vty"  [rw] <x>
    by  <object-fields-form-graph> [<TE_wf>,<C_wf>],
        <frel_is_graph_rool> ["λfspec.  TE |- C has_field fspec"/R,"fidx"/x,"vty"/y];

  have "(TE,HT) |- fldvals1 fldvals_conform_to C"
    by  <heapobj_conforms_to.object-fields-conform> [<heapobj_conforms>,"fldvals0"],
        <fldvals_conform_to.rool> ["fldvals0","TE","HT","C"],
        <fldvals_conform_to.derive> ["fldvals1","TE","HT","C"], <x>;

  have "(TE,HT) |- obj1 heapobj_conforms_to VT(ClassTy(C),0)"
    by  <heapobj_conforms_to>;

  have "TE |- heap1 heap_conforms_to HT"
    by <heap_conforms_to>,<eq_fsets>,<state_conforms_to>,<in_fdomain>;

 qed by <state_conforms_to>;
end;
```

The proof has clearly required extra detail: but although we have had to reason about the uniqueness of field descriptors, otherwise the proof follows essentially the same outline. Note that many of the explicit instantiations are not required *post facto* (we leave them in after completing the proof simply because there is little point in taking them out).

Note also that we have survived without naming many local facts. This is because the proof obligations happen to be simple enough, so that implicitly including most facts at each justification does not significantly confuse the automated engine.

7.3.3 Example 3: Monotonicity of Value Conformance Under Allocation

Remember $\hbar_\tau \trianglelefteq_{heap} \hbar'_\tau$ simply means $\hbar'_\tau$ records types for some locations not mentioned in $\hbar_\tau$. Clearly the conformance relations for values and other terms are monotonic under this relation:

Value conformance is monotonic under $\trianglelefteq_{heap}$.

> Given well-formed Γ, $\hbar_\tau$, $\hbar'_\tau$, v and τ, where $\Gamma, \hbar_\tau \vdash v \leq:_{val} \tau$ and $\hbar_\tau \trianglelefteq_{heap} \hbar'_\tau$ then $\Gamma, \hbar'_\tau \vdash v \leq:_{val} \tau$.

The proof is by induction over the derivation of $\Gamma, \hbar_\tau \vdash v \leq:_{val} \tau$, and the only non-trivial case is typing for addresses, when we must use the appropriate property of $\trianglelefteq_{heap}$. The proof in Declare is:

```
thm <val_conforms_to-mono-lemma>
```

```
if "TE wf_tyenv"
    "HT0 htyping_leq HT1"
    "(TE,HT0) |- val rval_conforms_to ty" <conforms>
then "(TE,HT1) |- val rval_conforms_to ty";
proof
  proceed by rule induction on <conforms> with val,ty variable;
    case Prim: qed;
    case Null: qed;
    case Addr: qed by <htyping_leq.rool>, <in_fdomain>,
                      <rval_conforms_to.Addr>;
    case Trans: qed by <rval_conforms_to.Trans>;
    case Void: qed;
  end;
end;
```

7.4 Errors Discovered

In this section we describe an error in the Java language specification that we independently rediscovered during the course of this work. We also describe one major error and a noteworthy omission in Drossopoulou and Eisenbach's original presentation of the type soundness proof.

7.4.1 An Error in the Java Language Specification

In the process of finishing the proofs of the lemmas described in Section 7.2.1 we independently rediscovered a significant flaw in the Java language specification that had recently been found by developers of a Java implementation [PB97]. In theory the flaw does not break type soundness, but the authors of the language specification have confirmed that the specification needs alteration.

The problem is this: in Java, all interfaces and arrays are considered subtypes of the type `Object`, in the sense that a cast from an interface or array type to `Object` is permitted. The type `Object` supports several "primitive" methods, such as `hashValue()` and `getClass()` (there are 11 in total). The question is whether expressions whose *static* type is an interface support these methods.

By rights, interfaces should indeed support the `Object` methods — any class that actually implements the interface will support these methods by virtue of being a subclass of `Object`, or an array. Indeed, the Sun JDK toolkit allows calling these methods from static interface types, as indicated by the successful compilation (but not execution) of the code:

```
public interface I { }

public class Itest {
  public static void main(String args[]) {
    I a[] = { null, null };
    a[0].hashCode();
```

```
    a[0].getClass();
    a[0].equals(a[1]);
  }
}
```

However, the existing language specification states explicitly that interfaces *only support those methods listed in the interface or its superinterfaces*, and that there is no 'implicit' superinterface (i.e. there is no analogue to the 'mother-of-all-classes' `Object` for interfaces). To quote:

> The members of an interface are all of the following:
>
> - Members declared from any direct superinterfaces
> - Members declared in the body of the interface.
>
> ...
> There is no analogue of the class `Object` for interfaces; that is, while every class is an extension of class `Object`, there is no single interface of which all interfaces are extensions.
>
> [GJS96], pages 87 and 185

The error was detected when trying to prove the existence of compatible methods and fields as we move from a type to a subtype, in particular from the type `Object` to an interface type.

7.4.2 Runtime Typechecking, Array Assignments, and Exceptions

In Drossopoulou and Eisenbach's original formulation the type soundness property was stated along the following lines (emphasis added):

Theorem 6 *If a well-typed term t is not ground, then it rewrites to some t' (and a new state s and environment Γ). Furthermore, either t'* **eventually rewrites** *to an exception, or t' has some narrower type than t, in the new state and environment.*

The iterated rewriting was an attempted fix for a problem demonstrated by the following program:

```
void silly(C arr[], C s) {
   arr[1] = s;
}
```

At runtime, `arr` may actually be an array of some narrower type, say `C'` where `C'` is a subclass of `C`. Then the array assignment appears to become badly typed *before* the exception is detected, because during the rewriting the left side becomes a narrower type than the right. Thus they allow the exception to appear after a number of additional steps.

However, `arr` can become narrower, and then subsequently fail to terminate! Then an exception is never raised, e.g.

```
arr[loop()] = s;
```

The problem occurs in even simpler cases, e.g. when both `arr` and `s` have some narrower types `C'[]` and `C'`. Then, after the left side is evaluated, the array assignment appears badly typed, but will again be well typed after the right side is evaluated.

Fixing this problem requires a different understanding of the role of the types we assign to terms. Types for intermediary terms only exist to help express the type soundness invariant of the abstract machine, i.e. to define the allowable states that a well-typed execution can reach. In particular, the array assignment rule must be relaxed to allow what appear to be badly typed assignments, but which later get caught by the runtime typechecking mechanism.

This problem is an interesting case where the attempted re-use of typing rules in a different setting (i.e. the runtime setting rather than the typechecking setting) led to a subtle error, and one which we believe would only have been detected with the kind of detailed analysis that machine formalization demands. The mistake could not be missed in that setting! The difference between the Java_S and Java_R rules is clearly necessary in retrospect, but failure to grasp this can lead to subtle errors. For example, see the discussion on the `types` mailing list, where researchers were concerned that subject reduction does not hold for the Java source language [Typ98].

7.4.3 Side-effects on Types

A significant omission in Drossopoulou and Eisenbach's original proof was as follows: when a term has two or more subterms, e.g. *arr*`[`*idx*`] := `*e*, and *arr* makes a reduction to *arr*', then the types of *idx* and *e* may change (become narrower) due to side-effects on the state. This possibility had not originally been considered by Drossopoulou and Eisenbach, and requires a proof that heap locations do not change type (our notion of heap conformity suffices). The foremost of these lemmas has been mentioned in Section 7.2.1. This problem was only discovered while doing detailed machine checking of the rough proof outline.

7.5 Appraisal

The previous section has given several examples of Declare proofs from our case study. We now address the following rather important question: what effect did adopting declarative proof techniques have on the execution of the case study?

We have already described many of the small-scale contributions of declarative proof in Chapter 3. The same pros and cons we have described there were played Ott again and again in small ways throughout the development of the proofs. For example, the flexibility in decomposition provided by Declare was used many times throughout the case study, but similarly the number of terms quoted in Declare proofs was always relatively high.

We can now step back to look at methodological issues:

- *Proof Refinement.* The declarative proof style meant we could repeatedly refine approximate proof scripts, starting with notes and finishing with a machine-checkable script.

- *Maintainability.* The declarative proof style meant that it was often simple to chase through the exact ramifications of a small change to the model. Primarily this is because so much information is explicit in a declarative proof, and the effects of a change could often be predicted even before checking a single step of a proof, either by searching or typechecking proofs.

- *Robustness.* The declarative style meant that proofs rarely broke because of modifications to Declare's automated prover.

- *Clarity.* The disciplined approach enforced when mechanizing a proof ensures errors like those described in Section 7.4 are detected. The declarative proof language allowed the author to think clearly about the language while preparing the proof outlines. The error described in Section 7.4.1 was found when simply preparing the proof outline, rather than when checking it in detail. When drafting a Declare proof the question "will a machine accept this proof?" is always in mind, and this ensures that unwarranted logical leaps are not made.

We discuss these further in the next chapter.

7.5.1 Related Work

As mentioned at the start of Chapter 6, our model and proofs for Java$_S$ were based on a paper version of similar proofs developed by Drossopoulou and Eisenbach. Our soundness results, while similar in many respects, differ from Drossopoulou and Eisenbach's in detail. The main differences are:

- *Heap Typing.* We use a heap typing, which we believe makes definitions more coherent and leads to a simpler problem statement.

- *Safety and Liveness.* We prove two complementary results, rather than attempting to combine safety and liveness in one property. Drossopoulou and Eisenbach's property does not prove that all transitions result in conforming configurations, just that there always exists at least one such transition. In the presence of non-determinism this could mean that extra transitions are possible to non-conforming states.

- *Conformance includes Widening.* The statement and proof of subject reduction is substantially simplified by using conformance over configurations, up to widening.

- *Conformance over Exceptional Configurations.* Exceptions are not mentioned in the statement of subject reduction, since conformance is also defined for exceptional configurations.

- *No Reasoning about Multiple Steps.* The statement of subject reduction does

Wright and Felleisen [WF94] have studied type soundness proof techniques for a wide range of language constructs, though not for Java itself. They have not mechanized their proofs.

Tobias Nipkow and David von Oheimb [Nv98] have developed a proof of the type soundness property for a similar subset of Java in the Isabelle theorem prover. The first version of their proof was developed at roughly the same time as our own, and they have since continued to extend the subset covered and refine their formalization. I am extremely grateful for the chance to meet with Nipkow and von Oheimb and have adopted some suggestions they have made (indeed this has been mutual). These two works are valuable "modern" case studies of theorem proving methods applied this kind of problem. Isabelle is a mature system and has complementary strengths to Declare, notably strong generic automation and manifest soundness. A tool which unites these strengths with Declare's would be an exciting prospect.

Several groups are working on type soundness properties for aspects of the JVM [SA98, Qia97]. These proofs have not yet been mechanized, and thus are somewhat removed from the concerns of this thesis.

Chapter 8

Summary

The aim of this dissertation has been to describe the use of a technique called "declarative theorem proving" to fully formalise reasoning about operational semantics. Part I concentrated on the technique itself, and at the heart of the technique lies our method of proof description, based around three simple constructs, as described in Chapter 3. We explained the impact of this and other techniques with respect to four aspects of theorem proving tools: specification, proof description, automated reasoning and interaction. We also proposed techniques for simplifying the extraction of results from theorems (Section 2.3), a method for validating specifications by translation to Mercury (Section 2.4) and a language for providing justifications and hints for the automated prover. Throughout we used the system Declare as an example implementation of these techniques.

This has, in many ways, been the description of a long experiment in attempting to conduct significant proofs while sticking to the "declarative ideal." The resulting techniques are, we claim, relatively faithful to this ideal, particularly in contrast to tactic based provers. When considered as a package, the approach we have proposed is quite novel, though it clearly draws from a range of ideas across the spectrum of theorem proving. However, novelty aside, we must now address the more important question: *do declarative techniques make for better theorem proving?* This is, of course, difficult to answer definitively, as it requires a balanced assessment in the context of a particular project. However, we can first consider the somewhat simpler question: is declarative proof a suitable mechanism for proof description? The arguments in favour are documented in Chapter 3 and Chapter 5. They include:

- A declarative style is more readable, uses far fewer proof description constructs, and encourages good mathematical style.
- A declarative style allows considerable flexibility when decomposing a problem.
- A declarative style is pragmatically useful, as it allows proofs to be typechecked without discharging obligations, error recovery is easy to implement, and it is possible to implement a relatively simple and coherent interactive development environment for developing such proofs.

One drawback is that declarative proofs require extensive term quotation in order to specify logical steps. We have presented a range of mechanisms to alleviate this problem without compromising the declarative ideal, but it remains a challenge for future work.

Looking beyond the simple issue of describing proofs, we turn to the methodological issues described at the end of the previous chapter: *proof refinement*, *maintainability*, *robustness* and *clarity*. In many ways, these issues form the heart of the matter. Proof is, after all, a social process as well as a formal one, as argued by De Millo, Lipton and Perlis [MLP79]. Presuming declarative proof description can be made at least as efficient as existing proof description techniques "in the small", then the benefits "in the large" may well tip the balance in its favour.

Part II has described a lengthy case study in the application of these techniques, and indeed this study has considerable interest in its own right. Aside from issues of declarative proof, it demonstrates how formal techniques can be used to help specify a major language. Java itself is far more complicated than Java$_S$, but we have still covered a non-trivial subset. Drossopoulou and Eisenbach's formalization was the original inspiration for this work. We suggest that in the long run theorem prover specifications may provide a better format for such formalizations, especially when flexible tools are provided to read, execute and reason about them. In addition, the independent rediscovery of the mistake in the Java language specification described in Section 7.4.1 indicates that errors in language specifications can indeed be discovered by the process of formal proof.

8.1 Future Work

Throughout this thesis we have hinted at places where future work looks particularly promising. The following summarizes these, with the addition of some topics we have not yet considered:

- *Generalization of techniques?* Isabelle has demonstrated how techniques in theorem proving can be made generic across a range of theorem provers. Many of the techniques presented here have been crafted for first order logic: it should be possible to generalise these via the typical parameterization mechanisms used in Isabelle.

- *Declarative proofs in other logics?* It may be useful to apply declarative proof languages to other logics. How must the proof language change in this case?

- *Automated Reasoning?* Chapter 4 has described the requirements for an automated engine in our problem domain, and indicated how our current engine fails to meet these requirements in some ways. Clearly further work is possible here, especially to utilise techniques from other theorem provers in our context.

- *Interfaces for declarative proof?* Chapter 5 has presented a prototype interface for Declare that takes advantage of some of the features of our declarative proof

language, e.g. the small number of constructs to provide debugging support for each. A lot of scope remains for finding and inventing the interactive mechanisms to best support declarative proof.

In addition, the case study of Part II could be greatly extended in scope, simply by increasing the range of language constructs considered. Similar techniques could be applied to a study of the Java Virtual Machine and other interesting operational systems.

Appendix A

An Extract from the Declare Model

This appendix contains an extended extract from the Declare sources for the case study described in Chapters 6 and 7. This covers the model as far as the well-formedness constraints on environments, and the definition of conformance. We have also included the statements of many theorems up to this point, plus a selection of proofs. We have used first order symbols rather than their ASCII equivalents.

A.1 `psyntax.art` - Primitives and types

```
datatype prim =
   Void | Bool ":bool" | Char ":uchar" | Byte ":int8" | Short ":int16"
 | Int ":int32" | Long ":int64" | Float ":ieee32" | Double ":ieee64";

datatype primTy =
   VoidTy | BoolTy | CharTy | ByteTy | ShortTy
 | IntTy | LongTy | FloatTy | DoubleTy;

datatype refTy = ClassRefTy ":id" | InterfaceRefTy ":id" | ArrayRefTy ":typ" | AnyRefTy
and typ = RefTy ":refTy" | PrimTy ":primTy";

def [autodefn] "ClassTy C = RefTy(ClassRefTy(C))";
def [autodefn] "InterfaceTy i = RefTy(InterfaceRefTy(i))";
def [autodefn] "ArrayTy i = RefTy(ArrayRefTy(i))";
def [autodefn] "intTy = PrimTy(IntTy)";
def [autodefn] "boolTy = PrimTy(BoolTy)";
def [autodefn] "voidTy = PrimTy(VoidTy)";
def [autodefn] "ObjectTy = RefTy(ClassTy `Object`)";

def "mk_array_ty n aty = repeatn n aty (fun ty -> ArrayTy(ty))";

type argTy = ":typ list";

//-----------------------------------------------------------------
// Assign types to primitive values
```

```
def  "prim_type pval =
        match pval with
          Bool(b) -> BoolTy
        | Byte(byte) -> ByteTy
        | Char(c) -> CharTy
        | Short(sh) -> ShortTy
        | Long(lng) -> LongTy
        | Int(i) -> IntTy
        | Float(fl) -> FloatTy
        | Double(db) -> DoubleTy
        | Void -> VoidTy";
```

A.2 `widens.art` - Environments, Widening and Visibility

```
import psyntax;
notation rels;

//--------------------------------------------------------------------
// Type environments.
//
// These contain
//   -- class and interface declarations
//   -- local variable declarations
// The class and interface hierarchies are derivable from these, at
// least for well-formed environments.

datatype classDecl =
   CLASS ": id option ×
            id fset ×
            (id |-?> typ) ×
            ((id × typ list) |-?> typ)";

datatype interfaceDecl =
   INTERFACE ":(id fset) ×
               ((id × typ list) |-?> typ)";

type classenv = ":id |-?> classDecl";
type interfaceenv = ":id |-?> interfaceDecl";
type tyenv = ":(id |-?> classDecl) ×
               (id |-?> interfaceDecl)";
type varenv = ":id |-?> typ";

reserve TE for ":tyenv"
and CE for ":classenv"
and IE for ":interfaceenv"
and VE for ":varenv"
and C for ":id"
and i for ":id";

//--------------------------------------------------------------------
// Now derive the class and interface hierarchies from the
```

```
// declarations in the environment.
//
// First well-formed types.

def "TE |- C wf_class ↔ (∃CE IE cdec. TE = (CE,IE) ∧ flookup CE C = Some cdec)";
mode "inp1 |- inp2 wf_class";

def "TE |- i wf_interface ↔ (∃CE IE idec. TE = (CE,IE) ∧ flookup IE i = Some idec)";
mode "inp1 |- inp2 wf_interface";

constant wf_reftype ":tyenv -> refTy -> bool";
constant wf_type ":tyenv -> typ  -> bool";

thm <wf_type> [defn,code]
 "TE |- ty wf_type ↔
              match ty with
                  RefTy rt -> TE |- rt wf_reftype
                | PrimTy(pt) -> true";

thm <wf_reftype> [defn,code]
 "TE |- rty wf_reftype ↔
         match rty with
           ClassRefTy C -> TE |- C wf_class
         | InterfaceRefTy i -> TE |- i wf_interface
         | ArrayRefTy(ty) -> TE |- ty wf_type
         | AnyRefTy -> false";
mode "inp1 |- inp2 wf_reftype";
mode "inp1 |- inp2 wf_type";

// Hmmm.. can we extend labelling so we don't have to restate these??
thm <prim-wf_type> [autorw,automeson] "TE |- PrimTy(pt) wf_type";
proof qed by <wf_type>; end;

thm <class-wf> [autorw]
    "TE |- ClassTy(C) wf_type ↔ TE |- C wf_class";
proof qed by <wf_type>,<wf_reftype>; end;

thm <interface-wf> [autorw]
    "TE |- InterfaceTy(i) wf_type ↔ TE |- i wf_interface";
proof qed by <wf_type>,<wf_reftype>; end;

thm <array-wf> [autorw]
    "TE |- ArrayTy(ty) wf_type ↔ TE |- ty wf_type";
proof qed by <wf_type>,<wf_reftype>; end;

def "TE |- AT wf_types ↔ all (λvt. TE |- vt wf_type) AT";
mode "inp1 |- inp2 wf_types";

def "TE |- VE wf_varenv ↔
        ∀id vt. flookup(VE)(id) = Some(vt) →
         (TE |- vt wf_type [<rool>])";
mode "inp1 |- inp2 wf_varenv";

//-----------------------------------------------------------------
// The subclass relationship, derived from the declarations in TE.
//
```

```
// nb. executable version does not terminate for circular class
// structures.
//
// Both arguments should be provably well-formed in the context
// where this predicate is used.

lfp subclass_of =
<Refl>
         // ------------------------------------
            "TE |- C subclass_of C"

<Step>
            "TE = (CE,IE) ∧ flookup CE C = Some (CLASS(Some(Csup),_x1,_x2,_x3)) ∧
             TE |- Csup subclass_of C''"
        // ------------------------------------------------------------
             "TE |- C subclass_of C''"
;
mode "inp1 |- inp2 subclass_of inp3";
mode "inp1 |- inp2 subclass_of out3";

//-------------------------------------------------------------------------
// The implements relationship, derived from the declarations in TE.

def "TE |- C implements i ↔
        ∃CE IE dec _x1 _x2 _x3 Is.
           TE = (CE,IE) ∧ flookup CE C = Some (CLASS(_x1,Is,_x2,_x3)) ∧ i ∈ Is";

mode "inp1 |- inp2 implements inp3";
mode "inp1 |- inp2 implements out3";

//-------------------------------------------------------------------------
// The subinterface relationship

lfp subinterface_of =
<Refl>
         // ------------------------------------
             "TE |- i subinterface_of i"

<Step> [ ]
          "TE = (CE,IE) ∧ flookup IE i = Some (INTERFACE(Is,methods)) ∧
            i' ∈ Is ∧
            TE |- i' subinterface_of i''"
         // ------------------------------------------
            "TE |- i subinterface_of i''"

;
mode "inp1 |- inp2 subinterface_of inp3";
mode "inp1 |- inp2 subinterface_of out3";

//-------------------------------------------------------------------------
// Widening/Narrowing, derived from the declarations in the environment.
//

lfp widens_to =
<Prim>      [automeson,autorw]
```

```
        // ------------------------------------
            "TE |- PrimTy(pt) widens_to PrimTy(pt)"

<ClassToClass>  [automeson,autorw]
            "TE |- C subclass_of C'"
        // ------------------------------------------------
            "TE |- ClassTy(C) widens_to ClassTy(C')"

<InterfaceToInterface> [automeson,autorw]
            "TE |- i subinterface_of i'"
        // ------------------------------------------------
            "TE |- InterfaceTy(i) widens_to InterfaceTy(i')"

<InterfaceToObject>   [automeson,autorw]

      // ------------------------------------------------
       "TE |- InterfaceTy(i) widens_to ObjectTy"

<ClassToInterface>  [automeson]
              "TE |- C subclass_of C' ∧
               TE |- C' wf_class ∧
               TE |- C' implements i ∧
               TE |- i wf_interface ∧
               TE |- i subinterface_of i'"
       // ------------------------------------------------------------
          "TE |- ClassTy(C) widens_to InterfaceTy(i')"

<ArrayToObject>  [automeson,autorw]
            "TE |- ty wf_type"
      // ------------------------------------
            "TE |- ArrayTy(ty) widens_to ObjectTy"

<Array>  [automeson,autorw]
            "TE |- ty widens_to ty'"
      // ------------------------------------
            "TE |- ArrayTy(ty) widens_to ArrayTy(ty')";
mode "inp1 |- inp2 widens_to inp3";

def "TE |- tys tys_widen_to tys' ↔
       len tys = len tys' ∧
       (∀j. j < len(tys) → TE |- el(j)(tys) widens_to el(j)(tys'))";
mode "inp1 |- inp2 tys_widen_to inp3";

//-----------------------------------------------------------------
// Search for field declarations, based off the declarations in TE.
// Sensibly defined for well formed hierarchies of interfaces and classes.

lfp VisField =
<Hit>     "TE = (CE,IE) ∧ flookup CE C = Some (CLASS(Csupo,Is,fields,methods)) ∧
           flookup(fields)(v) = Some(vt)"
         // ------------------------------------
            "VisField(TE,C,v)(C,vt)"

<Miss>    "TE = (CE,IE) ∧ flookup CE C = Some (CLASS(Some(Csup),Is,fields,methods)) ∧
           flookup(fields)(v) = None ∧
           VisField(TE,Csup,v)(res)"
```

```
         // ------------------------------------
             "VisField(TE,C,v)(res)";

mode "VisField(inp)(out)";

//-------------------------------------------------------------------
// Return all field declarations for a class, based off the declarations in
// TE. [FDecs(TE,C)] indicates all the fields in [C] ∧ all the superclasses
// of [C], including hidden fields.
//
// For well-formed TE, vt is unique for a given (f,C).

lfp FieldExists =
<Hit>     "TE = (CE,IE) ∧ flookup CE C = Some (CLASS(Csupo,Is,fields,methods)) ∧
           f ∈ fdomain(fields) ∧
           flookup(fields)(f) = Some(vt)"
         // ------------------------------------
             "((C,f),vt) ∈ FieldExists(TE,C)"

<Super>    "TE = (CE,IE) ∧ flookup CE C = Some (CLASS(Some(Csup),Is,fields,methods)) ∧
            (fidx,vt) ∈ FieldExists(TE,Csup)"
         // ------------------------------------
            "(fidx,vt) ∈ FieldExists(TE,C)";

mode "FieldExists(inp1)(out2)";

def "AllFields (TE,C) = fset_of_set (FieldExists(TE,C))";

//-------------------------------------------------------------------
// Return all versions of a method, based off the declarations in
// TE. MSigs(TE,C,m) indicates all the method declarations (i.e. both the class of
// the declaration and the signature) for method m in class C, or inherited
// from one of its superclasses, and not hidden by any of its superclasses.

lfp MSigsC =
<Hit>     "TE = (CE,IE) ∧ flookup CE C = Some (CLASS(Csupo,Is,fields,methods)) ∧
           flookup methods midx = Some(rt)"
         // ------------------------------------
             "MSigsC(TE,C)(midx,rt)"

<Miss>    "TE = (CE,IE) ∧ flookup CE C = Some (CLASS(Some(Csup),Is,fields,methods)) ∧
           MSigsC(TE,Csup)(midx,rt) ∧
           flookup methods midx = None"
         // ------------------------------------
             "MSigsC(TE,C)(midx,rt)"
;
mode "MSigsC(inp1)(out2)";

lfp MSigsI =

<Hit>     "TE = (CE,IE) ∧ flookup IE i = Some (INTERFACE(Is,methods)) ∧
           flookup methods midx = Some(rt)"
         // ------------------------------------
             "MSigsI(TE,i)(midx,rt)"

<Miss>    "TE = (CE,IE) ∧ flookup IE i = Some (INTERFACE(Is,methods)) ∧
```

```
           flookup methods midx = None ∧
           (i' ∈ Is ∧ MSigsI(TE,i')(midx,rt) ∨
            (Is = fempty ∧ MSigsC(TE,‘Object‘)(midx,rt)))"
         // ------------------------------------------------------
            "MSigsI(TE,i)(midx,rt)"
;
mode "MSigsI(inp1)(out2)";


// Arrays always support all methods found in ‘Object‘, unless they
// are overridden. I haven't yet got arrays supporting methods and
// fields generic to all arrays, i.e. "size" and "clone".

def  "MSigsA(TE)(midx,mt) ↔ MSigsC(TE,‘Object‘)(midx,mt)";

mode "MSigsA(inp1)(out2)";

def "MSigs(TE,refty)(midx,mt) ↔
       match refty with
         InterfaceTy(i) -> MSigsI(TE,i)(midx,mt)
       | ClassTy(C) -> MSigsC(TE,C)(midx,mt)
       | ArrayTy(ty') -> MSigsA(TE)(midx,mt)";
mode "MSigs(inp1)(out2)";
```

A.3 wfenv.art - Constraints on Environments

The proofs have been omitted from this file for brevity.

```
import psyntax widens;
notation rels;

reserve TE for ":tyenv"
and CE for ":classenv"
and IE for ":interfaceenv"
and C for ":id"
and i for ":id";

//=============================================================================
// PART 1. Define well-formed type environments
//
// At the roots of the tree we check that interfaces do not mess around
// with the return types of Object methods...

def "
 TE wf_tyenv ↔
 (∃dec methods. ((∃CE IE. TE = (CE,IE) ∧ flookup CE ‘Object‘ = Some(dec)) ∧
                 dec = CLASS(None,fempty,fpempty,methods)) [<Object_declared> [rw]]) ∧
 (∀C Csupo Is fields methods.
  (∃CE IE. TE = (CE,IE) ∧ flookup CE C = Some(CLASS(Csupo,Is,fields,methods))) →

   (match Csupo with
      Some(Csup) ->
         (TE |- Csup wf_class) [<class_superclass_declared>] ∧
         (¬(TE |- Csup subclass_of C)) [<no_circular_classes>] ∧
```

```
         (∀midx rt1. MSigsC(TE,Csup)(midx,rt1) →
          ∀rt2. flookup methods midx = Some rt2 →
              (TE |- rt2 widens_to rt1) [<class_return_types_wider> ])

     | None ->
         (C = `Object`) [<only_Object_has_no_superclass> [rw]] ∧
         (fields = fpempty) [<Object_has_no_fields> [rw]] ∧
         (Is = fempty) [<Object_implements_no_interfaces> [rw]]) ∧

    (∀midx rt. flookup methods midx = Some rt →
     (∀m AT. midx = (m,AT) → (TE |- AT wf_types ∧ TE |- rt wf_type) [<class-methtypes-wf>])) ∧

    (∀fld ty. flookup fields fld = Some(ty) →
     (TE |- ty wf_type) [<field-types-wf>]) ∧

    (∀i. i ∈ Is →
     (TE |- i wf_interface) [<class_superinterfaces_declared>] ∧
     (∀midx rt1. MSigsI(TE,i)(midx,rt1) →
      (∃rt2. MSigsC(TE,C)(midx,rt2) ∧
            TE |- rt2 widens_to rt1) [<interfaces_implemented>])))
∧
   (∀i Is methods.
     (∃CE IE. TE = (CE,IE) ∧ flookup IE i = Some(INTERFACE(Is,methods))) →
     (∀i'. i' ∈ Is →
       (¬(TE |- i' subinterface_of i)) [<no_circular_interfaces>] ∧
       (TE |- i' wf_interface) [<interface_superinterfaces_declared>]) ∧

    (∀midx rt. flookup methods midx = Some rt →
     (∀m AT. midx = (m,AT) → (TE |- AT wf_types ∧ TE |- rt wf_type) [<interface-methtypes-wf>])) ∧

     (∀i'. i' ∈ Is →
      ∀midx rt1 rt2.
          MSigsI(TE,i')(midx,rt1) ∧
          flookup methods midx = Some(rt2) →
            (TE |- rt2 widens_to rt1) [<interface_return_types_wider>]) ∧

     (∀midx rt1. flookup methods midx = Some(rt1) →
      ∀rt2. MSigsC(TE,`Object`)(midx,rt2) →
         (TE |- rt1 widens_to rt2) [<interface_return_types_wider_than_Object>]))";

mode "inp wf_tyenv";

//=============================================================================
// PART 2. Transitivity and Reflexivity for Widening

thm <widens_to-refl> [autorw]
 if "TE |- ty wf_type"
 then "TE |- ty widens_to ty";

//-----------------------------------------------------------------------------
// Object is always a well-formed class, type, implements no interfaces
// and has no superclasses.

thm <wf_class-Object> [autorw]
 if "TE wf_tyenv"  <TE_wf>
 then "TE |- `Object` wf_class";
```

```
thm <wf_type-Object> [autorw]
 if "TE wf_tyenv"
 then "TE |- ObjectTy wf_type";

thm <Object-implements-nothing> [autorw]
 if "TE wf_tyenv" <TE_wf>
    "TE |- i wf_interface"
 then "¬(TE |- `Object` implements i)";

thm <Object-subclass>[rw]
 if "TE wf_tyenv" <TE_wf>
    "TE |- C wf_class"
 then "TE |- `Object` subclass_of C ↔ C = `Object`";

thm <Object-widens> [rw]
 if "TE wf_tyenv"  <TE_wf>
    "TE |- rt wf_type"
 then "TE |- ObjectTy widens_to rt ↔ rt = ObjectTy";

//-------------------------------------------------------------------------
// widens_to is transitive.  Non-trivial as we must
// ensure confluence of the subtype graph.

thm <widens_to-trans>
  if "TE wf_tyenv" <TE_wf>
     "TE |- ty1 wf_type"
     "TE |- ty2 wf_type"
     "TE |- ty3 wf_type"
     "TE |- ty1 widens_to ty2" <a1>
     "TE |- ty2 widens_to ty3" <a2>
then "TE |- ty1 widens_to ty3";

//=========================================================================
// PART 3.  Decomposition results for widening for types of particular forms
//
// e.g. The only subtypes of an array type are covariant array
// types of the same dimension.

thm <array-widens-lemma>
  "ty0 = ArrayTy(aty0) ∧
   TE |- ty0 wf_type ∧
   TE |- ty1 wf_type ∧
   TE |- ty1 widens_to ty0
   → ∃aty1.
        ty1 = ArrayTy(aty1) ∧
        TE |- aty1 wf_type ∧
        TE |- aty1 widens_to aty0";

thm <prim-widens-lemma>
  "TE |- ty wf_type ∧
   TE |- ty widens_to PrimTy(pty)
   → ty = PrimTy(pty)";

thm <class-widens-lemma>
if "TE wf_tyenv"
```

```
   "TE |- ty wf_type"
   "TE |- C wf_class"
   "TE |- ty widens_to ClassTy(C)"
   "C <> `Object`"
then "∃C'. TE |- C' subclass_of C ∧
           TE |- C' wf_class ∧
           ty = ClassTy(C')";

thm <reference-widens-lemma>
if "TE wf_tyenv"
   "TE |- ty wf_type"
   "TE |- ty widens_to RefTy(rt)"
then "∃rt'. ty = RefTy(rt')";

//============================================================================
// PART 4. Preservation of Visibility

//----------------------------------------------------------------------------
// Dependent typing of AllFields, MSigs etc.
//
//  -- AllFields only finds wf. classes and wf. field types...
//  -- MSigs only finds well-formed method types...

thm <FieldExists-wf>
  if "TE wf_tyenv" <TE_wf>
     "TE |- C wf_class"
     "((C',f),ty) ∈ FieldExists(TE,C)" <a>
 then "TE |- C' wf_class ∧ TE |- ty wf_type";

thm <FieldExists-finite>
  if "TE wf_tyenv" <TE_wf>
     "TE |- C wf_class" <x>
  then "finite (FieldExists(TE,C))";

thm <AllFields-wf>
  if "TE wf_tyenv"
     "TE |- C wf_class"
     "((C',f),ty) ∈ AllFields(TE,C)"
 then "TE |- C' wf_class ∧ TE |- ty wf_type";

thm <MSigsC-wf>
  if "TE wf_tyenv" <TE_wf>
     "TE |- C wf_class"
     "MSigsC(TE,C)(midx,rt)" <a>
     "midx = (m,AT)"
 then "TE |- AT wf_types ∧ TE |- rt wf_type";

thm <MSigsI-wf>
  if "TE wf_tyenv" <TE_wf>
     "TE |- i wf_interface"
     "MSigsI(TE,i)(midx,rt)" <a>
     "midx = (m,AT)"
 then "TE |- AT wf_types ∧ TE |- rt wf_type";

thm <MSigs-wf>
  if "TE wf_tyenv" <TE_wf>
```

```
     "ty = RefTy(refty)"
     "TE |- ty wf_type"
     "MSigs(TE,refty)((m,AT),rt)"
 then  "TE |- AT wf_types ∧ TE |- rt wf_type";

//-------------------------------------------------------------------------
// subclass_of preserves field existence (thoug not necessarily visibility)

thm <inherited-fields-exist>
  if "TE |- C0 wf_class" <a>
     "TE |- C1 wf_class" <b>
     "TE wf_tyenv" <TE_wf>
     "TE |- C1 subclass_of C0" <subclass>
     "fspec ∈ AllFields(TE,C0)" <x>
then "fspec ∈ AllFields(TE,C1)" <y>;

//-------------------------------------------------------------------------
// subtyping preserves method visibility up to narrowing of return type.

thm <class-inherited-class-methods-are-narrower>
  if "TE wf_tyenv" <TE_wf>
     "TE |- C0 wf_class"
     "TE |- C1 wf_class"
     "TE |- C1 subclass_of C0" <C1_subclass>
     "MSigsC(TE,C0)(midx,rt0)" <search>
then "∃rt1. MSigsC(TE,C1)(midx,rt1) ∧
            TE |- rt1 wf_type ∧
            TE |- rt1 widens_to rt0";

thm <interface-inherited-interface-methods-are-narrower>
  if "TE wf_tyenv" <TE_wf>
     "TE |- i0 wf_interface"
     "TE |- i1 wf_interface"
     "TE |- i1 subinterface_of i0" <i1_subclass>
     "MSigsI(TE,i0)(midx,rt0)" <search>
then "∃rt1. MSigsI(TE,i1)(midx,rt1) ∧
            TE |- rt1 wf_type ∧
            TE |- rt1 widens_to rt0";

thm <class-inherited-interface-methods-are-narrower>
  if "TE = (CE,IE)"
     "TE wf_tyenv" <TE_wf>
     "TE |- C wf_class"
     "TE |- i wf_interface"
     "TE |- C implements i" <imp>
     "MSigsI(TE,i)(midx,rt0)" <search>
  then "∃rt1. MSigsC(TE,C)(midx,rt1) ∧
              TE |- rt1 wf_type ∧
              TE |- rt1 widens_to rt0";

thm <interface-inherited-Object-methods-are-narrower>
  if "TE wf_tyenv" <TE_wf>
     "TE |- i wf_interface"
     "MSigsC(TE,`Object`)(midx,rt0)" <base>
then "∃rt1. MSigsI(TE,i)(midx,rt1) ∧
            TE |- rt1 wf_type ∧
```

```
            TE |- rt1 widens_to rt0";

thm <array-inherited-Object-methods-are-identical>
    "TE wf_tyenv ∧
     MSigsC(TE,‘Object‘)(m,mt)
     → MSigsA(TE)(m,mt)";

thm <inherited-methods-exist>
  if "TE wf_tyenv"
     "ty0 = RefTy(refty0)"
     "ty1 = RefTy(refty1)"
     "TE |- ty0 wf_type"
     "TE |- ty1 wf_type"
     "MSigs(TE,refty0)(midx,rt0)"
     "TE |- ty1 widens_to ty0"  <a>
 then "∃rt1. MSigs(TE,refty1)(midx,rt1) ∧
             TE |- rt1 wf_type ∧
             TE |- rt1 widens_to rt0";

//-------------------------------------------------------------------------
// FieldExists only searches super classes.
//

thm <FieldExists-finds-subclasses>
   if "TE wf_tyenv"
      "TE |- C wf_class"
      "((Cf,f),ty) ∈ FieldExists(TE,C)" <deriv>
 then "TE |- C subclass_of Cf";

//-------------------------------------------------------------------------
// AllFields does not find more than one field type
// for a given field/class pair.

thm <object-fields-unique-lemma>
   if "TE wf_tyenv" <TE_wf>
      "TE |- C wf_class"
      "((Cf,f),ty1) ∈ FieldExists(TE,C)" <deriv1>
      "((Cf,f),ty2) ∈ FieldExists(TE,C)" <deriv2>
 then "ty1 = ty2";

//-------------------------------------------------------------------------
// And thus the graph found by AllFields form a partial function.

thm <object-fields-form-graph>
  if "TE wf_tyenv" <TE_wf>
     "TE |- C wf_class"
  then "frel_is_graph (AllFields(TE,C))";

//-------------------------------------------------------------------------
// Object has no visible fields...

thm <AllFields-Object>
   if "TE wf_tyenv" <TE_wf> then "¬(x ∈ AllFields(TE,‘Object‘))";
```

A.4 rsyntax.art - Syntax of Java$_R$

```
//------------------------------------------------------------------------
// Syntax of JavaR - configurations of the abstract machine,
// and structural operations on them.

import psyntax widens;
notation rels runtime;

datatype rval =
   RPrim ":prim"
 | RAddr ":int option";

type frame = ":(id |-?> rval)";

datatype rexp =
   RValue ":rval"
 | RStackVar ":id"
 | RAccess ":rexp × rexp"
 | RField ":rexp × id × id"
 | RNewClass ":id × ((id × id) |-?> typ)"
 | RNewArray ":typ × rexp list"
 | RCall ":rexp × (id × argTy) × rexp list"
 | RBody ":rstmt × frame"
and rstmt =
   RBlock ":rstmt list"
 | RIf ":rexp × rstmt × rstmt"
 | RWhile ":rexp × rstmt"
 | RReturn ":rexp"
 | RAssignToStackVar ":id × rexp"
 | RAssignToArray ":(rexp × rexp) × rexp"
 | RAssignToField ":(rexp × id × id) × rexp"
 | RExpr ":rexp";

reserve C for ":id"
and id for ":id"
and prog for ":cprog"
and mbody for ":cmethodbody"
and stmts for ":rstmt list"
and addr for ":int"
and val for ":rval"
and ty for ":typ";

//------------------------------------------------------------------------
// Heap Objects, Heaps, State and Configurations
//
// The type stored in an array indicates the type of elements
// stored in the array, not the type of the array itself

datatype heapobj =
   OBJECT ":((id × id) |-?> rval) × id"
 | ARRAY ":typ×rval list";

type heap = ":(int,heapobj)fpfun";
type state = ":frame × heap";
type 'a cfg = ":'a × state"
```

```
reserve heap for ":heap";

//-------------------------------------------------------------------------
// Heap operations

def "hoType(heapobj) =
        match heapobj with
           OBJECT(fldvals,C) -> ClassRefTy(C)
         | ARRAY(aty,vec) -> ArrayRefTy(aty)";

def "sAlloc(heap,heapobj) =
        let addr = freshi(fdomain(heap))
        in (heap <?++ (addr,heapobj),addr)";

//-------------------------------------------------------------------------
// initial values during allocation

def "initial ty =
        match ty with
          RefTy rty -> RAddr(None)
        | PrimTy(pt) ->
            match pt with
             BoolTy -> RPrim(Bool(false))
           | CharTy -> RPrim(Char(mk_uchar(32I)))
           | ByteTy -> RPrim(Byte(mk_int8(0I)))
           | ShortTy -> RPrim(Short(mk_int16(0I)))
           | IntTy -> RPrim(Int(mk_int32(0I)))
           | VoidTy -> RPrim(Void)
           | LongTy -> RPrim(Long(mk_int64(0I)))
           | FloatTy -> RPrim(Float(mk_ieee32(0I)))
           | DoubleTy -> RPrim(Double(mk_ieee64(0I)))";

// ------------------------------------------------------------------------
// Define ground expressions, values etc.
// What all good expressions aspire to be.
//

def "exp_ground exp = (∃v. exp = RValue(v))";
mode "exp_ground inp";

def "exps_ground exps = all exp_ground exps";
mode "exps_ground inp";

def "stmts_ground(stmts) = null(stmts)";
mode "stmts_ground inp";

def "stmt_ground(stmt) = (∃v. stmt = RExpr(RValue(v)))";
mode "stmt_ground inp";

// ------------------------------------------------------------------------
// Runtime type checking.  This must be executable.
// In principle we can return None for illegal typechecks,
// thus allowing us to reason that these never happen.

def "typecheck((TE,heap),sval,cell_ty) =
```

```
    match sval with
     RPrim(pval) -> Some (∃pt. cell_ty = PrimTy(pt) ∧ prim_type(pval) = pt)
   | RAddr(None) ->  Some (∃rt. cell_ty = RefTy(rt))
   | RAddr(Some(addr)) ->
      match flookup(heap)(addr) with
       Some(heapobj) ->
           Some(TE |- RefTy(hoType(heapobj)) widens_to cell_ty)
     | None -> Some false";
```

A.5 rstatics.art - Conformance and some proofs

```
//---------------------------------------------------------------------------
// Conformance for runtime structures,
// and preservation of this under various operations.

import psyntax rsyntax widens wfenv;
notation rels rstatics;

// A frame typing is the typing for local variables on the stack.
type ftyping = ":id |-?> typ";

// A heap typing is the typing for things in the heap
type htyping = ":int |-?> refTy";

reserve TE for ":tyenv"
and FT,FT0,FT1 for ":ftyping"
and HT,HT0,HT1 for ":htyping"
and frame for ":frame"
and heap for ":heap"
and C,id for ":id"
and ty for ":typ"
and refty for ":refTy"
and stmts for ":rstmt list"
and ST for ":ftyping ## htyping";

def "TE |- FT wf_ftyping ↔
        ∀id ty. flookup(FT)(id) = Some(ty) →
         (TE |- ty wf_type [<rool>])";

def "TE |- HT wf_htyping ↔
         (∀addr refty. flookup(HT)(addr) = Some(refty) →
          (TE |- RefTy(refty) wf_type) [<rool>])";

def "TE |- (FT,HT) wf_styping ↔
       (TE |- FT wf_ftyping [<frame>] ∧
        TE |- HT wf_htyping [<heap>])";

//---------------------------------------------------------------------------
//

lfp rval_conforms_to =
<NullToRef> [autorw,automeson]
     // ------------------------------------------------------------
           "(TE,HT) |- RAddr(None) rval_conforms_to RefTy(refty)"
```

```
<Addr> [autorw]
            "flookup(HT)(addr) = Some refty"
     // ------------------------------------------------------------
           "(TE,HT) |- RAddr(Some(addr)) rval_conforms_to RefTy(refty)"

<Prim> [autorw,automeson]
           "prim_type(p) = pt"
     // ---------------------------------------
            "(TE,HT) |- RPrim(p) rval_conforms_to PrimTy(pt)"

<Trans> [ ]
           "(TE,HT) |- val rval_conforms_to ty' ∧
            TE |- ty' wf_type ∧
            TE |- ty' widens_to ty"
     // ---------------------------------------
            "(TE,HT) |- val rval_conforms_to ty";

//-------------------------------------------------------------------------
// A heap conforms to a heap typing if:
//   -- All the objects in the heap have precisely the structure expected for
//      the type, including the correct runtime type tag.
//   -- All the values in the objects in heap conform w.r.t. the heap typing.
//      They may be narrower than their expected slots.

def "((TE,HT) |- fldvals fldvals_conform_to C) [<derive>] ↔
       (∀idx ty'. (idx,ty') ∈ AllFields(TE,C) →
           (∃val. flookup(fldvals)(idx) = Some(val) ∧
             (TE,HT) |- val rval_conforms_to ty') [<rool>])";

def "(E |- vec els_conform_to ty) [<derive>] ↔
      (∀j. j < len(vec) →
       (E |- el(j)(vec) rval_conforms_to ty) [<rool>])";

def "(E |- heapobj heapobj_conforms_to refty) [<derive>] ↔
     match heapobj with
       OBJECT(fldvals,C) ->
         (refty = ClassTy(C)) [<object-tag-matches>] ∧
         (E |- fldvals fldvals_conform_to C) [<object-fields-conform>]
     | ARRAY(aty,vec) ->
         (refty = ArrayTy(aty)) [<array-tag-matches>] ∧
         (E |- vec els_conform_to aty) [<array-elements-conform>]";

def "(TE |- heap heap_conforms_to HT) [<derive>] ↔
     ((fdomain heap = fdomain HT) [<domains-eq>] ∧
      (∀addr heapobj. flookup(heap)(addr) = Some(heapobj) →
       (∃refty. flookup(HT)(addr) = Some(refty) ∧
        ((TE,HT) |- heapobj heapobj_conforms_to refty)) [<rool>]))";

//-------------------------------------------------------------------------
// Frame conformance -- all the values in the frame conform to
// the given types w.r.t. the given heap typing (they may also be narrower).

def "(E |- frame frame_conforms_to FT) [<derive>] ↔
     (fdomain FT = fdomain frame) [<frame-domains-eq> [rw]] ∧
     (∀id ty. flookup(FT)(id) = Some(ty) →
       (∃val. flookup(frame)(id) = Some(val) ∧
```

```
             E |- val rval_conforms_to ty) [<stackvar-conforms>])";

//-------------------------------------------------------------------------
// Rules for expressions, statements, variables
//
// For various resons these relations are non-executable, e.g. we
// cannot guess the return type of the body of an expression
// (it changes during execution, and maybe indeterminate, e.g. if
// the return value has been reduced to null).

def rec "ralways_returns(stmt) =
           match stmt with
              RBlock(stmts) -> existsl ralways_returns stmts
            | RIf(e,stmt1,stmt2) -> ralways_returns(stmt1) ∧ ralways_returns(stmt2)
            | RReturn(e) -> true
            | _ -> false";

lfp rexp_conforms_to =

<StackVar>  "flookup(FT)(x) = Some(ty)"
        // ------------------------------------------
            "(TE,(FT,HT)) |- RStackVar(x) rexp_conforms_to ty"

<Access>   "(TE,ST) |- arr rexp_conforms_to ArrayTy(arrty) ∧
            (TE,ST) |- idx rexp_conforms_to intTy"
      // ---------------------------------------------------
           "(TE,ST) |- RAccess(arr,idx) rexp_conforms_to arrty"

<Field>  "(TE,ST) |- obj rexp_conforms_to ClassTy(C) ∧
           TE |- C wf_class ∧
           ((C',f),ty) ∈ AllFields(TE,C)"
      // ---------------------------------------------------
         "(TE,ST) |- RField(obj,C',f) rexp_conforms_to ty"

<Value>     "(TE,HT) |- v rval_conforms_to et"
     // ------------------------------------------------------------------------------
          "(TE,(FT,HT)) |- RValue(v) rexp_conforms_to et"


<NewClass>  "TE |- C wf_class ∧
             flds = fpfun_of_frel (AllFields(TE,C))"
      // ---------------------------------------------------------------
           "(TE,ST) |- RNewClass(C,flds) rexp_conforms_to ClassTy(C)"

<NewArray>
         "TE |- aty wf_type ∧
          (TE,ST) |- dims rexps_conform_to (replicate (len dims) intTy)"
      // ---------------------------------------------------------------------
           "(TE,ST) |- RNewArray(aty,dims) rexp_conforms_to (mk_array_ty (len dims) aty)"

<Call>   "TE |- ty wf_type ∧
          (TE,ST) |- e rexp_conforms_to ty ∧
          MSigs(TE,ty)((m,AT),rt) ∧
          (TE,ST) |- args rexps_conform_to AT"
      // ---------------------------------------------------
           "(TE,ST) |- RCall(e,(m,AT),args) rexp_conforms_to rt"
```

```
<Body>  "TE |- FT' wf_ftyping ∧
         (TE,HT) |- frame frame_conforms_to FT' ∧
         ST' = (FT',HT) ∧
         (TE,ST',rt) |- stmt rstmt_conforms ∧
         (rt <> voidTy → ralways_returns(stmt))"
    // ------------------------------------------------------------
        "(TE,(FT,HT)) |- RBody(stmt,frame) rexp_conforms_to rt"

<Trans>    "(TE,ST) |- exp rexp_conforms_to ty' ∧
            TE |- ty' wf_type ∧
            TE |- ty' widens_to ty"
     // ---------------------------------------
           "(TE,ST) |- exp rexp_conforms_to ty"

and rexps_conform_to =

<Exps>
           "len(exps) = len(etys) ∧
            (∀j. j < len(exps) → (TE,ST) |- el(j)(exps) rexp_conforms_to el(j)(etys))"
      // ---------------------------------------------------
           "(TE,ST) |- exps rexps_conform_to etys"


and rstmt_conforms =

<AssignToStackVar> [ ]
           "TE |- ty' wf_type ∧
            (TE,ST) |- e rexp_conforms_to ty' ∧
            ST = (FT,HT) ∧
            flookup(FT)(id) = Some(ty) ∧
            TE |- ty' widens_to ty"
     // ------------------------------------------------
           "(TE,ST,rt) |- RAssignToStackVar(id,e) rstmt_conforms"

<AssignToField> [automeson]
           "TE |- ty' wf_type ∧
            (TE,ST) |- rexp rexp_conforms_to ty' ∧
            TE |- C' wf_class ∧
            (TE,ST) |- obj rexp_conforms_to ClassTy(C') ∧
            ((C,f),ty) ∈ AllFields(TE,C')  ∧
            TE |- ty' widens_to ty"
     // ------------------------------------------------
           "(TE,ST,rt) |- RAssignToField((obj,C,f),rexp) rstmt_conforms"

<AssignToArray> [automeson]
           "TE |- ty wf_type ∧
            (TE,ST) |- e rexp_conforms_to ty ∧
            TE |- simpty wf_type ∧
            (TE,ST) |- arr rexp_conforms_to ArrayTy(aty) ∧
            (TE,ST) |- idx rexp_conforms_to intTy"
     // ------------------------------------------------------------------
           "(TE,ST,rt) |- RAssignToArray((arr,idx),e) rstmt_conforms"

<If> [autorw,automeson]
           "(TE,ST,rt) |- tstmt rstmt_conforms ∧
```

```
           (TE,ST,rt) |- estmt rstmt_conforms ∧
           (TE,ST) |- e rexp_conforms_to boolTy"
     // ------------------------------------------------------------------
          "(TE,ST,rt) |- RIf(e,tstmt,estmt) rstmt_conforms"

<Expr> [autorw,automeson]
          "TE |- ty wf_type ∧
           (TE,ST) |- e rexp_conforms_to ty"
      // ---------------------------------------
          "(TE,ST,rt) |- RExpr(e) rstmt_conforms"

<Block> [autorw,automeson]
          "all (λstmt. (TE,ST,rt) |- stmt rstmt_conforms) stmts"
      // ------------------------------------------------------------------------
          "(TE,ST,rt) |- RBlock(stmts) rstmt_conforms"

<Return>   "(TE,VE) |- e rexp_conforms_to rt"
      // ---------------------------------------
          "(TE,VE,rt) |- RReturn(e) rstmt_conforms"

<While>     "(TE,VE) |- exp rexp_conforms_to boolTy ∧
             (TE,VE,rt) |- bod rstmt_conforms"
      // ------------------------------------------------------------------------
           "(TE,VE,rt) |- RWhile(exp,bod) rstmt_conforms";

thm <rexp_conforms_to-trans> [ ]
if "TE wf_tyenv" <TE_wf>
   "TE |- ST wf_styping" <ST_wf>
   "TE |- ty' wf_type"
   "TE |- ty wf_type"
   "(TE,ST) |- exp rexp_conforms_to ty'"
   "TE |- ty' widens_to ty"
then "(TE,ST) |- exp rexp_conforms_to ty";
proof qed by <rexp_conforms_to.Trans>; end;


thm <rexps_conform_to-trans> [ ]
if "TE wf_tyenv"  <TE_wf>
   "TE |- ST wf_styping"  <ST_wf>
   "ST = (FT,HT)"
   "(TE,ST) |- exps rexps_conform_to tys'"
   "TE |- tys' wf_types"
   "TE |- tys wf_types"
   "TE |- tys' tys_widen_to tys"
then "(TE,ST) |- exps rexps_conform_to tys";
proof
    consider j st
      + "len exps = len tys'" [autorw]
      + "j < len exps"
      + "(TE,ST) |- (el j exps) rexp_conforms_to (el j tys')" <a>
      - "(TE,ST) |- (el j exps) rexp_conforms_to (el j tys)"
      by <tys_widen_to>,<rexps_conform_to>,<goal>;

    qed by <rexp_conforms_to.Trans> [<a>,"(el j tys)"/ty],
           <wf_types>,<tys_widen_to>,<all>;
end;
```

```
//-------------------------------------------------------------------
// Conformancs of configurations

def "(TE |- (frame,heap) state_conforms_to (FT,HT)) [<derive>] ↔
       (TE |- heap heap_conforms_to HT) [<heap>] ∧
       ((TE,HT) |- frame frame_conforms_to FT) [<frame>]";

def "(TE |- (exp,s) ecfg_conforms_to (ty,ST)) [<derive>]  ↔
     (TE |- s state_conforms_to ST) [<state>] ∧
     ((TE,ST) |- exp rexp_conforms_to ty) [<term>]";

def "(TE |- (exps,s) escfg_conforms_to (tys,ST)) [<derive>]  ↔
     (TE |- s state_conforms_to ST) [<state>] ∧
     ((TE,ST) |- exps rexps_conform_to tys) [<term>]";

def "((TE,rt) |- (stmt,s) scfg_conforms_to ST) [<derive>] ↔
     (TE |- s state_conforms_to ST) [<state>] ∧
     ((TE,ST,rt) |- stmt rstmt_conforms) [<term>]";

//-------------------------------------------------------------------------
// Narrowing/enlarging between heap typings

def "HT0 htyping_leq HT1 ↔
       (∀addr. addr ∈ fdomain HT0 →
        (flookup HT1 addr = flookup HT0 addr) [<rool> [rw] ])";

thm <htyping_leq-refl> [autorw,automeson] "HT htyping_leq HT";
proof qed by <htyping_leq>; end;

//-----------------------------------------------------------------------------
// State typings

def "(FT0,HT0) styping_leq (FT1,HT1) ↔ FT0 = FT1 ∧ HT0 htyping_leq HT1";

thm <styping_leq-refl> [autorw,automeson] "ST styping_leq ST";
proof  qed by <styping_leq>; end;

//-------------------------------------------------------------------------
// Lemmas: as heap and frame typings get narrower, typing judgemsnts
// remain identical.

thm <val_conforms_to-mono-lemma>
if "TE wf_tyenv"
    "HT0 htyping_leq HT1"
    "(TE,HT0) |- val rval_conforms_to val_ty" <hastype_in_HT0>
then "(TE,HT1) |- val rval_conforms_to val_ty";
proof
  proceed by rule induction on <hastype_in_HT0> with val,val_ty variable;
    case Prim: qed;
    case NullToRef: qed;
    case Addr: qed by <htyping_leq.rool>,<in_fdomain>,<rval_conforms_to.Addr>;
    case Trans: qed by <rval_conforms_to.Trans>;
  end;
end;
```

```
thm <frame-mono-lemma>
if "TE wf_tyenv"
   "HT0 htyping_leq HT1"
   "(TE,HT0) |- frame frame_conforms_to FT"
then "(TE,HT1) |- frame frame_conforms_to FT";
proof
  qed by <frame_conforms_to.derive> [<oblig>],
         <frame_conforms_to.stackvar-conforms>,
         <frame_conforms_to.frame-domains-eq>,
         <val_conforms_to-mono-lemma>;
end;

thm <heapobj-mono-lemma>
if "TE wf_tyenv"
   "HT0 htyping_leq HT1"
   "(TE,HT0) |- heapobj heapobj_conforms_to refty"
then "(TE,HT1) |- heapobj heapobj_conforms_to refty";
proof
  qed by structcases("heapobj"),
         <heapobj_conforms_to>,
         <fldvals_conform_to>,<els_conform_to>,
         <val_conforms_to-mono-lemma>;
end;



thm
if "TE wf_tyenv"
   "ST0 styping_leq ST1"
then <exp-mono-lemma>
  if "(TE,ST0) |- exp rexp_conforms_to ty" <exp_conforms_to>
  then "(TE,ST1) |- exp rexp_conforms_to ty"
and <exps-mono-lemma>
  if "(TE,ST0) |- exps rexps_conform_to tys" <exps_conform_to>
  then "(TE,ST1) |- exps rexps_conform_to tys"
and <stmt-mono-lemma>
  if "(TE,ST0,rt) |- stmt rstmt_conforms" <stmt_conforms>
  then "(TE,ST1,rt) |- stmt rstmt_conforms";
proof
  proceed by weak rule induction on
   <exp_conforms_to> with exp,ty,ST0,ST1 variable,
   <exps_conform_to> with exps,tys,ST0,ST1 variable,
   <stmt_conforms> with stmt,ST0,ST1,rt variable;

    case StackVar: qed by <styping_leq>,  <rexp_conforms_to.StackVar>;
    case Access: qed by <rexp_conforms_to.Access> ;
    case Field: qed by <rexp_conforms_to.Field>;
    case Value: qed by <val_conforms_to-mono-lemma>, <styping_leq>;
    case NewClass: qed;
    case NewArray:  qed by <rexp_conforms_to.NewArray>;
    case Call: qed by <rexp_conforms_to.Call>;
    case Trans: qed by <rexp_conforms_to.Trans>;
    case Exps: qed by <rexps_conform_to.Exps>;
    case Body:
      qed  by structcases("ty"),<rexp_conforms_to.Body> ["TE","ty"],
             <frame-mono-lemma>, <styping_leq>,<pair_forall_elim>;
```

```
    case AssignToStackVar:
      qed by <rstmt_conforms.AssignToStackVar>,
             <styping_leq>,<pair_forall_elim>;

    case AssignToField: qed by <rstmt_conforms.AssignToField>;
    case AssignToArray: qed by <rstmt_conforms.AssignToArray>;
    case If: qed by <rstmt_conforms.If>;
    case While: qed by <rstmt_conforms.While>;
    case Expr: qed by <rstmt_conforms.Expr>;
    case Return: qed by <rstmt_conforms.Return>;
    case Block: qed by <rstmt_conforms.Block>, <all>;
  end;
end;


//-------------------------------------------------------------------------
// Various operations on the state produce a narrower, conformant state.
// Allocation first.
//
// First prove "initial" creates values that conform

thm  <initial-values-conform>
  if "TE |- ty wf_type"
  then "(TE,HT) |- initial(ty) rval_conforms_to ty";
proof
  consider pt st "ty = PrimTy(pt)"
     by <initial>,<goal>,
       structcases("ty"),
       <rval_conforms_to.NullToRef> ["TE","HT"];
  qed by structcases("pt"),<initial>,<rval_conforms_to> ["initial(ty)","ty"],<prim_type>;
end;


//-------------------------------------------------------------------------
// Simple allocation preserves conformance, if all the conditions are right.
//

thm <object-alloc-conforms-lemma>
  if "TE wf_tyenv"
     "TE |- ST0 wf_styping"
     "TE |- C wf_class"
     "flds = fpfun_of_frel (AllFields(TE,C))"
     "fldvals = initial o_f flds"
     "heapobj = OBJECT(fldvals,C)"
     "sAlloc(heap0,heapobj) = (heap1,addr1)"
     "TE |- s0 state_conforms_to ST0"
     "s0 = (frame0,heap0)"
     "s1 = (frame0,heap1)"
  then "∃ST1. TE |- ST1 wf_styping ∧
              TE |- s1 state_conforms_to ST1 ∧
              flookup(snd(ST1))(addr1) = Some(ClassTy(C)) ∧
              ST0 styping_leq ST1" <oblig>;
proof
  have "∀fld ty. (fld,ty) ∈ AllFields(TE,C) → TE |- ty wf_type" <fields_wf>
     by <AllFields-wf>;
```

```
  let "ST0 = (FT0,HT0)";
  let "HT1 = HT0 <?++ (addr1,ClassTy(C))";

  have "HT0 htyping_leq HT1" // <HT0_leq>
    by <htyping_leq>,<sAlloc>,
       <freshi> ["fdomain heap0"],<state_conforms_to>,<heap_conforms_to>;

  have "(TE,HT1) |- fldvals fldvals_conform_to C"
    by  <fields_wf>,
        <object-fields-form-graph> ["TE","C"],
        <initial-values-conform> ["TE","HT1"],<fldvals_conform_to>,
        <frel_is_graph_rool> ["AllFields(TE,C)"/R];

  have "TE |- heap1 heap_conforms_to HT1"
    by <heap_conforms_to>,<sAlloc>, <freshi> ["fdomain heap0"],
       <eq_fsets>,<state_conforms_to>,
       <heapobj-mono-lemma>,<heapobj_conforms_to>;

  have "TE |- HT1 wf_htyping" by <wf_styping>,<wf_htyping>;

  qed by <wf_styping>, <state_conforms_to>,
         <oblig> ["(FT0,HT1)"], <frame-mono-lemma>, <styping_leq>;
end;
```

Bibliography

[AGMT98] J. S. Aitken, P. Gray, T. Melham, and M. Thomas. Interactive theorem proving: An empirical study of user activity. *Journal of Symbolic Computation*, 25(2):263–284, February 1998.

[And97] James H. Andrews. Executing formal specifications by translation to higher order logic programming. In Elsa L. Gunter and Amy Felty, editors, *Theorem Proving in Higher Order Logics*, Lecture Notes in Computer Science, pages 17–32. Springer-Verlag, 1997.

[Bai98] Anthony Bailey. *The Machine-checked literate formalisation of algebra in type theory*. PhD thesis, Department of Computer Science, University of Manchester, 1998.

[BM81] R. S. Boyer and J Strother Moore. Integrating decision procedures into heuristic theorem provers. Technical report, Univ. of Texas, 1981.

[BN98] F. Baader and T. Nipkow. *Term Rewriting and all that*. Cambridge University Press, Cambridge, 1998.

[Bou92] R. Boulton. Boyer-Moore automation for the HOL system. In L.J.M. Claesen and M.J.C. Gordon, editors, *International Workshop on Higher Order Logic Theorem Proving and its Applications*, pages 133–145, Leuven, Belgium, September 1992. IFIP TC10/WG10.2, North-Holland. IFIP Transactions.

[Bou95] R. J. Boulton. Combining decision procedures in the HOL system. *Lecture Notes in Computer Science*, 971:75–??, 1995.

[Bou97] R. J. Boulton. A tool to support formal reasoning about computer languages. *Lecture Notes in Computer Science*, 1217:81, 1997.

[CM92] J. Camilleri and T.F. Melham. Reasoning with inductively defined relations in the HOL theorem prover. Technical Report 265, University of Cambridge Computer Laboratory, August 1992.

[COR+95] Judy Crow, Sam Owre, John Rushby, Natarajan Shankar, and Mandayam Srivas. A tutorial introduction to PVS. In *Proceedings of the*

Workshop on Industrial-Strength Formal Specification Techniques, Baco Raton, Florida, 1995.

[DE97a] S. Drossopoulou and S. Eisenbach. Java is type safe — probably. *Lecture Notes in Computer Science*, 1241:389ff, 1997.

[DE97b] Sophia Drossopoulou and Susan Eisenbach. Is the Java type system sound? (version 2.01). Technical report, Imperial College, University of London, Cambridge, CB2 3QG, U.K., January 1997. This version was distributed on the Internet. Please contact the authors if a copy is required for reference.

[DE98] Sophia Drossopoulou and Susan Eisenbach. What is Java binary compatibility? Accepted for publication at Object Oriented Programming, Systems, Languages and Applications, Vancouver, Canada, 1998.

[DFW96] Drew Dean, Edward W. Felten, and Dan S. Wallach. Java security: from HotJava to Netscape and beyond. In IEEE, editor, *1996 IEEE Symposium on Security and Privacy: May 6–8, 1996, Oakland, California*, pages 190–200, 1109 Spring Street, Suite 300, Silver Spring, MD 20910, USA, 1996. IEEE Computer Society Press.

[FGMP90] Amy Felty, Elsa Gunter, Dale Miller, and Frank Pfenning. λprolog. In M. E. Stickel, editor, *Proceedings of the 10th International Conference on Automated Deduction*, volume 449 of *LNAI*, pages 682–681, Kaiserslautern, FRG, July 1990. Springer Verlag.

[Fro93] Jacob Frost. A Case Study of Co-induction in Isabelle HOL. Technical Report 308, University of Cambridge, Computer Laboratory, August 1993.

[GJS96] James Gosling, Bill Joy, and Guy Steele. *The Java Language Specification*. Addison-Wesley, 1996.

[GM93] M.J.C Gordon and T.F Melham. *Introduction to HOL: A Theorem Proving Assistant for Higher Order Logic*. Cambridge University Press, 1993.

[GM95] E. Gunter and S. Maharaj. Studying the ML module system in HOL. *The Computer Journal*, 38(2):142–151, 1995.

[GMW77] Michael Gordon, R. Milner, and Christopher Wadsworth. Edinburgh LCF. Internal Report CSR-11-77, University of Edinburgh, Department of Computer Science, September 1977.

[Gun94] Elsa L. Gunter. A broader class of trees for recursive type definitions for HOL. In Jeffery Joyce and Carl Seger, editors, *Higher Order Logic Theorem Proving and Its Applications*, volume 780 of *Lecture Notes in Computer Science*, pages 141–154. Springer-Verlag, February 1994.

[Har95] J. Harrison. Inductive Definitions: Automation and Application. In E.T. Schubert, P.J. Windley, and J. Alves-Foss, editors, *8th International Workshop on Higher Order Logic Theorem Proving and its Applications*, volume 971 of *Lecture Notes in Computer Science*, pages 200–213, Aspen Grove, Utah, USA, September 1995. Springer-Verlag.

[Har96a] J. Harrison. HOL light: A tutorial introduction. In M. Srivas and A. Camilleri, editors, *First international conference on formal methods in computer-aided design*, volume 1166 of *Lecture Notes in Computer Science*, pages 265–269, Palo Alto, CA, USA, November 1996. Springer Verlag.

[Har96b] J. Harrison. A Mizar Mode for HOL. In J. Von Wright, J. Grundy, and J. Harrison, editors, *Ninth international Conference on Theorem Proving in Higher Order Logics TPHOL*, volume 1125 of *Lecture Notes in Computer Science*, pages 203–220, Turku, Finland, August 1996. Springer Verlag.

[Har97a] John Harrison. First order logic in practice. In Maria Paola Bonacina and Ulrich Furbach, editors, *Int. Workshop on First-Order Theorem Proving (FTP'97)*, RISC-Linz Report Series No. 97-50, pages 86–90. Johannes Kepler Universität, Linz (Austria), 1997.

[Har97b] John R. Harrison. Proof style. Technical Report 410, University of Cambridge Computer Laboratory, Cambridge, CB2 3QG, U.K., January 1997.

[HJ89] Ian J. Hayes and Cliff B. Jones. Specifications are not (necessarily) executable. Technical Report UMCS-90-12-1, Department of Computer Science, University of Manchester, Manchester M13 9PL, England, December 1989.

[Hut90] Matthew Hutchins. Machine assisted reasoning about Standard ML using HOL, November 1990. Australian National University Honours Thesis.

[Inw96] Myra Van Inwegen. *The machine-assisted proof of programming language properties.* PhD thesis, University of Pennsylvania, December 1996.

[Jac88] M. I. Jackson. An overview of VDM. *SafetyNet*, 2, September 1988.

[JDD94] J. Joyce, N. Day, and M. Donat. S: A machine readable specification notation based on higher order logic. *Lecture Notes in Computer Science*, 859:285ff, 1994.

[JH94] Sverker Janson and Seif Haridi. An introduction to AKL, A multi-paradigm programming language. In B. Mayoh, E. Tyugu, and J. Penjaam, editors, *Constraint Programming: Proceedings 1993 NATO ASI Parnu, Estonia*, NATO Advanced Science Institute Series, pages 411–443. Springer, 1994.

[JJM+95] J. Gulmann, J. Jensen, M. Jørgensen, N. Klarlund, T. Rauhe, and A. Sandholm. Mona: Monadic second-order logic in practice. In U.H. Engberg, K.G. Larsen, and A. Skou, editors, *TACAS*, pages 58–73. Springer Verlag, LNCS, 1995.

[KB70] D. E. Knuth and P. B. Bendix. Simple word problems in universal algebra. In J. Leech, editor, *Computational problems in abstract algebra*, pages 263–297. Pergamon Press, Elmsford, N.Y, 1970.

[KM96a] Matt Kaufmann and J. Strother Moore. ACL2: An industrial strength version of Nqthm. *COMPASS — Proceedings of the Annual Conference on Computer Assurance*, pages 23–34, 1996. IEEE catalog number 96CH35960.

[KM96b] Matt Kaufmann and J Strother Moore. ACL2: An industrial strength version of nqthm. In *Compass'96: Eleventh Annual Conference on Computer Assurance*, page 23, Gaithersburg, Maryland, 1996. National Institute of Standards and Technology.

[Lov68] D. W. Loveland. Mechanical Theorem Proving by Model Elimination. *Journal of the ACM*, 15:236–251, 1968.

[LP92] Zhaohui Luo and Robert Pollack. The LEGO proof development system: A user's manual. Technical Report ECS-LFCS-92-211, University of Edinburgh, May 1992.

[LY97] Tim Lindholm and Frank Yellin. *The Java Virtual Machine Specification*. The Java Series. Addison-Wesley, Reading, MA, USA, January 1997.

[Mau91] M. Mauny. Functional programming using CAML. Technical Report RT-0129, National Institute for Research in Computer and Control Sciences (INRIA), 1991.

[MBBC95] Z. Manna, N. Bjoerner, A. Browne, and E. Chang. STeP: The Stanford Temporal Prover. *Lecture Notes in Computer Science*, 915:793ff, 1995.

[McA89] David McAllester. *Ontic*. The MIT Press, Cambridge, MA, 1989.

[ME93] M. van Inwegen and E.L. Gunter. HOL-ML. In J.J. Joyce and C.-J.H. Seger, editors, *International Workshop on Higher Order Logic Theorem Proving and its Applications*, volume 780 of *Lecture Notes in Computer*

Science, pages 59–73, Vancouver, Canada, August 1993. University of British Columbia, Springer-Verlag, published 1994.

[Mel88] Thomas F. Melham. Automating recursive type definitions in higher order logic. Technical Report 146, University of Cambridge Computer Laboratory, Cambridge CB2 3QG, England, September 1988.

[Mel91] T. F. Melham. A mechanized theory of the π-calculus in HOL. In *Informal Proceedings of the Second Logical Framework Workshop*, May 1991.

[Mel92] Thomas F. Melham. The HOL logic extended with quantification over type variables. In L.J.M. Claesen and M.J.C. Gordon, editors, *International Workshop on Higher Order Logic Theorem Proving and its Applications*, pages 3–18, Leuven, Belgium, September 1992. IFIP TC10/WG10.2, North-Holland. IFIP Transactions.

[Mil80] Robin Milner. *A Calculus for Communicating Systems*, volume 92 of *Lecture Notes in Computer Science*. Springer-Verlag, Berlin, 1980.

[MLP79] Richard A. De Millo, Richard J. Lipton, and Alan J. Perlis. Social processes and proofs of theorems and programs. *Communications of the ACM*, 22(5):271–280, May 1979.

[MS97] Muller and Slind. Treating partiality in logic of total functions. *COMPJ: The Computer Journal*, 40, 1997.

[MTDR88] M.J.C. Gordon, T.F. Melham, D. Sheperd, and R. Boulton. *The UNWIND Library*. Manual part of the HOL system, 1988.

[MTHM97] Robin Milner, Mads Tofte, Robert Harper, and David MacQueen. *The Definition of Standard ML (Revised)*. The MIT Press, 1997.

[MW97] William McCune and Larry Wos. Otter—the CADE-13 competition incarnations. *Journal of Automated Reasoning*, 18(2):211–220, April 1997.

[Nes92] M. Nesi. Formalizing a modal logic for CSS in the HOL theorem prover. In L.J.M. Claesen and M.J.C. Gordon, editors, *International Workshop on Higher Order Logic Theorem Proving and its Applications*, pages 279–294, Leuven, Belgium, September 1992. IFIP TC10/WG10.2, North-Holland. IFIP Transactions.

[Nip89] Tobias Nipkow. Equational reasoning in Isabelle. *Science of Computer Programming*, 12(2):123–149, July 1989.

[Nip96] T. Nipkow. More Church-Rosser proofs (in Isabelle/HOL). *Lecture Notes in Computer Science*, 1104:733ff, 1996.

[NL95] William Newman and Mik Lamming. *Interactive Systems Design*. Addison-Wesley, December 1995.

[NN96] D. Nazareth and T. Nipkow. Formal verification of algorithm W: The monomorphic case. In J. Von Wright, J. Grundy, and J. Harrison, editors, *Ninth international Conference on Theorem Proving in Higher Order Logics TPHOL*, volume 1125 of *Lecture Notes in Computer Science*, pages 331–346, Turku, Finland, August 1996. Springer Verlag.

[NO79] C. G. Nelson and D. C. Oppen. Simplifications by cooperating decision procedures. *ACM Transactions on Programming Languages and Systems*, 1(2), 1979.

[NO80] Greg Nelson and Derek C. Oppen. Fast decision procedures based on congruence closure. *Journal of the ACM*, 27(2):356–364, April 1980.

[Nor98] Michael Norrish. *C Formalized in HOL*. PhD thesis, University of Cambridge, August 1998. Submitted for examination.

[Nv98] T. Nipkow and D. von Oheimb. Java$_{light}$ is type-safe — definitely. In *25th ACM Symp. Principles of Programming Languages*. ACM Press, 1998.

[Pau90] L. C. Paulson. Isabelle: The next 700 theorem provers. In P. Odifreddi, editor, *Logic and Computer Science*, pages 361–385. Academic Press, 1990.

[Pau94] L.C. Paulson. A fixedpoint approach to implementing (co)inductive definitions. In Alan Bundy, editor, *Proceedings of the 12th International Conference on Automated Deduction*, volume 814 of *LNAI*, pages 148–161, Berlin, June/July 1994. Springer.

[Pau97] L.C. Paulson. Proving properties of security protocols by induction. In *PCSFW: Proceedings of The 10th Computer Security Foundations Workshop*. IEEE Computer Society Press, 1997.

[PB97] Roly Perera and Peter Bertelsen. The unofficial java bug report, June 1997. Published on the WWW at `http://www2.vo.lu/homepages/gmid/java.htm`.

[Plo91] Gordon D. Plotkin. A structural approach to operational semantics. Technical report, Computer Science Department, Aarhus University, DK-8000 Aarhus C. Denmark, September 1991.

[PM93] Chr. Paulin-Mohring. Inductive definitions in the system Coq; rules and properties. *Lecture Notes in Computer Science*, 664:328ff, 1993.

[PR98] F. Pessaux and F. Rouaix. *The O'Caml-Tk implementation*, 1998. From the O'Caml system distribution., available at `http://pauillac.inria.fr/caml`.

[Qia97] Zhenyu Qian. A Formal Specification of Java Virtual Machine Instructions. Technical report, Universität Bremen, FB3 Informatik, D-28334 Bremen, Germany, November 1997.

[Ras95] Ole Rasmussen. The Church-Rosser theorem in Isabelle: A proof porting experiment. Technical Report 364, University of Cambridge, Computer Laboratory, March 1995.

[Rud92] P. Rudnicki. An overview of the MIZAR project, 1992. Unpublished; available by anonymous FTP from `menaik.cs.ualberta.ca` as `pub/Mizar/Mizar_Over.tar.Z`.

[Rus93] John Rushby. Formal methods and the certification of critical systems. Technical Report SRI-CSL-93-7, Computer Science Laboratory, SRI International, Menlo Park, CA, December 1993. Also issued under the title *Formal Methods and Digital Systems Validation for Airborne Systems* as NASA Contractor Report 4551, December 1993. A book based on this material will be published by Cambridge University Press in 1998/9.

[SA98] R. Stata and M. Abadi. A type system for Java bytecode subroutines. In *25th Annual ACM Symposium on Principles of Programming Languages*, pages 149–160, January 1998.

[SHC96] Zoltan Somogyi, Fergus Henderson, and Thomas Conway. The execution algorithm of Mercury, an efficient purely declarative logic programming language. *Journal of Logic Programming*, 29(1–3):17–64, October–November 1996.

[Sho84] R.E. Shostak. Deciding Combinations of Theories. *Journal of the ACM*, 31(1):1–12, January 1984.

[Sli96] K. Slind. Function definition in higher order logic. In J. Von Wright, J. Grundy, and J. Harrison, editors, *Ninth international Conference on Theorem Proving in Higher Order Logics TPHOL*, volume 1125 of *Lecture Notes in Computer Science*, pages 381–398, Turku, Finland, August 1996. Springer Verlag.

[Spi67] Michael Spivak. *Calculus*. W. A. Benjamin, Inc., New York, 1967.

[Spi88] J. M. Spivey. *Understanding Z*. Cambridge Tracts in Theoretical Computer Science 3. Cambridge University Press, 1988. ISBN 0-521-33429-2.

[SS97] Geoff Sutcliffe and Christian B. Suttner. The CADE-13 ATP system competition. *Journal of Automated Reasoning*, 18(2):137–138, April 1997.

[Sta95] Richard Stallman. *GNU Emacs manual*. Free Software Foundation, 675 Mass Ave, Cambridge, MA 02139, USA, Tel: (617) 876-3296, USA, 11th, Emacs version 19.29 edition, June 1995. Includes GNU Emacs reference card.

[Sym93] D. Syme. Reasoning with the Formal Definition of Standard ML in HOL. In J.J. Joyce and C.-J.H. Seger, editors, *International Workshop on Higher Order Logic Theorem Proving and its Applications*, volume 780 of *Lecture Notes in Computer Science*, pages 43–59, Vancouver, Canada, August 1993. University of British Columbia, Springer-Verlag, published 1994. ftp://ftp.cl.cam.ac.uk/hvg/papers/MLinHOL.hug93.ps.gz.

[Sym95] D. Syme. A new interface for HOL — ideas, issues and implementation. *Lecture Notes in Computer Science*, 971:324–??, 1995.

[Sym97a] Don Syme. DECLARE: A prototype declarative proof system for higher order logic. Technical Report 416, University of Cambridge Computer Laboratory, Cambridge, CB2 3QG, U.K., March 1997.

[Sym97b] Don Syme. Proving Java type soundness. Technical Report 427, Computer Laboratory, Univeristy of Cambridge, June 1997.

[Tar55] A. Tarski. A fixed point theorem and its applications. *Pacific J. Math.*, pages 285–309, 1955.

[Typ98] Types mailing list archive, 1998. Available at http://www.cs.indiana.edu/types on the WWW.

[WF94] Andrew K. Wright and Matthias Felleisen. A syntactic approach to type soundness. *Information and Computation*, 115(1):38–94, November 1994.

[Wil70] Stephen Willard. *General Topology*. Addison-Wesley, New York, 1970.

[WLB98] Tim Wahls, Gary T. Leavens, and Albert L. Baker. Executing formal specifications with constraint programming. Technical Report 97-12a, Department of Computer Science, Iowa State University, 226 Atanasoff Hall, Ames, Iowa 50011, August 1998. Available by anonymous ftp from ftp.cs.iastate.edu or by e-mail from almanac@cs.iastate.edu.

Index