THE MACHINE-ASSISTED PROOF OF PROGRAMMING LANGUAGE PROPERTIES

MYRA VANINWEGEN

A DISSERTATION

in

COMPUTER AND INFORMATION SCIENCE

Presented to the Faculties of the University of Pennsylvania in Partial Fulfillment of the Requirements for the Degree of Doctor of Philosophy.

August 1996

Carl Gunter

Supervisor of Dissertation

Peter Buneman

Graduate Group Chairperson

ABSTRACT

THE MACHINE-ASSISTED PROOF OF PROGRAMMING LANGUAGE PROPERTIES

Myra VanInwegen

Advisor: Carl Gunter

The goals of the project described in this thesis are twofold. First, we wanted to demonstrate that if a programming language has a semantics that is complete and rigorous (mathematical), but not too complex, then substantial theorems can be proved about it. Second, we wanted to assess the utility of using an automated theorem prover to aid in such proofs. We chose SML as the language about which to prove theorems: it has a published semantics that is complete and rigorous, and while not exactly simple, is comprehensible. We encoded the semantics of Core SML into the theorem prover HOL (creating new definitional packages for HOL in the process). We proved important theorems about evaluation and about the type system. We also proved the type preservation theorem, which relates evaluation and typing, for a good portion of the language. We were not able to complete the proof of type preservation because it is not true: we found counterexamples. These proofs demonstrate that a good semantics will allow the proof of programming language properties and allow the identification of trouble spots in the language. The use of HOL had its plusses and minuses. On the whole the benefits greatly outweigh the drawbacks, enough so that we believe that these theorems could not been proved in amount of time taken by this project had we not used automated help.

Contents

List of Tables

Chapter 1

Introduction

As computer (hardware and software) systems come to play a larger role in our lives, it becomes increasingly important to make sure that these systems operate as intended. While no method can guarantee that a system will work perfectly, there are several ways to increase confidence that a system will perform as expected when used in the setting for which it was designed.

A system *specification* is a precise statement of what the system must do, without saying how it is to be done. The specification must also include a description of the environment in which the system will operate. Engineers use *validation*, *verification*, and *testing* to help ensure correctness of systems. Validation is the process of checking that the specification is correct, that is, that the assumptions about the environment are valid and that the system designers and the customer agree on what the system should do. Testing involves running the system with a collection of inputs and checking that the outputs conform to the specification. Verification consists of looking at the system itself in order to determine if it meets the specification, without actually running it. Verification usually involves the use of mathematical logic to prove that an algorithm or circuit meets its specification or satisfies some given properties.

The problem with testing is that in complex systems there are too many cases to test them all. Occasionally one can prove that the input can be divided up into a small number of classes, and that it is sufficient to test one input from each class. This is often not the case. When only a few of the inputs can be tested, testing does not ensure the correctness of the system. This is what gives verification its appeal. When a system (or some abstraction of it) is verified, it has been *proved* to be correct (at that level of abstraction).

There are two big obstacles to verifying systems. The first is that one needs to have precise mathematical descriptions of what the system is supposed to do (the specification) and of what the system actually does. An example of the first is a finite state automata that gives the action of the system in each state and indicates how the system goes from state to another. An example of the second specification is a mathematical model of a circuit. Creating these mathematical descriptions is a bigger problem than one might guess. Much of the work in formal methods (which is the use of logic-based mathematics to increase confidence in systems) has been in formulating these mathematical descriptions.

The second obstacle is that even if one has a rigorous (mathematical) description of the desired properties and actual function of the system, proofs of correctness are usually large and complex. This not only makes the proofs long and tedious, but also increases the chances that they will contain mistakes. Thus one would like to have some kind of automated help in doing the proofs. Not only can an automated proof tool help organize and carry out the proof, but it acts as a proof checker, preventing errors.

Automated proof tools can be roughly divided into two classes. The first is used to check properties of systems that can be modeled as finite state automata. These systems, called *model checkers*, are fully autonomous: when given an input consisting of a finite state automata and a property to be checked, they will, after some computation, report either that the system satisfies the property, or return a state for which property fails. A good reference for the use of model checkers in verifying properties of systems is [McMillan92].

Model checkers do not work in all situations. Sometimes the system is not easily describable as a finite state automaton, either because one would need an infinite number of states, or because a more complex mathematical model is needed. For instance, a system that can be configured in an variety of ways or can have any number of components cannot be easily described as a finite state automaton. An example of this would be a cache coherence protocol, where model checking can only be used to verify the protocol in certain fixed configurations.

The other class of proof tools are semi-automated, general-purpose theorem provers. (For the rest of this thesis, the term "theorem prover" will refer to this second class of proof tools.) Theorem provers implement a logic (for example, first-order logic) and provide some automated support for proving theorems in the logic. One would like it to be the case that, when the theorem prover is given a statement, if the statement is true then a proof is returned, and if it is not true, then some proof of its falseness (perhaps

a counterexample) is returned. However, the logics implemented by theorem provers are usually powerful enough that it is impossible (for theoretical, not just practical, reasons) to give an algorithm that will do this. Thus these theorem provers can only find proofs for a limited number of statements, and they often require a great deal of human input to guide them in the proof.

The human effort required can make using a theorem prover a tedious process. One of the annoyances with using a theorem prover is that it demands that everything be proved in great detail. Statements that are "obviously true" to the mathematically-knowledgeable human are not obvious to the theorem prover: it must be convinced with a detailed proof.[1] Theorem provers can be augmented with decision procedures to help with this problem. Decision procedures are programs that when given a statement of a certain form (say, in a restricted subset of the logic) will determine whether or not it is true and, if it is true, return a proof. There is still much work to be done before theorem provers are sufficiently powerful that they can be feasibly applied to prove correctness of large, real-world systems.[2]

It is useful to note that even if verification becomes more easily applicable to real-world problems, it will never completely supplant testing. Mathematics is sometimes not completely convincing or understandable. An analogy can be drawn between formal verification of systems and formal definitions of mathematical concepts. It is often the case that a definition is not fully understood until several examples have been given. For the same reason, people will never be contented with seeing a mathematical proof of correctness of a system—there is always the possibility that there is something wrong with the theorem prover, or it may be the case that the mathematical description of the functionality of the system fails to match the actual implementation of the system. They will need to look at some of the proofs, to read them and understand them, and they will want to plug some real inputs into the system and check the results.

We are interested in formal verification of software. Our approach is to prove properties of programming languages. The motivation for doing this is that if one proves a property of a language, then this property is true of any program written in the language. Examples of properties that one might like to prove include determinism (for any program, if it is run twice with the same input, it will return the same result) and that certain language

[1] This is one of its advantages as well. Often statements thought to be "obviously true" turn out to be subtly false.

[2] Verifications have been done for large, real-world systems, but only at a relatively high level of abstraction.

constructs cannot cause side effects. Some languages have a strong notion of type: each phrase in the language can be given a type, and the type system determines how these types fit together in order for the program to be legal. A typechecker looks over the code to make sure that the rules of the type system are satisfied by the program. The typechecker thus does a consistency check on the program. This check can catch many common errors, such as mistakenly using a composite data item in a place where a component of the item is needed or passing arguments to functions in the wrong order (if the arguments being reversed have different types).

A property that one would like to be satisfied by a programming language with a typechecker is *type soundness*: for any program, if the program passes the typechecker, then it will not have a type error when it is executed. Type errors include the attempted use of a non-function as if it were a function (such as applying the number 3 to an argument) and the use of a function with the wrong type of argument (such as the addition of 3 and false). This is not trivial: a great deal of processing can take place in computing the value of a function or an argument, and there are many opportunities for things to go wrong. The most common sources of error in programming languages are constructs that allow a value (such as a function) to have different types in different situations. An example of such a function is a list reverse function that can be used on any sort of list. Proving properties about typechecking in a language with such constructs can be tricky.

In order to prove properties of a language, one needs to have a mathematical specification of the language, that is, a mathematical description of the form of programs (the grammar of the language), a specification of program evaluation, and, if the language has a typechecker, a description of how one confirms that the types are consistent. If the semantics of a language is not given in a mathematical form, one will have to be created for it. The difficulty of formalizing the semantics depends on the precision, clarity, and complexity of the existing specification.

Proofs of programming language properties have been accomplished for languages with mathematical semantics (for example, see [CW85], [Tofte90], [WF92]), but these have almost always been for "toy" languages: they were formulated for the purpose of demonstrating programming language concepts and proof techniques. To the author's knowledge, up to the present, there has been only one project involving proofs of programming language properties for "real" (in the sense of having a user community and a good compiler)

programming languages.[3]

Programming language specifications have been with us for over three decades, roughly since the publication of the original Algol 60 report [Naur60]. Perhaps the most important lesson to be learned from studying the history of programming language specifications is that they are difficult to get right, where getting it right can be taken to mean that the language constructs are unambiguously described and that the description is understandable. Chapter 3 gives some examples of programming language specifications in various styles and points out their advantages and deficiencies.

Standard ML (SML) is a "real" language with a complete published mathematical semantics, *The Definition of Standard ML* [MTH90]. SML is a mostly-functional language with a strong notion of type, and all implementations of SML provide a typechecker. SML has features that make it not only pleasant for programming, but also a good object of study for programming language instruction. The semantics is phrased in mathematical terms, making it precise, but the techniques used are not so deep as to be impenetrable. The evaluation and type system of the language are given by proof rules that state when a program can be given a certain type and determine the value to which a program evaluates.

Because of the size of SML (the number of phrases in the grammar and the complication of some of the rules), it is natural to try to use a theorem prover to help with proving properties of the language. HOL (for Higher Order Logic) is a good choice for such a theorem prover. It is typed and has sound, easily-extensible definitional principles, which allow the encoding of the syntax and semantics of SML. In addition, its higher order features enable the definition of relations (such as the evaluation relations) as the smallest relations satisfying a set of rules.

The project proceeded in this manner. First, we encoded the dynamic semantics of Core SML, as given in the Definition. In order to do this we needed to define a mutually recursive types definition package for HOL. The method we used for encoding semantics is described in Chapter 5. We then proved some theorems about evaluation, the most important of which was that evaluation is deterministic. This proof is described in Section 6.2. We decided that, in order to give the project a goal to aim at, we would try to prove type preservation for Core SML. Type preservation is, like type soundness, a property that relates the static (type system) and dynamic (evaluation) semantics of the language; a proof of type preservation is usually one of steps used in a proof of type soundness.

[3]This is the work of Donald Syme, described in Chapter 2.

Type preservation says that if a program is given a type τ by the type system, and the program evaluates to a value v, then v has type τ. Type preservation, and our efforts to prove it, are described in detail in Chapters 7 and 8. In order to allow a proof of type preservation, we changed the semantics of evaluation to fix some well known bugs [Kahrs93, Kahrs94] and encoded the static semantics, also fixing known bugs. In addition, we wrote a package to automate the definition of mutually recursive relations in HOL. This package is described by example in Section 5.3. We set about trying to prove type preservation, and we succeeded in completing the proof for most of the language (Section 8.2).

When we tried to prove type preservation for the part of the language including expressions, we ran into problems with substitution. That is, we found that we had to prove a large number of lemmas about substitution in order to proceed with the proof. The difficulties we experienced with substitution are described in Sections 8.2–8.4. During the course of proving substitution lemmas, the close examination of the semantics uncovered some problems (described in Section 8.5) with the semantics of SML that would prevent a proof of type preservation. We did not see any clear way to fix these problems, and, having spent already a year since beginning the proof of type preservation, we decided to wrap up the project. We had done enough work to demonstrate that, given a good (not too complex) mathematical semantics for a language, one can prove significant properties of the language and find mistakes and potential trouble spots in its definition. We also felt that we had enough experience with HOL to assess its utility in this situation. Our conclusions about SML and HOL are in Chapter 9.

In the process of proving theorems, we found that some parts of the language were causing a great deal of difficulty because of their complexity . The most prominent of these were imperative types, equality types, and the special rules for scoping explicit type variables. We eliminated these parts of the language in order to allow the proofs to be accomplished in a reasonable amount of time. A complete list of changes is in Appendix A, and a discussion of why they are reasonable is in Section 9.1. We refer to the changed version of the language as HOL-SML.

Chapter 2

Related Work

In this chapter we briefly describe several projects that are similar to this one. The first two are projects at Computational Logic Inc., involving the programming language Gypsy. The other two are ML-related. One is an encoding of the syntax and dynamic semantics of Core SML in HOL88 by Donald Syme. The other is the encoding of the syntax and static and dynamic semantics of Mini-ML, a functional ML-like language, in Typol.

Gypsy is an imperative language with many constructs, including support for concurrency and data abstraction. It is designed to support program verification. A Gypsy program includes, along with the instructions to be executed to compute its result, the specification of what that result should be. One of the ways in which Gypsy supports reasoning about programs is that it disallows aliasing: no non-local variables are allowed in procedures, there is no construct allowing a variable access to another variable's value (say through a pointer assignment), and structure sharing is not allowed among call-by-reference procedure parameters. The Gypsy Verification Environment [Cohen89] allows a user to develop, specify, and prove correctness of Gypsy programs.

Here we discusses two aspects of Computational Logic's Gypsy work. The first is the work of Donald Good, Ann Siebert, and William Young on the Middle Gypsy 2.05 Definition, a mathematical description of most of the Gypsy language in the logic of the Boyer-Moore theorem prover. The second is the work of William Young on a verified code generator for the language Micro-Gypsy.

The language Middle Gypsy differs little from the language obtained by removing concurrency and data abstraction from Gypsy. The Middle Gypsy 2.05 Definition [GSY90] presents a mathematical specification of this language in the logic of the Boyer-Moore

theorem prover. The definition is given via two functions, **F** and **P**, which describe the mathematical and computational aspects of programs. **P** (the computer program mechanism) models the computational parts of the program, including the change of state occurring as the program executes. It is allowed to reflect the limitations of a machine running the program, such as a finite amount of memory. **F** (the mathematical expressions part of the language) models the application of mathematical functions. The mathematical expressions are used for checking that the procedural interpretation satisfies the operating constraints, such as the entry and exit specifications.

The functions **F** and **P** can be compared to the evaluation relations we defined for HOL-SML. Our relations cannot be encoded as HOL functions (which must be everywhere defined) because SML programs are not guaranteed to terminate and because evaluation is not defined for all programs that can be constructed from SML syntax. Their language definition can be expressed using functions, even in the presence of recursive procedures, by the inclusion of parameters that state the maximum number of recursive calls allowed during the course of evaluation. As with SML, there are some programs that, because of problems such as type errors, do not evaluate correctly. This problem is handled in the Gypsy definition by defining the result of the evaluation to be a marked object, which is a pair $[m, o]$ of a mark m and an object o. The mark is used to indicate errors that occur during the evaluation of the phrase. If the mark is *nil*, then no errors have occurred during the evaluation of the phrase. Both **F** and **P** take as arguments a phrase to be evaluated (an expression or program segment), an environment giving values for the free variables in the phrase, and an integer stating the maximum of steps to be taken, which ensures that the functions terminate. **F** returns a marked, typed value, and **P** returns a marked, typed store.

Several things should be noted about this definition. First, a quote from the Preface says something about the complexity and utility of this definition:

> ...because the definition is in rigorous mathematical form, it is possible to state and prove invariant properties about the definition. ...However, rigorous formulation and proof of these properties was not possible within the scope of this effort. However, doing these proofs would have increased substantially our belief that the 200 pages of Boyer-Moore logic does say the "right" thing.

The fact that the definition indeed fills 200 pages with Boyer-Moore logic will make proving

theorems about the definition very hard.

Second, the expression or procedure to be evaluated is expressed as a sequence of ASCII characters, which may not even be legitimate Gypsy program phrases. If there are syntax errors or typing errors, these are indicated by the mark in the evaluation result. Evaluation is similar to LISP evaluation in the sense that errors are reported only if they occur in the section of code that actually gets evaluated. This is not part of the Gypsy definition; the static and dynamic semantics can be checked separately if so desired. While it does not seem to be useful to evaluate ASCII sequences that are not even generated by the Gypsy grammar, the way the evaluator is expressed enables a proof of a stronger theorem than the type preservation theorem. Type preservation can be phrased roughly as: if a phrase P has type τ and it evaluates to a value v, then v has type τ. For this system, if there were a separate static semantics, one could try to prove a result something like: if a program P typechecks, then either (a) there does not exist an n such that P terminates after n steps, or (b) there exists an n such that P terminates after n steps with a marked store $[m, s]$, and either m is *nil* and for each variable x if x is declared to have type τ then the value associated with x in the resulting store has type τ, or else m is one of a restricted class of run-time errors, such as divide by zero. This is analogous to the statement of strong type soundness for SML: if the phrase passes the typechecker (that is, it elaborates to some type τ), then either the program gets into some sort of infinite loop, or else it evaluates to a value that has type τ.

The Micro-Gypsy work is part of a CLI effort called the *Short Stack*, which consists of the high-level language Micro-Gypsy, the assembly language Piton, the FM8502 processor, and a gate-level design for the processor. The connections between the levels—the code generator translating Micro-Gypsy to Piton, an assembler that translates Piton to machine code for the FM8502, and the design of the FM8502—have been formally verified, allowing trusted programs to be written in Micro-Gypsy.

Young's work, the formally verified Micro-Gypsy code generator, is the top of the stack. Micro-Gypsy is a small subset of Gypsy including simple data types, arrays, and procedural abstraction. The operational semantics of Micro-Gypsy is an interpreter written as a recursive function (called MG-MEANING) in the Boyer-Moore theorem prover. As the focus of this paper was on the code generator and its verification, the MG-MEANING function is taken for granted and is not described in depth; however, it is indicated in [GSY90] that the semantics of Micro-Gypsy is similar to the mathematical semantics for

Middle Gypsy.

The code generator is implemented by a function in the Boyer-Moore logic called MAP-DOWN, which not only translates Micro-Gypsy statements into Piton, but also translates the background against which these statements are executed (called an mg-state) into equivalent structures for Piton (a p-state). The mg-state includes the binding of variables to values (the combination of SML's environment and memory) and current condition (analogous to SML's exceptions). MAP-UP is a mapping from a p-state to an mg-state that, given information on the expected types of the variables, reconstructs the bindings for an mg-state from the temporary stack of a p-state. The correctness of the code generator is phrased in terms of the functions MAP-DOWN, MAP-UP, MG-MEANING, and Piton interpretation (which is not given a name). The idea is to prove that MG-MEANING = MAP-UP $\circ$ Piton-interp $\circ$ MAP-DOWN, that is, the mg-state resulting from applying MAP-DOWN, running the Piton interpreter on the resulting p-state, and then applying MAP-UP is the same as MG-MEANING applied to the initial mg-state. The actual proof eliminates MAP-UP by showing that it is a left inverse of MAP-DOWN and then showing that MAP-DOWN $\circ$ MG-MEANING = Piton-interp $\circ$ MAP-DOWN. This is done by induction on the evaluation defined by MG-MEANING.

Donald Syme's encoding of Core SML uses HOL88, an older version of HOL than HOL90, the version used for our work. His encoding has many similarities to ours, as well as some significant differences. The encoding of the reduced syntax is similar to ours. However, instead of developing a mutually recursive type definition package of his own, he used one developed by Claudio Russo of Edinburgh. To encode the semantic objects needed for evaluation he also developed a finite map theory for HOL; however, as Russo's package did not allow the nesting of type definitions under constructors, he used an axiomatization to express the properties that the types needed to satisfy. As noted in Section 4.1, axioms should be avoided in HOL since it is easy to introduce inconsistency.

The evaluation relations were defined in a somewhat different way than ours. As explained in Chapter 5, we defined our relations as the smallest relations satisfying the evaluation rules. Syme first defined *inference trees*, structures in which each node is a statement about the evaluation of an SML phrase. An inference tree is valid if the statement in each node can be inferred from its children via one of the evaluation rules. Then a sentence is defined to be inferable if there exists a valid inference tree with that sentence as the conclusion. Syme notes that he used an encoding of addresses and the reference

creation rule (Rule 114) that eliminates nondeterminism in the choice of the new address.

Syme then defined HOL inference rules and tactics based on the evaluation relations and rules. These rules and tactics allow him to prove properties of evaluation in a manner similar to that described in Chapter 6 for our system. Using these tactics, he proved determinacy of SML evaluation and the pattern matching theorem.

He then implemented a symbolic evaluator that helps significantly in proving properties of specific properties. Syme realized that he could not prove correctness of programs without some notion of the types of the values involved. However, his approach is primitive—he attempts to select out subsets of the SML values that satisfy simple structural properties. This approach only can provide a crude approximation to types: a list is defined as anything that is constructed with a backbone of `::` (`cons`es), and a pair is defined as value that is a record with labels `1` and `2`. As explained in Section 7.2, this idea will work for verifying only a few functions. Using this technology, Syme proved the correctness of the append function and an interpreter for a tiny expression language.

Mini-ML is a small applicative language including mutually recursive higher-order functions, function definition via pattern matching, and polymorphism. In [CDDK86] Dóminique Clément, Joëlle Despeyroux, Thierry Despeyroux, and Gilles Kahn describe the static and dynamic semantics of the language and give a translation to an abstract machine, the Categorical Abstract Machine (CAM). The paper first presents the syntax of the language and then shows two different natural semantics formulations of static semantics for the language and proves them equivalent. One of these forms of static semantics is deterministic, allowing type inference. The dynamic semantics are also shown in a natural semantics formulation. The CAM is described, and a translation from Mini-ML code to CAM code is given. This translation has been proved correct by Thierry Despeyroux. The static and dynamic semantics of this language, the CAM, and the translation from Mini-ML to CAM code were all were encoded into Typol, which is a system that allows natural semantics specifications to be rapidly entered into a computer, and then allows these specifications to be executed.

Chapter 3

Programming Language Specifications

This chapter describes five programming language specifications (for Algol 60, LISP, Scheme, Algol 68, and Standard ML) in order to give some examples of different specification styles and the difficulties involved in describing the meaning of program constructs in a clear, accurate, and understandable way. The first section, which examines the specifications of the first four languages listed, may be skipped without much loss, as its main goal is to provide examples of specifications that can be contrasted with that of SML.

3.1 A Brief History

A programming language specification consists of descriptions of the syntax of the language and a precise description of the effects or values of the definitions, expressions, and commands of the language. Occasionally there are constraints other than the syntax that need to be satisfied in order for the program to be considered proper. One such set of constraints is a type system, which is checked by a typechecker.

Usually there are some general concepts that need to be explained, such as the scope of variables and the different forms of formal parameters to procedures. Programming language specifications usually serve two audiences: users and implementors. The specification should include everything a user needs to know in order to program in the language and everything that an implementor needs to know in order to create a compiler or interpreter for the language. Often, language specifications leave open some details about

implementation, particularly in areas such as real arithmetic or input/output, and both users and implementors need to know the guidelines that implementations must follow.

It is difficult to specify exactly what a programming language construct does. The confusion comes from two sources. The first is that the designers may have had some concept in mind that they did not describe clearly in the semantics. This leads to confusion in the mind of the reader, while there was none in the mind of the designer. These issues are not hard to resolve, at least in theory: one can ask the designers. An example of this problem, described below, is the incorrect specification of Algol 60's if-then-else forms for both expressions and statements. The second source of confusion is that the designers may not have had a clear idea of exactly what they wanted the programming language construct to do, or, even worse, different designers had different ideas as to what the construct should do. These issues can be difficult to resolve. An example of this is the question of whether Algol 60 functions can have side effects.

In this section, one of the issues addressed is determinism. Recall that a deterministic program is one that gives the same answer every time it is evaluated on the same input. A programming language is deterministic if every program written in the language is deterministic. Many modern programming languages are not intended to have this property. For example, the semantics of a language that supports multiple threads of computation often specifies that any thread that is currently ready for execution can be chosen to run. In contrast, one expects that a classic single-threaded language should be deterministic, and, although determinism of a language is not often stated, it is implicit in the presentation of the constructs—specifications refer to "the" result of a computation. However, many informal (non-mathematical) programming language specifications can be interpreted in such a way that the language is nondeterministic. We will examine instances of this below.

The "Report on the Algorithmic Language ALGOL 60" [Naur60] is considered to have founded the field of programming language specification. It was the first report to use the Backus-Naur Form (named after two of the authors of the report) of grammar specification, and the meanings of programming language constructs were explained more carefully than in previous language reports. The language was not based on an existing implementation, but was designed by an international committee. In fact, the authors stress that the choice of constructs and their semantics were chosen without regard to how easily they could be implemented.

The Algol 60 report is exemplary in its organization, giving a clear syntax, a semantics

(a description of what the construct is intended to do), and examples for each construct. The semantics is written informally, that is, using prose rather than mathematics to express the meaning of the constructs. The language is specified well enough that a user familiar with programming languages could write substantial programs in the language. Yet, despite its overall quality, there are many mistakes and ambiguities in the report. It is worth looking at some of these problems, as they are typical of the informal descriptions of programming languages. These problems are discussed in detail by Donald Knuth in his 1967 paper "The Remaining Trouble Spots in ALGOL 60" [Knuth67].

The first ambiguity that becomes apparent to the reader is that errors are not well described. Niklaus Wirth writes [Wirth74] that "the real challenge in compiling is not the detection of correct sentential forms, but coping with ill-formed, erroneous programs, in diagnosing the mistakes and in being able to proceed in a sensible way". Thus the programming language specification must be specific about what constitutes an error and what ranges of options the implementor has in dealing with the error. Questions that must be answered are: which errors are supposed to be detected at compile-time? Which errors must be reported at run-time? What errors are allowed to go undetected?

The Algol 60 report fails miserably in this. In Section 3.1.4.2 on subscripts, we find:

> The value of the subscripted variable is defined only if the value of the subscript expression is within the subscript bounds of the array.

In Section 3.3.6, on **real** arithmetic, we find:

> Numbers and values of type **real** must be interpreted in the sense of numerical analysis, i.e. as entities defined inherently with only a finite accuracy. ...No exact arithmetic will be specified, however, and it is understood that different hardware representations may evaluate arithmetic expressions differently.

What exactly do "undefined" and "unspecified" mean? There is a footnote in Section 1 on this, but it is not helpful:

> Whenever the precision of arithmetic is stated as being unspecified, or the outcome of a certain process is said to be undefined, this is to be interpreted in the sense that a program only fully defines a computational process if the accompanying information specifies the precision assumed, the kind of arithmetic assumed, and the course of action to be taken in all such cases as may occur during the course of the execution of the instruction.

It is a mistake to lump together actions that should produce an error (for example, subscripts out of range) with actions whose effects are allowed to vary between implementations. In fact, Knuth points out that because of this lumping together of two different concepts (and the observation that if a program does not terminate, its result is surely undefined) and the wording of the semantics of a go to statement applied to an undefined label, one can interpret the report as saying that the implementation of the language must solve the halting problem! Another problem is that if a result of a computation is unspecified, it can return a different value for different executions of the program, resulting in nondeterminism.

The explanations of what is meant by the **if-then-else** construct provide examples (mentioned by Knuth) of unclear or misleading explanations in language specifications. One of the problems concerns the syntax and semantics of the **if-then-else** construct. When this construct is used with expressions, the **else** part is mandatory.

⟨if clause⟩ ::= **if** ⟨Boolean expression⟩ **then**
⟨arithmetic expression⟩ ::= ⟨simple arithmetic expression⟩|
⟨if clause⟩ ⟨simple arithmetic expression⟩ **else** ⟨arithmetic expression⟩

Yet in the explanation of the result produced by an arithmetic expression formed with a succession of if clauses, we find:

> The construction:
> **else** ⟨simple arithmetic expression⟩
> is equivalent to the construction:
> **else if true then** ⟨simple arithmetic expression⟩

However, the second construction is not allowed by the syntax, thus rendering this explanation useless.

Another mistake is associated with **if-then-else** constructs for statements. It is stated in Section 4.5.3.2 that there are two forms of the statement (with and without the **else**), and the effects of each when there are multiple if clauses is given. One of the sentences in this section is:

> If none of the Boolean expression of the if clauses is true, then the effect of the whole conditional statement will be equivalent to that of a dummy statement.

Knuth points out that either this sentence should be either deleted or it should be made

clear that it refers only to the form of the conditional in which there is no **else** part. He then says:

> This well-known error ...would have been fixed in the Revised Report except for the fact that these proposals were tied to other ones involving side effects; in the heated discussions which took place, the less controversial issues were overlooked.

One of the controversial issues for Algol 60 was that of side effects and the consequent questions over order of evaluation. There are two forms of Algol procedures. One form is essentially shorthand for a set of statements and is used only for the changes it effects on the program state (the values of variables). The other form, a *function*, returns values that can be used in expressions. If a function, in addition to returning a value, also changes the state, then it is said to have side effects. The report is not clear as to whether side effects are allowed. Because functions can do everything that procedures can, one might think that functions can have side effects. However parts of the report assume that no side effects occur—for example, the quote from Section 4.5.3.2 indicates that, if all Boolean expressions evaluate to false and there is no else clause, then either no side effects have occurred in the evaluation of the Boolean expressions, or if they have, then they must be undone.

Thus the debate arose: *should* side effects be allowed? Arguments were made on both sides. On the one hand, side effects can be useful (for example, in a function that returns a "random" real number). On the other hand, side effects can obscure the execution of a program: a reader of a program usually does not expect that a function will have side effects. If side effects are allowed, then there is good reason to specify the order of evaluation. In Section 3.3.5 on arithmetic expressions, the report reads "The sequence of operations within one expression is generally from left to right", but this means that, for example, $a/b \times c$ is to be interpreted at $(a/b) \times c$ rather than $a/(b \times c)$. In evaluating a/b, it is not specified whether a or b should be evaluated first. The problem is that if a or b are expressions containing calls to functions with side effects, the result can be different depending on which is evaluated first. If an order of evaluation is not specified, then an implementation is allowed to choose a different order of evaluation for different runs of a program, and the language is nondeterministic. Thus allowing side effects implies that the order of evaluation should be specified. However, there is a reason to leave order of

evaluation unspecified: implementors might want to evaluate expressions in a certain way in order to speed up the execution. This conflict was never resolved for Algol 60.

The original LISP report [McCarthy] was published in 1960, the same year as the first Algol 60 report, but the method of language description is very different: the LISP paper appears to be more a paper in mathematics than in computer science. This paper gives a formal description for the meaning of programs through the use of an interpreter for the language written in a language of mathematical concepts.

The structure of the paper is as follows: First general mathematical notions such as conditional expressions and Church's λ-notation are discussed. Then symbolic expressions (S-expressions), the syntactic mechanism that describes the form of all LISP's code and data structures, are defined. Then the basic functions on them (atom, eq, car, cdr, and cons) are described mathematically. Then several other functions (such as subst and append) are defined using the mathematical notions and the basic functions. Then it is shown how to represent as S-expressions the basic mathematical notions and functions on S-expressions. Then comes the most interesting part: the description of how to evaluate a program, which is given via the definition of a function eval. Eval takes two arguments, an expression to be evaluated and a list of pairs that acts as an environment for the evaluation. Eval is one of several functions (along with evcon and evlist) that define the evaluation of LISP programs.

The LISP report, while mathematically precise about what correct programs ought to do, does not make clear what an implementation should do with incorrect programs. For example, the description of car specifies that "car[x] is defined if and only if x is not atomic. car $[(e_1 \cdot e_2)] = e_2$" (here $(e_1 \cdot e_2)$ is an ordered pair). This specifies the mathematical meaning of car (it is a partial function whose domain is the non-atomic objects), but it does not say what happens if the object to which it is applied is atomic, which is important computationally. The description of the implementation on the IBM 704 does not help: car is implemented by a machine instruction that assumes that the address of the cell containing the pair is in the appropriate machine register. This does not say whether or not the compiler should do any checks as to whether the argument is non-atomic before issuing the car instruction.

Some questions about the implementation are answered only by a careful look through

the code for the interpreter. For example: is the language call-by-value (the expressions for the actual parameters are evaluated before being paired with the formal parameters) or call-by-name (the expressions for the actual parameters are used without being evaluated first)? The English description of the interpreter says "the evaluation of $((\text{LAMBDA}\,(x_1 \ldots x_n)\,\mathcal{E})\,e_1 \ldots e_n)$ is accomplished by evaluating $\mathcal{E}$ with the list of pairs $((x_1\,e_1)\ldots(x_n\,e_n))$ put on front of the previous list a". This suggests that the interpreter is call-by-name. However, examining the code for the interpreter one finds a different story. It is clear from the code that eval is mapped over the list of arguments before they are paired with the formal parameters and put into the environment a.

A confusing aspect of this report is the lack of an explicit statement as to what the allowed forms of programs are. For example,

$$((\lambda x.(\text{eq}\,x\;1 \rightarrow \lambda y.\text{car}\,y, \text{T} \rightarrow \lambda y.\text{cdr}\,y))1)\,(3 \cdot 4)$$

is in the mathematical language presented in the paper and thus can be translated into LISP's S-expressions. Yet eval will not evaluate this term: the cases the eval function provides for the application of an expression F to a collection of arguments do not include the possibility that F is itself an application.

Algol 68 [vW60] was the official successor to Algol 60. The designers hoped that it would be the language of the future: it would include everything that a programmer could need and would have a precise syntax and semantics. To this end, all debates over Algol 60 were resolved, and many features were added to the language, while some of the more problematic features (such as call-by-name procedure parameters) were abolished. The precise semantics was provided by a new form of grammar, the vW grammar, after its inventor and the main author of the Algol 68 report, A. van Wijngaarden. This grammar allowed a precise and complete specification of the syntax of the language and included many aspects of the type system, such as ensuring that a variable is defined consistently with its use. Unfortunately, this power and precision came at the expense of comprehensibility. This two-level, context-sensitive grammar was difficult to understand. To make matters worse, the authors invented new terminology for many familiar features. For example, "multiple value" and "denotation" were used instead of "array" and "constant". This was done to avoid preconceived notions of what the components of a programming language should do, but the effect was to make the specification nearly unreadable.

The combination of a complicated grammar and new terminology proved to be the

undoing of Algol 68: its programming language innovations were overwhelmed by a nearly inscrutable specification. However, some of these ideas, such as coercions between different types and the ability to create and manipulate pointers appeared later in other languages, such as C [KR78].

Scheme, a version of LISP, was first described in 1975. The "Revised^3 Report on the Algorithmic Language Scheme" [RC86] shows how far informal specifications of programming languages had come since the original Algol report. It has the same general format as the Algol 60 report. In addition, in a section (Section 7) towards the end of the paper, the formal syntax and semantics are given. The syntax in Section 7 is given using BNF notation, and the semantics for a large portion of the language is given using a form of denotational semantics. It is clear that this programming language specification owes much to its predecessor; in fact the Scheme report is "Dedicated to the Memory of ALGOL 60". The Scheme specification is much more complete than that of Algol 60, but it is also much longer (43 pages as opposed to 16 for Algol 60). A good portion of its additional length comes from describing portions of the language left completely up to implementations by the Algol 60 report such as input/output routines and real arithmetic.

The biggest improvement over the Algol 60 report is that the Scheme report is specific about what constitutes an error, and errors are distinguished from what is left open to the choice of the implementor. In addition, requirements are established for those parts of the language that are allowed to vary with the implementation.

There are still some questions left open by the specification. Order of evaluation is not unambiguously described. Furthermore, it is treated differently in the informal and formal parts of the report. In a note in Section 4.1.3, it is state that "In contrast to other dialects of Lisp, the order of evaluation is unspecified." Since Scheme functions are allowed to have side effects, if a different order of evaluation is chosen in different runs of a program, then Scheme is nondeterministic. Thus one wonders: does "unspecified" allow a different choice of evaluation order on different runs of a program?

In Section 7.2 we find:

> The order of evaluation within a call is unspecified. We mimic that here by applying arbitrary permutations *permute* and *unpermute*, which must be inverses, to the arguments in a call before and after they are evaluated. This still requires that the order of evaluation be constant throughout a program (for any given number of arguments), but it is a closer approximation to the

intended semantics that a left-to-right evaluation would be.

This paragraph not only does not clarify what the "intended semantics" of unspecified order of evaluation are, but it calls into question the relationship between the formal and informal semantics. The wording in the quote above suggests that this is just one interpretation of the informal description. However, at the beginning of Section 7 we find: "This chapter provides formal descriptions of what has already been described informally in previous chapters of the report," which indicates that this is the official explanation of what is meant by an unspecified order of evaluation.

The question "Is Scheme nondeterministic?" was recently posed to a collection of Scheme experts. The answer supplied by most (including Matthias Blume, Jim Miller, and Morry Katz) was that Scheme is in fact a nondeterministic language. This, however, is not a commonly known fact: Mitchell Wand was under the impression that different executions of the same procedure call must have the same order of evaluation, and thus the language was deterministic.[1]

One final aspect of the denotational semantics in the Scheme report will be mentioned: it is hard to read. It relies on notations that are only summarized in the report. These notations make the resulting semantics nearly incomprehensible to someone new to this type of semantics. It is difficult to understand even for those who have studied denotational semantics because it is somewhat involved and because the definitions of some of the meaning functions have been omitted. As an example of the mathematical complication of this semantics, this, the clause for an `if` without an else branch, is one of the shortest in the semantics:

$$\mathcal{E}[\![(\texttt{if}\, E_0\, E_1)]\!] = \lambda\rho\kappa.\mathcal{E}[\![E_0]\!]\rho(\mathit{single}(\lambda\epsilon.\mathit{truish}\,\epsilon \to \mathcal{E}[\![E_1]\!]\rho\kappa, \mathit{send}\,\mathit{unspecified}\kappa))$$

The problems with Algol 60 and Scheme semantics mentioned above illustrate some of the ambiguities inherent in informal programming language specifications. These ambiguities, combined with complicated constructs, can make it difficult to formulate a mathematical semantics for a languages.

We examined three mathematical specifications above: the Algol 68 report, the LISP report, and the section on denotational semantics in the Scheme report. While these specifications have the advantage of being more precise than their informal counterparts, they

[1] This poll was conducted by the author over email in September 1994.

are also deficient in some regards. Not only the mathematical but also the computational aspects of programs need to be specified; in particular, errors and the ways in which they are dealt with need to be clearly described. The mathematical notation used needs to be as clear and as simple as possible, completely explained within the specification, and, in addition, there needs to be some informal discussion of the formal semantics. The LISP semantics fails to provide information regarding the computational properties of the language. The denotational semantics in the Scheme report fails to define and explain the methods used. Finally, the Algol 68 report fails because of the inherent complexity of its specification and because of the new terminology employed.

3.2 SML and its Specification

The specification of SML is given by a small book called *The Definition of Standard ML* [MTH90]. Since its publication in 1990, it has been widely recognized as setting the standard for the rigorous specification of a "real" programming language (as opposed to a toy language such as Fun in [CW85]). A companion volume, *Commentary on Standard ML* [MT91], provides explanations and examples for the material covered mathematically in the Definition. These publications are referred to in the rest of this document as the Definition and the Commentary respectively.

This section consists of two parts. The first describes the features that distinguish SML from other languages. The second gives an overview of how SML is specified by explaining a few of the typechecking and evaluation rules. Some of the ideas is this presentation were clarified by [Appel92]. This section can be skipped by readers familiar with SML and its specification.

SML: The Language. SML is divided into two levels—the Core and Modules. The Core is essentially the programming part of the language, while the Modules system is used to provide structure and extra generality to large and complex programming tasks by collecting together related types, functions, and data items. In order to make the task of encoding and proving properties about the language more manageable, we have only dealt with the Core language. Gunter and Maharaj have built on this work to encode and prove properties about the Modules [MG93]. As the work described in this thesis concerns the Core language, the following description will concentrate on the Core.

Before a program is compiled or is executed by an interpreter, it is checked for type

correctness. The typechecker can catch a large portion of simple errors, for example, calling a function with the wrong number of arguments or putting the arguments in the wrong order. In languages without a typechecker, programs with obvious flaws can be compiled or interpreted and will work fine—until the defective section of code is encountered. The effects of not catching such mistakes early on depend on the programming language. With LISP-like languages such as Scheme, these problems are detected at run-time as the bad code is about to be executed, and an error message is printed out. With C, there is no run-time checking[2], and the compiled code is executed exactly as programmed[3], which often results in corrupting the program memory. This may result in the program crashing while executing code entirely unrelated to the section of code containing the problem, which makes tracking down the bug extremely difficult.

SML's close interaction between typechecking and evaluation results in *run-time safety*. Run-time safety means that a program will execute without encountering errors that result in the program behaving unpredictably. Scheme and SML are both safe. C is not, and strange things can happen when program errors corrupt the stack. Run-time safety for Scheme is accomplished by run-time checking, while for SML it is ensured by *type soundness*. Type soundness means that if a program passes the typechecker, then it will not have a type error when it is executed. Type soundness is achieved by another property, *type preservation*, which means that if the typechecker verifies that a program has a certain type, then when it is executed the result is a value that has the same type. Thus compiled code can assume that data objects really are what they are supposed to be, and run-time checks can be dispensed with. Thus, in addition to catching errors earlier (during typechecking), SML can potentially execute faster than Scheme.

Type preservation ensures type soundness by catching during typechecking errors that could cause the program to go wrong, such as trying to add an integer to a boolean. However, there are some errors that cannot be caught before execution, such as division by zero. These errors are dealt with by the *exception mechanism*. This mechanism captures information pertaining to the error in a *packet*, which is passed back in place of a value as the result of the computation. These packets can be caught and handled by the user, and if they are not caught the program terminates. In addition, users can declare and raise their own exceptions, thus allowing more control over the flow of programs. As exceptions

[2]The C specification [KR78] does not require such run-time checking. However, there are a variety of quality C implementations that have run-time checking, and some even ensure complete run-time safety.

[3]Of course, many C gurus see this as a feature, not a bug.

are part of SML's language definition and type system, they provide a carefully specified way of dealing with errors that occur during execution.

Despite its power in finding program errors, a type system might be considered too much of a bother if it gets in the way of programming. Consider Pascal, in which every variable and function has to be given a type explicitly. This results in a significant amount of program text being dedicated to supplying type information. There are two major problems with this. The first is that much of this information is not needed. For example, in the line of code `b := y > 0`, it must be the case the `b` is a boolean. Thus, with some form of *type inference*—the process of inferring the types of variables and functions from their contexts—many of these declarations could be omitted. The second problem is that sometimes the types of functions end up being too specific. The code for inserting an item in a list is essentially the same no matter what type of data the list contains, yet with languages such as Pascal, these functions must be re-implemented every time a new list type is created.

SML gets around these difficulties by using polymorphic type inference. Polymorphism means that the same object can have many different types. There are many different forms of polymorphism; the kind used in SML is *parametric polymorphism*. A function is parametrically polymorphic if its arguments can have any of a usually-infinite number of types (all having the same general structure), and the result is computed in a uniform way for each type of argument. See [CW85] for a discussion of the forms of polymorphism and good descriptions of each.

List processing functions are the typical example of polymorphic functions—once the generic type of lists has been defined, functions that do not depend on the type of element in the list will be applicable to all lists. For example, a function that reverses the elements in a list can be given the type $(\tau\,\texttt{list}) \rightarrow (\tau\,\texttt{list})$ for any type τ.[4]

Type inference involves using rules of program structure to determine the types of variables and functions. For example, if a function is applied to the argument `5` and returns a value that is then concatenated to a string, then the type of the function must be `int` $\rightarrow$ `string`. It is important to distinguish type inference from the type system. The type system is given by a set of rules that specify the type that a phrase can be given, based

[4]Lists are so useful that they, along with some basic list-processing functions, have been specified in the Definition. However, the types and functions that result from these specifications are no different from those that users can program themselves, and re-implementing list types and functions is often given as an example in SML books and classes.

on the types of the subphrases. Type inference is an algorithm the figures out that types of phrases based on program structure. When the term "typechecker" was introduced in this thesis, it was defined as an algorithm that verifies that the type system rules are satisfied by a program. A type inference algorithm, if it is to be of any use, must also agree with the type system, in the sense that types of a phrase and its subphrases determined by the type inference algorithm must be related according to the rules of the type system. In this document, the term typechecker is used with SML to mean an algorithm that does type inference as well as verifying that the type rules are satisfied.

In SML, the type inference algorithm must satisfy an additional constraint: it must return the *most general type* for a phrase. For example, the identity function `fn x => x`, which returns whatever argument was given it, can be given the type $\tau \rightarrow \tau$ for any type τ. Thus it can be given the types, for example, `int` $\rightarrow$ `int`, or (`bool list`) $\rightarrow$ (`bool list`), or $\alpha \rightarrow \alpha$ for a type variable α. Its most general type is the last one, since any other permissible type can be obtained from it by replacing the α with some other type.

Despite powerful type inference, explicit types in the syntax are occasionally necessary. Sometimes they are needed to resolve overloaded functions, such as `+`, which can be used with both integers and reals, or to specify the complete type of a record argument to a function. However, only a few type annotations suffice to guide the typechecker in finding the type for a function.

In SML, functions are first-class data objects. This means that they can be used in the same sorts of situations as the basic data types such as booleans or integers. In particular, functions can be created on the fly and passed back as the return values of functions and can be used as arguments to functions. This allows quite a bit of flexibility in programming. For example, the ability to create and return functions with local state can be used for object-oriented programming.

While SML includes non-functional features such as references (mutable values) and exceptions, it emphasizes functional programming by segregating the functional part of the language from the non-functional part. Reference cells (variables whose values can be changed via assignment) have different types than other values: a reference cell that contains an integer has type `int ref`. Assignment and dereference operations work only on values with `ref` type. This is unlike Scheme, which, while mainly considered a functional language, allows the value of any variable to be changed. The segregation of non-functional aspects of the language by SML simplifies reasoning about programs—if a variable does

not have reference type, then its value will not change over the course of evaluation, that is, within the scope of a variable, the entity referred to by the variable does not change.

SML is usually run under a read-eval-print loop. This interactive method of compilation helps facilitate the development and testing of small pieces of code.

The final feature that distinguishes SML from other languages is the use of pattern matching for defining functions. This is easiest to explain with an example. Below is a function that adds up the elements in an integer list:

```
fun add (b::bs) = b + (add bs)
  | add [] = 0
```

The pattern `b::bs` matches a value that is the constructor `::` (SML's infix equivalent to Scheme's cons) applied to two values that match `b` and `bs`. The pattern `[]` matches only the empty list. This definition says that the function `add` applied to a non-empty list is the value of the first item in the list added to the sum of the rest of the list, and the sum of an empty list is 0.

It is informative to see how one could infer the type of the function in the paragraph above. The argument must be of type τ `list` for some τ, since the constructor `::` is a constructor for the list type and `[]` is the empty list. The function must return an integer, since the returned value for an argument that is the empty list is 0. Thus we have pinned down the type of the function to τ `list` $\rightarrow$ `int`. Now we only need to determine what τ is. The variable `b` must be of type τ and `bs` must be of type τ `list` because the type of `::` here is $(\tau * \tau$ `list`$) \rightarrow \tau$ `list` ($*$ represents the pair type). Now `b` is an argument to `+`, so `b` must be either an integer or a real. We know that the return value of `add` is `int`, thus `(add bs)` has type `int`, and since the two arguments of `+` must have the same type, then `b` must also be an integer. Thus τ is `int`, and the type of the function is (`int list`) $\rightarrow$ `int`. The type inference algorithm, called algorithm $\mathcal{W}$ [Milner78], works by assigning type variables as the types of all unknown identifiers and then uses rules typified by the reasoning above, in conjunction with unification, in order to determine the types of functions and variables.

SML's specification. The syntax of Core SML is given in Chapter 2 of [MTH90] using a modified BNF style that uses line breaks instead of | to indicate alternatives. For example,

the syntax of expressions *exp* is:

exp	::=	*atexp*	atomic expressions
		exp atexp	application (L)
		exp_1 *id* exp_2	infixed application
		exp : *ty*	typed (L)
		exp `handle` *match*	handle exceptions
		`raise` *exp*	raise exception
		`fn` *match*	function

where an L in the rightmost column means that the construct is left associative. This little section of syntax mentions three of the sixteen phrase classes: there is Exp, ranged over by *exp*, AtExp (atomic expressions), ranged over by *atexp*, and Match (matches, which are clauses that use pattern-matching to specify a function), ranged over by *match*. It is worth noting that the syntax specification is a mutually recursive definition of the phrase classes in the language.

Note the absence of the forms that one would expect to see, such as an if-then-else clause. This is because the grammar presented here is the *bare* language, a subset of the full grammar. The rest of the constructs are derived forms that can be expressed in terms of the bare language.[5]

As mentioned above, the dynamic semantics (evaluation) and static semantics (type system) are defined by sets of rules. The Definition gives the following overview of the use of rules to define these relations.

> The job of a language-definer is twofold. First—as we have already suggested—he must create a world of meanings appropriate for the language, and must find a way of saying precisely what these meanings are. ...
>
> The second part of the definer's job is to define *evaluation* precisely. This means that he must define at least *what* meaning, M, results from evaluating any phrase P of his language ...
>
> We shall now explain the keystone of our semantic method. First, we need a slight but important refinement. A phrase P is never evaluated *in vacuo* to a meaning M, but always against a *background*; this background—call it B—is

[5]These derived forms are given in the Definition's Appendix A.

itself a semantic object, being a distillation of the meanings preserved from evaluation of earlier phrases. ...

The keystone of this method, then, is a certain kind of assertion about evaluation; it takes the form[6]

$$B \vdash P \Downarrow M$$

and may be pronounced: 'Against the background B, the phrase P evaluates to the meaning M'. *The formal purpose of this Definition is no more, and no less, than to decree exactly which assertions of this form are true.* This could be achieved in many ways. We have chosen to do it in a structured way, as others have, by giving rules which allow assertions about a *compound* phrase P to be inferred from assertions about its *constituent* phrases $P_1, \ldots, P_n$.

Later it is clarified that the rules for elaboration (the type system rules) as well as evaluation are expressed using assertions involving a background, a phrase, and a result. The assertions for elaboration are of the form $B \vdash P : M$ and can be pronounced 'Against the background B, the phrase P has type M'.

While the rules defining typechecking and evaluation can look obscure, this is due to the complexity of the meaning of the specific language constructs being described, not due to the inherent complexity of the method of specification (see [Tofte90] for much simpler rules in the same style). In addition, while descriptive material is somewhat scarce in the Definition, the Commentary gives explanations of the rules and examples of how they are used. Armed with these two books, one can rapidly become acquainted with the general idea of how the rules specify the language, although it takes a long time and careful study to fully understand the rules and their implications.

Chapter 4 of the Definition gives the static semantics (type rules) for the Core.[7] This chapter of the Definition follows the plan set out in the overview. Thus it begins with the definition of the meanings for the static semantics: that is, the objects (called the *semantic objects*) that are the results and backgrounds for the typing assertions. The most important definitions are various forms of types (record types, function types, constructed

[6] In the Definition, the symbol $\Rightarrow$ is used for "evaluates to" or "has the type"; however we use the symbols $\Downarrow$ and : respectively. This avoids conflict with the use of $\Rightarrow$ in propositions as logical implication, and the use of different symbols for dynamic and static semantics help avoid confusion between the two.

[7] To be more exact, it gives the rules of a subset of the bare language that is obtained by the removal of all references to infix operators. The directives that establish infix functions (and later revoke them) and the syntax forms for infixed applications are used only for parsing.

types), type schemes (polymorphic types—types that have some of their type variables universally quantified), and variable environments (maps from variables to type schemes). Variable environments are one of the components of environments, which in turn are one of the components of a context. Types and environments are the meanings to which programs elaborate, and contexts are the background against which program phrases are elaborated. Section A.2 gives the differences between the static semantics for Core SML and those for HOL-SML.

The next step, according to the outline, is to give rules that precisely define which meanings result from elaborating SML phrases. Each rule is of the form:

$$\frac{C_1 \vdash P_1 : A_1 \quad \ldots \quad C_n \vdash P_n : A_n \quad SC_1 \ \ldots \ SC_m}{C \vdash P : A}$$

Here each C_i is a context and each A_i is either a type or an environment. Generally, expression-like phrases (including expressions and expression rows) elaborate to types, while declaration-like phrases (including declarations and value bindings) elaborate to environments. Each sentence $C_i \vdash P_i : A_i$ specifies the elaboration of some subphrase of P. Usually each C_i is C with some additional assumptions, and the final result A is some combination of the intermediate results $A_1 \ldots A_n$. The SC_i are side conditions. They are used to put constraints on the semantic objects appearing in the rule, for example, specifying that a variable is in the domain of the environment. The form as shown here is slightly inaccurate, as the side conditions are actually mixed in with the hypotheses that are elaborations (which are called premises).

In order to give a flavor of the elaboration rules, the following paragraphs explain two of the rules. The static and dynamic rules are numbered, and it is easiest to refer to them by number. Rule 6 applies to atomic expressions of the form `let` *dec* `in` *exp* `end`. The rule is:

$$\frac{C \vdash dec : E \quad C \oplus E \vdash exp : \tau}{C \vdash \texttt{let}\ dec\ \texttt{in}\ exp\ \texttt{end} : \tau}$$

Here, C is a context, E is an environment, and τ is a type. The $\oplus$ operation is the modification of C by E—the bindings in E augment (or, if there are conflicts, replace) the bindings in the environment component of C (and other components in C can be affected as well). This rule says if the declarations in *dec* elaborate to an environment E and, in the context created by modifying C by E, *exp* elaborates to τ, then the entire `let` phrase elaborates to τ.

As mentioned before, SML implementations have typecheckers that perform type inference. This algorithm uses the typing rules in a more active way than that suggested above. The way Rule 6 would be interpreted is more like the following: to find the type of (`let` *dec* `in` *exp* `end`) in the context C, we first elaborate the declarations in *dec* to get an environment E consisting of bindings for variables, constructors, etc. Then we add these new bindings to C and elaborate *exp*, yielding a type τ, which is the type of the entire `let` expression.

Polymorphic types enter the typing system through the use of the Clos operation in Rules 17 and 29. Rule 17 generalizes the type of variables declared with value bindings, while Rule 29 generalizes the type of value constructors (like `::`, the SML equivalent of LISP's `cons`). Rule 17[8] is:

$$\frac{C \vdash \mathit{valbind} : VE \quad VE' = \mathrm{Clos}_{C,\mathit{valbind}} VE}{C \vdash \texttt{val}\ \mathit{valbind} : VE' \text{ in Env}}$$

See page 120 for the definition of Clos.

An example of a value declaration is

```
val third = fn lst => hd (tl (tl lst))
```

where `hd` and `tl` are SML functions that take the head and tail, respectively, of a list. This function will return the third element of its argument (assuming the list has at least three elements; if there are less than three elements, then one of the applications of `hd` or `tl` will raise an exception), and we expect that the declaration will elaborate to the type scheme $\forall\alpha.(\alpha\,\texttt{list}) \to \alpha$. Rule 19 shows that the value binding elaborates to this type by being instantiated as follows (where *vb* is short for `third = fn lst => hd (tl (tl lst))`)[9]

$$\frac{\begin{array}{c} C_0 \vdash \texttt{third = fn lst => hd (tl (tl lst))} : \{\mathit{third} \mapsto (\alpha\,\texttt{list}) \to \alpha\} \\ \{\mathit{third} \mapsto \forall\alpha.(\alpha\,\texttt{list}) \to \alpha\} = \mathrm{Clos}_{C_0,vb}\{\mathit{third} \mapsto (\alpha\,\texttt{list}) \to \alpha\} \end{array}}{\begin{array}{c} C_0 \vdash \texttt{val third = fn lst => hd (tl (tl lst))} : \\ \{\mathit{third} \mapsto \forall\alpha.(\alpha\,\texttt{list}) \to \alpha\} \text{ in Env} \end{array}}$$

C_0 here is the initial context, which contains in its environment component bindings for `hd` and `tl`. These are mapped to $\forall\alpha.\,\alpha\,\texttt{list} \to \alpha$ and $\forall\alpha.\,\alpha\,\texttt{list} \to \alpha\,\texttt{list}$ respectively. Briefly, `third = fn lst => hd (tl (tl lst))` elaborates to $\{\text{third} \mapsto (\alpha\,\texttt{list}) \to \alpha\}$

[8] The Rule 17 examined here is essentially the HOL-SML version, which is simplified from the Definition.

[9] SML syntax will be written in typewriter font. The program variables, when shown on their own, will be written in italics.

because of the rule for variables, which specifies that a variable can have any type that is a specialization of its type scheme as given by the context (thus the type of `hd` in *valbind* is α `list` $\rightarrow \alpha$), and the rule for application, which says that if one applies a function of type $\tau_1 \rightarrow \tau_2$ to an argument of type τ_1, then the result has type τ_2.

The operation Clos generalizes the type of *third*. The expression that binds *third* is `fn lst => hd (tl (tl lst))`. This is non-expansive (see page 121) because it is a function expression, and therefore the type associated with it in the variable environment can be generalized (there are no free type variables in the initial context). Thus the result of elaborating the declaration is the environment whose variable environment component is $\{\mathit{third} \mapsto \forall\alpha.(\alpha\ \texttt{list}) \rightarrow \alpha\}$ and whose other components are empty.

Chapter 5 in the Definition gives the dynamic semantics (evaluation rules) for the Core. First, the *reduced syntax* is described. The idea behind the reduced syntax is that the types are largely dealt with in the static semantics, so a great deal of type-related information can be eliminated from the language before executing it. The Definition, however, eliminates too much: datatype declarations need to be retained in order to provide information concerning value constructors for the defined types (see [Kahrs93]). In HOL-SML, essentially only the type tags on terms is removed in going to the reduced syntax. The differences the dynamic semantics for HOL-SML and that for Core SML is given in Section A.3.

The semantic objects for evaluation are values (including special values, basic values, closures, records, packets, and addresses), environments (in this chapter of the Definition, environments are maps from program variables to values), and the components of the state (including the memory, which is a map from addresses to values). Some of the values deserve some explanation. Special values are objects such as numbers and strings, atomic values that can be represented explicitly in a program. Basic values are SML's built-in functions, such as arithmetic and input/output operations. A closure is the value to which a function evaluates. The three components of a closure are the *match* that specifies the arguments and body of the function, an environment that contains bindings for the free variables in the match, and another environment used for implementing recursive definitions. A packet is a raised exception.

It is interesting to note how the segregation of the non-functional aspects of the language are reflected in the structure and use of the state. For most languages with non-functional features, an environment maps variables to *locations*, which are meant to model addresses

in the computer memory, and a *store* then maps these locations to values (as in the denotational semantics of Scheme, Section 7 of [RC86]). Thus to find the value associated with a given variable, one must access both the environment and the store. This abstraction is needed because the values of all variables can be changed, and because there may be *aliasing*. Aliasing occurs when a value in the store can be accessed (and changed) through more than one variable. One way this can come about is if the same variable is used twice as an actual parameter in a function call. Locations model aliasing by placing the value that can be multiply accessed in the store and, in the environment, mapping all the variables that can access it to the location holding the value. Then, if an assignment is made to one of the variables, the value in the store is changed, and all variables mapped to that location will report the new value as their value.

However, as noted above, SML segregates out its assignable variables from non-assignable variables. The assignable variables have reference type: if a variable is a reference cell that can contain a value of type τ, then the type of the variable is τ `ref`. An environment for SML maps variables to values, among which are addresses. Assignable variables are mapped to addresses, and other variables are mapped to other forms of value. Let us consider why non-assignable variables need not be mapped to addresses. While a non-assignable variable can be used twice as an actual parameter in a function call, its value cannot be changed within the function, so there is no need to share a location in memory.

If a language does not allow assignment (such as Mini-ML [CDDK86]) or allows assignment but forbids aliasing (such as Gypsy [Cohen89, GSY90]), then locations can be done away with entirely, and variables can be mapped directly to values.

The form of the evaluation rules is similar to that of the typing rules, except that now there is a state to deal with. The general form is thus $s, A \vdash \mathit{phrase} \Downarrow A', s'$ where s is the initial state and s' is the final state. A is an environment (except for patterns, where there is also a value—the value against which the pattern is being matched—to the left of the $\vdash$) and A' is either a value, a packet, an environment (for declarations), or FAIL (indicating a failed pattern match).

There are sufficiently many cases for the possible results of evaluations that some conventions are used to reduce the clutter in the specification. In order to describe these conventions, the following paragraphs show their effect on Rule 135. This rule was chosen because each of the conventions have some effect on the rule.

The first convention (also used for typechecking rules) concerns angle brackets. Some

phrases have optional parts, and this is indicated by the use of angle brackets in the syntax. For example, one of the forms that value bindings can take is pat `=` $exp\ \langle$`and` $valbind\rangle$. In the semantics of this phrase, one rule is made to suffice for phrases with or without this optional part, and the parts of the rule referring to the optional part of the phrase are put into angle brackets. Rule 135, one of the rules for this form of value bindings, is:

$$\frac{E \vdash exp \Downarrow v \quad E, v \vdash pat \Downarrow VE \quad \langle E \vdash valbind \Downarrow VE' \rangle}{E \vdash pat \texttt{ = } exp\ \langle \texttt{and } valbind \rangle \Downarrow VE \langle + VE' \rangle}$$

This rule is shorthand for the following two rules:

$$\frac{E \vdash exp \Downarrow v \quad E, v \vdash pat \Downarrow VE}{E \vdash pat \texttt{ = } exp \Downarrow VE}$$

$$\frac{E \vdash exp \Downarrow v \quad E, v \vdash pat \Downarrow VE \quad E \vdash valbind \Downarrow VE'}{E \vdash pat \texttt{ = } exp \texttt{ and } valbind \Downarrow VE + VE'} \tag{3.1}$$

The second convention is the state convention. Note that the above equations do not mention state at all. If the rule does not explicitly alter the state, it is left implicit. The state is put back in by replacing each sentence in the rule of the form $A \vdash phrase \Downarrow A'$ with one of the form $s, A \vdash phrase \Downarrow A', s'$ and hooking the states together in a way suggested by their arrangement in the rule. This is best illustrated with an example. After applying the state convention, Equation (3.1) becomes

$$\frac{s, E \vdash exp \Downarrow v, s' \quad s', E, v \vdash pat \Downarrow VE, s'' \quad s'', E \vdash valbind \Downarrow VE', s'''}{s, E \vdash pat \texttt{ = } exp \texttt{ and } valbind \Downarrow VE + VE', s'''} \tag{3.2}$$

The meaning of this now fully expanded rule can be explained two ways. The first is a "top-down", more proof-theoretic way: if we prove the hypotheses of the rule, then using this rule we have a proof of the conclusion of the rule. The other is a "bottom-up", more active, proof-search way: in order to find what (pat `=` exp `and` $valbind$) evaluates to, we first evaluate the expression to get a value v and new state s'. Then, using this new state and environment E we match the pattern pat against the value in order to get a state s'' and a variable environment (map from variables to values) VE that contains bindings for the variables in the pattern. Finally, in state s'' and environment E we evaluate the rest of the value bindings, getting a new value environment VE'. Finally, the result of the entire value binding is the modification of VE by VE'. Note that these two different interpretations correspond with those for the static semantics: the top-down approach is like typechecking, and the bottom-up approach is like type inference.

The final convention is the exception convention, which generates yet more rules. According to this convention, for each hypothesis of the form $s, A \vdash phrase \Downarrow A', s'$ if $phrase$

could evaluate to a packet (a raised exception), we add another rule where (a) this hypothesis has been transformed to $s, A \vdash phrase \Downarrow p, s'$ (where p is a packet), (b) this modified hypothesis is the last hypothesis in the rule, and (c) the result and final state are the packet and state resulting from this last hypothesis. Again, an example is the best way to explain this. When the exception convention is applied to Equation (3.2), we get the following two additional rules (according to the Definition, the evaluation of patterns cannot result in packets):

$$\frac{s, E \vdash exp \Downarrow p, s'}{s, E \vdash pat \texttt{ = } exp \texttt{ and } valbind \Downarrow p, s'}$$

$$\frac{s, E \vdash exp \Downarrow v, s' \quad s', E, v \vdash pat \Downarrow VE, s'' \quad s'', E \vdash valbind \Downarrow p, s'''}{s, E \vdash pat \texttt{ = } exp \texttt{ and } valbind \Downarrow p, s'''}$$

The meaning of this first additional rule is that the evaluation of *exp* resulted in an exception, so instead of trying to continue the evaluation we pass the packet on as the result of the evaluation. The meaning of the second additional rule is that the evaluation of *exp* went fine, but the evaluation of *valbind* resulted in an exception, and this exception is the result of the evaluation.

The application of these conventions means that there are quite a few more rules than is initially apparent. For HOL-SML there are 61 rules written out in Section A.3. After we expand them according to the conventions and eliminate the duplicates we get 104 rules.

Together, these rules specify the evaluation of the thirteen phrase classes (AtExp, Exp, ExpRow, AtPat, Pat, PatRow, Match, Mrule, Dec, ValBind, DatBind, ConBind, ExBind) in the reduced syntax. Note that these rules are specifying thirteen mutually recursive relations, where each is a relation between an environment, an initial state, possibly (for patterns) a value, a program phrase, some kind of evaluation result, and a final state.

Because of the number of different phrase classes (and thus evaluation and typing relations) and the number of different evaluation rules involved, any attempt to prove a non-trivial property of the evaluation or typechecking of SML programs will involve the analysis of a great number of cases. This suggests that SML is a language for which an encoding in a theorem prover would prove to be worth the effort involved.

Now that we have seen examples of both evaluation and typechecking rules, it is sensible to draw some comparisons between them. Although there are more evaluation rules, and the evaluation relations have more components than the elaboration relations, the evaluation rules are generally more straightforward. The main reason for this is that the side conditions in the evaluation rules are simple functions such as looking up a variable

in an environment or checking that a value has a certain form. The side conditions for elaboration rules can, however, be quite complicated, such as the one involving the Clos operation in Rule 17 described above.

Chapter 4

Automated Theorem Provers and HOL

There are quite a variety of theorem provers being used today. The work described here was accomplished using HOL. Below several theorem provers (Coq [HFWHD91], HOL [GM93], and Nqthm [BM79]) are described in brief in order to give an overview of the use of theorem provers and to provide the means to assess HOL's suitability for the task.

Each theorem prover implements a logic and provides methods for defining new data types, constants, and axioms, which allow users to encode their problems in the logic. Coq was developed by Gérard Huet of INRIA Rocquencourt. Its logic is the Calculus of Inductive Constructions, which is similar to Martin-Löf's intuitionistic type theory [M-L84]. HOL, as its name suggests, implements higher order logic. It was developed by Mike Gordon of the University of Cambridge. Nqthm, developed by Robert Boyer and J. Strother Moore at Computational Logic, Inc., is based on an untyped, quantifier free first-order theory of inductively defined data structures.

All of these theorem provers require a great deal of human interaction in order to prove theorems. This interaction is accomplished in different ways. Some theorem provers (for example, Coq and HOL) are mainly used in a *goal-directed* fashion: the user sets a goal (a statement to be proved) and uses *tactics* to break the goal down into smaller pieces and to solve goals. Goal-directed theorem proving will be described in detail in the next section. Coq has a nice window-oriented interface to help in goal-directed theorem proving. A similar interface is being developed for HOL. Some goal-directed theorem provers (for example, HOL) allow the user to write new tactics, while others (Coq) do not.

Nqthm is highly automated: the user presents a goal, and it tries to prove the goal. It has a great deal of deductive machinery at its disposal, including induction and rewriting. However, Nqthm cannot prove complicated theorems on its own; it must be given hints in the form of proved lemmas. Thus, in practice, Nqthm does not appear to be more powerful than goal-directed theorem provers. For users who prefer goal-directed theorem proving, there is a goal-directed interface available for Nqthm.

HOL was used in this work to prove properties of SML. It is well-suited to this task for several reasons. The first is that it is typed. That is, each HOL term has a type, and terms can only be combined into larger terms if their types match up according to certain rules.[1] This helps simplify the statement of theorems since it eliminates the need to include hypotheses that provide type information.[2] In addition, HOL provides flexible constant and type definition packages that can be extended to enable definition of even more complicated constants and types. The extensibility of these packages was needed in order be able to define the syntax and semantics of SML.

The higher order features of HOL enable us to define relations (such as the evaluation relations and typing relations) as the smallest relations satisfying a set of rules. We accomplish this by specifying that a collection of terms (representing an SML phrase and some semantic objects) is in the intersection of all relations satisfying the rules. How this works is explained in Section 5.3.

Proving properties of SML evaluation or typing involves proving one case for each rule that defines the relation. The ability to write new tactics for HOL enables one to define tactics that can automatically satisfy many of the simpler goals, leaving only the ones requiring more involved treatment.

The next section gives a brief description of HOL. This can be skipped by the reader familiar with HOL.

[1] The typing rules for HOL are much simpler than those for SML because HOL terms can have one of only four forms (constants, variables, applications, and lambda-abstractions). There are no type schemes in HOL: type variable are left free in terms. A type variable in a theorem can be instantiated to any type (via the basic rule `INST_TYPE`, shown in Table 4.1) to create a theorem about that type. HOL has a type inference mechanism, so it is not necessary to explicitly give the types of all variables used in HOL terms. However, if the inferred type contains type variables, HOL insists that the user provide names for them.

[2] In contrast, statements of theorems proved in Nqthm are peppered with LISP-like predicates that constrain the forms of variables.

4.1 A Brief Description of HOL

The version of HOL used for this work, HOL90, is implemented on top of SML (the implementation is SML/NJ, Standard ML of New Jersey [AM91]), and SML is its meta-language. Therefore every HOL term is an SML object of type `term`. It is easy to become confused about whether objects and types being manipulated belong to the SML in which HOL90 is implemented (SML/NJ), HOL itself, or the SML that we have encoded into HOL, which we call HOL-SML. We will try to be as unambiguous as possible in the following to avoid these kinds of confusions.

HOL provides a parser that allows users to enter HOL terms using ASCII characters that resemble the logical symbols they are intended to represent. However, in order to avoid inflicting this notation on the reader, all HOL terms presented here will be written using standard mathematical connectives rather than their HOL ASCII equivalents.

HOL propositions are HOL terms with HOL type `bool`. There is no HOL object for a *proof*. There are *theorems*, represented by SML type `thm`, but all an object of type `thm` says is what has been proved, not how it was accomplished. Proofs can be done either "forwards", "backwards", or a combination of the two. Forward proof starts from axioms and inference rules without hypotheses (the two are distinguished because HOL has a specific technical meaning for axiom) and builds up the desired conclusion through the application of inference rules. Table 4.1 shows HOL's basic inference rules, out of which all other rules are constructed. The name of the rule, shown to the right, is also the name of the SML function that implements it. See Section 16.3 of [GM93] for a detailed explanation of these rules. The notion of substitution (used in the rules `BETA_CONV` and `INST_TYPE`) is important for the work accomplished here and is explained in detail in Section 8.3.

Backwards proof (the goal-directed theorem proving mentioned above) involves setting a goal, which is essentially a statement of the desired theorem. The hypotheses of the desired theorem form the assumptions of the goal, and the conclusion of the desired theorem forms the conclusion of the goal. Tactics are used to break down the goal into smaller goals (*subgoals*) that are more comprehensible and easier to solve, and these subgoals can be further broken down and simplified until the user sees a way to solve them. Technically, a tactic is function that, given a goal, will return the subgoals and a *validation function*. A validation function tells how to reassemble theorems satisfying the subgoals to form a theorem satisfying the goal. A theorem satisfies a goal when the conclusion of the theorem

$$\frac{}{t \vdash t} \quad (\texttt{ASSUME})$$

$$\frac{}{\vdash t = t} \quad (\texttt{REFL})$$

$$\frac{}{\vdash (\lambda x.t_1)t_2 = t_1[t_2/x]} \quad (\texttt{BETA\_CONV})$$

$$\frac{\Gamma_1 \vdash t_1 = t_1' \quad \ldots \quad \Gamma_n \vdash t_n = t_n' \quad \Gamma \vdash t[t_1, \ldots, t_n]}{\Gamma_1 \bigcup \ldots \bigcup \Gamma_n \bigcup \Gamma \vdash t[t_1', \ldots, t_n']} \quad (\texttt{SUBST})$$

$$\frac{\Gamma \vdash t_1 = t_2}{\Gamma \vdash (\lambda x.t_1) = (\lambda x.t_2)} \quad (\texttt{ABS})$$

$$\frac{\Gamma \vdash t}{\Gamma \vdash t[\sigma_1, \ldots, \sigma_n/\alpha_1, \ldots, \alpha_n]} \quad (\texttt{INST\_TYPE})$$

$$\frac{\Gamma \vdash t_2}{\Gamma - \{t_1\} \vdash t_1 \Rightarrow t_2} \quad (\texttt{DISCH})$$

$$\frac{\Gamma_1 \vdash t_1 \Rightarrow t_2 \quad \Gamma_2 \vdash t_1}{\Gamma_1 \bigcup \Gamma_2 \vdash t_2} \quad (\texttt{MP})$$

Table 4.1: HOL Basic Inference Rules

is the same as the conclusion of the goal, and the hypotheses of the theorem are included in the assumptions of the goal. Some tactics solve goals by returning an empty list of subgoals and a validation function that, when applied to the empty list, returns a theorem satisfying the goal.

Tactics can be applied until the resulting subgoals are satisfied by axioms (or inference rules without hypotheses), but a more usual approach is a combination of forward and backwards proof. Using this method, one uses a series of tactics to break the goal up into more manageable subgoals and then uses forward inference to build theorems that satisfy these subgoals.

One of the most useful aspects of goal-directed theorem proving is that HOL manages the goals using its *goalstack*. The initial goal is set using the function `set_goal`. A goal is displayed by the goalstack with the conclusion of the goal printed above a row of underscores and the assumptions of the goal printed below the underscores. The `expand` function applies its argument to the top goal, and the subgoals are printed out and put on the stack.

The statement above that there is no HOL object for a proof is correct, but there are *proof scripts*. While proving a theorem, either in a forwards or backwards fashion, users generally enter into a file the tactics or rules used to prove a theorem, in the form of SML/NJ code that can be re-run to prove the theorem. This code has many uses. First, it can be used to re-prove the theorem in case the machine crashes before the theory can be saved (see below). Second, it can be used as a template to prove theorems with a similar structure (for example, another theorem that must be proved by induction over lists). Finally, if the definitions of objects in the statement of the theorem change, the code used to prove the theorem the first time dramatically speeds up the process of proving the theorem about the modified objects.

One problem with proof scripts is that they are close to unreadable if they are longer than about ten lines. This is because (for backwards proof) even if a user knows all the tactics used in the script, after a few lines of code, each of which transforms the goal in some way, she has lost track of the form of the goal being operated on by the tactic. This is not an insurmountable problem, however: setting the goal and re-entering the tactics a few at a time allows the user to see the how the goals evolve and to again understand the proof.

HOL is structured into *theories*. Each theory consists of some type and term constants,

$\vdash \quad \mathsf{T} = ((\lambda x_{bool}.x) = (\lambda x_{bool}.x))$
$\vdash \quad \forall = \lambda P_{\alpha \to bool}.\ P = (\lambda x.\mathsf{T})$
$\vdash \quad \exists = \lambda P_{\alpha \to bool}.\ P(\varepsilon P)$
$\vdash \quad \mathsf{F} = \forall b_{bool}.b$
$\vdash \quad \neg = \lambda b.\ b \Rightarrow F$
$\vdash \quad \wedge = \lambda b_1 b_2.\forall b.\ (b_1 \Rightarrow (b_2 \Rightarrow b)) \Rightarrow b$
$\vdash \quad \vee = \lambda b_1 b_2.\forall b.\ (b_1 \Rightarrow b) \Rightarrow ((b_2 \Rightarrow b) \Rightarrow b)$
$\vdash \quad \mathsf{One\_One} = \lambda f_{\alpha \to \beta}.\forall x_1 x_2.\ (f x_1 = f x_2) \Rightarrow (x_1 = x_2)$
$\vdash \quad \mathsf{Onto} = \lambda f_{\alpha \to \beta}.\forall y.\exists x.\ y = fx$
$\vdash \quad \mathsf{Type\_Definition} = \lambda P_{\alpha \to bool}\ rep_{\beta \to \alpha}.$
$\quad\quad \mathsf{One\_One}\ rep \wedge (\forall x.Px = (\exists y.\ x = rep\ y))$

Table 4.2: Definition of Basic Logical Constants

axioms, definitions, and proved theorems.[3] The same rules of inference are used for each theory. Some of the inference rules do not have hypotheses, but HOL distinguishes these from "axioms", which are specific to each theory.

Axioms are strongly discouraged because it is too easy to introduce inconsistency into the logic. Instead, there are definitional principles for constants and types that ensure that the resulting theory remains consistent. Many of HOL's logical constants have been given via definitions (see Table 4.2). The basic HOL system (including all the theories loaded when a user starts up the system) uses exactly five axioms; these are shown in Table 4.3. These axioms state facts about the set-theoretic model that underlies HOL, so they do not introduce inconsistency. The axioms could have been given as inference rules without hypotheses, as `REFL` is, but since they are most easily stated in terms of the constants defined in Table 4.2, they are presented as axioms in the same theory in which these constants are defined. There are many layers of theories built up on top of this theory, and each includes definitions of types and/or constants and theorems about these new objects. The theory that is loaded when the user starts up the system is already quite sophisticated, including list and tree types and a substantial number of definitions and theorems about arithmetic on natural numbers.

The basic definitional principles are primitive. They can be extended through programming, but these extensions must, after rearranging terms and proving various theorems,

[3] The mathematical notion of theory includes *all* theorems that are provable from the axioms and definitions. Since these theorems are usually infinite in number, and because HOL has no way of automatically proving these theorems, HOL's notion of theory includes only the theorems that have actually been proved.

BOOL_CASES_AX	$\vdash \forall b.\ (b = \mathsf{T}) \vee (b = \mathsf{F})$
IMP_ANTISYM_AX	$\vdash \forall b_1 b_2.\ (b_1 \Rightarrow b_2) \Rightarrow (b_2 \Rightarrow b_1) \Rightarrow (b_1 = b_2)$
ETA_AX	$\vdash \forall f_{\alpha \rightarrow \beta}.\ (\lambda x. f x) = f$
SELECT_AX	$\vdash \forall P_{\alpha \rightarrow bool} x.\ Px \Rightarrow P(\varepsilon P)$
INFINITY_AX	$\vdash \exists f_{ind \rightarrow ind}.\ \mathsf{One\_One}\ f \wedge \neg(\mathsf{Onto}\ f)$

Table 4.3: Basic Axioms

eventually invoke the basic principles to accomplish the definition.

The type definition functions `new_type_definition` and `define_type` are examples of a basic definitional principle and an extension of it. Both are SML functions that define new HOL types and return theorems about them. The basic principle, `new_type_definition`, allows a type to be created from a non-empty subset of another type (the representation type). The subset is specified as a predicate over the representation type: the element is in the subset if and only if predicate holds of the element. In order to guarantee that the subset is non-empty, the user provides a theorem stating that there exists an element of the representation type satisfying the predicate. The result of invoking `new_type_definition` is that the new type is defined and a theorem is proved and returned. This theorem states that there exists a bijective function (called the representation function) between the new type and the subset of the representation type that satisfies the predicate.

The extension, `define_type`, allows simple types (limited forms of recursive types, not mutually recursive types) to be specified in a manner that looks somewhat like the SML type definition. An example is the easiest way to describe it. Below is the SML definition of a datatype `bar`:

```
datatype 'a bar = BAR1 of 'a |
     BAR2 of bool | BAR3 of ('a bar * 'a bar);
```

This is the definition (via the SML function `define_type`) of the HOL type that corresponds to it:

```
define_type {name = "bar_Axiom",
             fixities = [Prefix, Prefix, Prefix],
             type_spec = `bar = BAR1 of 'a |
                BAR2 of bool | BAR3 of bar=>bar`}
```

It is worth clarifying the difference between the two type definitions above. The first is an SML declaration that declares a new SML type `bar`. This new SML type can now be used in SML, as in

```
let val b = BAR1 false in BAR3 (b, b) end
```

The second is the application of an SML function in order to create a new HOL type `bar`. This new HOL type can only be used in HOL, as for example in the HOL term:

$$\forall\,(b : \alpha\ \texttt{bar}).\,(\exists\,(x : \alpha).\,b = \texttt{BAR1}\ x\,)$$

The way in which `define_type` reduces to `new_type_definition` is quite complicated. It is explained fully in [Melham88].

Chapter 5

Encoding HOL-SML in HOL

Encoding HOL-SML into HOL required defining several classes of mutually recursive types and mutually recursive relations. To define the typing system, we first needed to encode the syntax and the semantic objects, the latter of which are the type environments and the meanings to which phrases elaborate. Both the syntax and semantic objects are represented by a set of mutually recursive HOL types. Then typing relations (one for each phrase class in the syntax) are defined. These relations link the syntax and semantic objects, specifying the results of elaboration.

A similar process is needed for encoding evaluation. Since evaluation is defined over a reduced syntax (described on page 129), we defined a separate syntax for evaluation and a function that translates the full syntax to the reduced syntax. The semantic objects were also defined, and then the evaluation relations (one for each phrase class in the reduced syntax) were defined. The differences between the evaluation semantic objects for HOL-SML and those of Core SML are given in Section A.3, along with the entire set of evaluation rules for HOL-SML.

This chapter describes the encoding of the static semantics. The encoding of the dynamic semantics is similar.

5.1 Encoding the Syntax

Section A.1 shows the complete syntax of HOL-SML. Section 2.9 of the Definition lists some syntactic restriction for the language. We need these same restrictions for HOL-SML, but, as in the Definition, we do not take account of these restrictions in the grammar

itself. Instead, they are incorporated into a collection of predicates (`proper_`*phrase*, for each *phrase* in the grammar) that ensure that the phrases satisfy the restrictions. These predicates will be described in more detail in Chapter 8.

We encountered some technical difficulties in encoding the syntax: the type definition facilities of HOL (described briefly in Section 4.1 of this thesis and in detail in Chapter 20 of [GM93]) allow the definition of recursive types but not *mutually* recursive types, while the syntax of HOL-SML is a definition of sixteen mutually recursive phrase classes. One could make mutually recursive types by brute force, but the brute force method did not work for a definition as large as the syntax of SML. Thus before we could proceed with the encoding, we (Elsa Gunter and the author) defined a package for the definition of mutually recursive data types. A feature added later (by Elsa Gunter and Healf Goguen) was the ability to allow the recursive types being defined to be nested within other type constructors. In addition to defining the types, the package proves useful theorems about the resulting types (for example, an induction theorem and theorems that assert the distinctness and one-to-one-ness of the constructors for the types) and includes a mechanism for defining primitive recursive functions over the types. This package is now being distributed with HOL90.

Note the presence in the grammar (Section A.1) of certain "hooks" into the Modules system, such as long identifiers (which may be quantified by a series of structure identifiers) and `open` declarations. Although the work described here involves only the Core of SML, the hooks into the Modules have been preserved in HOL-SML. This enabled the encoding of the syntax and evaluation of the Modules to be defined later [MG93].

Another aspect of the syntax worth noting is that the phrase classes can be divided up into several groups. The syntactic type phrases (Ty and TyRow) are defined only in terms of each other. The pattern phrases (AtPat, Pat, and PatRow) are defined only in terms of each other and the type phrases. The expression phrases (Exp, AtExp, and ExpRow), however, depend on themselves and all other phrases. Because of this structure, we define these groups of phrase classes separately: first the type phrases, then the pattern phrases, and so on. At each step the phrases in the group being defined depend only on each other and the phrases in the groups previously defined. These separate definitions allow for modularity (for example, if a function needs to be defined over type phrases we need not worry about other phrases), and the separate definitions run faster than would one definition that defines all the phrases at once. The increase in speed of definition is due to the fact

that the intermediate theorems that allow the definition to be reduced to `define_type` (and thence to `new_type_definition`) are much smaller for smaller definitions.

The grammar shown in Section A.1 is the *concrete* syntax, that is, what would be typed into an implementation. We encoded a syntax that more resembles the *abstract* syntax that is the output of a parser. In the abstract syntax, constructors are wrapped around objects to create objects of another type. As an example, the abstract syntax for expressions is:

exp ::= `ATEXPexp` *atexp*
| `APPexp` *exp atexp*
| `TYPEDexp` *exp ty*
| `HANDLEexp` *exp match*
| `RAISEexp` *exp*
| `FNexp` *match*

Here, `ATEXPexp`, `APPexp`, etc, are the constructors. Note that each HOL-SML phrase class is an HOL type, and the constructors are HOL functions. For example, the HOL type of `APPexp` is `exp` → `atexp` → `exp`.

The encoding of the syntax for evaluation (the reduced syntax) is similar to that for the elaboration grammar. To distinguish elements of the reduced syntax, "_e" is tagged onto the end of the phrase and constructor names. For example, the abstract syntax for the reduced syntax for expressions is:

exp_e ::= `ATEXPexp_e` *atexp_e*
| `APPexp_e` *exp atexp_e*
| `HANDLEexp_e` *exp_e match_e*
| `RAISEexp_e` *exp_e*
| `FNexp_e` *match_e*

5.2 Encoding the Semantic Objects

HOL-SML's static semantic objects include basic objects, such as type variables and type constructor names (stamps that distinguish the types resulting from datatype declarations), and compound objects such as types, type schemes, and environments. The semantic objects form a mutual recursion. As with the grammar, the encoding of the semantic objects uses HOL type constructors to construct elements of one class of semantic objects from elements of other classes, and the semantic objects can be divided up into groups

allowing them to be defined a group at a time, with the later groups depending on the former groups.

The structure of this recursion is more complex than that of the grammar and presents a few additional difficulties. Several of the semantic objects are described as finite functions between others which are mutually recursive with them. For example, in the Definition's Figure 10, a record type (in HOL-SML) is described as a finite map from labels to (HOL-SML) types, while a type is described as a disjoint sum of record types with other types:

$$\begin{aligned} \varrho &\in \text{RecType} = \text{Lab} \xrightarrow{fin} \text{Type} \\ \tau &\in \text{Type} = \ldots \cup \text{RecType} \cup \ldots \end{aligned}$$

In order to encode this we created a theory of finite maps, *finmap*. The internal representation of the finite maps are lists of pairs, where the first item in the pair is a member of the domain of the finite map, and the second item is the value of the map at that item. In order to simplify the definition of certain finmap manipulation functions (such as the modification of one map by another), we keep the domain items sorted. We have a predicate, `proper_finmap`, that checks that the domain is sorted according to some function that operates as a less-than operator, and that this comparison function is in fact a total order. We could have created, through the use of `new_type_definition`, a new type of maps whose domains are sorted (say, *sortmap*), but due to the limitations of our mutually recursive type definition package, sortmap could not be used to define the semantic objects. Thus we end up including in some of the statements of the theorems we proved hypotheses about the properness of finite maps.

Another problem with defining the semantic objects arises from the fact that several of them are left under-specified in the definition. Both TyVar (type variables) and TyName (type constructor names) are only specified to be sets, with elements of TyName possessing an arity that indicates how many parameters the type constructor takes.[1] One could parameterize by these types, but we chose to use a concrete instance instead. We represent TyVars by an HOL constructor wrapped around a string and TyNames by an HOL constructor wrapped around two natural numbers, the first representing the stamp that distinguishes this (HOL-SML) type from others, and the second representing the arity.

[1] For example, the type constructor name associated with the type constructor `list` has arity one, which determines the type of the elements of the list.

5.3 Encoding the Typing Relations

After encoding the syntax and static semantic objects of HOL-SML into HOL, the next step was to define the typing relations. We created a package that, given HOL terms representing the rules, automatically defines as HOL constants the mutually recursive relations, and proves two useful theorems. The first theorem states that the relations satisfy the rules. The second is an induction theorem, allowing one to prove properties of the objects in the relations by proving one case for each rule.[2]

Since this is the core of the encoding of HOL-SML into HOL, and because it is the main use of the higher order features of HOL in this work, we will explain how the package goes about making the definitions, using the typing relations as the example.

For each phrase class *phrase* in the grammar, the relation associated with it will be called `tych_`*phrase* (the name is meant to be suggestive of typechecking). Thus, for example, `tych_exp` *C exp type* will encode the relation $C \vdash exp : type$. Each relation is an HOL function taking a collection of terms to `bool`. The arguments of the functions are the language phrase being elaborated and various semantic objects, including possible results (types and environments) and backgrounds (contexts and, for the elaboration of constructor bindings, a type). The relation holds if, against the background, the phrase elaborates to the result.

The typing rules of HOL-SML (shown in Section A.2) specify the conditions under which one can conclude that the typing relation holds of a phrase and some semantic objects: if the hypotheses (the terms above the horizontal line) are true, then the conclusion (the term below the line) is true. However, we do not want the elaboration relations to be just any set of relations that obeys the rules. Consider the relations that always return true. Then the rules are certainly satisfied, since the conclusion (which is an application of one of the relations) is true no matter what happens to the hypotheses. The problem with this collection of relations is that we need to be able to *invert* the rules: if the conclusion of the rule holds, then we need to be able to conclude that the hypotheses of the rule also hold.[3] What we need is that the relations are true *only* if they are can be proved by a chain of reasoning using the rules. In other words, we need the smallest (in the sense of

[2] This package is similar to one written by Tom Melham [CM92], but ours allows the definition of mutually recursive inductive relations.

[3] Actually, this inversion works only if the conclusions of the rules are distinct. If this is not the case, then we can conclude only that the hypotheses of one of the relevant rules hold.

having as few tuples in the relations as possible) relations that satisfy these rules.

The challenge, then, is to express this idea (that the elaboration relations are the smallest relations satisfying the rules) in HOL. The general outline is this: first we encode the rules as a predicate (call it the type-rule predicate) on relations: if this predicate is true of a collection of relations, then the relations satisfy the typing rules. We then define the elaboration relations as the smallest relations satisfying the type-rule predicate. Specifically, using higher order logic, the elaboration relations are defined as being the intersection of all relations satisfying the type-rule predicate. Then, after defining the elaboration relations, we prove that the resulting relations actually do satisfy the elaboration rules. The induction theorem, which we also prove, is equivalent to stating that the elaboration relations are the smallest such relations.

The type-rule predicate. Examining the rules for elaboration in Section A.2 carefully, one notes that there are several separate groups of relations given there, corresponding to the groups of phrase classes. Therefore, the elaboration relations can be defined in phases, in the same way that the phrase classes are defined in phases. The reasons for doing this are the same as those for breaking up the definitions of the phrases: modularity (if we wish to prove a property of the typing of a pattern, we will not have to consider a collection of irrelevant cases) and speed of execution.

Therefore, to define the elaboration relations, we define several separate type-rule predicates, each of which takes as arguments potential elaboration relations and returns true if the relations satisfy the elaboration rules for those phrases. To be clearer about the encoding of the rules into predicates, let us examine the most complicated predicate, `tych_exp_pred`, which is the predicate for elaboration relations for phrases AtExp, ExpRow, Exp, Match, Mrule, Dec, and ValBind. The predicate `tych_exp_pred` is defined so that it is true of a set of relations if they satisfy the rules; its general form can be read "`tych_exp_pred` holds of this collection of relations if the relations satisfy the first rule, *and* the relations satisfy the second rule, *and* ...". Thus each rule shows up in one conjunct (a conjunct is one of a collection of things that are "and"ed together) in the definition of `tych_exp_pred`.

In order to give the flavor of what `tych_exp_pred` looks like, we show how one of the rules, Rule 8, is reflected in its conjuncts. Rule 8 refers to the typing of expression rows,

$\forall$(*tych_atexp* : `context` $\rightarrow$ `atexp` $\rightarrow$ `type` $\rightarrow$ `bool`)
(*tych_exprow* : `context` $\rightarrow$ `exprow` $\rightarrow$ `rectype` $\rightarrow$ `bool`)
(*tych_exp* : `context` $\rightarrow$ `exp` $\rightarrow$ `type` $\rightarrow$ `bool`)
(*tych_match* : `context` $\rightarrow$ `match` $\rightarrow$ `type` $\rightarrow$ `bool`)
(*tych_mrule* : `context` $\rightarrow$ `mrule` $\rightarrow$ `type` $\rightarrow$ `bool`)
(*tych_dec* : `context` $\rightarrow$ `dec` $\rightarrow$ `env` $\rightarrow$ `tyname set` $\rightarrow$ `bool`)
(*tych_valbind* : `context` $\rightarrow$ `valbind` $\rightarrow$ `varenv` $\rightarrow$ `statusmap` $\rightarrow$ `bool`).
`tych_exp_pred` *tych_atexp tych_exprow tych_exp tych_match tych_mrule tych_dec tych_valbind* =
...
($\forall$ *C lab exp type*.
tych_exp C exp type $\Rightarrow$
tych_exprow C (`EXPROW` *lab exp* `NONE`)
(`insert_into_rectype empty_rectype` *lab type*)) $\wedge$

($\forall$ *C lab exp exprow type rec*.
tych_exp C exp type $\wedge$ *tych_exprow C exprow rec* $\Rightarrow$
tych_exprow C (`EXPROW` *lab exp* (`SOME` *exprow*))
(`add_rectype` (`insert_into_rectype empty_rectype` *lab type*) *rec*)) $\wedge$
...

Table 5.1: Part of `tych_exp_pred`

which evaluate to records. It is written in Section A.2 as

$$\frac{C \vdash exp : \tau \quad \langle C \vdash exprow : \varrho \rangle}{C \vdash lab \texttt{ = } exp \langle \texttt{, } exprow \rangle : \{lab \mapsto \tau\} \langle + \varrho \rangle}$$

When the optional parts of the rule have been expanded out, the result is the following two rules:

$$\frac{C \vdash exp : \tau}{C \vdash lab \texttt{ = } exp : \{lab \mapsto \tau\}} \tag{5.1}$$

$$\frac{C \vdash exp : \tau \quad C \vdash exprow : \varrho}{C \vdash lab \texttt{ = } exp\texttt{, } exprow : \{lab \mapsto \tau\} + \varrho} \tag{5.2}$$

Table 5.1 shows the general form of the definition of `tych_exp_pred` and the two conjuncts corresponding to Rule 8. The *tych_exprow* and *tych_exp* in this table are variables that have the same type and are named the same as the elaboration relations that will be defined as constants later: `tych_exp_pred` can be applied to any relations of the appropriate type, and it will be true of these relations only if they satisfy the rules encoded in its conjuncts. (For the rest of this paper, we will use typewriter font for constants and types, and italics for variables.)

Notation in rule	Encoding
lab `=` *exp*	`EXPROW` *lab exp* `NONE`
lab `=` *exp*`,` *exprow*	`EXPROW` *lab exp* (`SOME` *exprow*)
$C \vdash exprow : \varrho$	*tych_exprow C exprow rec*
$\{lab \mapsto \tau\}$	`insert_into_rectype empty_rectype` *lab type*
$r1 + r2$	`add_rectype` $r1\ r2$

Table 5.2: Encoding into HOL

Each rule is encoded as a conjunct in the following manner. The rule is encoded as an implication, where the antecedent (the term before the $\Rightarrow$) is the conjunction of terms representing the hypotheses of the rule, and the consequent (the term after the $\Rightarrow$) is the conclusion. All the variables (other than those for the elaboration relations) in the rule are universally quantified. The Greek variables are replaced with English ones, so τ becomes *type* and ϱ becomes *rec*. The mathematical notation is encoded into HOL functions and constructors: the concrete syntax is replaced with abstract syntax; variables representing elaboration relations take the place of the mathematical notation, and notation in the results of the elaborations are replaced with HOL functions (which we had to defined). Examples of each of these encodings is given in Table 5.2.

Defining the elaboration relations. Note that `tych_exp_pred` only specifies when the potential elaboration relations must return true, so functions satisfying `tych_exp_pred` may return true even when the rules do not justify it. Thus we must define the elaboration relations to be the smallest relations satisfying `tych_exp_pred`, that is, the intersection of all relations satisfying `tych_exp_pred`. We specify that a tuple is in the relation if and only if it is in every possible elaboration relation satisfying `tych_exp_pred`. For example, the term used to define `tych_exp` is:

$\forall$ *C exp type.*
`tych_exp` *C exp type* =
$\forall$ *poss_tych_atexp poss_tych_exprow poss_tych_exp*
poss_tych_match poss_tych_mrule poss_tych_dec
poss_tych_valbind.
`tych_exp_pred` *poss_tych_atexp poss_tych_exprow*
poss_tych_exp poss_tych_match poss_tych_mrule
poss_tych_dec poss_tych_valbind $\Rightarrow$
poss_tych_exp C exp type

After defining the elaboration relations, it remains to be shown that the resulting relations do indeed satisfy `tych_exp_pred`. That is, the elaboration relations satisfy the rules. We have written a tactic that does this automatically. This tactic works by first expanding out all the definitions, then by acting like a miniature Prolog interpreter. That is, it checks if the conclusion of the goal is among its hypotheses. If so, then it is done, and if not it does backchaining: it finds among the assumptions an implication whose conclusion matches the desired goal and replaces the goal with the antecedent of this implication and then breaks up the conjunction in the new goal. Only a few iterations need to be done because of the common structure of our rules.

Finally, we proved *induction theorems* for the elaboration relations. These theorems allow one to prove facts about the arguments of elaboration relations. There is an induction theorem for each phrase class. The induction theorem for expressions is below, and the one for patterns is shown in Equation 5.4.

$$\begin{array}{l} \forall\, \mathit{atexp\_prop}\ \mathit{exprow\_prop}\ \mathit{exp\_prop}\ \mathit{match\_prop} \\ \quad \mathit{mrule\_prop}\ \mathit{dec\_prop}\ \mathit{valbind\_prop}. \\ \quad \texttt{tych\_exp\_pred}\ \mathit{atexp\_prop}\ \mathit{exprow\_prop}\ \mathit{exp\_prop} \\ \quad \mathit{match\_prop}\ \mathit{mrule\_prop}\ \mathit{dec\_prop}\ \mathit{valbind\_prop} \Rightarrow \\ \quad (\forall C\ \mathit{exp}\ \mathit{type}.\, \texttt{tych\_exp}\ C\ \mathit{exp}\ \mathit{type} \Rightarrow \\ \quad\ \mathit{exp\_prop}\ C\ \mathit{exp}\ \mathit{type}) \end{array} \tag{5.3}$$

Here, each *phrase_prop* is a property for the phrase class *phrase*. It has the same type as `tych_`*phrase* and is thus is a property of the arguments of `tych_`*phrase*. This theorem says that in order to show some property *exp_prop* holds of an expression and semantic objects satisfying `tych_exp`, one must show that this property (along with corresponding properties for other phrase classes) satisfies the rules: the predicate `tych_exp_pred` applied to the properties states that the properties satisfy the rules. One may wonder: why do the other properties enter in if we only want to prove something about expressions? The answer is that the elaboration of expressions depends on the elaboration of other phrase classes, so if a property is to hold of the elaboration of expressions, then in general it must be the case that a similar property holds for the elaboration of the other phrase classes.

The induction theorem is almost a rewording of the definitions of the elaboration relations as the smallest relations satisfying the rules. If *exp_prop*, *atexp_prop*, etc., satisfy the rules and the arguments C, *exp* and *type* are in the relation `tych_exp`, then since `tych_exp` is the smallest relation satisfying the rules, then C, *exp* and *type* are in *exp_prop* as well.

Because this statement is so close to the definition of the elaboration relations, it is easy to prove.

As noted above, the induction theorems are proved automatically by the package we created for defining mutually recursive relations. The form shown for Equation 5.3 is unacceptable, since the predicate `tych_exp_pred` is defined by the package, not the user. Thus the constant `tych_exp_pred` is replaced by its definition (that is, the encodings of the rules for the phrases) before being returned to the user. The returned theorem has this form:

$$\begin{array}{l}\forall\, \mathit{atexp\_prop}\ \mathit{exprow\_prop}\ \mathit{exp\_prop}\ \mathit{match\_prop}\\ \quad \mathit{mrule\_prop}\ \mathit{dec\_prop}\ \mathit{valbind\_prop}.\\ \quad \langle\!\langle \mathit{atexp\_prop},\ \mathit{exprow\_prop},\ \mathit{exp\_prop},\ \mathit{match\_prop},\ \mathit{mrule\_prop},\\ \quad \mathit{dec\_prop},\ \mathit{and}\ \mathit{valbind\_prop}\ \mathit{satisfy}\ \mathit{the}\ \mathit{rules}\rangle\!\rangle \Rightarrow\\ \quad (\forall C\ \mathit{exp}\ \mathit{type}.\ \texttt{tych\_exp}\ C\ \mathit{exp}\ \mathit{type} \Rightarrow\\ \quad \mathit{exp\_prop}\ C\ \mathit{exp}\ \mathit{type})\end{array}$$

where $\langle\!\langle \mathit{atexp\_prop},\ \mathit{exprow\_prop},\ \mathit{exp\_prop},\ \mathit{match\_prop},\ \mathit{mrule\_prop},\ \mathit{dec\_prop},\ \mathit{and}\ \mathit{valbind\_prop}\ \mathit{satisfy}\ \mathit{the}\ \mathit{rules}\rangle\!\rangle$ stands for a big conjunction of terms, one for each rule, with each conjunct stating that the properties satisfy that rule. For example, the conjunct corresponding to the rule in Equation 5.1 is:

$$\begin{array}{l}\forall\, C\ \mathit{lab}\ \mathit{exp}\ \mathit{type}.\\ \quad \mathit{exp\_prop}\ C\ \mathit{exp}\ \mathit{type} \Rightarrow\\ \quad \mathit{exprow\_prop}\ C\ (\texttt{EXPROW}\ \mathit{lab}\ \mathit{exp}\ \texttt{NONE})\\ \quad\ (\texttt{insert\_into\_rectype}\ \texttt{empty\_rectype}\ \mathit{lab}\ \mathit{type})\end{array}$$

Note that this is the same as the first conjunct in Table 5.1, except that *exp_prop* and *exprow_prop* take the place of *tych_exp* and *tych_exprow*.

The form of the induction theorem for patterns is:

$$\begin{array}{l}\forall \mathit{atpat\_prop}\ \mathit{patrow\_prop}\ \mathit{pat\_prop}.\\ \quad \langle\!\langle \mathit{atpat\_prop},\ \mathit{patrow\_prop},\ \mathit{and}\ \mathit{pat\_prop}\ \mathit{satisfy}\ \mathit{the}\ \mathit{rules}\rangle\!\rangle \Rightarrow\\ \quad (\forall C\ \mathit{pat}\ \mathit{VE}\ \mathit{SM}\ \mathit{type}.\ \texttt{tych\_pat}\ C\ \mathit{pat}\ \mathit{VE}\ \mathit{SM}\ \mathit{type} \Rightarrow\\ \quad \mathit{pat\_prop}\ C\ \mathit{pat}\ \mathit{VE}\ \mathit{SM}\ \mathit{type})\end{array} \qquad (5.4)$$

Note that this induction theorem involves properties only of the other pattern phrases. This is because the definition of pattern elaboration is mutually recursive only with elaborations

of pattern rows and atomic patterns. Thus if we want to prove a property of patterns alone, we do not have to deal with properties of other phrase classes.

Although the form of these theorems may look somewhat obscure, it will be shown in Chapters 6 and 8 that many useful theorems can be phrased in a form allowing proof via these induction theorems.

Chapter 6

Proving Properties of Evaluation

In this chapter we demonstrate how to prove properties about HOL-SML by describing several theorems we proved about evaluation. One proves properties about evaluation using the induction theorems for evaluation (which are similar to those for elaboration). The first difficulty in proving a property is putting it into a form by which it can be proved via the induction theorems. We start off with a theorem that is easy to prove, the pattern matching theorem, Theorem 2.1 from the Commentary [MT91]. This is:

Theorem 6.1 (Pattern Matching is Well-Behaved) *Let E, V, and pat be any environment, value, and pattern. Suppose that*

$$s, E, v \vdash pat \Downarrow r, s'$$

can be inferred for states s, s′ and result r. Then r is either a variable environment VE, or else the special result FAIL*—it cannot be an exception packet p. Moreover, s = s′.*

This theorem is a bit misleading, as the fact that the evaluation of patterns cannot result in packets is specified by the Definition. The general form of the HOL-SML relation involving pattern phrases[1] (see page 136) is:

$$s, E, v \vdash pat \Downarrow VE/\text{FAIL}, s'$$

This states clearly that the only allowable results are a variable environment and FAIL, so there is nothing to prove.

[1] The pattern evaluation relation in HOL-SML takes the same arguments as that for SML, but the Definition does not mention the state in the general form shown in the heading for the pattern rules. We find this omission confusing; thus we include the state in the general form here.

The assertion that $s = s'$ remains to be proved. When translated according to our encoding, the theorem we want to prove is

$$\forall\, s\ E\ v\ pat\ r\ s'.\ \texttt{eval\_pat}\ s\ E\ v\ pat\ r\ s' \Rightarrow (s = s') \tag{6.1}$$

The induction theorem for evaluation of patterns is:

$$\begin{array}{l}\forall\, atpat\_prop\ patrow\_prop\ pat\_prop.\\ \quad \langle\!\langle atpat\_prop,\ patrow\_prop,\ pat\_prop\ satisfy\ the\ rules \rangle\!\rangle \Rightarrow\\ \quad\ (\forall\, s\ E\ v\ pat\ r\ s'.\ \texttt{eval\_pat}\ s\ E\ v\ pat\ r\ s' \Rightarrow\\ \quad\ pat\_prop\ s\ E\ v\ pat\ r\ s')\end{array}$$

where $\langle\!\langle atpat\_prop,\ patrow\_prop,\ pat\_prop\ satisfy\ the\ rules \rangle\!\rangle$ stands for a big conjunction of terms, with each conjunct representing a pattern evaluation rule.

In order to prove our theorem (Equation 6.1), we need to make the conclusion of the induction theorem match the theorem we want to prove. Thus we see that we need to specialize the variable *pat_prop* to $\lambda\, s\ E\ v\ pat\ r\ s'.\ (s = s')$. We must also instantiate the properties *atpat_prop* and *patrow_prop*. It is clear that these latter properties should assert that the final state is the same as the initial state, so they are $\lambda\, s\ E\ v\ atpat\ r\ s'.\ (s = s')$ (for atomic patterns) and $\lambda\, s\ E\ rec\ patrow\ r\ s'.\ (s = s')$ (for pattern rows).

Thus, in order to prove our theorem, we must prove that these three properties satisfy the pattern evaluation rules. We will show how the properties satisfy one rule. The rule is one of the rules resulting from expanding out Rule 159 (shown on page 136). It is:

$$\frac{s, E, v \vdash pat \Downarrow VE, s'}{s, E, v \vdash var\ \texttt{as}\ pat \Downarrow \{var \mapsto v\} + VE, s'}$$

This is encoded as:

$$\begin{array}{l}\forall\, s\ E\ v\ var\ pat\ VE\ s'.\\ \quad pat\_prop\ s\ E\ v\ pat\ (\texttt{VARENVvef}\ VE)\ s' \Rightarrow\\ \quad pat\_prop\ s\ E\ v\ (\texttt{LAYEREDpat\_e}\ var\ pat)\\ \qquad (\texttt{VARENVvef}\\ \qquad\ (\texttt{add\_varenv\_e}\\ \qquad\ (\texttt{insert\_into\_varenv\_e}\ \texttt{empty\_varenv\_e}\ var\ v)\ VE))\\ \qquad s'\end{array} \tag{6.2}$$

Note that *pat_prop* is used here instead of `eval_pat`, since the conjunct states that the properties satisfy the rules, not that the evaluation relations satisfy the rules. The HOL

constructor `VARENVvef` indicates that the result is a variable environment rather than FAIL.

When *pat_prop* is replaced by $\lambda\, s\ E\ v\ pat\ r\ s'.(s = s')$, we are left with the following to prove:

$$\forall\, s\ E\ v\ var\ pat\ VE\ s'.(s = s') \Rightarrow (s = s')$$

This is easily proved. After proving that the properties satisfy the rules, yielding a theorem T, an application of the rule `MP` (modus ponens) to T and the induction theorem proves the desired theorem.

Note that because the pattern evaluation relations were defined separately from the relations for all other phrase classes, this proof, which only involved properties of patterns, did not need to consider cases for other phrase classes.

The next example is Theorem 2.2 in the Commentary.[2] This is:

Theorem 6.2 *For any phrase, let the sentence*

$$s, A \vdash phrase \Downarrow A', s'$$

be inferred, where $s = (mem,\ ens,\ cns)$ *and* $s' = (mem', ens', cns')$ *and* A, A' *are semantic objects. Then*

$$Dom(mem) \subseteq Dom(mem')\ and\ ens \subseteq ens'\ and\ cns \subseteq cns'$$

This is a more complicated theorem. The first complication is that it refers to all phrase classes in the reduced syntax rather than just some of them. Furthermore, in order for the property to hold for one group of phrases, it must hold for all groups of phrases upon which the group depends. For example, the group including the expressions (AtExp, ExpRow, Exp, Match, Mrule, Dec, and ValBind) depends on the pattern phrase group: Rule 135 (shown on page 134) for value bindings refers to the execution of patterns. In order to prove the property for expressions we must first prove it for patterns.

Thus in order to prove this theorem we have to define properties for all the phrase classes and then prove the theorem in separate steps, one for each group of mutually recursive phrases. The exact order in which we prove the theorems does not matter, as

[2] As we have added a semantic object for constructor names to HOL-SML, we include ConNameSet in the state in our presentation of the theorem.

long as the theorems for more basic phrase groups are proved before theorems for phrase groups that depend on them. For example, the pattern phrase group and exception binding phrases do not depend on each other, so either one can be proved first.

The second complication is that it takes a little bit of thought to see how to encode this theorem into HOL. The problem is how to encode $\mathrm{Dom}(mem) \subseteq \mathrm{Dom}(mem')$. HOL has a `set` theory, and this provides the predicate `SUBSET` for testing set containment. Then the Dom function must be encoded in HOL. This is easily done, resulting in an HOL function `dom_set` that takes as an argument a finite map and returns the set of its domain elements. Then the statement of the theorem for expressions is:

$$\begin{array}{l}\forall\, s_1\ E\ exp\ vp\ s_2\,.\ \texttt{eval\_exp}\ s_1\ E\ exp\ vp\ s_2 \Rightarrow \\ \quad \texttt{dom\_set}\,(\texttt{mem\_of\_state}\ s_1)\ \texttt{SUBSET}\ \texttt{dom\_set}\,(\texttt{mem\_of\_state}\ s_2) \wedge \\ \quad \texttt{exnames\_of\_state}\ s_1\ \texttt{SUBSET}\ \texttt{exnames\_of\_state}\ s_2 \wedge \\ \quad \texttt{connames\_of\_state}\ s_1\ \texttt{SUBSET}\ \texttt{connames\_of\_state}\ s_2\end{array}$$

Here `mem_of_state`, `exnames_of_state` and `connames_of_state` are HOL functions that return the memory, exception name set, and constructor name set components of a state. Comparing with the induction theorem for expressions:

$$\begin{array}{l}\forall\, atexp\_prop\ exprow\_prop\ exp\_prop\ match\_prop \\ \quad mrule\_prop\ dec\_prop\ valbind\_prop. \\ \quad \langle\!\langle atexp\_prop,\ exprow\_prop,\ exp\_prop,\ match\_prop,\ mrule\_prop, \\ \quad dec\_prop,\ and\ valbind\_prop\ satisfy\ the\ rules \rangle\!\rangle \Rightarrow \\ \quad (\forall\, s_1\ E\ exp\ vp\ s_2\,. \\ \quad\ \texttt{eval\_exp}\ s_1\ E\ exp\ vp\ s_2 \Rightarrow exp\_prop\ s_1\ E\ exp\ vp\ s_2)\end{array} \tag{6.3}$$

we see that the property for expressions must be

$$\begin{array}{l}\lambda\, s_1\ E\ exp\ vp\ s_2\,. \\ \quad \texttt{dom\_set}\,(\texttt{mem\_of\_state}\ s_1)\ \texttt{SUBSET}\ \texttt{dom\_set}\,(\texttt{mem\_of\_state}\ s_2) \wedge \\ \quad \texttt{exnames\_of\_state}\ s_1\ \texttt{SUBSET}\ \texttt{exnames\_of\_state}\ s_2 \wedge \\ \quad \texttt{connames\_of\_state}\ s_1\ \texttt{SUBSET}\ \texttt{connames\_of\_state}\ s_2\end{array}$$

The properties for the other phrase classes are defined similarly. We will not go describe how one would go about proving this theorem, since the proof of determinism, which is a long and complicated proof, will be explored carefully.

6.1 Inversion Theorems

Inversion theorems are standard theorems to prove about inductively-defined relations. In Tom Melham's inductive relations definition package, they are referred to as case analysis theorems and are proved automatically (that is, he has written functions that, given the induction theorem for the relation and a theorem stating that the rules are satisfied, proves and returns the case theorem). Our (mutually recursive) relations definition package also automatically proves these theorems. Since they are important theorems, we will describe the theorems and how to phrase them in terms of the induction theorems.

Inversion theorems can be summarized this way. Say relation R holds of some arguments $v_1\ v_2 \ldots v_n$. Then the inversion theorem for this relation gives the possible forms that the arguments $v_1\ v_2 \ldots v_n$ can have, and for each combination of permissible forms, indicates what properties they must have in order for it to be the case that R holds of those arguments. Both the possible forms of arguments for the relation and the properties for them are derived directly from the rules. To explain this, we will show the form for the inversion theorem for atomic expression and show how Rule 108 fits into this form. Rule 108 is:

$$\frac{E \vdash dec \Downarrow E' \quad E + E' \vdash exp \Downarrow v}{E \vdash \texttt{let}\ dec\ \texttt{in}\ exp\ \texttt{end} \Downarrow v}$$

When the state conventions and exception conventions have been applied to it, the result is the following three rules:

$$\frac{s, E \vdash dec \Downarrow E', s' \quad s', E + E' \vdash exp \Downarrow v, s''}{s, E \vdash \texttt{let}\ dec\ \texttt{in}\ exp\ \texttt{end} \Downarrow v, s''}$$

$$\frac{s, E \vdash dec \Downarrow p, s'}{s, E \vdash \texttt{let}\ dec\ \texttt{in}\ exp\ \texttt{end} \Downarrow p, s'}$$

$$\frac{s, E \vdash dec \Downarrow E', s' \quad s', E + E' \vdash exp \Downarrow p, s''}{s, E \vdash \texttt{let}\ dec\ \texttt{in}\ exp\ \texttt{end} \Downarrow p, s''}$$

The inversion theorem for atomic expressions has the form

$$\forall\, s_1\ E\ atexp\ vp\ s_2.$$
$$\texttt{eval\_atexp}\ s_1\ E\ atexp\ vp\ s_2 \Rightarrow \langle\langle big\ disjunction \rangle\rangle$$

where $\langle\langle big\ disjunction \rangle\rangle$ contains one disjunct[3] for each rule for atomic expressions. These disjuncts state what form the arguments of `eval_atexp` can have, and the hypothesis of the

[3] A disjunction is a collection of terms that are "or"ed together. A disjunct is one of these terms.

rule that allows one to conclude that `eval_atexp` holds of those arguments. The disjuncts corresponding to the three rules for 108 are:

$$
\begin{array}{l}
(\exists\ dec\ v\ E'\ s'.\\
\quad (atexp = \texttt{LETatexp\_e}\ dec\ exp \wedge\\
\quad (vp = \texttt{VALvp}\ v) \wedge\\
\quad \texttt{eval\_dec}\ s_1\ E\ dec\ (\texttt{ENVep}\ E')\ s' \wedge\\
\quad \texttt{eval\_exp}\ s'\ (\texttt{add\_env\_e}\ E\ E')\ exp\ (\texttt{VALvp}\ v)\ s_2)) \vee
\end{array}
$$

$$
\begin{array}{l}
(\exists\ dec\ exp\ p.\\
\quad (atexp = \texttt{LETatexp\_e}\ dec\ exp) \wedge\\
\quad (vp = \texttt{PACKvp}\ p) \wedge\\
\quad \texttt{eval\_dec}\ s_1\ E\ dec\ (\texttt{PACKep}\ p)\ s_2) \vee
\end{array}
$$

$$
\begin{array}{l}
(\exists\ dec\ exp\ p\ E'\ s'.\\
\quad (atexp = \texttt{LETatexp\_e}\ dec\ exp) \wedge\\
\quad (vp = \texttt{PACKvp}\ p) \wedge\\
\quad \texttt{eval\_dec}\ s_1\ E\ dec\ (\texttt{ENVep}\ E')\ s' \wedge\\
\quad \texttt{eval\_exp}\ s'\ (\texttt{add\_env\_e}\ E\ E')\ exp\ (\texttt{PACKvp}\ p)\ s_2)
\end{array}
$$

As before, the way we prove properties is via the induction theorems. The induction theorem for atomic expressions is:

$$
\begin{array}{l}
\forall\ atexp\_prop\ exprow\_prop\ exp\_prop\ match\_prop\\
\quad mrule\_prop\ dec\_prop\ valbind\_prop.\\
\quad \langle\!\langle atexp\_prop,\ exprow\_prop,\ exp\_prop,\ match\_prop,\ mrule\_prop,\\
\quad dec\_prop,\ and\ valbind\_prop\ satisfy\ the\ rules \rangle\!\rangle \Rightarrow\\
\quad (\forall\ s_1\ E\ atexp\ vp\ s_2.\\
\quad\ \texttt{eval\_atexp}\ s_1\ E\ atexp\ vp\ s_2 \Rightarrow atexp\_prop\ s_1\ E\ atexp\ vp\ s_2)
\end{array}
$$

Comparing this theorem against the statement of the match theorem, we see that the property associated with *atexp_prop* must be:

$$\lambda\ s_1\ E\ atexp\ vp\ s_2.\ \langle\!\langle big\ disjunction \rangle\!\rangle$$

In order to prove the inversion theorem for atomic expressions we must define similar properties for the other phrase classes (Exp, ExpRow, Match, Mrule, Dec, and ValBind)

and prove that these properties satisfy the rules. All of this is done automatically by our mutually recursive relations definition package.

Note that similar theorems can be proved for all groups of phrases, for example, the pattern phrases. However, since the theorems only state that the rules that define the relations can be inverted, the properties for the different phrase groups are independent. This is in contrast to, for example, Theorem 6.2, where the theorem for the group including expressions depended on the theorem for the pattern phrase group.

6.2 Determinism

The determinacy theorem is Theorem 2.4 in the Commentary.

Theorem 6.3 (Determinacy) *Let the two sentences*

$$s, A \vdash \mathit{phrase} \Downarrow A', s' \quad s, A \vdash \mathit{phrase} \Downarrow A'', s''$$

both be inferred. Then (A'', s'') *only differs from* (A', s') *by a one-to-one change of addresses and exception names which do not occur in* (s, A).

As mentioned in Section A.3 (page 131), the choice of next address and exception name in HOL-SML is deterministic. Thus our final states and results are *the same* instead of differing by exact choice of addresses and exception names. Therefore, our determinacy theorem is stated as follows (this is the version for expressions):

$$\begin{array}{l} \forall\, s_1\ E\ \mathit{exp}\ \mathit{vp}\, s_2\ \mathit{vp}'\, s_2'. \\ \quad \texttt{eval\_exp}\, s_1\ E\ \mathit{exp}\ \mathit{vp}\, s_2\ \wedge\ \texttt{eval\_exp}\ s_1\ E\ \mathit{exp}\ \mathit{vp}'\, s_2' \Rightarrow \\ \quad (s_2 = s_2')\ \wedge\ (\mathit{vp} = \mathit{vp}') \end{array} \tag{6.4}$$

This does not quite fit the form of the conclusion of the induction theorem for expressions (Equation 6.3). However, an equivalent statement of the theorem:

$$\begin{array}{l} \forall\, s_1\ E\ \mathit{exp}\ \mathit{vp}\, s_2. \\ \quad \texttt{eval\_exp}\, s_1\ E\ \mathit{exp}\ \mathit{vp}\, s_2 \Rightarrow \\ \quad \forall\, s_2'\ \mathit{vp}'.\, \texttt{eval\_exp}\ s_1\ E\ \mathit{exp}\ \mathit{vp}'\, s_2' \Rightarrow (s_2 = s_2')\ \wedge\ (\mathit{vp} = \mathit{vp}') \end{array} \tag{6.5}$$

does fit the induction theorem. Thus we see that *exp_prop* must be

$$\begin{array}{l} \lambda\, s_1\ E\ \mathit{exp}\ \mathit{vp}\, s_2.\, \forall\, s_2'\ \mathit{vp}'. \\ \quad \texttt{eval\_exp}\, s_1\ E\ \mathit{exp}\ \mathit{vp}'\, s_2' \Rightarrow (s_2 = s_2') \wedge (\mathit{vp} = \mathit{vp}') \end{array}$$

The properties for the other phrase classes are similar.

As with Theorem 6.2, the dependencies among the phrase groups give rise to dependencies among the determinacy theorems for the phrase groups, and thus must be proved in steps, with the theorems for more basic phrase groups (such as patterns and exception bindings) being proved before the theorems for the phrase groups that depend on them, such as the group including the expressions.

We will trace through the process of proving determinacy for group including the expressions. We will show how to prove the case for one of the rules resulting from Rule 108. The rule is

$$\frac{s, E \vdash dec \Downarrow E', s' \quad s', E + E' \vdash exp \Downarrow v, s''}{s, E \vdash \texttt{let}\ dec\ \texttt{in}\ exp\ \texttt{end} \Downarrow v, s''}$$

The encoding of this rule, as a condition to be satisfied by properties, is:

$$\begin{array}{l}
\forall\, s_1\ E\ dec\ exp\ v\, s_2. \\
\quad (\exists\, E'\, s'. \\
\qquad dec_prop\, s_1\ E\ dec\, (\texttt{ENVep}\ E')\, s' \wedge \\
\qquad exp_prop\, s'\, (\texttt{add_env_e}\ E\ E')\ exp\, (\texttt{VALvp}\ v)\, s_2) \Rightarrow \\
\quad atexp_prop\, s_1\ E\, (\texttt{LETatexp_e}\ dec\ exp)\, (\texttt{VALvp}\ v)\, s_2
\end{array}$$

When we instantiate *dec_prop*, *exp_prop*, and *atexp_prop*, we find that we must prove the following (this is shown in a manner mimicking what one would see while doing goal-directed theorem proving in HOL, that is, the goal is shown above a horizontal line, and the assumptions—statements one can use in doing the proof—are below the line):

$$\begin{array}{l}
\forall\, s_1\ E\ dec\ exp\ v\ s_2. \\
\quad (\exists\, E'\, s'. \\
\qquad (\forall\, ep'\, s_2'. \\
\qquad\quad \texttt{eval_dec}\ s_1\ E\ dec\ ep'\ s_2' \Rightarrow (s' = s_2')\ \wedge\ (\texttt{ENVep}\ E' = ep')) \wedge \\
\qquad (\forall\, vp'\, s_2'. \\
\qquad\quad \texttt{eval_exp}\ s'\, (\texttt{add_env_e}\ E\ E')\ exp\ vp'\ s_2' \Rightarrow \\
\qquad\quad (s_2 = s_2')\ \wedge\ (\texttt{VALvp}\ v = vp'))) \Rightarrow \\
\quad (\forall\, vp'\, s_2'. \\
\qquad \texttt{eval_atexp}\ s_1\ E\, (\texttt{LETatexp_e}\ dec\ exp)\ vp'\ s_2' \Rightarrow \\
\qquad (s_2 = s_2')\ \wedge\ (\texttt{VALvp}\ v = vp'))
\end{array}$$

After a bit of elementary processing (basically rearranging the terms and choosing instances for the existentially quantified variables in the antecedents of the goal), we get:

$$\frac{(s_2 = s_2') \wedge (\texttt{VALvp}\ v = vp')}{\begin{array}{l}\forall\, ep'\, s_2'.\\ \quad \texttt{eval\_dec}\ s_1\ E\ dec\ ep'\ s_2' \Rightarrow (s' = s_2') \wedge (\texttt{ENVep}\ E' = ep')\\ \forall\, vp'\, s_2'.\\ \quad \texttt{eval\_exp}\ s'\ (\texttt{add\_env\_e}\ E\ E')\ exp\ vp'\ s_2' \Rightarrow\\ \quad (s_2 = s_2') \wedge (\texttt{VALvp}\ v = vp'))\\ \texttt{eval\_atexp}\ s_1\ E\ (\texttt{LETatexp\_e}\ dec\ exp)\ vp'\ s_2'\end{array}}$$

This is where the inversion theorems come in. We have

$$\texttt{eval\_atexp}\ s_1\ E\ (\texttt{LETatexp\_e}\ dec\ exp)\ vp'\ s_2'$$

among the assumptions, and this should tell us something about the evaluations of the *dec* and *exp* within the let expression. The inversion theorem is the agent that allows us to draw these conclusions. Using forward inference, we instantiate the inversion theorem for atomic expressions to the arguments of `eval_atexp` above, and then using the fact that different forms of atomic expressions are distinct, we conclude that

$$\begin{array}{l}(\exists\, dec'\ exp'\ v\ E'\ s'.\\ \quad (dec = dec') \wedge (exp = exp') \wedge (vp' = \texttt{VALvp}\ v) \wedge\\ \quad \texttt{eval\_dec}\ s_1\ E\ dec'\ (\texttt{ENVep}\ E')\ s' \wedge\\ \quad \texttt{eval\_exp}\ s'\ (\texttt{add\_env\_e}\ E\ E')\ exp'\ (\texttt{VALvp}\ v)\ s_2') \vee\\ (\exists\, dec'\ exp'\ p.\\ \quad (dec = dec') \wedge (exp = exp') \wedge (vp' = \texttt{PACKvp}\ p) \wedge\\ \quad \texttt{eval\_dec}\ s_1\ E\ \ dec'\ (\texttt{PACKep}\ p)\ s_2) \vee\\ (\exists\, dec'\ exp'\ p\ E'\ s'.\\ \quad (dec = dec') \wedge (exp = exp') \wedge (vp' = \texttt{PACKvp}\ p) \wedge\\ \quad \texttt{eval\_dec}\ s_1\ E\ dec'\ (\texttt{ENVep}\ E')\ s' \wedge\\ \quad \texttt{eval\_exp}\ s'\ (\texttt{add\_env\_e}\ E\ E')\ exp\ (\texttt{PACKvp}\ p)\ s_2)\end{array}$$

When we use `STRIP_ASSUME_TAC` with this information to add it to the assumptions, we get three subgoals. We will show how to solve two of them. The first one is:

$$\frac{(s_2 = s_2') \wedge (\texttt{VALvp}\, v = vp')}{\begin{array}{l}\forall\, ep'\, s_2'.\\ \quad \texttt{eval\_dec}\, s_1\, E\, dec\, ep'\, s_2' \Rightarrow (s' = s_2') \wedge (\texttt{ENVep}\, E' = ep')\\ \forall\, vp'\, s_2'.\\ \quad \texttt{eval\_exp}\, s'\, (\texttt{add\_env\_e}\, E\, E')\, exp\, vp'\, s_2' \Rightarrow\\ \quad (s_2 = s_2') \wedge (\texttt{VALvp}\, v = vp')\\ \texttt{eval\_atexp}\, s_1\, E\, (\texttt{LETatexp\_e}\, dec\, exp)\, vp'\, s_2'\\ dec = dec'\\ exp = exp'\\ vp' = \texttt{VALvp}\, v'\\ \texttt{eval\_dec}\, s_1\, E\, dec'\, (\texttt{ENVep}\, E'')\, s''\\ \texttt{eval\_exp}\, s''\, (\texttt{add\_env\_e}\, E\, E'')\, exp'\, (\texttt{VALvp}\, v')\, s_2'\end{array}}$$

To solve this, we take the assumption

$$\texttt{eval\_dec}\, s_1\, E\, dec'\, (\texttt{ENVep}\, E'')\, s''$$

and combine it with

$$dec = dec'$$

to get

$$\texttt{eval\_dec}\, s_1\, E\, dec\, (\texttt{ENVep}\, E'')\, s''$$

and match this against the assumption

$$\forall\, ep'\, 2'.\, \texttt{eval\_dec}\, s_1\, E\, dec\, ep'\, s_2' \Rightarrow (s' = s_2') \wedge (\texttt{ENVep}\, E' = ep')$$

to conclude that

$$(s' = s'') \wedge (\texttt{ENVep}\, E' = \texttt{ENVep} E'')$$

and from the second conjunct, we get

$$E' = E''$$

because the constructor `ENVep` is one-to-one. Then using this information, we conclude from the assumptions

$$\texttt{eval\_exp}\, s''\, (\texttt{add\_env\_e}\, E\, E'')\, exp'\, (\texttt{VALvp}\, v')\, s_2'$$

and

$$exp = exp'$$

that

$$\texttt{eval\_exp}\ s'\ (\texttt{add\_env\_e}\ E\ E')\ exp\ (\texttt{VALvp}\ v')\ s_2'$$

Then we match this against the assumption

$$\forall\ vp'\ s_2'.\ \texttt{eval\_exp}\ s'\ (\texttt{add\_env}\ E\ E')\ exp\ vp'\ s_2' \Rightarrow (s_2 = s_2')\ \wedge\ (\texttt{VALvp}\ v = vp')$$

getting the result that

$$(s_2 = s_2')\ \wedge\ (\texttt{VALvp}\ v = \texttt{VALvp}\ v')$$

Now we combine this information with the assumption

$$vp' = \texttt{VALvp}\ v'$$

and we have proved the goal.

The second goal we will slow how to solve is:

$$\frac{(s_2 = s_2')\ \wedge\ (\texttt{VALvp}\ v = vp')}{\begin{array}{l}\forall\ ep'\ s_2'.\\ \quad \texttt{eval\_dec}\ s_1\ E\ dec\ ep'\ s_2' \Rightarrow (s' = s_2')\ \wedge\ (\texttt{ENVep}\ E' = ep')\\ \forall\ vp'\ s_2'.\\ \quad \texttt{eval\_exp}\ s'\ (\texttt{add\_env\_e}\ E\ E')\ exp\ vp'\ s_2' \Rightarrow\\ \quad (s_2 = s_2')\ \wedge\ (\texttt{VALvp}\ v = vp')\\ \texttt{eval\_atexp}\ s_1\ E\ (\texttt{LETatexp\_e}\ dec\ exp)\ vp\ s_2''\\ dec = dec'\\ exp = exp'\\ vp' = \texttt{PACKvp}\ p\\ \texttt{eval\_dec}\ s_1\ E\ dec'\ (\texttt{PACKep}\ p)\ s_2'\end{array}}$$

Here, when we match

$$\texttt{eval\_dec}\ s_1\ E\ dec\ (\texttt{PACKep}\ p)\ s_2'$$

against the assumption

$$\forall\ s_2'\ ep'.\ \texttt{eval\_dec}\ s_1\ E\ dec\ ep'\ s_2' \Rightarrow (s' = s_2')\ \wedge\ (\texttt{ENVep}\ E' = ep')$$

we conclude that

$$(s' = s'_2) \wedge (\texttt{ENVep}\, E' = \texttt{PACKep}\, p)$$

which is a contradiction, since the constructors for `env_pack` (`ENVep` and `PACKep` are among these) are distinct. Thus when we rewrite this conclusion with the distinctness theorem, we get `F:` from the assumptions we have proved false. We use `CONTR_TAC` and the subgoal is proved.

Note that for the goals shown here, the determinacy of subphrases was given by assumptions (this is how the induction hypothesis shows up in this proof). For phrases that include subphrases from other groups (like patterns or exception bindings), the determinacy of subphrases is given by the theorems for determinacy for those phrase groups, rather than by induction hypotheses. This is why, for example, we must prove the determinacy for the pattern phrase group before proving it for the group including expressions.

The number of cases to be dealt with in proving this theorem is huge. In our example we started with a goal deriving from one rule, and got three subgoals from the use of the inversion theorem. This results in roughly 1600 subgoals in the proof! It was obvious that we needed some automation here. We strove to write one tactic that would solve as many of these goals as possible. The process by which the tactic solved the goals is essentially similar to that shown for the goals solved above. However there are many special cases that show up in the subgoals, a result of side conditions in the rules. Unfortunately, so many subgoals are generated in the process of proving this theorem that even the special cases must be included in the tactic in order for it to be able to treat most of the subgoals. Thus the resulting tactic is ugly and unreadable, but it works: after applying this tactic to the more than 1600 subgoals resulting from breaking up the rules according to the cases presented by the inversion theorems, there are only 5 subgoals remaining. These are "super-special cases" that need to be handled by separate tactics.

The process of finding one tactic that works on all but 5 of the 1600 or so subgoals was an iterative process: first, all of the goals were printed into a file, and we sampled these randomly and wrote a tactic that would solve these goals. Then we applied this tactic to all the subgoals and found that it had solved about one third of them. The remaining subgoals were again examined, and the tactic was refined to solve a sampling of these subgoals. After applying this, only about 60 subgoals remained. After a bit more tweaking, the resulting tactic yielded 5 remaining subgoals. We decided that generalizing the tactic enough to solve these remaining goals would complicate it far too much, and so

special tactics were written for these cases.

Doing the proof of determinacy allowed us to find many mistakes in our encoding. These mistakes would show up as subgoals that either were unsolvable or looked strange. When we traced them back, we invariably found that they were due to something like a missing prime on a variable in our encoding of the rules.

One aspect of using a theorem prover that became apparent to the author during the proof of this theorem was that theorem provers do not let a user get away with anything. Often, during the course of a proof on paper, people will look at a small, basic lemma and say "this is obvious" and not make a formal proof of it. However, a theorem prover demands that one actually work out a proof of the lemma. As an example, a lemma that we needed for the proof of the determinacy theorem is:

$$\forall\,(l : \alpha\ list)\,(l' : \alpha\ list)\,(f : \alpha \to \beta).$$
$$\texttt{ONE\_ONE}\ f \Rightarrow ((\texttt{MAP}\ f\ l = \texttt{MAP}\ f\ l') \iff (l = l'))$$

Chapter 7

Relating Static and Dynamic Semantics

When an SML program is evaluated, one of three things could possibly happen. The first possibility is that the evaluation does not terminate because it gets into an infinite loop. The second possibility is that it does not evaluate because there is no rule that applies to the phrase (we say that evaluation is stuck). For example, there is no rule for the application of `5` to an atomic expression. The third possibility is that the evaluation could succeed and produce some result, which is, for expressions, either a value or a packet. One would hope that SML has the property of type soundness: that is, the second possibility, that the evaluation gets stuck because no rule applies, does not occur.

Unfortunately, natural semantics (for form of semantics used in the Definition) is not well suited to proving this sort of theorem. The problem is that it is difficult to distinguish between not being able to proceed at all (there is no rule that applies to the phrase) and there being no end to the computation (the evaluation gets into an infinite loop): they are both reflected in there being no result A' or final state s' such that $s, A \vdash \mathit{phrase} \Downarrow A', s'$ is derivable. However, the reason for the lack of results is different. Say we want to evaluate the program `5 true`. This is an application of the expression `5` to the atomic expression `true`. There are several rules that have as their conclusion a sentence of the form $s, E \vdash \mathit{exp}\ \mathit{atexp} \Downarrow v/p, s'$ for some v/p and s'. Each of them has hypotheses that describe the evaluation of *exp* (say to v_1) and *atexp* (say to v_2), and there are various side conditions or further evaluations to represent the application of v_1 to v_2. There are no rules for when v_1 is a special value, namely the integer `5`. Thus our attempt to evaluate

this phrase has stopped dead.

On the other hand, say we want to evaluate

```
let val rec f = fn x => f x in f 3 end
```

The declaration[1] `val rec f = fn x => f x` will result in an environment where f is bound to the closure

$$(\texttt{x => f x}, E, \{f \mapsto (\texttt{x => f x}, E, \{\})\})$$

The first component is the match that forms the body of the function. The second component, E, is the environment in which the expression is evaluated. The third component is a variable environment that allows for the function to be applied recursively by giving a value for the occurrence of f in the body of the function.

In the evaluation of `f 3` in the body of the let, f will evaluate to the closure above, and `3` evaluates to itself. Two rules could apply here:

$$\frac{s, E \vdash exp \Downarrow (match, E', VE), s' \quad s', E \vdash atexp \Downarrow v, s'' \quad s'', E' + \text{Rec } VE, v \vdash match \Downarrow v', s'''}{s, E \vdash exp\ atexp \Downarrow v', s'''}$$

$$\frac{s, E \vdash exp \Downarrow (match, E', VE), s' \quad s', E \vdash atexp \Downarrow v, s'' \quad s'', E' + \text{Rec } VE, v \vdash match \Downarrow \text{FAIL}, s'''}{s, E \vdash exp\ atexp \Downarrow [\texttt{Match}], s'''}$$

The first is for normal function evaluation, that is, one of the clauses in *match* matches the value v, and the evaluation results in a value v'. The second rule is for when none of the clauses in *match* matches v.

It seems that we should use the first rule, since `3` matches the pattern in `x => f x`, which is just the variable x. However, in trying to evaluate the body of the function with x bound to `3` (as required by Rule 127, page 133), we find that we again need to evaluate `f 3`, which is the evaluation for which we are currently attempting to find the result. Thus an infinite loop is represented by an never-ending search for a way to complete a proof tree.

[1] We remind the reader that concrete SML syntax (that is, not using the constructors with which it is encoded into HOL) will be written in typewriter font. The program variables, when shown separately, will be written in italics.

7.1 Methods of Proving Type Soundness

Type soundness is usually proved as a corollary of a property called *strong type soundness*. This property states that if a program is well-typed and has type τ, then its evaluation either goes on forever or results in a value of type τ. Since a well-typed value cannot result from a type error (this must be proved), this implies type soundness.

Some ways of expressing the dynamic semantics of programs lend themselves more easily to proofs of strong type soundness. For example, the syntactic reduction approach used by Wright and Felleisen in [WF92] allows the theorem to be stated and proved in a straightforward way. In this presentation of dynamic semantics, evaluation proceeds a step at a time. Basic steps are written $phrase \longrightarrow phrase'$ (everything is syntax here; substitution[2] is used instead of environments, and an expression called a ρ-expression is used instead of a separate state).

Examples of the reduction steps (phrased in SML-like syntax) are

$$\texttt{let } x \texttt{ = } v \texttt{ in } exp \texttt{ end} \longrightarrow exp[v/x]$$

for reducing a let expression (v is a syntactic representation of a value and $exp[v/x]$ is exp with v substituted for x) and

$$b\ v \longrightarrow \mathrm{APPLY}(b, v)$$

for applying a basic value to another value. Evaluation contexts K are used to determine which reduction will be done next. The "reduction-in-context" rule is:

$$\frac{e \longrightarrow e'}{K[e] \longmapsto K[e']}$$

Evaluation contexts are given as follows:

$$K ::= [\,] \mid K\ e \mid v\ K \mid \texttt{let x = K in e}$$

for e an expression. This essentially says that if our expression is the application of an expression to another expression, then we must reduce the expression that is acting as a function first, until we get a value, and only then can we start applying reductions to the expression to which the function is being applied. In let expression, we first reduce the expression to which the variable is bound until it is a value. Then, after a let reduction, the body (with the value substituted for the variable) is reduced.

[2]Substitution is explained in Section 8.3.

The relation $\longmapsto\!\!\!\!\!\rightarrow$ is the reflexive transitive closure of $\longmapsto$.

This method can be contrasted with the SML rule for the application of a basic value:

$$\frac{s, E \vdash exp \Downarrow b, s' \quad s', E \vdash atexp \Downarrow v, s'' \quad \text{APPLY}(b, v) = v'}{s, E \vdash exp\ atexp \Downarrow v', s'''}$$

Here, the expression is first fully evaluated, then the atomic expression is fully evaluated, and then the apply is done. In syntactic reduction semantics, the evaluations of the expression and atomic expression is done step by step, until values are gotten, and then the basic step $b\ v \longrightarrow \text{APPLY}(b, v)$ is done. The natural semantics incorporates structure (which is given by evaluation contexts and the reduction-in-context rule in the Wright-Felleisen semantics) and operation (the reduction of $b\ v$ to $\text{APPLY}(b, v)$) into one rule.

A simple evaluation in the Wright-Felleisen style is given as

$$\texttt{let x = succ 3 in sq x} \longmapsto \texttt{let x = 4 in sq x} \longmapsto \texttt{sq 4} \longmapsto 16$$

where `succ` is the successor function and `sq` is a squaring function. The first step is justified by the following instance of the reduction-in-context rule:

$$\frac{\texttt{succ 3} \longrightarrow \texttt{4}}{\texttt{let x = succ 3 in sq x} \longmapsto \texttt{let x = 4 in sq x}}$$

and the other steps are justified by rules where the reduction context is the entire expression.

This method of dynamic semantics is more suited for proofs of strong type soundness because an infinite evaluation is distinguished from a stuck evaluation. An infinite evaluation is represented as an infinite chain of reductions. A stuck evaluation is represented by a chain of reductions that ends in an expression (that is not yet a value) for which no next step can be taken. Faulty expressions are those with basic type errors (the use of a basic function like + with the wrong type of argument, or the use of a non-function as if it were a function). Faulty expressions are a superset of the stuck expressions (they are a proper superset: some expressions evaluate to a value even if they are faulty, as the faulty part of the expression may not be evaluated).

Strong type soundness for this system is proved by first proving three other properties: *type preservation*, *uniform evaluation*, and *untypability of faulty expressions*. Untypability of faulty expressions states that faulty expressions cannot be given a type: there is no C and τ such that $C \vdash e : \tau$. Type preservation says that for any expression e, if e has type τ and $e \longmapsto\!\!\!\!\!\rightarrow e'$, then e' has type τ. Uniform evaluation says that for any expression e,

the evaluation of e either results in an infinite loop, gets stuck, or results in a value v. Strong type soundness is proved from these properties as follows. Say e has type τ, and its evaluation is not infinite. Then, by uniform evaluation, $e \longmapsto\!\!\!\!\!\rightarrow e'$ where e' is a stuck expression or a value. By untypability of faulty expressions, if e' were stuck, then it would be faulty, and thus it would not have a type. By type preservation, e' has type τ, which eliminates this possibility. Thus e' is a value v which by type preservation has type τ.

Another form of dynamic semantics that is well suited to the statement and proof of strong type soundness is that used in [Leroy93]. Here an abstract machine[3] is used to specify the evaluation of a language with references, polymorphism "by name", and continuations. The evaluation is done a step at a time, using two relations that are defined in terms of each other. One relation breaks up the term, placing structural information in the continuation and doing reductions until a value is reached. The other relation determines how to apply the continuation to a value. Leroy has augmented the usual semantics for programs in order to eliminate stuck evaluations: if none of the evaluation rules are matched, the evaluation results in the distinguished answer `wrong`. Like the Wright-Felleisen semantics, programs that get into infinite loops are expressed by an infinite sequence of reductions. This differentiates them from programs with type errors, which result in `wrong`. Since there are two relations used here, there are two parts to strong type soundness. First, if an expression e has type τ and the continuation k accepts values of type τ, then if we evaluate e with continuation k, the result is not `wrong`. Second, if a value v has type τ and the continuation k accepts values of type τ, then if we pass on v to k, the result is not `wrong`.

SML is specified using natural semantics, so we must take a different route. The standard approach for proving strong type soundness in natural semantics is shown in [ACPP89]. The idea is that, as in Leroy's semantics, evaluation will be defined for every program, whether or not it is type-correct. The object `wrong` is returned whenever there is a type error in the program, so we add `wrong`-generation rules such as:

$$\frac{s, E \vdash exp \Downarrow v, s' \quad v \text{ is not applicable}}{s, E \vdash exp\ atexp \Downarrow \texttt{wrong}, s'}$$

Here, "is not applicable" means that v is not one of the values that can be used as a

[3]Leroy refers to his semantics as being in a "structural operational style", but a close examination shows that this form of operational semantics is similar to the CEK abstract machine as in [FF86].

function. We also need `wrong`-propagation rules such as:

$$\frac{s, E \vdash exp \Downarrow v, s' \quad v \text{ is applicable} \quad s', E \vdash atexp \Downarrow \texttt{wrong}, s''}{s, E \vdash exp\ atexp \Downarrow \texttt{wrong}, s'''}$$

If the `wrong` rules are added in such a way as to ensure that there is a rule that applies to every syntactically correct program, then an attempt to evaluate an expression will either result in a infinite search for a proof tree (reflecting an evaluation that gets into an infinite loop), or the program will evaluate and the answer will either be `wrong` or a proper result such as a value or a packet. As with the Wright-Felleisen semantics, type preservation (if e has type τ and e evaluates to v, then v has type τ) is proved. Since `wrong` is not given a type, if e has type τ and results in an answer a, then a has type τ (and is thus not `wrong`).

Thus, once one has assured that there *is* a rule for every syntactic form, the difficulty in this theorem lies in proving type preservation. This is proved by induction on the length of the inference that $s, E \vdash exp \Downarrow v, s'$.

Let us consider what would be involved in taking this approach to proving strong type soundness for Core SML. Adding `wrong`-propagation rules (one for each hypothesis) and `wrong`-generation rules would dramatically increase the number of evaluation rules, at least doubling it. This would make any proof about the system large and difficult.

Another approach, given in [GR93], uses the idea of a *partial proof*. The idea here is that logic variables can represent an evaluation result that is as yet unknown. If a logic variable occurs as the result of an evaluation in the hypotheses of a rule, then other hypotheses may use this unknown result, but the results of these evaluations are required to be unknown also. If a proof tree has logic variables in it, then it can be considered a partial proof. This form of semantics has two types of sentences, $exp \downarrow \omega$ and $exp \Downarrow \omega$, where ω is either a value or a logic variable. States are not present here because there are no references in [GR93], and environments are not present because substitution is used instead. The latter sentence $exp \Downarrow \omega$ is roughly analogous to the $\longrightarrow$ operation in the Wright-Felleisen semantics (although in addition to a single reduction step, the evaluation of the reduction is completely carried out). The former $exp \downarrow \omega$ roughly corresponds to $\longmapsto\!\!\!\!\!\rightarrow$, that is, the full evaluation of a term, including many steps taken within contexts. Examples of evaluation rules are:

$$\frac{e \downarrow \omega \quad e' \downarrow \omega' \quad \omega\, \omega' \Downarrow \omega''}{e\, e' \downarrow \omega''} \quad \text{(application-search)}$$

$$\frac{e[v/x] \downarrow v'}{(\texttt{fn x => e})\ v \Downarrow v'} \quad \text{(application-reduce-1)}$$

where there would be quite a few application reduction rules depending on the form of the value being applied. In addition, we have

$$v \Downarrow v \quad \text{(value)}$$

far values v.

The advantage of this approach is that, again, we can distinguish between phrases to which no rule applies (resulting in stuck evaluations) and phrases that execute forever. Phrases to which no rule applies are represented by partial proofs in which no refinements (extensions of the proof towards the leaves in order to try to replace logic variables by values) can be made, and phrases that execute forever are represented by partial proofs in which refinements can always be made. An example of a stuck evaluation is the following (where X is a logic variable):

$$\dfrac{\dfrac{\dfrac{}{\texttt{5} \Downarrow \texttt{5}}}{\texttt{5} \downarrow \texttt{5}} \quad \dfrac{\dfrac{}{\texttt{true} \Downarrow \texttt{true}}}{\texttt{true} \downarrow \texttt{true}} \quad \texttt{5 true} \Downarrow X}{\texttt{5 true} \downarrow X}$$

Evaluation for this system can be described as a series of proofs, each of which is a refinement of the previous one. Evaluation ends when the sentence in the conclusion of the proof is $exp \downarrow v$ for some value v. Strong type soundness for this system is stated as follows. Say e has type τ and P is a partial proof with $e \downarrow \omega$ as its conclusion. Then if there is no P' that refines P, then P is a total proof (it contains no logic variables), ω is a value v, and v has type τ. A consequence of this theorem is that evaluation cannot get stuck for lack of a rule that applies.

We will not take this approach either. It would require formulating partial proofs in HOL and recasting the rules into the two forms of evaluation relation needed for this system. These seems to be no theoretical difficulty in formulating partial proofs in HOL; in fact Syme [Syme93] has formulated proof trees (but without logic variables) in his encoding of SML in HOL. However, recasting SML evaluation rules into the two forms of evaluation relation would be a substantial departure from the rules given in the Definition, and it would be more difficult to say that the encoded language is close to that given in the Definition.

Therefore instead of proving type soundness, we will aim toward proving type preservation. When type preservation is proved for a system without `wrong` rules, it does not ensure type soundness. Consider the following rules for a let expression:[4]

$$\frac{e_2\,[e_1/x] : \tau}{\texttt{let}\ x = e_1\ \texttt{in}\ e_2 : \tau} \quad \text{(let typing)}$$

$$\frac{e_1 \Downarrow v_1 \quad e_2[v_1/x] \Downarrow v_2}{\texttt{let}\ x = e_1\ \texttt{in}\ e_2 \Downarrow v_2} \quad \text{(let evaluation)}$$

If x does not occur in e_2, then a type error in e_1 will not get caught by the typechecker, but it will cause a type error in the evaluation of the let expression. Thus for this system, type preservation would not guarantee type soundness.

To be a bit more precise, we will state type preservation as follows: Say that $C \vdash exp : \tau$ and $s, E \vdash exp \Downarrow v/p, s'$, where v/p is a value or a packet. In addition, assume that E agrees with C, that is, that the values associated with variables in E can be given the types associated with them in C. Then if v/p is a value v, then v has type τ.

This statement of the property needs to be refined. For example, we need constraints on the types of the values in the range of the state: we need to ensure that for each address a in the domain of the memory, if E maps the variable v to the address a, then $s(a)$ has type $C(v)$. This property will be refined, and the notions "agrees with" and "has type" defined, in Chapter 8.

7.2 Why Prove Type Preservation?

Neither type soundness nor type preservation for all of Core SML has been proved. Type soundness has been proved for various languages with some of Core SML's key features. Wright and Felleisen [WF92] proved type soundness for a language with polymorphism, references, exceptions, and continuations, using a syntactic reduction semantics. Leroy [Leroy93] proved type soundness for a language with polymorphism by name, references, and continuations, using an abstract machine semantics. Abadi, Cardelli, Pierce, and Plotkin [ACPP89] proved type soundness for a functional language with type dynamic, using natural semantics and `wrong` rules. Tofte proved type preservation for a language with polymorphism and references, using a semantics almost exactly like the one in the Definition (this is not a surprise; he is one of the authors of the Definition).

[4] Thanks to Frank Pfenning for this example.

Since we will not prove type soundness and thus will not demonstrate that evaluation of type-correct programs does not result in a run-time type error, one wonders: why bother trying to prove type preservation? Furthermore, few people doubt that SML, at least SML as implemented, is actually type-sound: it has been in use long enough that one presumes that any errors would have been caught and fixed long ago. Thus one might ask: why prove type preservation when no one doubts that a stronger theorem (type soundness) in fact holds of the language? There are several answers to this question.

The first is that implementations are a different sort of object than a formal language definition. The fact that some version of SML has had all known bugs fixed is no more than a statement that it has been extensively tested. There may well be a program that no one has written yet that will demonstrate that type soundness does not hold. This does occur occasionally, especially when major changes have been made to the compiler. Furthermore, all implementations differ from the Definition. In fact, the rules in the Definition are not type-sound. (The most obvious error, which is explained by Kahrs in [Kahrs93], is the conflicting treatment of constructors in the evaluation and typechecking rules, specifically with respect to the `ref` constructor. This is explained briefly in Section A.3.) Thus it would be useful to have one language that we are certain satisfies a strong property, such as type preservation.

The second is that type preservation is a more basic property than type soundness. Type preservation is what ensures that the type system and the operational semantics fit together properly. For example, it is what ensures that if a program is given the type $\texttt{int} \rightarrow \texttt{int}$ by the type system, then, if it is evaluated with an integer argument, then the result really is an integer. Once type preservation is proved, type soundness is then a matter of checking that type system checks all subparts of the term, and that the evaluation system has a rule for each possible case for a correctly-typed term (for example, there is an evaluation rule for the application of each form of value that can be given function type).

A third answer is that, as noted, type preservation is a necessary and significant step towards proving type soundness. Furthermore many, if not most, of the errors that would prevent a proof of type soundness are likely to show up in an attempt to prove type preservation.

Fourth, there have been many changes proposed to Standard ML, such as weak polymorphism (a form of which has been implemented in SML/NJ), first-class continuations,

and polymorphism by name. There have been proofs of type preservation or type soundness published for each of these extensions [HMV92, WF92, Leroy93, HDM93]. However, these proofs have been done only in small subsets of SML, and there is no guarantee that they would not interact badly with some language feature not included in the study. Thus a proof of the type preservation of SML, if it were easily understood and extendible, could provide a framework into which researchers could investigate the effects of new language features or extensions to the typechecking rules.

Fifth, proving type preservation of Core SML would demonstrate the utility of mathematical specifications of programming languages. Informal language specifications are often easier to understand than mathematical ones; however, they lack the precision required to prove desirable properties about a language. A mathematical specification can more accurately give the meanings of programming language constructs, but if the mathematics is too complicated it will be useless. A proof of an important property such as type preservation, then, can act as a demonstration that the specification is a sensible one.

Sixth, in addition to its theoretical importance, a proof of type preservation would be useful to researchers working on verification of SML programs. For example, say that we would like to prove the correctness of a list append function. Since we only need to show that an append function performs its duties correctly on values, we would like to factor out the evaluation of the arguments of the function from the evaluation of the function itself. A possible statement of this theorem is then:

Theorem 7.1 *Let* $\langle\langle$*list_exp_1*$\rangle\rangle$ *and* $\langle\langle$*list_exp_2*$\rangle\rangle$ *be list expressions, that is,*

$$s, C_0 \vdash \langle\langle \textit{list\_exp\_1} \rangle\rangle : \tau \texttt{ list}$$

and

$$s, C_0 \vdash \langle\langle \textit{list\_exp\_2} \rangle\rangle : \tau \texttt{ list}$$

for some τ*, where* C_0 *is the initial context. Let* E *be such that*

$$s, E_0 \vdash \texttt{val l1 = } \langle\langle \textit{list\_exp\_1} \rangle\rangle \texttt{;val l2 = } \langle\langle \textit{list\_exp\_2} \rangle\rangle \Downarrow E, s'$$

where E_0 *is the initial environment. Say that*

$$s', E_0 + E \vdash \texttt{let val append = } \langle\langle \textit{append\_fun} \rangle\rangle \texttt{ in append l1 l2 end} \Downarrow v/p, s''$$

Then v/p *is a value* v *and not a packet,* $s' = s''$*, and* $v =$ APPEND E(`l1`) E(`l2`) *(where* APPEND *is a mathematical description of the intended effect of the append function).*

Type preservation would ensure that the values associated with variables $l1$ and $l2$ are in fact proper list values.

Chapter 8

Towards Proving Type Preservation

The first step in proving type preservation is precisely defining the notions used in the statement of type preservation at the end of Section 7.1. In particular, we need to define what "E agrees with C" and "v has type τ" means, and we need to include a hypothesis that ensures that the state agrees with the context. It is not at all clear how to phrase these definitions, and, as they are crucial in the proof, we expound on them in the first section of this chapter.

Then we restate the type preservation property in a precise way and discuss the progress made towards proving it. The biggest accomplishment to date has been the proof of typechecking under substitution. This theorem and its proof are discussed in the last section of this chapter.

8.1 How to Give a Type to a Value

The published proof that is closest to a proof of type preservation for SML is in [Tofte90]. There, Tofte uses a natural semantics that is close to that used in the Definition, including a similar treatment of state and environments in the evaluation rules. Thus we used Tofte's definitions as a starting point for our own. The idea is that, boiled down to the basics, E maps variables to values, and C maps variables to type schemes, so "E agrees with C" if, for every variable, its value in E can have the type scheme associated with it in C.

We will start to describe how to give a type to a value. For simple semantic objects such

as integers and strings, this is easy. It is only slightly more complicated for basic values (such as addition) and assignment: the types for these values is given by the Definition, so we just check that the type matches the expected one. For records and constructed values (such as `5::l` for an integer list `l`) this is straightforward: the type is defined structurally, using the types given by the context for the value constructors. For closures it is a bit more complicated, as the closure contains a match, which is a syntactic phrase, along with the evaluation environment at the time the match was put into the closure. To give a type for the closure, then, one must find a context that agrees with the evaluation environment and elaborate the match in this context.

What do we do if the value is an address? Now we have to consider the contents of the store (the store is the same as what is referred to in the Definition as the memory—a mapping from addresses to values). Here we introduce the notion of a *store typing*, a map from addresses to types. This store typing is needed to take account of two facets of references—aliasing (two variables can refer to one reference cell) and the fact that the contents of a reference cell can change over time. The store typing ensures that if variables x and y refer to the same reference cell, containing a value of type τ, then they have the same type, namely τ `ref`. One might protest x and y do not need to have the same type if the contents of the cell is a polymorphic object, such as the identity function. In this case, `x` could have type (`int` $\rightarrow$ `int`)`ref` while `y` could have type (`bool` $\rightarrow$ `bool`)`ref`. However, if the call is later assigned the successor function, then `y`'s type would then be incorrect. To avoid this problem, types of items in reference cells are required to be simple, that is, not polymorphic. This is why a store typing is a map from addresses to *types*, not type schemes.

The presence of reference cells means there are two relations to be defined: when a store matches a store typing, and when a value has a type. Tofte combines these two relations into one. He writes $m : ST \models v : \tau$ to mean that given a typed store $m : ST$, value v has type τ. As part of the definition, he ensures that store m agrees with the store typing ST—if v is an address a, then τ must be $(ST(a))$`ref` and $m : ST \models m(a) : ST(a)$. The problem with this definition is that it cannot be defined by regular induction, as there can be circularities in the store. Consider the following three SML statements:

```
val f = ref (fn x:int => 5);
val g = ref (fn x => if x = 1 then 6 else (!f)(x - 1));
f := (fn x => if x = 1 then 5 else (!g)(x - 1));
```

The result of this is a store in which f refers to g and vice versa. Thus, in trying to determine that $m : ST \models a_f : (\texttt{int} \rightarrow \texttt{int})\,\texttt{ref}$, where m is the resulting store, ST is the resulting store typing, and a_f is the address to which f is mapped, we must check that $m : ST \models a_g : (\texttt{int} \rightarrow \texttt{int})\texttt{ref}$, which in turn requires showing that $m : ST \models a_f : (\texttt{int} \rightarrow \texttt{int})\,\texttt{ref}$.

Because of the circularity present here, Tofte must use maximal fixed points to give the meaning of this relation. The principle of co-induction comes from this definition, just as our induction theorems for the evaluation relations come from the definition of the relations as the least relations satisfying the rules. Co-induction is an unfamiliar principle, and it would be nice to dispense with it, if possible.

Leroy [Leroy93] defines these concepts in a different way, resulting in a simpler definition. The key is that one does not need to know the value at a given address in a store in order to know its type: one only has to look up the address in the store typing. Thus he defines (using Tofte's terminology) two different relations: $ST \models v : \tau$ and $\models m : ST$. The first determines when a value has a given type, depending on the store typing, and the second determines that a store agrees with its store typing. Using these definitions, the store resulting from the SML statements above does not present definitional circularities, despite the fact that the store itself is circular: verifying that $ST \models a_f : (\texttt{int} \rightarrow \texttt{int})\,\texttt{ref}$ involves only looking up the type of a_f in ST. Verifying that $\models m : ST$ does involve checking that the contents of a_f and a_g both have type $\texttt{int} \rightarrow \texttt{int}$, but the check that the value at address a_f has type $\texttt{int} \rightarrow \texttt{int}$ involves only checking the type of a_g, not its contents. The definitions given in the style of Leroy avoid the use of maximal fixed points as a definitional method and thus avoid co-induction as a method of proving lemmas.

The definitions of these relations must be expanded considerably in order to account for datatype declarations. E and C must be included in the value-has-type relation (written above as $ST \models v : \tau$). E is needed to determine which constructors are associated with the constructor names in the value,[1] and C is needed to find the type of this constructor. Similarly, E and C must be included in the store-has-storetype relation (written above as $\models m : ST$). In addition, we will use several mutually recursive relations in order to clarify the presentation. These relations match the structure of values. There is be a value-has-type relation (which will be written $\models_v$) that covers the assignment operator, special values (integers and strings), basic values (built-in operators like + and `div`), and

[1] See Section A.3 for an discussion of constructor names.

address values. There are conval-has-type ($\models_c$) and exval-has-type ($\models_e$) relations that specify when ConVals and ExVals have a given type. There is a closure-has-type ($\models_f$) relation for closures, and a record-has-rectype ($\models_r$) relation that specifies when a record has a given record type. We will refer to the collection of these relations as the val-type relations.

Let VE_s be a variable environment for the static semantics (that is, a map from variables to type schemes) and VE_d be a variable environment for the dynamic semantics (a map from variables to values). We need a relation that states that VE_d agrees with VE_s when the status of identifiers in VE_s is given by SM (status maps are explained in Section A.2). This relation is written $C, E_d, ST \models VE_d : VE_s, SM$, where C is a context and E_d a dynamic environment. We first check that the domains of VE_d and VE_s are the same. For any i in the domain of VE_d, if $SM(i)$ is c_stat or e_stat, we assume that this entry is the result of the original definition of the constructor name or exception name and do not check further.

If the status is of i is v_stat, then we need to determine that the value $VE_d(i)$ agrees with the type scheme $VE_s(i)$. The natural way to do this (which is the method used by Tofte) is to say that a value v agrees with a type scheme σ if v has type τ for any τ that is an instantiation of σ. Unfortunately these relations will have to be defined with our mutually recursive relations package, and Tofte's definition cannot be formulated in the package—the hypotheses of the rule can only be single instances of the relations being defined, or side conditions that do not mention the relations. We cannot have hypotheses that are universally quantified instances of the relations being defined. We solve this difficulty by including in the val-type relations yet another argument, a set of type variables tv, and arrange so that the relation returns true if the value has the type, and the type variables in tv can be substituted for. The relationship for values is written $C, E_d, ST \models_v v : tv, \tau$.

The notion that a set of type variables in a type can be substituted for needs some explanation. Say the type scheme σ is $\forall\vec{\alpha}.\tau'$. The statement "$v$ has type τ for any τ that is an instantiation of σ" means that for any substitution S with domain $\vec{\alpha}$, v has type $\tau'[S]$ (where $\tau'[S]$ is the substitution S applied to τ'). Thus, to show that $C, E_d, ST \models_v v : tv, \tau$, we must show that v has type τ and that for any substitution with domain tv, v can also have type $\tau[S]$.

In order to describe how to define $C, E_d, ST \models_v v : tv, \tau$ we first note that the relation will be defined inductively, with the types of values depending on the sub-parts of the values, so tv may contain variables not in the values. Table 8.1 shows the definitions of

mk_var_SM VE	$=$	SM such that Dom SM = Dom VE and Rng SM = {v_stat}
basval_type $b\,\tau$	$=$	true iff τ is the type of the basic value b
type_of_sval s	$=$	`int` if the value is an integer, `string` if the value is a string
conname_type $C\ E_d\ cn\,\tau$	$=$	$\exists\, longcon\,\sigma.$ $E_d(longcon) = cn$ and $C(longcon) = \sigma$ and $(\sigma \succ \tau)$
exname_type $C\ E_d\ tv\ en\,\tau$	$=$	$\exists\, longexcon.$ $(tv \bigcap \text{tyvars}\,\tau = \emptyset)$ and $E_d(longexcon) = en$ and $C(longexcon) = \tau$
closure_VE $C\ VE_d\,VE_s$	$=$	Dom VE_d = Dom VE_s and $\forall\, i \in$ Dom $VE_d.\ \exists match.$ reduce $match = match_d$ and $C \vdash match : \tau$ and $\vec{\alpha} \bigcap$ free_tyvars $C = \emptyset$ (where $VE_s(i) = \forall\, \vec{\alpha}.\,\tau$ and $VE_d(i) = (match_d, E, \{\}))$
constructors_ok $E_d\,C$	$=$	$(\forall\, longcon_1\ longcon_2\ cn\,\sigma_1\ \sigma_2.$ $E_d(longcon_1) = cn$ and $E_d(longcon_2) = cn$ and $C(longcon_1) = \sigma_1$ and $C(longcon_2) = \sigma_2 \Rightarrow$ $\sigma_1 = \sigma_2)$ and $(\forall\, longexcon_1\ longexcon_2\ en\,\sigma_1\ \sigma_2.$ $E_d(longexcon_1) = en$ and $E_d(longexcon_2) = en$ and $C(longexcon_1) = \sigma_1$ and $C(longexcon_2) = \sigma_2 \Rightarrow$ $\sigma_1 = \sigma_2)$
env_of_str $(strname, E_s)$	$=$	E_s
mem_of_state (mem, es, cs)	$=$	mem

Table 8.1: Primitive Functions for val_has_type

the basic relations and functions used by the val-type relations. The functions prefix and body are defined so that if $\sigma = \forall\vec{\alpha}.\,\tau$, then body $\sigma = \tau$ and prefix $\sigma = \vec{\alpha}$. Table 8.2 shows the rules that define the value-has-type relation. Table 8.3 shows the rules that define the other val-type relations, those stating when an ExVal, ConVal, record, and closure have a given type.

First we consider the base cases in the definition, the simple (non-compound) values. Special values (integers and strings) and HOL-SML's basic values (basic operators such as plus) can only have one type, and so τ must be that type. Since τ in this case contains no type variables, for any substitution S, $\tau[S] = \tau$, so the type variables in tv can be

substituted for.

The assignment operator can have any τ that is an instantiation of $\forall\alpha.\,\alpha\,\texttt{ref}*\alpha \to \texttt{unit}$. Now we have to determine whether the type variables in tv can be substituted for. Let S be any substitution with domain tv. We have to determine if the assignment operator can have type $\tau[S]$. Now, if τ is an instantiation of $\forall\alpha.\,\alpha\,\texttt{ref}*\alpha \to \texttt{unit}$, then there exists a τ' such that $(\alpha\,\texttt{ref}*\alpha \to \texttt{unit})[\tau'/\alpha] = \tau$. Then

$$(\alpha\,\texttt{ref}*\alpha \to \texttt{unit})[\tau'[S]/\alpha] = \tau[S]$$

thus the assignment operator has type $\tau[S]$. An example will help explain this. Say we want to show that the assignment operator has type $(\beta*\gamma)\,\texttt{ref}*(\beta*\gamma) \to \texttt{unit}$ and β can be substituted for. The substitution (the τ'/α used above) that effects the instantiation of the type scheme for assignment to this type is $(\beta*\gamma)/\alpha$. Let S be any substitution with domain $\{\beta\}$. Then $S = \tau_2/\beta$ for some type τ_2. Then we need to show that the assignment operator has type

$$((\beta*\gamma)\,\texttt{ref}*(\beta*\gamma) \to \texttt{unit})[\tau_2/\beta] = (\tau_2*\gamma)\,\texttt{ref}*(\tau_2*\gamma) \to \texttt{unit}$$

The substitution $((\beta*\gamma)[\tau_2/\beta])/\alpha = (\tau_2*\gamma)/\alpha$ shows this is the case.

For addresses, we look up the address in the store type to find the type it must have, and this must be the same as τ. Since the range of the store type consists of simple types, an address can only have one type, and so none of its type variables (if there are any) can be substituted for. Thus it must be the case that there is no intersection between tv and the type variables in τ.

In order to define the exval-has-type relation, we need to know what type an exception name can have. To find whether an exception name has a type τ, we need to make use of both the dynamic environment and the context. First, we look in the dynamic environment to find a longExCon *longexcon* such that $E(longexcon)$ is the exception name. Then we get the type scheme by looking up the longExCon in the context. In a proper context, this type scheme will be a simple type—none of the type variables will be quantified over—so τ must be this type. As with addresses, we cannot substitute for any of the variables in this type, so we check that none of the variables in tv are in the type.

In order to define the conval-has-type relation, we need to know what type a constructor name can have. In HOL-SML we keep information on constructors in the dynamic environment (see page 130). If a dynamic environment E and context C agree with each

$$\frac{}{C, E_d, ST \models_v (\texttt{:=}) : tv, (\tau\ \texttt{ref} * \tau) \rightarrow \texttt{unit}}$$

$$\frac{\text{basval\_type}\ b\ \tau}{C, E_d, ST \models_v b : tv, \tau}$$

$$\frac{C, E_d, ST \models_c conval : tv, \tau}{C, E_d, ST \models_v conval : tv, \tau}$$

$$\frac{C, E_d, ST \models_e exval : tv, \tau}{C, E_d, ST \models_v exval : tv, \tau}$$

$$\frac{C, E_d, ST \models_r r : tv, \varrho}{C, E_d, ST \models_v r \text{ in Type} : tv, \varrho \text{ in Val}}$$

$$\frac{ST(a) = \tau \quad tv \bigcap \text{tyvars}\ \tau = \emptyset}{C, E_d, ST \models_v a : tv, \tau\ \texttt{ref}}$$

$$\frac{ST \models_f closure : tv, \tau}{C, E_d, ST \models_v closure : tv, \tau}$$

Table 8.2: Rules for $\models_v$

other, for every longCon *longcon*, $C(longcon)$ returns its type scheme, and $E(longcon)$ returns the constructor name. Because of this the case for constructor names is similar to that for exception names, except that we must take into consideration the fact that the type schemes are not simple types. First we look in the dynamic environment to find a longCon *longcon* such that $E(longcon)$ is the constructor name. Then we look up that longCon in the context to get the type scheme. To check that a constructor name has type τ, we check that τ is an instantiation of the type scheme.

If the constructor name has type τ, then we must check that the type variables in tv can be substituted for. If there are free (unquantified) variables in the type scheme, then (as with exception names) those variables cannot be substituted for. If there are no free variables, then (as with the assignment operator) the substitution that effects the instantiation of the type scheme to τ can be modified so that it effects an instantiation to $\tau[S]$, where S is any substitution with domain tv. As discussed in Section A.2 (page 123),

type schemes that are returned by looking up a longCon are the results of the original declaration of the constructor, and (we have proved this) these type schemes have no free variables. Thus any of the type variables can be substituted for.

The case for closures (needed for the closure-has-type relation) is complicated. A closure, written $(match_d, E_d, VE_d)$, consists of a match phrase $match_d$, a dynamic environment E_d, and a dynamic variable environment VE_d. E_d is the environment at the time the function expression containing the match was evaluated. The match phrase is a phrase in the syntax of the dynamic semantics, that is, in the reduced syntax. The match $match_d$ may be part of a mutually recursive collection of functions. If it is, the variable environment VE_d gives declarations of any recursive functions that are referred to in the match. If the match is not part of a mutually recursive collection of functions, VE_d is empty.

Thus there are two cases for checking $ST \models_f (match_d, E_d, VE_d) : tv, \tau$. If VE_d is empty, then we need that there exists a context C and a match in the unreduced syntax *match* such that E_d agrees with C, $match_d$ is the reduced version of *match*, $C \vdash match : \tau$, and none of the type variables in tv are free in C. The idea behind this definition is that C is the context in which the *match* was elaborated. We need the fact that none of the type variables in tv are free in C since the elaboration of the match may have depended on the type of these free variables. For the same reason, type variables that are free in C are not generalized in the use of Clos in Rule 17.

If VE_d is not empty, then the variables in the domain of VE_d may appear in $match_d$ (and hence in *match*), so we need to check the type of *match* in a context in which these variables are given types. Thus, in addition to *match* and C, we need that there exists a static variable environment VE_s with the same domain as VE_d, that gives the types for the mutually recursive functions, and we check the type of *match* against the context $C + VE_s$. In addition, we must check that the closures in the range of VE_d have the type scheme given them in VE_s.

The compound values are compound exception values (a form of ExVal, shown in the dynamic semantic objects in Figure 13 in the Definition as ExName $\times$ Val), compound constructor values (a form of ConVal, shown in Section A.3 of this paper as ConName$\times$Val), and records. A compound exception matches a type τ_2 if the exception name has type $\tau_1 \rightarrow \tau_2$ and the value with which it is paired has type τ_1. A compound constructor value matches a type τ_2 if the constructor name has type $\tau_1 \rightarrow \tau_2$ and the value with which it is paired has type τ_1. In both cases we check that the type variables in tv can be substituted

$$\frac{\text{conname\_type } C\ E_d\ cn\,\tau}{C, E_d, ST \models_c cn : tv, \tau}$$

$$\frac{\text{conname\_type } C\ E_d\ cn\,(\tau_1 \to \tau_2) \quad C, E_d, ST \models_v v : tv, \tau_1}{C, E_d, ST \models_c (cn, v) : tv, \tau_2}$$

$$\frac{\text{exname\_type } C\ E_d\ tv\ en\,\tau}{C, E_d, ST \models_e en : tv, \tau}$$

$$\frac{\text{exname\_type } C\ E_d\ tv\ en\,(\tau_1 \to \tau_2) \quad C, E_d, ST \models_v v : tv, \tau_1}{C, E_d, ST \models_e (en, v) : tv, \tau_2}$$

$$\frac{\text{Dom } r = \text{Dom } \varrho \quad \forall l \in \text{Dom } r.\ C, E_d, ST \models_v r(l) : tv, \varrho(l)}{C, E_d, ST \models_r r : tv, \varrho}$$

$$\frac{\begin{array}{c}\text{reduce } match = match_d \quad ST \models E_2 : C \\ C \vdash match : \tau \quad tv \bigcap \text{free\_tyvars } C = \emptyset\end{array}}{ST \models_f (match_d, E_2, \{\}) : tv, \tau}$$

$$\frac{\begin{array}{c}VE_d \neq \{\} \quad SM = \text{mk\_var\_SM } VE \\ \text{closure\_VE } (C + (VE_s, SM))\ VE_d\ VE_s \quad \text{reduce } match = match_d \\ C + (VE_s, SM) \vdash match : \tau \quad tv \bigcap \text{free\_tyvars } C = \emptyset \quad ST \models E_2 : C\end{array}}{ST \models_f (match_d, E_2, VE_d) : tv, \tau}$$

$$\frac{C, E_d, ST \models_v v : \emptyset, \tau}{C, E_d, ST \models_{vp} v : \tau}$$

$$\frac{}{C, E_d, ST \models_{vp} p : \tau}$$

Table 8.3: Rules for $\models_c$, $\models_e$, $\models_r$, $\models_f$, $\models_{vp}$,

$$\frac{ST \models E_d : E_s \quad \text{constructors\_ok}\ E_d\ (tynames, E_s)}{ST \models E_d : (tynames, E_s)}$$

$$\frac{C, E_d, ST \models SE_d : SE_s \quad C, E_d, ST \models VE_d : VE_s}{C, E_d, ST \models (SE_d, VE_d, EE_d) : (SE_s, TE_s, VE_s, EE_s, SM_s)}$$

$$\frac{\begin{array}{c}\text{Dom}\ VE_d = \text{Dom}\ VE_s \quad \text{Dom}\ VE_s = \text{Dom}\ SM \\ (\forall i \in \text{Dom}\ VE_d.\ SM(i) = \text{v\_stat} \Rightarrow \\ C, E_d, ST \models_v VE_d(i) : \text{prefix}\ (VE_s(i)), \text{body}\ (VE_s(i)))\end{array}}{C, E_d, ST \models VE_d : VE_s, SM}$$

$$\frac{\begin{array}{c}\text{Dom}\ SE_d = \text{Dom}\ SE_s \\ \forall i \in \text{Dom}\ SE_d.\ C, E_d, ST \models SE_d(i) : \text{env\_of\_str}\ (SE_s(i))\end{array}}{C, E_d, ST \models SE_d : SE_s}$$

$$\frac{\begin{array}{c}mem = \text{mem\_of\_state}\ s \quad \text{Dom}\ mem = \text{Dom}\ ST \\ \forall a \in \text{Dom}\ mem.\ C, E_d \models mem(a) : ST(a)\end{array}}{C, E_d \models s : ST}$$

Table 8.4: Rules for Environments and Store Types

for. For a proper context, if $C(longexcon)$ is a function type, the range type can only be `exn`, so for a proper context, τ will be `exn`.

For record values, we check that the labels in the record are the same as the labels in the record type, and that the value associated with a label in the record has the type associated with it in the record type. In Table 8.3, the value-has-type relation is universally quantified in a hypothesis of the rule. This is not a permissible form for a rule in our mutually recursive relations package. Thus the rule as written is not how it is actually encoded. Both record values and record types are encoded as finite maps, which are actually lists of pairs, with each pair having a label and a value (for record values) or type (for record types). The labels are sorted, so we use an auxiliary relation that steps through the lists, checking at each step that the labels are the same and that the value in the record has the type in the record type. The same trick is used for structure environments and variable environments.

Other semantic objects that can be returned as the result of an evaluation, such as FAIL or packets, can have any type. Thus we define yet another relation $\models_{vp}$, with the rules shown in Table 8.3.

To define $ST \models E_d : C$, stating that dynamic environment E_d agrees with static context C, we need two mutually recursive definitions: $E, C, ST \models SE_d : SE_s$, stating that a dynamic structure environment agrees with a static structure environment, and

$ST \models E_d : E_s$, saying that a dynamic environment agrees with a static environment. The relation $E, C, ST \models SE_d : SE_s$ is defined similarly to that for dynamic and static variable environments, that is, the domains must be the same and for each structure id i in the domain, the dynamic environment associated with i in SE_d must match the static environment in SE_s. The relation $ST \models E_d : E_s$ is satisfied if the dynamic and static structure environments agree and the dynamic and static variable environments agree.

Finally, $ST \models E_d : C$ if $ST \models E_d : E_s$, where E_s is the environment component of the context, and constructors_ok is satisfied. The predicate constructors_ok is defined in Table 8.1. It checks that if a constructor or exception name is in the dynamic environment in two different places (represented by two different longCons or longExCons), then the type schemes at these places in the context are identical. If constructors_ok is satisfied, the definition of conname_type does not depend on which longCon or longExCon is used. The rules for these relations are given in Table 8.4.

With all these definitions behind us, stating when a state matches a store type is easy. It is shown in Table 8.4.

8.2 The Type Preservation Theorem

As shown in the previous section, we need to use store types in order to state the type preservation theorem. Using Tofte's terminology, we will refer to the pairing of a state and a store type, where the memory in the state and the store type have the same domain, as a typed store, and will write this as $s : ST$ for a state s and store type ST. A typed store *succeeds* another, written $s : ST \sqsubseteq s' : ST'$, if $s = (mem, ens, cns)$ and $s' = (mem', ens', cns')$ and $ens \subseteq ens'$ and $cns \subseteq cns'$ and $\mathrm{Dom}\,(mem) \subseteq \mathrm{Dom}\,(mem')$ and $ST \subseteq ST'$ and $mem \subseteq mem'$.[2]

Now we are prepared to formally state the type preservation theorem.

Theorem 8.1 (Type Preservation) *Say the following hold:*

$$C, E \models s : ST \qquad ST \models E : C$$
$$C \vdash exp : \tau \qquad s, E \vdash \texttt{reduce}\ exp \Downarrow v/p, s'$$

[2]The inclusion on maps is meant to be inclusion of the sets of pairs $(a, ST(a))$ or $(a, mem(a))$ in the graph of the map. For example, $ST \subseteq ST'$ means that the domain of ST is included in the domain of ST', and for each a in the domain of ST, $ST(a) = ST'(a)$.

Then there exists a store typing ST' *such that* $C, E \models s' : ST'$ *and* $s : ST \sqsubseteq s' : ST'$ *and* $C, E, ST' \models_{vp} v/p : \tau$.

Note that the theorem above states the type preservation property only for expressions. A similar property was formulated for each of the phrase classes, and then we set about trying to prove the theorems. The proof of these theorems proceeds by induction over the elaboration rules. We succeeded in proving type preservation for the phrase class groups including the exceptions, constructor bindings, datatype bindings, and pattern phrases. In short, we proved the property for all groups of phrases except for the group including the expressions and declarations.

The problem with proving the theorem for expressions is that substitutions must be dealt with. Substitutions come into play, for example, in the case for variables. Since $C \vdash \mathit{longvar} : \tau$, then, by inversion of Rule 2, $C(\mathit{longvar}) = \sigma$ and $\sigma \succ \tau$. For evaluation, if the result is a packet, it can have any type, so the interesting case is that for a result that is a value. The reduced syntax for a variable is just that variable, and $s, E \vdash \mathit{longvar} \Downarrow v, s'$ implies, by inversion of Rule 104, that $E(\mathit{longvar}) = v$. Say $\sigma = \forall \vec{\alpha}.\tau'$ Since $ST \models E : C$, we know (again by inversion of the defining rules) that[3] $C, E, ST \models_v v : \vec{\alpha}, \tau'$. We need to show that $C, E, ST \models_v v : \emptyset, \tau$. Since $\sigma = \forall \vec{\alpha}.\tau' \succ \tau$, we end up showing that our definition of $C, E, ST \models_v v : tv, \tau$ properly captures the notion that the type variables in tv can be substituted for.

The general statement of the theorem we need to prove is: Say $C, E, ST \models_v v : tv, \tau$ and S is a substitution with domain tv. We need to prove that $C, E, ST \models_v v : \emptyset, \tau[S]$. This must be proved by induction over the rules that define the val-type relations. The most difficult case in this proof is for closures. Recall that a closure contains a match phrase, and in order for it to have a given type while tv can be substituted for, there must be a context C' such that $tv \bigcap \mathrm{free\_tyvars}\, C' = \emptyset$ and $C' \vdash \mathit{match} : \tau$. In order to prove that the closure has type $\tau[S]$, we need to show that $C' \vdash \mathit{match} : \tau[S]$.

Thus we need a theorem that we call *typechecking under substitution*. This corresponds to Lemma 4.2 in [Tofte90]. The above paragraph describes roughly what needs to be proved. The details are as follows. Say that we have closure value $(\mathit{match}_d, E_d, \{\})$ and we have that $C, E, ST \models_v v : tv, \tau$. By inversion of the first rule for closures in Table 8.3 we

[3] We have written $\vec{\alpha}$, which is a vector of type variables, in a position that needs a set of type variables. In our HOL encoding we have to explicitly convert this to a set.

have

$$\begin{aligned}&\exists\, match\, C'.\\ &\text{reduce } match = match_d \text{ and } ST \models E_d : C' \text{ and}\\ &C' \vdash match : \tau \text{ and } tv \bigcap \text{free\_tyvars}\, C' = \emptyset\end{aligned} \tag{8.1}$$

In order to prove $C, E, ST \models_v v : \emptyset, \tau[S]$, we need to show that

$$\begin{aligned}&\exists\, match'\, C''.\\ &\text{reduce } match' = match_d \text{ and } ST \models E_d : C'' \text{ and}\\ &C'' \vdash match' : \tau[S]\end{aligned} \tag{8.2}$$

Since we are given Equation 8.1, we do not get to choose the value for variables *match* and C'; they are given to us. We will refer to these values by the variable names. So we can assume

$$\begin{aligned}&\text{reduce } match = match_d \text{ and } ST \models E_d : C'\\ &C' \vdash match : \tau \text{ and } tv \bigcap \text{free\_tyvars}\, C' = \emptyset\end{aligned} \tag{8.3}$$

We must prove Equation 8.2, so we get to choose the witness for the existential quantifiers. We will choose C'' to be the same context as in Equation 8.1, that is, C'. We would like to choose $match'$ to be the same as *match*, but this will not work. Say *match* is the indentity function with a type constraint `fn x:'a => x`. According to the typing rules, this can only elaborate to one type, $\alpha \rightarrow \alpha$.[4] This is because, according to Rule 45, a typed pattern (the formal argument `x:'a`) can only have the type that is explicitly given, which is α. Thus in order to elaborate to a type $\tau_2 \rightarrow \tau_2$ for some type τ_2, we must change the match phrase. The easiest way to do this is to remove the type tags from the phrases.[5] Let strip_tags be a function that does this.[6] We let (strip_tags *match*) be the value for $match'$. Then Equation 8.2 becomes

$$\begin{aligned}&\text{reduce (strip\_tags } match) = match_d \text{ and}\\ &ST \models E_d : C' \text{ and } C' \vdash \text{strip\_tags } match : \tau[S]\end{aligned}$$

[4] Note that `'a` is used in the syntax, while we use α to mean the same type variable in semantic objects. This is standard usage for SML-like languages.

[5] We could try to substitute the type variables in the syntax with syntactic representations of the types in the range of the substitution, but this will not work. In order to have a syntactic representation of a type we need that for each type name in the type, the context has a type constructor that maps to that type name (see Rule 49). There may have been new datatypes declared since the *match* was typechecked, and the type names associated with those declarations would not be in the context C'.

[6] The function strip_tags must do something a bit different with exception declarations. If we strip the type tag from an exception declaration, this means that the exception does not carry a value—see Rules 31*a* and 31*b*. Thus instead of stripping off the type tag, we replace the type in the type tag with a new syntactic type `genericty`, which can have any type (see Rule 51.1).

The first conjunct, reduce(strip_tags *match*) = $match_d$, is easy to prove since reduce *match* = $match_d$, and reduce removes type tags anyway. We have the second conjunct, $ST \models E_d : C'$, already from Equation 8.3.

We are left to prove $C' \vdash$ strip_tags *match* : $\tau[S]$, assuming that $C' \vdash$ *match* : τ and $tv \bigcap$ free_tyvars$C' = \emptyset$. We call this theorem *typechecking under substitution*, and it was the largest and most complicated theorem we proved about HOL-SML. It is not surprising that proving this theorem and all the lemmas it entailed should be so complicated. Substitution has caused problems in mathematical logic from the beginning. One of Gödel's main complaints about Russell's *Principia Mathematica* is the lack of formality in the syntax; that in particular, he did not prove needed substitution theorems (see [Gödel83]).

After completing the proof of typechecking under substitution we did not go on to try to finish the proof of type substitution for the class of phrases containing expressions. During the course of proving these theorems, we found some examples of expressions for which type preservation does not hold. These are discussed in Section 8.5.

8.3 Substitution

We use some slightly nonstandard notation here, and as proving properties of substitutions played such a large part in this project, we will explain it here. For the purposes of this thesis, substitutions are maps from type variables to types. Occasionally we will use a substitution that maps type variables to type variables; it is to be understood as a map to types that are just type variables.

Substitutions arise in the instantiation of type schemes. We show explicitly the domain and range of the substitution using the following notation: τ/α, pronounced "τ for α", means that α is in the domain of the map, and τ is the value of the map at that variable. Simultaneous substitutions involving substitutions that have disjoint domains are represented by splicing the substitutions together with commas, as in $\tau_1/\alpha_1, \tau_2/\alpha_2$. If $\vec{\tau}$ is a collection of types $\tau_1 \ldots \tau_n$ and $\vec{\alpha}$ is a collection of type variables $\alpha_1 \ldots \alpha_n$, then the substitution $\vec{\tau}/\vec{\alpha}$ is short for $\tau_1/\alpha_1, \tau_2/\alpha_2, \ldots, \tau_n/\alpha_n$.

A substitution S applied to a type τ is written $\tau[S]$. Some examples will help explain the concept.

$$(\alpha\,\texttt{list})[\texttt{int}/\alpha] = \texttt{int list}$$

$$(\alpha\,\texttt{list})[\texttt{int}/\beta] = \alpha\,\texttt{list}$$

$$(\alpha\,\texttt{list})[(\beta * \texttt{int})/\alpha] = (\beta * \texttt{int})\,\texttt{list}$$

$$((\beta * \alpha)\,\texttt{list})[\texttt{int}/\alpha] = (\beta * \texttt{int})\,\texttt{list}$$

A type scheme $\sigma = \forall\,\vec{\alpha}.\tau$ is instantiated to a type with a substitution. Let $\vec{\tau}$ be a collection of types the same length as $\vec{\alpha}$. If $\tau[\vec{\tau}/\vec{\alpha}] = \tau'$, then we write $\sigma \succ \tau'$ and say "τ' instantiates σ" or "σ is a generalization of τ'". We say that the substitution $\vec{\tau}/\vec{\alpha}$ "effects the instantiation of σ to τ'". An example of instantiation is that $\forall\,\alpha.\,\alpha\,\texttt{list} \succ (\beta * \texttt{int})\,\texttt{list}$. The substitution that effects the instantiation is $(\beta * \texttt{int})/\alpha$.

The type variables in $\vec{\alpha}$ are said to be *bound* in $\forall\,\vec{\alpha}.\,\tau$, and $\vec{\alpha}$ is said to be the *prefix* of the type scheme, while τ is said to be the *body*. We often say that the type variables in $\vec{\alpha}$ have been *generalized* or *quantified* to make σ. Type variables appearing in τ and not in $\vec{\alpha}$ are said to be *free* in $\forall\,\vec{\alpha}.\,\tau$.

Two type schemes $\sigma = \forall\,\vec{\alpha}.\,\tau$ and $\sigma_2 = \forall\,\vec{\beta}.\,\tau'$ are *alpha-equivalent* if there is some rearrangement $\vec{\beta}_2$ of the prefix of σ_2 such that $\tau' = \tau[\vec{\beta}_2/\vec{\alpha}]$. Note that this means that $\vec{\alpha}$ and $\vec{\beta}$ must have the same length: type schemes here are not allowed to have type variables in the prefix that are not in the body (see the footnote on page 109). The easy way to think of alpha-equivalence is to think of the variables in the prefix as having been renamed—there is a one to one correspondence of the variables such that replacing one set with the other in the body of one type scheme results in the body of the other (and these corresponding variables must be the contents of the prefixes). For example, $\forall\,\alpha.\,\alpha * \beta$ is alpha-equivalent to $\forall\,\gamma.\,\gamma * \beta$, but is not alpha-equivalent to $\forall\,\beta.\,\beta * \beta$.

Alpha-conversion involves picking new names for the bound variables and applying the transformation to a type scheme (the new names must be different from the free variables in the type scheme), resulting in a type scheme that is alpha-equivalent to the first. Type schemes are often considered equal up to alpha-conversion, that is, an equal sign may appear between two type schemes if they are not exactly the same, but are alpha-equivalent. We do not adopt this notion here; when we want to allow type schemes to be alpha-equivalent to each other, we will state so.

Two variable environments VE_1 and VE_2 are alpha-equivalent if they have the same domain and for each identifier i in the domain, $VE_1(i)$ is alpha-equivalent to $VE_2(i)$. Similarly, two structure environments SE_1 and SE_2 are alpha-equivalent if they have the same domain, and for each s in the domain, if $SE_1(s) = (m_1, E_1)$ and $SE_2(s) = (m_2, E_2)$, then $m_1 = m_2$ and E_1 is alpha-equivalent to E_2. Two environments $(SE_1, TE_1, VE_1, EE_1, SM_1)$

and $(SE_2, TE_2, VE_2, EE_2, SM_2)$ are alpha-equivalent if SE_1 is alpha-equivalent to SE_2, $TE_1 = TE_2$, VE_1 is alpha-equivalent to VE_2, $EE_1 = EE_2$, and $SM_1 = SM_2$.

We will need to substitute in type schemes. Substituting in the presence of bound variables can be complicated. Two things can go wrong if we ignore the bound variables and just substitute on the body. We could end up substituting for the bound variables. An example of this is $(\forall\,\alpha.\,\alpha\,\texttt{list})[\texttt{int}/\alpha] \neq \forall\,\alpha.\,((\alpha\,\texttt{list})[\texttt{int}/\alpha]) = \forall\,\alpha.\,\texttt{int list}$, where we have clearly indicated what does *not* happen with a $\neq$ sign. Another problem is that a variable in the range of the substitution could be captured. An example of this is $(\forall\,\alpha.\,\alpha * \beta)[\alpha\,\texttt{list}/\beta] \neq \forall\,\alpha.\,((\alpha * \beta)[\alpha\,\texttt{list}/\beta]) = \forall\,\alpha.\,\alpha * \alpha\,\texttt{list}$.

In order to perform a substitution S on type scheme $\forall\,\vec{\alpha}.\,\tau$ correctly, first we alpha-convert the type scheme to $\forall\,\vec{\beta}.\,\tau[\vec{\beta}/\vec{\alpha}]$, where $\vec{\beta}$ are chosen to be different from the domain and (the type variables in the) range of the substitution and from the free type variables in $\forall\,\vec{\alpha}.\,\tau$. Then we just perform the substitution on the body of the type scheme. Thus $(\forall\,\vec{\alpha}.\,\tau)[S] = (\forall\,\vec{\beta}.\,(\tau[\vec{\beta}/\vec{\alpha}]))[S] = \forall\,\vec{\beta}.\,((\tau[\vec{\beta}/\vec{\alpha}])[S])$. For example, $\forall\,\alpha.\,\alpha * \beta$ is alpha-equivalent to $\forall\,\gamma.\,\gamma * \beta$, and $(\forall\,\alpha.\,\alpha * \beta)[\alpha\,\texttt{list}/\beta] = \forall\,\gamma.\,\gamma * \alpha\,\texttt{list}$. The function that performs substitutions on type schemes chooses new variable names deterministically based on the original names and the domain and range of the substitution, so if γ is the renaming chosen for α, the equality above is in fact real equality, not alpha-equivalence.

8.4 Typechecking under Substitution

In addition to being an important lemma needed in an eventual proof of type preservation, typechecking under substitution is an important theorem in itself. It demonstrates that the SML formulation of polymorphism makes sense: it shows that the type variables that are quantified by the Clos operation in Rule 17 can indeed be generalized. It accomplishes this by showing that the phrase can elaborate to any type that is the result of applying a substitution whose domain is those variables.

The formal statement of the theorem is:

Theorem 8.2 (Typechecking under Substitution 1 (for Match)) *Say* $C \vdash$ *match* : τ. *Then for any substitution* S *such that* Dom $S \bigcap$ free_tyvars$C = \emptyset$, $C \vdash$ strip_tags *match* : $\tau[S]$.

Note that the statement of the theorem above is for Match phrases. As the theorem must be proved by induction over the typing rules, which involve all the phrases in the

grammar, similar properties must be defined for all phrases. Declarations result in environments. These environments might have free type variables, as in the result of the declaration in the let in the phrase

```
val f = fn x => let val y = [x, x] in (y, y) end
```

Thus we need to perform substitution on the environments that result from elaborating declarations. This will lead to the complications mentioned in Section 8.3, including alpha-conversion and renaming in order to avoid capture of free type variables. Since local and let statements add environments into contexts, we will have to do substitution on contexts. Thus in order to do the induction, we will need to prove a stronger theorem (we phrase it for expressions this time):

Theorem 8.3 (Typechecking under Substitution 2) *Say* $C \vdash exp : \tau$. *Then for any substitution* S, $C[S] \vdash \text{strip\_tags } exp : \tau[S]$.

The proof of most of the cases for this theorem were relatively straightforward, if tedious to prove. Many lemmas had to be proved. For example, consider the case for a variable *longvar*. We have that $C \vdash longvar : \tau$ and we want to show that $C[S] \vdash longvar : \tau[S]$ (since strip_tags has no effect on a variable). From the former statement we conclude that there exists a σ such that $C(longvar) = \sigma$ and $\sigma \succ \tau$. To prove the latter statement, we need that there exists a σ' such that $C[S](longvar) = \sigma'$ and $\sigma' \succ \tau[S]$. We let σ' equal $\sigma[S]$, and then we must prove two lemmas, showing that $C[S](longvar) = \sigma[S]$ and $\sigma[S] \succ \tau[S]$. The proof of the first lemma is straightforward. The second lemma is not easy to prove. It involves reasoning about renaming of bound type variables, exchanging the order of substitutions, and substituting in substitutions.

The complicated case is Rule 17, for value declarations. It is not surprising that this is the hard case, since it is where type schemes are created. It is also where Tofte had the most difficulty in proving his Lemma 4.2, the equivalent theorem for his language.

We are given that

$$C \vdash \texttt{val}\ valbind : VE' \text{ in Env}, SM \tag{8.4}$$

and must show that

$$C[S] \vdash \text{strip\_tags } (\texttt{val}\ valbind) : VE'[S] \text{ in Env}, SM \tag{8.5}$$

By the inversion of Rule 17 applied to 8.4, we get that $C \vdash valbind : VE, SM$ and alpha_equiv $(\text{Clos}_{C, valbind} VE)\, VE'$. In order to show Equation 8.5, we need to show that

there exists a variable environment VE_2 such that $C[S] \vdash valbind' : VE_2, SM$ and that alpha_equiv $(\mathrm{Clos}_{C[S],valbind'}\, VE_2)\, VE'[S]$ where $valbind'$ is strip_tags $valbind$.

The task then is to choose VE_2 so that its closure with respect to $valbind'$ and $C[S]$ is alpha-equivalent[7] to $VE'[S]$. It would be convenient if we could set VE_2 to $VE[S]$. But this will not work. One problem is that we could substitute for type variables that should be generalized. An example will illustrate the problem. Say $VE(i) = \alpha * \texttt{int}$ for some identifier i, $VE'(i) = \sigma$, α is in the domain of S, and $S(\alpha) = \tau$. Say α is not free in C, so it will be generalized when $\alpha * \texttt{int}$ is closed with respect to $valbind$ and C. Thus σ will be alpha-equivalent to $\forall\beta.\beta * \texttt{int}$. Since there are no free type variables in this, $\sigma[S]$ will also be $\forall\beta.\beta * \texttt{int}$. However, $(VE[S])(i) = (\alpha * \texttt{int})[S] = \tau * \texttt{int}$. Unless τ happens to be another type variable that is not free in $C[S]$, the closure of this with respect to $valbind'$ and $C[S]$ will not be alpha-equivalent to $\forall\beta.\beta * \texttt{int}$. The problem here was that we substituted for a type variable that will be generalized in the closure.

Another thing that could go wrong is that clashes between the variables that should be generalized and the variables in the range of the substitution can prevent can prevent the variables from being generalized. Now, say $VE(i) = \alpha * \beta$, that $S = \alpha\,\texttt{list}/\beta$, that α is not free in C, but β is free in C. Again, let $\sigma = VE'(i)$. This time, σ is alpha-equivalent to $\forall\gamma.\gamma * \beta$. Thus $\sigma[S] = \forall\gamma.\gamma * (\alpha\,\texttt{list})$. However, $(\alpha * \beta)[S] = \alpha * (\alpha\,\texttt{list})$. Furthermore, when this is closed with respect to $C[S]$, α will not be generalized: the β occurring free in C will be substituted for, leaving α free in $C[S]$. This is a problem, since $\alpha * (\alpha\,\texttt{list})$ is certainly not equivalent to $\forall\gamma.\gamma * (\alpha\,\texttt{list})$.

The above two paragraphs demonstrate why we cannot set VE_2 equal to $VE[S]$. What we need to do is rename the variables in VE that will be generalized so they do not conflict with the domain or range of the substitution, nor with the free variables in the context or the variables in the types in the range of VE. We also need to deal with the effect of the substitution S on the free variables in the closure of the type. There is another complication as well: if the expression that gives a type to an identifier is expansive (see page 121), then no type variables will be generalized. For such identifiers, then we do not have to worry about renaming or about which type variables will be free in the resulting substituted type scheme, and we can just apply the substitution to the type. Thus we will define a function subs_right_vars such that the closure of subs_right_vars VE with respect

[7]Note that we explicitly include alpha-equivalence in Rule 17. The Definition assumes that all type schemes are equal up to alpha-conversion, but we found that this is the only place where alpha-equivalence is needed. This is explained on page 98.

to *valbind'* and $C[S]$ is alpha-equivalent to $VE'[S]$. The function will take more arguments than just *VE*, but we will see exactly what arguments are needed as we describe it.

Let us take a first cut at determining how to define subs_right_vars. Given an identifier i, what should be done with the type (call it τ) associated with it in *VE*? If the expression giving a type to the identifier is expansive, then (subs_right_vars *VE*)(i) should be just $(VE(i))[S]$, since none of its type variables will be generalized. Note that in order to determine whether or not the expression is expansive, we need to include *valbind'* as an argument to subs_right_vars.[8] Also, it is clear that S must be an argument as well.

If the expression is not expansive, then we must rename the type variables that will be bound in order to avoid clashes with the substitution. First, we find the type variables that will be generalized. These are the type variables in the type that are not free in C, call them $\vec{\alpha}$.[9] Then $VE'(i)$ is $\forall\,\vec{\alpha}.\,\tau$. We choose new type variables that are different from all the type variables in τ, the domain and range of S, and the free type variables in C. Call these new type variables $\vec{\beta}$.

We need to avoid substituting for $\vec{\alpha}$, the variables that will be generalized. Let S_2 be S with the type variables $\vec{\alpha}$ removed from the domain. To be precise, let $DS = \mathrm{Dom}\, S$. Then $S_2 = S|_{(DS-\vec{\alpha})}$, that is, S restricted to the type variables not in $\vec{\alpha}$. The result of this restriction is that $\vec{\alpha}$ will not be substituted for.

Then subs_right_vars should, for τ, return $\tau[\vec{\beta}/\vec{\alpha}, S_2]$, that is, we simultaneously rename the type variables that would be generalized and substitute for the type variables that would not be generalized. Now let us see why this works. We claim that when this type is closed with respect to *valbind'* and $C[S]$, the type variables that are generalized are $\vec{\beta}$. First, $\vec{\beta}$ are among the type variables that are generalized, since they are in $\tau[\vec{\beta}/\vec{\alpha}, S_2]$ and are not free in $C[S]$ (they were chosen to be not free in C, and they are not in the range of S). Furthermore, they are the only such type variables. Let us take an inventory of the variables in $\tau[\vec{\beta}/\vec{\alpha}, S_2]$. We have already accounted for $\vec{\beta}$. There are the variables that were in τ that were unaffected by the substitutions: they are not among the $\vec{\alpha}$, and are not in the domain of S. These variables will not be generalized because they will also be free in $C[S]$ (they are not among the $\vec{\alpha}$ because they are free in C). Then there are the type variables in the range of S_2. If a type variable γ in the range of S_2 shows up in

[8] Note that stripping the type tags from an expression does not change whether or not it is expansive, see the definition of expansive. Thus whether or not the expression is looked up in *valbind* or *valbind'* does not matter.

[9] Thus we see that one argument to subs_right_vars must be C, or at least the free type variables in C.

$\tau[\vec{\beta}/\vec{\alpha}, S_2]$, this is because there was a type variable δ that was in τ and was free in C, and $S(\delta)$ contained γ. But since δ was free in C, the substitution will substitute for δ in C, so γ is free in $C[S]$, so γ will not be generalized. Thus $\vec{\beta}$ are type variables that are generalized.

Thus, the closure of $\tau[\vec{\beta}/\vec{\alpha}, S_2]$ with respect to *valbind'* and $C[S]$ will be $\forall\vec{\beta}.(\tau[\vec{\beta}/\vec{\alpha}, S_2])$. We need that this typescheme is alpha-equivalent to $(VE'[S])(i)$, which is alpha-equivalent to $(\forall\vec{\alpha}.\tau)[S]$. Here, S acts only on the variables free in $\forall\vec{\alpha}.\tau$. That is, it does not substitute for $\vec{\alpha}$, and so its effect is that of S_2, which is S with $\vec{\alpha}$ removed from the domain. In addition, the type variables may have to be renamed in order to avoid clashes with the substitution. This exactly what we have done to get $\forall\vec{\beta}.(\tau[\vec{\beta}/\vec{\alpha}, S_2])$, and so this is the correct value of subs_right_vars if τ is non-expansive.

We have shown that $\tau[\vec{\beta}/\vec{\alpha}, S_2]$ will work, that is, when it is generalized with respect to *valbind'* and $C[S]$, it will be alpha-equivalent to $VE'(i)$. But there is something else to be shown, namely, that $C[S] \vdash$ *valbind'* $: VE_2, SM$. The problem with the description above is that new names for the bound variables are chosen independently for each type in VE. Consider the following program

```
let val (x as y) = fn z => z in x end
```
(8.6)

In the elaboration (against the initial context) of the value binding in the let statement, the resulting variable environment is $\{x \mapsto \alpha \to \alpha, y \mapsto \alpha \to \alpha\}$ for some α. The type variable α would be generalized, since it is not free in the context. In order to prove typechecking under substitution here, we would have to rename α to avoid clashes with the substitution. However, in the definition of subs_right_vars above, nothing constrains α to be replaced by the same type variable in the two types in the range of the variable environment (the new type variables are chosen independently for each type). Say that two different variables, β and γ are chosen. We would like to show that

$$C[S] \vdash \texttt{(x as y) = fn z => z} : \{x \mapsto \beta \to \beta, y \mapsto \gamma \to \gamma\}, SM$$

(where $SM = \{x \mapsto \text{v\_stat}, y \mapsto \text{v\_stat}\}$), but according to Rule 46 for elaborating layered patterns, x and y must have the same type, so the elaboration fails.

Thus the renaming must be done uniformly over the variable environment. Let $\vec{\alpha}_2$ be all the variables that could be generalized, that is the type variables in the range of VE minus the free type variables in C. Let S' be S with the type variables $\vec{\alpha}_2$ removed from

the domain. That is, let $DS = \mathrm{Dom}\, S$ and $S' = S|_{(DS - \vec{\alpha}_2)}$. Let $\vec{\beta}_2$ be the renaming, that is, new type variables that are not in the domain or range of S, in the types in the range of VE, nor free in C. Let $S_3 = \vec{\beta}_2/\vec{\alpha}_2, S'$. Then subs_right_vars returns, for each τ in the range of VE, $\tau[S]$ if the expression is expansive, or $\tau[S_3]$ if the expression is non-expansive. The same reasoning as above shows that the closure of $\tau[S_3]$ with respect to *valbind'* and $C[S]$ is alpha-equivalent to S applied to the closure of τ with respect to *valbind* and C: the extra variables in the domain of the substitution do not interfere with the effect of the substitution. The arguments needed by subs_right_vars are: the free type variables in C, the type variables in VE, *valbind*, and, of course, VE.

The complicated process described above is why we need alpha-equivalence in Rule 17. In the process of proving typechecking under substitution, when we perform a substitution on VE', the bound type variables may be renamed differently for different type schemes. If we were to show that $\mathrm{Clos}_{C[S],valbind'}\, VE_2 = VE'[S]$ (where the = is exact equality, not equality up to alpha-conversion), then VE_2 would have to contain those renamed variables. However, as we saw in the program above (Equation 8.6), the renaming of variables that will be bound must be done uniformly in order to ensure that $C[S] \vdash valbind' : VE_2, SM$. Hence the need for alpha-equivalence in Rule 17.

The only remaining detail in the proof for this case is: by what mechanism do we prove that $C[S] \vdash valbind' : \mathrm{subs\_right\_vars}\, S\, V_C\, V_V\, valbind'\, VE, SM$ (where V_C is the set of type variables free in the context and V_V is the set of type variables in variable environment)? We prove it by induction along with the main substitution property: it is another fact to prove for value bindings. There is a complication here. The value binding that we are elaborating may be just a part of the entire value binding (see Rule 26), and thus the statement of this property must take account of the fact that there are two value bindings being discussed, one of which (the one that is being elaborated) is a part of the other (which is used to determine whether or not an expression is expansive and is the source of V_V). The formal statement for this property is:

$$
\begin{aligned}
&\forall C\ VE\ valbind\ valbind_2\ SM\ S\ V_V. \\
&\quad C \vdash valbind : VE, SM \wedge \mathrm{proper\_valbind}\ valbind_2 \wedge \\
&\quad \mathrm{subvalbind}\ valbind_2\ valbind \wedge \mathrm{tyvars\_in\_varenv}\ VE \subseteq V_V \Rightarrow \\
&\quad C[S] \vdash \mathrm{strip\_tags}\ valbind : \\
&\qquad \mathrm{subs\_right\_vars}\ S\ (\mathrm{free\_tyvars}\, C)\ V_V\ valbind_2\ VE, SM
\end{aligned}
\tag{8.7}
$$

The predicate proper_valbind checks that all the syntactic restrictions in Section 2.9 of the

Definition hold of the value binding, and subvalbind $valbind_2\ valbind$ checks that $valbind$ is a subpart of $valbind_2$.

We will summarize briefly the cases for proving this property. The case for Rule 26 is that if we are given

$$C \vdash pat \texttt{ = } exp\ \langle \texttt{and}\ valbind \rangle : VE\,\langle + VE' \rangle, SM\,\langle + SM' \rangle$$

and all the other hypotheses of Equation 8.7, we must show

$$\begin{aligned} C[S] \vdash \text{strip\_tags}\,(pat \texttt{ = } exp\ \langle \texttt{and}\ valbind \rangle) : \\ VE_2\,\langle + VE_2{}' \rangle, SM\,\langle + SM' \rangle \end{aligned}$$

where we use the shorthand $VE_2 = \text{subs\_right\_vars}\ S\ (\text{free\_tyvars}\,C)\ V_V\ valbind_2\,VE$ and $VE_2{}' = \text{subs\_right\_vars}\ S\,(\text{free\_tyvars}\,C)\,V_V\ valbind_2\,VE'$.

The induction hypothesis applied to $valbind$ takes care of the possible (`and` $valbind$) in the phrase. If exp is expansive, then VE_2 is just $VE[S]$ (because subs_right_vars will find that exp is the expression associated with the variables in the pattern, and so will choose the same substitution for all types in the range of the variable environment). The theorem for typechecking under substitution of patterns gives $C[S] \vdash \text{strip\_tags}\ pat : (VE[S], SM, \tau[S])$, and the induction hypothesis (for typechecking under substitution) for expressions gives $C[S] \vdash \text{strip\_tags}\ exp : \tau[S]$, so we are done. (Note that we need that the value binding is proper in order for this to be the case—if the value binding were allowed to bind variables more than once, exp might not be the unique expression giving a type and value to the variables in pat.)

If exp is not expansive, then $VE_2 = VE[S_3]$, where S_3 is a substitution that renames the type variables that will be bound and substitutes for the other type variables in the domain of S, as described above. Then by the theorem for typechecking under substitution of patterns, we get $C[S_3] \vdash \text{strip\_tags}\ pat : (VE[S_3], SM, \tau[S_3])$, and by the induction hypothesis (for typechecking under substitution) for expressions we get $C[S_3] \vdash \text{strip\_tags}\ exp : \tau[S_3]$. Now, $C[S_3]$ is alpha-equivalent to $C[S]$: it is not equal since different variables may have been chosen for the renaming of bound type variables. Thus we needed to prove additional lemmas, stating that for expressions and patterns, the results of elaboration are the same in contexts that are alpha-equivalent to each other. Then we can conclude that $C[S] \vdash \text{strip\_tags}\ pat : (VE[S_3], SM, \tau[S_3])$ and $C[S] \vdash \text{strip\_tags}\ exp : \tau[S_3]$, and this, combined with the induction hypothesis applied to the smaller value binding, proves this case.

$\forall\, C\ \mathit{atexp}\ \tau.\ C \vdash \mathit{atexp} : \tau \Rightarrow$
$\quad(\forall\, S.$
$\qquad \text{proper\_atexp}\ \mathit{atexp} \wedge \text{proper\_context}\ C \wedge \text{proper\_subs}\ S \Rightarrow$
$\qquad C[S] \vdash \text{strip\_tags}\ \mathit{atexp} : \tau[S]) \wedge$
$\forall\, C\ \mathit{exprow}\ \varrho.\ C \vdash \mathit{exprow} : \varrho \Rightarrow$
$\quad(\forall\, S.$
$\qquad \text{proper\_exprow}\ \mathit{exprow} \wedge \text{proper\_context}\ C \wedge \text{proper\_subs}\ S \Rightarrow$
$\qquad C[S] \vdash \text{strip\_tags}\ \mathit{exprow} : \varrho[S]) \wedge$
$\forall\, C\ \mathit{exp}\ \tau.\ C \vdash \mathit{exp} : \tau \Rightarrow$
$\quad(\forall\, S.$
$\qquad \text{proper\_exp}\ \mathit{exp} \wedge \text{proper\_context}\ C \wedge \text{proper\_subs}\ S \Rightarrow$
$\qquad C[S] \vdash \text{strip\_tags}\ \mathit{exp} : \tau[S]) \wedge$
$\forall\, C\ \mathit{match}\ \tau.\ C \vdash \mathit{match} : \tau \Rightarrow$
$\quad(\forall\, S.$
$\qquad \text{proper\_match}\ \mathit{match} \wedge \text{proper\_context}\ C \wedge \text{proper\_subs}\ S \Rightarrow$
$\qquad C[S] \vdash \text{strip\_tags}\ \mathit{match} : \tau[S]) \wedge$
$\forall\, C\ \mathit{mrule}\ \tau.\ C \vdash \mathit{mrule} : \tau \Rightarrow$
$\quad(\forall\, S.$
$\qquad \text{proper\_mrule}\ \mathit{mrule} \wedge \text{proper\_context}\ C \wedge \text{proper\_subs}\ S \Rightarrow$
$\qquad C[S] \vdash \text{strip\_tags}\ \mathit{mrule} : \tau[S]) \wedge$
$\forall\, C\ \mathit{dec}\ E\ \mathit{tns}.\ C \vdash \mathit{dec} : E, \mathit{tns} \Rightarrow$
$\quad(\forall\, S.$
$\qquad \text{proper\_dec}\ \mathit{dec} \wedge \text{proper\_context}\ C \wedge \text{proper\_subs}\ S \Rightarrow$
$\qquad C[S] \vdash \text{strip\_tags}\ \mathit{dec} : E[S], \mathit{tns}) \wedge$
$\forall\, C\ \mathit{valbind}\ \mathit{VE}\ \mathit{SM}.\ C \vdash \mathit{valbind} : \mathit{VE}, \mathit{SM} \Rightarrow$
$\quad\text{proper\_valbind}\ \mathit{valbind} \wedge \text{proper\_context}\ C \Rightarrow$
$\quad(\forall\, S.\ \text{proper\_subs}\ S \Rightarrow$
$\qquad C[S] \vdash \text{strip\_tags}\ \mathit{dec} : \mathit{VE}[S], \mathit{SM}) \wedge$
$\quad(\forall\, S\ \mathit{valbind}_2\ V_V.\ \text{proper\_subs}\ S \wedge \text{proper\_valbind}\ \mathit{valbind}_2 \wedge$
$\qquad \text{subvalbind}\ \mathit{valbind}_2\ \mathit{valbind} \wedge \text{tyvars\_in\_varenv}\ \mathit{VE} \subseteq V_V \Rightarrow$
$\qquad C[S] \vdash \text{strip\_tags}\ \mathit{valbind} :$
$\qquad\ \text{subs\_right\_vars}\ S\ (\text{free\_tyvars}\, C)\ V_V\ \mathit{valbind}_2\ \mathit{VE}, \mathit{SM})$

Table 8.5: Full Statement of Typechecking Under Substitution

The case for Rule 27 is simpler. Since each expression in a recursive value binding must be a function expression, which is non-expansive, the resulting variable environment returned by subs_right_vars is $VE[S_3]$, where S_3 is, as above, the substitution that renames bound variables and substitutes free variables. Thus by the inductive hypothesis (for typechecking under substitution) applied to value bindings, we get that $C[S_3] \vdash \text{strip\_tags } \mathit{valbind} : VE[S_3], SM$ and as before, since $C[S_3]$ is alpha-equivalent to $C[S]$, we get that $C[S] \vdash \text{strip\_tags } \mathit{valbind} : VE[S_3], SM$.

Finally, note the presence of proper_valbind in the property in Equation 8.7. In order for us to use this property to prove the case for Rule 17 for the proof of the main theorem (for typechecking under substitution), we need to know that the value binding is proper. Because the proof is by induction over the terms, we thus need to know that all the phrases are proper. This fact must be added to the hypotheses of Theorem 8.3.

In order to clarify exactly what we proved, we state in full the theorem for typechecking under substitution, including the cases for all the phrases in the class with expressions and the extra clause for value bindings stating that value bindings can elaborate to the variable environment modified by subs_right_vars. This is given in Table 8.5. In this table, proper_subs checks that all the types in the range of the substitution are proper types. Keep in mind that this is not all we needed to prove: we needed to prove related properties for all the phrase classes, as well as multitudes of substitution lemmas.

8.5 Problems with the Formulation

The first problem is pointed out by Stefan Kahrs in [Kahrs93]. Consider the program:

```
let datatype A = C of bool -> int in
    fn (C x) => x true
end
let datatype B = C of int in
    C 5
end
```

In this program, the returned value of the first let statement is a function which is applied to the return value of the second let statement. According to the semantics for HOL-SML (given in Section A.2), and also according to the semantics for Core SML in the Definition,

this can elaborate, with the type name given to type `A` being the same as that given to `B`. In the semantics given in the Definition, evaluation will result in a run-time type error, since the constructor C for `A` will be considered the same as the constructor C for `B`. In the semantics for HOL-SML (given in Section A.3), the evaluation will result in the packet [`Match`], since the two constructors will be given different constructor names. Thus this problem does not break type soundness (since no run-time type error occurs). It does not even interfere with type preservation: as described in Section 8.1, a packet can have any type. However, it does something unintuitive: the function in the first let statement covers all the possibilities for type `A`, so its application should not result in [`Match`].

The fix for this problem, described in [Kahrs93], is to include among the hypotheses of Rule 6 the side condition: tynames $\tau \bigcap T = \emptyset$, where T is the set of type names generated by the declaration in the let. This ensures that local type names do not escape their intended scope. This fix was not included in the semantics presented here because if it were included, in order to prove typechecking under substitution, we would have to prove typechecking under renaming of local type names. This is because these local type names may have been reused in datatype definitions later, and thus the range of the substitution might contain these type names. The new side condition prevents the substituted type of the let expression from containing these type names. Thus we would have to prove that different type names could have been chosen as the local type names, and then the type names in the range of the substitution would not clash with the local type names. Not wanting to prove yet more theorems about substitution, we did not include this fix in our semantics.

The same motivation was the main reason we include a T field in the elaboration of declarations, as described in Section A.2. If we adopt the use of $\oplus$ as the Definition does, then type names in the range of substitutions end up in the TyNameSet, and then these type names cannot be used in local type declarations (they are considered to have been "used" already).

There is a similar problem with Rule 16, pointed out in [Kahrs94]. The use of a recursive function that takes a function as an argument can again result in a local data type escaping its expected scope, even though the type of the let expression using the local type declaration does not contain local type names. The fix suggested here is the use of $\oplus$ to prevent the use of type names in the arguments of the function as local type names. This fix was not included here for the same reason that the correction for Rule 6 was omitted:

it would require us to prove facts about the renaming of local type names. It is clear that in order to prove type soundness for SML, this proof would have to be made, and the fixes for Rules 6 and 16 would have to be included. It is interesting to note that although the more careful accounting of type names used by the T parameter in declarations eliminates the use of $\oplus$ elsewhere in the semantics, it seems that it would have to remain here.

An alternative (also mentioned in [Kahrs93]) to the suggestions above is to keep track of all type names generated, not just the ones currently in use, and to not try to reuse local type names. This fixes the problems with both Rules 6 and 16, as well as avoiding problems in case there are any more rules with subtle problems like Rule 16. A similar approach is taken, for example, with addresses in the dynamic semantics. This accounting of type names is the approach taken by all known implementations of SML. It introduces a form of state into the static semantics, thus complicating it. However, it prevents the problems associated with the escape of supposedly local type names, which is the root cause of the problems with Rules 6 and 16.

In [Tofte96], Tofte suggests keeping the original way of dealing with type names, that is, using $\oplus$ to add all type names used to the context, instead of keeping track of just those type names generated by datatype declarations as we do for HOL-SML. To fix Rule 6 he suggests adding side conditions tynames $\tau \subseteq (T \text{ of } C)$, which is also proposed by Kahrs as a possible solution. For Rule 16, the side condition proposed is tynames $VE \subseteq (T \text{ of } C)$. The problem with this solution is that it complicates the proof of typechecking under substitution by disallowing the substituted type of a let statement to contain new types names. One solution could be to alter the definition of substituting a context such that the type names in the range of the substitution are added to T of C. This was the approach we adopted at first, before we discovered that it would result in requiring a proof of substitutions under renaming of type names.

A more serious problem (in that a solution has not yet been formulated) is that constructor and exception names are not adequately dealt with in the semantics presented here or in the Definition. Consider the following program:

```
let exception hold_int of int in hold_int 0 end
```

The result of executing this program (call it P) is an ExVal that is the pairing of an exception name (call it en) with the integer 0. Type preservation for this program states that if $C \vdash P : \texttt{exn}$ and $E_d \vdash P \Downarrow (en, 0)$ and $ST \models E_d : C$ then $C, E_d, ST \models_v (en, 0) :$

$\emptyset$, `exn`. However, since the exception declaration was a local one, there is no way to check if the pairing of en with 0 makes sense: neither C nor E_d has any record of the exception *hold_int* or its exception name en. The problem can also occur with constructor names, even if the problems with Rules 6 and 16 mentioned above are resolved. The fix to Rule 6 prevents local type names from occurring in the type of let declarations, but constructor names can escape their local bindings with the help of exceptions:

```
let datatype foo1 = D
    exception my_ex of foo1 in
    my_ex D
end
```

This problem does not occur when exception or constructor names are used in functions, as in:

```
let exception hold_int of int in
    fn x => if x > 0 then x - 1 else raise hold_int x
end
```

Here, the closure will contain a dynamic environment giving a value for the exception constructor, and the definition of the val-type relation $\models_c$ ensures that there is a context that agrees with the environment, giving a type for the exception constructor.

Another problem is that if an exception or value constructor is reused, there remains no record of the name of the first use of the constructor in the dynamic semantics. Consider the following series of declarations:

```
datatype foo1 = A | B
val x = A
datatype foo2 = A | B
val y = x
```

The last declaration, for y, is elaborated in an environment that includes the results of the previous declarations. We would like to show that evaluation and elaboration of the last declaration results in static and dynamic environments such that the value of y in the dynamic environment has the type of y given in the static environment. The type of y is `foo1`. The value of y is a constructor name, but that constructor name is not in the

environment in which the declaration is evaluated—it was present after the declaration of type `foo1` only as the value of constructor A, but after the declaration of type `foo2`, it was overwritten by the mapping of A to the new constructor name.

A first cut at trying to solve these problems might be to have the static semantics generate constructor and exception names and thread environments containing information on these names though the elaboration much as the state is threaded through the dynamic semantics. Then, when we want to know the type of an exception name or constructor name, we just look up the name in the these environments. The problem with this is that local declarations will only be elaborated once, while they may be evaluated many times. Thus a straightforward generation of names would result in the names being chosen differing between the static and dynamic semantics.

We leave the job of formulating a viable solution for future work.

Chapter 9

Assessment

The work presented here demonstrates that it is possible to prove significant properties of programming languages given a definition that is both formal (given in mathematical terms) and not too complicated. Below we discuss why the work described here was useful (that is, what it tells us about SML and programming languages in general) and weight the benefits and drawbacks of using a theorem prover (namely, HOL).

9.1 What Have We Learned About SML?

One may wonder: why do we believe that the proofs accomplished here are useful? HOL-SML is different from that used in the Definition, and is different from any implementation of SML. What then do these proofs have to do with SML as in the Definition or as implemented?

The differences between HOL-SML and Core SML are discussed at length in Appendix A. Some of the changes complicate the semantics while others are simplifications. None of the changes are major: they preserve the key characterists of Core SML, that is, the ability to define new parameterized datatypes and polymorphic functions, the use of pattern matching in functions, functions as first-class values, and the presence of imperative features such as reference cells and exceptions,

Some of the changes serve to bring the Definition closer to what is done by implementations. An example is the distinguishing of the `ref` constructor from other constructors in the dynamic semantics, which we accomplish here with constructor names.

Other changes seem at first glance to be a departure not only from the Definition,

but also from implementations. An example of this is the use of the value restriction in determining which type variables to generalize in creating polymorphic types. The value restriction results in a simpler semantics than in the Definition, while at least one implementation, Standard ML of New Jersey (SML/NJ), goes the other way. SML/NJ uses a method called weak typing, which uses type variables with numbers in them to approximate how many times a function can be applied before a reference cell is generated. Wright, in [Wright93], shows that not only is the value restriction simple, but it is also useful: in a study of a large body of SML code, there are few instances of the use of value in a way that takes advantage of applicative/imperative type variables, much less weak typing. In even large programs (a few tens of thousands of lines), there were fewer than ten instances of polymorphic uses of expansive expressions. There has been talk of changing SML/NJ to this simplified semantics.[1]

Furthermore, Mads Tofte is working on a project to define a language called Simplified SML [Tofte96], which is a modification of SML that simplifies the parts of the language that cause the most problems both in the semantics and in implementations. Modifications to the Core include the use of the value restriction, inclusion of identifier status in the semantics (a change we also made), and clarification of the scope of explicit type variables.[2]

We also eliminated equality types from the language. We justify this because it is the single most complicated part of the definition of the Core. The notions associated with equality, such as equality attributes and whether or not semantic objects admit or respect equality, are confusing and present great challenges for an automated theorem prover. Furthermore, the system as presented in the Definition is undesirable for two reasons. First, it is insufficient in that it misses many legitimate application of equality (see [GGM93]). Second, the resulting algorithm automatically defined for a datatype may not be what the user requires, so she would be better off writing her own. Because of all this, we decided that preserving equality types in HOL-SML was not worth the additional complexity it presented.

Thus, although HOL-SML is a departure from the semantics in the Definition, future versions of ML may look more like more like HOL-SML than the Definition. One could argue that future versions of ML *should* look more like HOL-SML, in the sense that the semantics should not contain features that are too hard to describe. With the increasing

[1] This was related to the author in conversation with Andrew Appel.

[2] Tofte's change regarding the scope of explicit types variables is less drastic than our elimination of the special rules, but it also recognizes that these rules are confusing.

importance of verification of programs, if one hopes to prove properties of a language and programs written in a language, the more complicated the semantics, the easier it is for a programmer to write subtly incorrect programs in the language, and the harder it is to implement and use verification tools for the language.

Another question about the validity of this work is that the proofs were accomplished using an encoding of the language into HOL. The version of HOL we use, HOL90, is written in a version of SML, Standard ML of New Jersey (SML/NJ). Thus it is conceivable that a mistake in the Definition (a property that we expect to be satisfied but does not hold of the language) could result in a mistake in SML/NJ, which would result in a mistake in HOL, which would lead to a mistake in the proof of this property. Thus we would have a proof of the property when in fact the property does not hold of the language.

There are several reasons why this apparent circularity is not a problem. One is that the chances that a problem in the Definition could work its way through an implementation and a theorem prover and an encoding of HOL-SML into a false proof of the property in question, without causing problems that would have been caught somewhere along the way, is extremely unlikely. The semantics of HOL-SML is ultimately represented in SML/NJ, but a given aspect of the semantics of HOL-SML has no direct connection to corresponding aspect of SML/NJ. This is because the semantics is encoded in an abstract way—the definitions for the grammar and semantic objects are the results of a mutually recursive HOL type definition, and the evaluation and elaboration relations are defined using a mutually recursive relations definition package. This mechanism is so different from how the language is implemented in SML/NJ that the encoding is largely independent of the specific behavior of SML/NJ.

Another reason lies in the security of HOL. As described in Chapter 4, HOL has a simple core, based on a few inference rules, axioms, and definitional principles. HOL is set up so that theorems can be proved only from these basic pieces and other already proved theorems. Thus, when traced back, all theorems are based on this core. This part of HOL has been coded carefully and exercised a great deal. It is exceptionally unlikely that a mistake in SML/NJ could cause a mistake in HOL resulting in the proof of theorems that are not true, for HOL-SML or any other encoded problem. Mistakes in SML/NJ are far more likely to result in core dumps.

A final reason why the fact that HOL90 is written in SML/NJ should not result in incorrect theorems is that a byproduct of a proof with a goal-directed theorem prover like

HOL is a minute inspection of every aspect of the semantics involved in the proof. HOL does not go off and prove the theorem on its own, coming back after a certain amount of time with a proof or disproof. It must be lead carefully to a proof. Thus it is the user who actually proves the theorem; HOL's main task is to organize the proof and prevent mistakes, and to automate some special-purpose tasks (such as defining types). This guidance by the user means that the user knows, in detail, how each theorem is proved. A false theorem would appear suspicious and would likely be caught by the user. In fact, this close inspection and contemplation of the problem, resulting from the attempt to prove facts about it, is part of what makes verification with a theorem prover so useful for finding errors.

We will point out two instances where close inspection of the rules brought to our attention details in the semantics requiring changes in the rules. One is that although it is pointed out in Section 4.5 of the Definition that type schemes will be considered equal if they are alpha-equivalent,[3] we found only one place (Rule 17) where alpha-equivalence is needed (this is described on page 98). Since our type schemes are encoded as a list of type variables and a type, it is much more convenient to use HOL's equality (which requires that the prefix contain the same type variables in the same order, and that the body is the same) rather than inserting many statements of alpha-equivalence. Thus we put alpha-equivalence explicitly in the rule only where it is needed, and assume exact equality elsewhere.

Another difference is in the encoding of Rule 27. As noted on page 46, our encoding of semantic objects uses a theory of finite maps, and we needed to ensure that the finite maps are proper, that is, their domains are sorted. Thus we have defined a collection of predicates, proper_*object* for each semantic object *object*, that constrain the structures of the objects, including that all finite maps are proper. Other constraints are included as well, such as, for ConsTypes, that the number of types combined with a type name matches the arity of type name.[4] These properness constraints are present in almost every theorem we proved about the evaluation and elaboration of HOL-SML. We proved that if we start with proper semantic objects (contexts, states, environments), then the results

[3]The Definition also states that type schemes are equivalent if one can be obtained from another by deleting type variables from the prefix that do not appear in the body. However, we noted that the semantics only generates type schemes such that all variables in the prefix occur in the body, and so we included this in our definition for proper type schemes.

[4]We have also defined predicates proper_*phrase* for each HOL-SML syntax phrase *phrase*, that check that the phrases satisfy the syntactic restrictions in Section 2.9 of the Definition.

(such as values, types, and environments) are also proper objects. For Rule 27, we found that due to the recursive nature of the rule, elaborations resulting in improper types could result. For example, the elaboration of the declaration

```
val rec f = fn x => g x and g = fn y => f y
```

can result in a variable environment in which f and g are mapped to the type `[]list`, where the type name associated with the list constructor has not been given any arguments to indicate what the type of the list items should be.

Thus in our encoding, we included in Rule 27 a side condition stating that all the types in the range of VE were proper. We did not include this side condition in the rule written on page 126. This was because it was a result of the peculiarities of our encoding: the properties checked in the side condition are either inherent (that the number of types in a ConsType matches the arity of the type name) or irrelevant (that the labels in a finite map are sorted) in the definition of Type in the Definition.

9.2 Was HOL Useful?

A prominent feature of this work was that it was accomplished with a theorem prover, HOL. We did have a good idea of what would be needed to prove the theorems before starting the proofs with HOL. We knew the general structure of the theorem, that is, what forms of induction we would use, and what types of lemmas we would need to prove. However, we did not attempt to do any proofs on paper before beginning the proofs in HOL. Partly this was because the task was so large that the idea of writing it all down was frightening. There seemed to be no use in simply selecting a few cases and proving them: something similar to this had been done by others, especially in [Tofte90]. Partly, we needed the help of the HOL's goal directed theorem proving in order to see exactly what needed to be proved: some of the rules were so complex that it was easier to let HOL set up the induction to determine exactly what needed proving than to do it by hand.

Part of the goal of this work was to determine if theorem provers are useful in proving properties of large programming languages. Thus, a legitimate question to ask is: was HOL useful in this situation? That is, was it more of a hindrance or a help?

Some background on the time this project took will be useful. It took two months to make the first cut at the mutually recursive types definition package, to encode the

syntax and semantic objects for the dynamic semantics, and to encode the evaluation relations (but not to create a general mutually recursive relations definition package). Then about one month was spent adding functionality to the types definition package and creating a mutually recursive relations definition package. About three months was spent proving inversion theorems and determinism for evaluation. Less than one month was spent encoding the grammar, semantic objects, and rules for elaboration. About one year was spent relating the static and dynamic semantics, including defining the val-type relations, proving that evaluation and elaboration resulted in proper semantic objects, proving type preservation for all phrase classes except for that including the expressions, and proving typechecking under substitution.

A great deal of time was spent on this project. We believe that without the help of the theorem prover, the task would not have been accomplished at all. Although there are drawbacks in using HOL, its ability to check that definitions are sensible and used consistently and the ability to modify proof scripts to reprove theorems when definitions have been changed more than made up for the disadvantages its use imposed.

We have already mentioned (page 66) one of the drawbacks in using HOL: it demands complete proofs of all lemmas. Some of these lemmas seemed to be obviously true, but were annoyingly complicated to prove.

One of the most annoying "obvious" lemmas was: say we are given a collection of type variables $\vec{\alpha}$ that are in a type τ. Say we choose new names of the type variables, $\vec{\beta}$. Then it should be the case that $\tau[\vec{\beta}/\vec{\alpha}]$, which is the result of replacing the old type variables with the new ones, should contain all the variables in $\vec{\beta}$. We needed to prove this to show that the result of renaming the bound variables of a proper type scheme was another proper type scheme. (As mentioned in a footnote at the bottom of page 109, the type variables in the prefix of a proper type scheme must be contained in the body.) Proving this lemma was moderately involved, since substitution is defined structurally, and the subparts of τ do not contain all the variables in $\vec{\alpha}$. Thus in doing the proof by induction, we had to restrict the substitution to the type variables in the subparts, prove that the result of substituting the subparts with the restricted substitution was the same as the result of substituting with the original substitution, use the induction hypothesis to conclude that the range of the restricted substitutions was contained in the substituted subparts, and then show that the type variables in the substituted type, which is the union of the type variables in the substituted subparts, is the same as the range of the original substitution.

On the other hand, this demand for detailed proof can be seen as an advantage of HOL as well: sometimes "obvious" lemmas turn out to be false. It was obvious that the substitution $\vec{\beta}/\vec{\alpha}, S_2$, described on page 97, would suffice as the definition of the effect of subs_right_vars on a type associated with a non-expansive expression. Of course we could choose the new variables ($\vec{\beta}$ here) differently for each type in the variable environment: after all, these are the variables that are going to be generalized, so there will be no connection between the type variables in the different types. However, in doing the proof, we discovered that type variables resulting from the elaboration of one *pat* = *exp* pair must be renamed the same way, as demonstrated by the program in Equation 8.6. Thus the fact that HOL forced us to thoroughly prove all facts prevented an incorrect definition of subs_right_vars, which might have resulted in an incorrect proof.

Another difficulty with using HOL was that there were so many lemmas of all descriptions to be proved that it was difficult to keep track of which ones had and had not been proved. We gave long descriptive names to all theorems, yet after a few months of steady work, it was impossible to remember the names of the theorems. We resorted to using emacs to search the files for the statements of the theorems. Sometimes, when we looked for theorems using a slightly different phrasing than that used in the actual theorem, we did not find them, and we ended up proving a few theorems over again. These theorems were generally small (usually little lemmas about sets) and thus not much time was wasted. This problem is certainly not unique to goal-directed theorem provers. It probably would have been an even worse problem for a paper proof. Since with a more automated theorem prover like Boyer-Moore, the proof has to be worked out in advance in order to lead the prover to a solution, there would be just as much a problem there.

The converse of this problem, having two theorems with the same name, occurred once or twice as well, when we neglected to check if a theorem with a certain name already existed before giving that name to another theorem. This was not too hard to fix, as it was easy to determine from the context which theorem was needed.

The problems with keeping track of the names of the theorems seems to point to the need for a better way of organizing proofs. A great deal of attention has been devoted to structured programming, that is, how to organize a large program so that it can be broken down into independent steps that can be accomplished on their own. HOL theories help in structuring the proofs, since they gather together related definitions and proved theorems, but this is not useful for developing large proofs in one of the theories. It seems that there

is more work to be done in structuring large proofs.

Another difficulty with using HOL, mentioned in Sections 5.1 and 5.3, was that before we could encode the semantics, we had to create packages to allow us to automate definitions of mutually recursive types and relations. Even when these definitional packages were up and running, we could not encode our definitions in the most straightforward way. The best example of this is in Section 8.1, where, because we could not universally quantify over premises, we had to include a set of type variables in the val-type relations in order to check that these type variables could be substituted for, and (as described on page 87) we had to create auxiliary relations to check that semantic objects defined with finite maps match up. Thus our relations were not defined in the most intuitive way; however, the resulting relations had the same effect as the intuitive versions.

If we were doing a paper proof, we could define our relations however we wanted. However, using HOL's definitional packages has certain benefits, even if does force us to make definitions in a different way. One benefit is HOL's typechecker. It took several iterations before we decided on final definitions for the val-type relations, and the HOL types of the relations changed as we added more and more arguments. As we revised the definition, the typechecker politely informed us when we had neglected to update a rule to reflect a new argument. This forced us to consider carefully how each new argument should be dealt with by each rule and thus resulted in a definition that was more likely to be correct than one written on paper. Another benefit of using HOL's definitions is the automatic proof of useful theorems, like the induction and inversion theorems. It is not too difficult to see from the rules defining the relations what these theorems should be, but sometimes it can help a user to see them clearly stated.

Another drawback to using HOL, as mentioned in Section 4.1 is that the proof scripts are unreadable. They can, however, be understood by replaying them, and thus the proofs can be described in a clear fashion (evidenced, we hope, by the descriptions of proofs in Sections 6.2 and 8.4).

One great benefit of HOL is that it allows one to keep track of exactly what conditions are needed for a property to hold. For example, on page 99, we point out that we needed the value binding to be proper (that is, to satisfy the syntactic restrictions of Section 2.9 of the Definition) in order to prove that a value binding (stripped of its type tags) elaborates to a variable binding modified by subs_right_vars. In order for this hypothesis to be available to this smaller proof (which is proved along with typechecking under substitution), the

properness of the expressions must be passed around as a condition in the theorem (see Table 8.5). Thus it became clear that a expression must satisfy the syntactic restrictions in order for typechecking under substitution to hold of it. In writing such a proof on paper, it is easy to lose track of assumptions we had to make for proofs of lemmas to go through.

By far the greatest advantage of using HOL was the ability to change definitions and rapidly rerun proofs on the modified definitions. We started off with Core SML as modified by the simplest of Stefan Kahrs [Kahrs93, Kahrs94] suggestions for fixing the problems that would have an effect on type preservation. In the course of proving the theorems described here, we were constantly changing the definitions, until we ended up with the semantics described here as HOL-SML. Each change had a definite purpose. For example, status maps were used in order to be able to state properties about proper contexts and to be able to use there properties to conclude facts about identifiers looked up in the context. After each revision of the semantics, we needed to go back and reprove the theorems relating to those definitions.

With paper proofs, this would have been a daunting task. We would have had to examine closely each proof to see if it used the definition and to determine exactly what change, if any, the new definition made in the proof. Most likely, only a superficial glance would be given to make sure that the theorem would still hold, and many subtleties would be missed.

Using HOL, confirming that an old proof still works for a new definition involves updating the statement of the theorem to account for any changes in the arguments to the terms (HOL's typechecker helps with this), and then rerunning the proof script on the statement of the theorem. HOL will either report that the proof script works, or it will return an error. If an error is returned, tactics from the script can be entered a piece at a time until the error is reached, and then the problem is apparent and can be fixed. Usually the fixes are simple, and if they are not simple, it is because the definition has changed substantially enough that there is something significant to be proved.

In our many episodes of doing these re-proofs, we found that, as expected, lemmas relating directly to the change took some time to change. As the theorems related less and less directly with the change, then proof scripts were easier to fix, until at a certain level of abstraction, the proofs went through unaltered. HOL's ability to automatically check these reproofs and quickly locate places where changes needed to be made in the proofs was a tremendous time saver. It meant that effectively, we had to prove everything

only once. If the proofs were on paper, they would be almost as difficult to check when definitions were changed as they were to prove originally, and would be much less likely to be modified correctly to account for the change.

Proving theorems about substitutions (and related operations such as alpha-conversion) required far more time and HOL code than any other variety of theorems. This is not a surprise, since lemmas about substitution abound in papers on proofs of programming language properties. In [Tofte90] the proof of Lemma 4.2, the analogy of the theorem we call typechecking under substitution, is one of the longest proofs in the paper. One would like to find a way to avoid such complications.

One possible solution for this problem would be to write packages for HOL (similar to our type and relational definitions packages) that will automatically define and prove theorems about substitutions. If HOL is going to be used for proving properties about programming languages, it will be essential to do this to prevent these theorems from being proved over and over.

Another approach to solving the problems with substitution might be to use *higher order abstract syntax*. Higher order abstract syntax (HOAS) involves using the binding and substitution properties of the language of the theorem prover (here, HOL terms) to handle the binding and substitution properties of the language being studied (here, SML). Since the semantics of SML uses bound variables and substitution only in type schemes, we contemplate using HOAS for this aspect of the language.

The first problem we note is that we cannot use HOAS in type schemes in HOL. In our encoding type variables are represented by an HOL constructor wrapped around a string, and type variables are explicitly listed as one of the possibilities for an SML type. In an HOAS version there would not be an explicit case for type variables. Instead, an HOL variable of type `type` (which is the HOL type of semantic objects representing SML types) would be used instead. To make a type scheme from a type, we need to quantify over a collection of type variables. This is done in our encoding by wrapping a constructor around a list of encoded type variables (the prefix) and a type (the body).

In an HOAS encoding, for each type variable being quantified over, we would use the HOL λ to abstract over the "type variable" (an HOL variable) and use a constructor to inject the function into an HOL type for type schemes, call it `typescheme2`. Since we can quantify over several type variables, the term to which we apply the λ-abstraction would need to be a `typescheme2`, so we would need the constructor to inject the function

type `type` → `typescheme2` into `typescheme2`. There is a theoretical difficulty in doing this: the HOL function type contains *all* functions from one type to another, not just the computable ones. As a result the function type `type` → `typescheme2` has an uncountable number of elements, since the number of elements in `type` and in `typescheme2` is each countably infinite. Therefore, the function type `type` → `typescheme2` cannot be injected into `typescheme2`.

Thus in order to encode type schemes using HOAS, we would need to use a system that handles function types differently. There are several systems that are designed to be used with HOAS, such as λProlog [NM88] and Elf [Pfenning91]. One might contemplate encoding SML in one of these systems and using this to prove properties of the language. There may be difficulties here as well. In particular, the way the function Clos (discussed in Section A.2) is defined, one needs to be able to compute the type variables in a type that are not free in the context. Since there is no explicit encoding for type variables, it may not be possible to write in the language of the theorem prover a function that returns the type variables in a type.

Even if HOAS cannot be used to avoid problems with bound variables and substitution in SML as described in the Definition, it may be possible to reformulate the semantics in a way that allows an efficient encoding using HOAS. Some preliminary work in this area has been done in [Chirimar95].

Appendix A

Semantics of HOL-SML

HOL-SML is the language about which we proved our theorems. It started out as Core SML modified as per Stefan Kahrs' suggestions as to how to fix the semantics to allow type preservation to be proved. Through the course of the work presented here, that starting language has been changed substantially. Most of these latter changes were simplifications in order to make the proofs more feasible. This appendix gives the grammar and the static and dynamic semantics after all the changes made. In order for the semantics presented here to be most easily compared with that in the Definition, the rules are presented in a style that closely mimics the style used there. In addition, some of the definitions of functions used in the rules (for example, Abs, used in Rule 20) use a wording close to that in the Definition.

A.1 Grammar

The grammar of HOL-SML is described below. As noted in the Commentary and in [Kahrs93], identifier status is not unambiguously explained in the Definition. For HOL-SML, we have taken the following approach: Var (variables), Con (value constructors), ExCon (exception constructors), and a catch-all Id are all separate semantic objects. The first three are used in the syntax. Id is the domain of VarEnv (variable environments), and thus elements of Var, Con, and ExCon are converted to Id to be placed in a VarEnv. The domain of ConEnv (constructor environments) is also Id, in order to make converting them to VarEnvs (as is done in Rule 29) easier.

The grammar is given in Tables A.1 and A.2. Only the parts of the grammar dealt with

atexp	::=	*scon*	special constant
		longvar	value variable
		longcon	value constructor
		longexcon	exception constructor
		`{` ⟨*exprow*⟩ `}`	record
		`let` *dec* `in` *exp* `end`	local declaration
		`(` *exp* `)`	parenthesized expression
exprow	::=	*lab* `=` *exp* ⟨`,` *exprow*⟩	expression row
exp	::=	*atexp*	atomic expression
		exp atexp	application
		exp `:` *ty*	typed expression
		exp `handle` *match*	handle exception
		`raise` *exp*	raise exception
		`fn` *match*	function
match	::=	*mrule* ⟨`|` *match*⟩	
mrule	::=	*pat* `=>` *exp*	
dec	::=	`val` *valbind*	value declaration
		`type` *typbind*	type declaration
		`datatype` *datbind*	datatype declaration
		`abstype` *datbind* `with` *dec* `end`	abstype declaration
		`exception` *exbind*	exception declaration
		`local` dec_1 `in` dec_2 `end`	local declaration
		`open` $longstrid_1 \ldots longstrid_n$	open declaration($n \geq 1$)
			empty declaration
		dec_1⟨`;`⟩dec_2	sequential declaration
valbind	::=	*pat* `=` *exp* ⟨`and` *valbind*⟩	
		`rec` *valbind*	recursive binding
typbind	::=	*tyvarseq tycon* `=` *ty* ⟨`and` typbind⟩	
datbind	::=	*tyvarseq tycon* `=` *conbind* ⟨`and` datbind⟩	
conbind	::=	*con* ⟨`of` ty⟩ ⟨`|` *conbind*⟩	
exbind	::=	*excon* ⟨`of` ty⟩ ⟨`and` *exbind*⟩	
		excon `=` *longexcon* ⟨`and` *exbind*⟩	

Table A.1: Grammar: Expressions, Matches, Declarations, and Bindings

atpat	::=	`_`	wildcard
		scon	special constant
		var	variable
		longcon	constant
		longexcon	exception constant
		`{` ⟨*patrow*⟩ `}`	record pattern
		`(` *pat* `)`	parenthesized pattern
patrow	::=	`...`	wildcard
		lab `=` *pat* ⟨`,` *patrow*⟩	pattern row
pat	::=	*atpat*	atomic pattern
		longcon atpat	value construction
		longexcon atpat	exception construction
		pat `:` *ty*	typed pattern
		var⟨`:` *ty*⟩`as` pat	layered pattern
ty	::=	*tyvar*	type variable
		`{` ⟨*tyrow*⟩ `}`	record type expression
		tyvarseq tycon	type construction
		ty `->` *ty′*	function type expression (R)
		`(` *ty* `)`	parenthesized type
		`genericty`	generic type
tyrow	::=	*lab* `:` *ty* ⟨,*tyrow*⟩	type-expression rows

Table A.2: Grammar: Patterns and Type Expressions

by the semantics are shown; derived forms and infix and precedence directives are omitted. Aside from the absence of these directives, the only difference between this grammar and that for Core SML given in Section 2.8 of the Definition is that one additional phrase, `genericty`, has been added to the type expressions. Its use is described on page 90. All the syntactic restrictions in Section 2.9 of the Definition hold here as well.

A.2 Static Semantics

The static semantics of HOL-SML has been simplified significantly from that of Core SML. The changes in the semantic objects and rules are listed below, as well as the complete set of rules for typechecking HOL-SML.

Semantic Objects

Here we list the differences between the semantic objects for Core SML and HOL-SML. There is an additional simple semantic object, Stat, that indicates the status of an

identifier. The possible values for Stat are v_stat (variable status), c_stat (constructor status), and e_stat (exception constructor status). The simple semantic objects TyVar (type variables) and TyName (type constructor names) differ only in that TyVars have neither equality nor imperative attributes, and TyNames no longer have equality attributes. Another change is that we have pared down the base types (those whose type names appear in the initial static basis) by omitting `real`, `instream`, and `outstream`. Consequently, we omit in the dynamic semantics all basic values operating on these types. The compound semantic object TyVarSet, which is used in accounting for scope of explicit type variables, has been omitted. The class StatMap of status maps and their place in the static environment will be described below.

These are the changes in the compound semantic objects:

$$\begin{aligned} CE &\in \text{ConEnv} = \text{Id} \xrightarrow{\text{fin}} \text{TypeScheme} \\ VE &\in \text{VarEnv} = \text{Id} \xrightarrow{\text{fin}} \text{TypeScheme} \\ SM &\in \text{StatMap} = \text{Stat} \xrightarrow{\text{fin}} \text{Stat} \\ E \text{ or } (SE, TE, VE, EE, SM) &\in \text{Env} = \text{StrEnv} \times \text{TyEnv} \times \text{VarEnv} \times \\ &\qquad \text{StrEnv} \times \text{ExConEnv} \times \text{StatMap} \\ C \text{ or } T, E &\in \text{Context} = \text{TyVarSet} \times \text{Env} \end{aligned}$$

Changes in the Rules and Semantic Functions

In the rules for declarations below, a set of type names is returned. This is the set of type names introduced by `datatype` declarations. Each use of $\oplus$ in the Definition is replaced here by an explicit addition of type names into the context. The purpose of this is to account more carefully for the generation of type names and to include in (T of C) only those type names that have been generated by `datatype` declarations, rather than to attempt to include all type names currently in use. The motivation for this is given in Section 8.5.

It is worth noting that two mistakes in the static semantics of the Definition remain unfixed in the semantics here. The problems are are in Rules 6 and 16 and deal with the scope of type names. This is described in Section 8.5.

The function Clos is used on constructor environments in Rule 29 and on variable environments in Rule 17. If a constructor environment CE has the form $\{id_i \mapsto \tau_i; 1 \leq i \leq n\}$ then Clos CE is $\{id_i \mapsto \forall(\text{tyvars}\,\tau_i).\tau_i; 1 \leq i \leq n\}$.

The syntactic restrictions in Section 2.9 of the Definition specify that a value binding bind may bind an identifier only once. Say a value environment VE has the form

$$\{id_i \mapsto \tau_i; 1 \leq i \leq n\}$$

and results from the elaboration of a value binding *valbind*, which has the form

$$pat_1 \texttt{=} exp_1 \mid \ldots \mid pat_m \texttt{=} exp_m$$

Then for each $i \leq n$, id_i will be mentioned in exactly one pattern in the value binding; say this is pat_k. Then exp_k will be the expression that binds id_i, that is, gives it its value and type. Let get_exp(*valbind, id*) return the expression that binds *id* in *valbind*.

In order to avoid generalizing the types of reference cells, if get_exp (*valbind, id*) can create a reference cell, we will not generalize its type. Expressions whose evaluation cannot create new reference cells are functions (`fn` expressions), value constructors, and variables; these will be called *non-expansive*.[1] All other expressions will be called *expansive*.

The closure of VE, $\mathrm{Clos}_{C,valbind}VE$ is

$$\{id_i \mapsto \sigma_i; 1 \leq i \leq n\}$$

where

$$\sigma_i = \begin{cases} \forall(\text{tyvars } \tau_i \backslash \text{tyvars } C).\tau_i & \text{if get\_exp(}valbind,\ id\text{) is non-expansive} \\ \tau_i & \text{if get\_exp(}valbind,\ id\text{) is expansive} \end{cases}$$

This definition is considerably simpler than that given in the Definition. In the Definition, there are two varieties of type variables, *applicative* and *imperative*. Imperative variables are those that could end up in reference cells, while applicative variables cannot. The idea in the semantics in the Definition is that an non-expansive expression cannot create new reference cells, so all type variables (that are not free in the context) can be generalized. Expansive expressions can create reference cells, so only the applicative type variables can be generalized. The idea presented here, that that all type variables are generalized if the expression is non-expansive and none are generalized if the expression is expansive, is called the *value restriction*, and is described in [Wright93].

In Section 4.5 of the Definition, it is stated that two type schemes are considered equal if they are alpha-equivalent. Since HOL handles structural equality much better than

[1] Tofte in [Tofte96] suggests allowing more expressions to be non-expansive, including records (if the expression associated with each label is non-expansive) and constructors applied to non-expansive expressions.

alpha-equivalence, we tried to do without alpha-equivalence wherever possible. For the theorems we proved, we found that we only needed to allow for alpha-equivalence only in Rule 17, where generalization of free type variables in variable environments is done. We have mentioned alpha-equivalence explicitly in the Rule 17 below to indicate this. This is explained on page 98.

In Rule 17 in the Definition, there is a quantity U that is the set of explicit type variables scoped at this value binding. The side conditions involving U ensure that an explicit type variable can be quantified at the point at which it is scoped. The definition of this concept is complex and easy to misinterpret. The biggest argument against it is that it can catch the user unaware, as it imposes a somewhat arbitrary scope on top of syntax with no scoping constructs. We have therefore eliminated the special rules in order to simplify the semantics, and we treat all type variables equally, whether or not they are explicit in the syntax. Tofte, in [Tofte96] takes another approach, by introducing a binding constructor for explicit type variables.

The function rng_tynames (used in Rules 19 and 20 below) collects the type names that are given meaning in a type environment:

$$\text{rng\_tynames}\, TE = \{t \mid (t, CE) \in \text{Ran}\, TE \text{ for some } CE\}$$

The side condition $\text{rng\_tynames}\, TE \bigcap (T \text{ of } C) = \emptyset$ in Rules 19 and 20 is equivalent to that given in the Definition, $\forall (t, CE) \in \text{Ran}\, TE, t \notin (T \text{ of } C)$.

Rules 19 and 20 below are simplified from the Definition by the removal of the side condition: TE maximizes equality. This reflects the fact that equality types have been done away with. Equality in HOL-SML is only defined for the basic types `int`, `string`, and `bool`. This is a major simplification, as the definition of when types and other semantic objects do and do not admit equality is involved and confusing. Eliminating equality types simplifies the definition of Abs.

For any TE of the form

$$TE = \{tycon_i \mapsto (t_i, CE_i); 1 \leq i \leq k\}$$

where no CE_i is the empty map, and for any E, we define $\text{Abs}(TE,\ E)$ to be the environment obtained from E and TE as follows. First, let $\text{Abs}(TE)$ be the type environment $\{tycon_i \mapsto (t_i, \{\}); 1 \leq i \leq k\}$ in which all the constructor environments have been replaced by the empty map. Then $\text{Abs}(TE, E) = \text{Abs}(TE) + E$.

Rule 51.1 below (for the type expression `genericty`) is needed to allow the proof of typechecking under substitution to go through. It is used in the "`of` *ty*" qualifications in exception bindings to allow the type expression to match any type. For a similar reason we remove explicit type tags from expressions and patterns while doing this proof. (The function that executes this task is strip_tags and is discussed in Section 8.2.)

Appendix B of the Commentary describes status maps, which are finite maps from identifiers to a status indicating whether the identifier is used as a value constructor, an exception constructor, or a variable. In the grammar used for HOL-SML, we presume that the status of the identifier in the syntax has been resolved. However, we found that we still needed to know the status of identifiers to express well-formedness constraints on contexts. We needed to be assured, for example, that if an identifier can be looked up as a value constructor, then it was placed into a variable environment as a value constructor, and that if a lookup of the constructor succeeded, then the type scheme returned satisfied certain properties particular to value constructors.

The last paragraph of this appendix suggests that status maps can be integrated into the static semantics. We found this to be the most convenient way to handle status maps. Thus in the static enironments, each variable environment has an associated status map which records the status of each identifier. These status maps are built up in the same way the variable environments are (see Rules 30*a*, 31*a*, and 35), and when identifiers are looked up in contexts, it is checked that the status under which they are looked up is the same status with which they were put in the variable environment.

Note that according to the elaboration rules below, the only way an identifier enters a variable environment with status c_stat (for constructor) is via Rules 29 and 19 (or by an `open` declaration), and the only way an identifier enters a variable environment with status e_stat (for exception constructor) is via Rule 21 (or by an `open` declaration). When value or exception constructors enter static variable environments via Rules 29 or 21, these are products of the original declarations of the objects and thus they are paired with the original type schemes. In Rules 130 and 137.1, constructor or exception names are put the into an analogous location in the dynamic environment.

Later, variables can be assigned constructor values, resulting in there being a longId *longid* such that $E(longid)$ is the name of the constructor, and $C(longid)$ is an instantiation of the original type scheme.[2] However, these secondary declarations can only be looked

[2]For example `val elist = []:('a*int)list`.

up as identifiers, so they are distinguishable from the originals. Since `open` declarations copy the values and type schemes in environments exactly, if $E(longcon_1) = E(longcon_2)$ (that is, there are two longCons returning the same constructor name), and $C(longcon_1) = \sigma_1$ and $C(longcon_2) = \sigma_2$ (that is, both declarations entered the static environment as constructors), then $\sigma_1 = \sigma_2$.

Atomic Expressions $\boxed{C \vdash atexp : \tau}$

$$\frac{}{C \vdash scon : \mathrm{type}(scon)} \tag{1}$$

$$\frac{C(longvar) = \sigma \quad \sigma \succ \tau}{C \vdash longvar : \tau} \tag{2}$$

$$\frac{C(longcon) = \sigma \quad \sigma \succ \tau}{C \vdash longcon : \tau} \tag{3}$$

$$\frac{C(longexcon) = \tau}{C \vdash longexcon : \tau} \tag{4}$$

$$\frac{\langle C \vdash exprow : \varrho \rangle}{C \vdash \texttt{\{}\langle exprow \rangle\texttt{\}} : \{\}\langle +\varrho \rangle \text{ in Type}} \tag{5}$$

$$\frac{C \vdash dec : E, T \quad C + (T, E) \vdash exp : \tau}{C \vdash \texttt{let } dec \texttt{ in } exp \texttt{ end} : \tau} \tag{6}$$

$$\frac{C \vdash exp : \tau}{C \vdash \texttt{(}exp\texttt{)} : \tau} \tag{7}$$

Expression Rows $\boxed{C \vdash exprow : \varrho}$

$$\frac{C \vdash exp : \tau \quad \langle C \vdash exprow : \varrho \rangle}{C \vdash lab \texttt{ = } exp \langle \texttt{, } exprow \rangle : \{lab \mapsto \tau\}\langle +\varrho \rangle} \tag{8}$$

Expressions $\boxed{C \vdash exp : \tau}$

$$\frac{C \vdash atexp : \tau}{C \vdash atexp : \tau} \tag{9}$$

$$\frac{C \vdash exp : \tau' \to \tau \quad C \vdash atexp : \tau'}{C \vdash exp\ atexp : \tau} \tag{10}$$

$$\frac{C \vdash exp : \tau \quad C \vdash ty : \tau}{C \vdash exp \ : \ ty : \tau} \tag{11}$$

$$\frac{C \vdash exp : \tau \quad C \vdash match : \texttt{exn} \to \tau}{C \vdash exp \ \texttt{handle} \ match : \tau} \tag{12}$$

$$\frac{C \vdash exp : \texttt{exn}}{C \vdash \texttt{raise} \ exp : \tau} \tag{13}$$

$$\frac{C \vdash match : \tau}{C \vdash \texttt{fn} \ match : \tau} \tag{14}$$

Matches $\boxed{C \vdash match : \tau}$

$$\frac{C \vdash mrule : \tau \quad \langle C \vdash match : \tau \rangle}{C \vdash mrule \ \langle \texttt{|} \ match \rangle : \tau} \tag{15}$$

Match Rules $\boxed{C \vdash mrule : \tau}$

$$\frac{C \vdash pat : (VE, SM, \tau) \quad C + (VE, SM) \vdash exp : \tau'}{C \vdash pat \ \texttt{=>} \ exp : \tau \to \tau'} \tag{16}$$

Declarations $\boxed{C \vdash dec : E, T}$

$$\frac{C \vdash valbind : VE, SM \quad \text{alpha\_equiv} \ (\text{Clos}_{C, valbind} VE) \, VE'}{C \vdash \texttt{val} \ valbind : VE' \text{ in Env}, SM} \tag{17}$$

$$\frac{C \vdash typbind : TE}{C \vdash \texttt{type} \ typbind : TE \text{ in Env}, \{\}} \tag{18}$$

$$\frac{\begin{array}{c} C + (\text{rng\_tynames} \, TE, \, TE) \vdash datbind : VE, \, TE, SM \\ \text{rng\_tynames} \, TE \bigcap (T \text{ of } C) = \emptyset \end{array}}{C \vdash \texttt{datatype} \ datbind : (VE, TE) \text{ in Env}, \text{rng\_tynames} \, TE} \tag{19}$$

$$\frac{\begin{array}{c} C + (\text{rng\_tynames} \, TE, \, TE) \vdash datbind : VE, \, TE, SM \\ \text{rng\_tynames} \, TE \bigcap (T \text{ of } C) = \emptyset \\ C + (\text{rng\_tynames} \, TE, \, VE, \, TE, SM) \vdash dec : E, T \end{array}}{\begin{array}{c} C \vdash \texttt{abstype} \ datbind \ \texttt{with} \ dec \ \texttt{end} : \\ \text{Abs}(TE, E), \text{rng\_tynames} \, TE \bigcup T \end{array}} \tag{20}$$

$$\frac{C \vdash exbind : EE, SM \quad VE = EE}{C \vdash \texttt{exception} \ exbind : (VE, EE, SM) \text{ in Env}, \{\}} \tag{21}$$

$$\frac{C \vdash dec_1 : E_1, T_1 \quad C + (T_1, E_1) dec_1 : E_2, T_2}{C \vdash \texttt{local}\ dec_1 \texttt{in}\ dec_2\ \texttt{end} : E_2, T_1 \bigcup T_2} \tag{22}$$

$$\frac{C(longstrid_1) = (m_1, E_1) \quad \ldots \quad C(longstrid_n) = (m_n, E_n)}{C \vdash \texttt{open}\ longstrid_1 \ldots longstrid_n : E_1 + \ldots + E_n, \{\}} \tag{23}$$

$$\frac{}{C \vdash \quad : \{\} \text{ in Env}, \{\}} \tag{24}$$

$$\frac{C \vdash dec_1 : E_1, T_1 \quad C + (T_1, E_1) \vdash dec_2 : E_2, T_2}{C \vdash dec_1 \langle ; \rangle dec_2 : E_1 + E_2, T_1 + T_2} \tag{25}$$

Value Bindings $\boxed{C \vdash valbind : VE, SM}$

$$\frac{C \vdash pat : (VE, SM, \tau) \quad C \vdash exp : \tau \quad \langle C \vdash valbind : VE' \rangle}{C \vdash pat\ \texttt{=}\ exp\ \langle \texttt{and}\ valbind \rangle : VE\,\langle + VE' \rangle, SM\,\langle + SM' \rangle} \tag{26}$$

$$\frac{C + (VE, SM) \vdash valbind : VE, SM}{C \vdash \texttt{rec}\ valbind : VE, SM} \tag{27}$$

Type Bindings $\boxed{C \vdash typbind : TE}$

$$\frac{tyvarseq = \alpha^{(k)} \quad C \vdash ty : \tau \quad \langle C \vdash typbind : TE \rangle}{\begin{array}{c} C \vdash tyvarseq\ tycon\ \texttt{=}\ ty\ \langle \texttt{and}\ \text{typbind} \rangle : \\ \{tycon \mapsto (\Lambda\alpha^{(k)}.\tau, \{\})\} \langle + TE \rangle \end{array}} \tag{28}$$

Data Type Bindings $\boxed{C, \tau \vdash datbind : VE, TE, SM}$

$$\frac{\begin{array}{c} tyvarseq = \alpha^{(k)} \quad C, \alpha^{(k)} t \vdash conbind : CE, SM \\ \langle C \vdash datbind : VE, TE, SM' \quad \forall (t', CE) \in \text{Ran } TE, t \neq t' \rangle \end{array}}{\begin{array}{c} C \vdash tyvarseq\ tycon\ \texttt{=}\ conbind\ \langle \texttt{and}\ \text{datbind} \rangle : \\ \text{Clos } CE \langle + VE \rangle, \\ \{tycon \mapsto (t, \text{Clos } CE)\} \langle + TE \rangle, SM \langle + SM \rangle \end{array}} \tag{29}$$

Constructor Bindings $\boxed{C, \tau \vdash conbind : CE, SM}$

$$\frac{\langle C, \tau \vdash conbind : CE, SM \rangle}{C, \tau \vdash con\ \langle \texttt{|}\ \text{conbind} \rangle : \{con \mapsto \tau\} \langle + CE \rangle, \{con \mapsto \text{c\_stat}\} \langle + SM \rangle} \tag{30a}$$

$$\frac{C \vdash ty : \tau' \quad \langle C, \tau \vdash conbind : CE, SM \rangle}{\begin{array}{c} C, \tau \vdash con\ \texttt{of}\ ty\ \langle \texttt{|}\ \text{conbind} \rangle : \\ \{con \mapsto \tau' \to \tau\}\langle + CE \rangle, \{con \mapsto \text{c\_stat}\}\langle + SM \rangle \end{array}} \quad (30b)$$

Exception Bindings $\boxed{C \vdash exbind : EE, SM}$

$$\frac{\langle C \vdash exbind : EE, SM \rangle}{\begin{array}{c} C \vdash excon\ \langle \texttt{and}\ \text{exbind} \rangle : \\ \{excon \mapsto \texttt{exn}\}\langle + EE \rangle, \{excon \mapsto \text{e\_stat}\}\langle + SM \rangle \end{array}} \quad (31a)$$

$$\frac{C \vdash ty : \tau \quad \langle C \vdash exbind : EE, SM \rangle}{\begin{array}{c} C \vdash excon\ \texttt{of}\ ty\ \langle \texttt{and}\ \text{exbind} \rangle : \\ \{excon \mapsto \tau \to \texttt{exn}\}\langle + EE \rangle, \{excon \mapsto \text{e\_stat}\}\langle + SM \rangle \end{array}} \quad (31b)$$

$$\frac{C(longexcon) = \tau \quad \langle C \vdash exbind : EE, SM \rangle}{\begin{array}{c} C \vdash excon\ \texttt{=}\ longexcon\ \langle \texttt{and}\ \text{exbind} \rangle : \\ \{excon \mapsto \tau\}\langle + EE \rangle, \{excon \mapsto \text{e\_stat}\}\langle + SM \rangle \end{array}} \quad (32)$$

Atomic Patterns $\boxed{C \vdash atpat : (VE, SM, \tau)}$

$$\frac{}{C \vdash \_ : (\{\}, \{\}, \tau)} \quad (33)$$

$$\frac{}{C \vdash scon : (\{\}, \{\}, \text{type}(scon))} \quad (34)$$

$$\frac{}{C \vdash var : (\{var \mapsto \tau\}, \{var \mapsto \text{v\_stat}\}, \tau)} \quad (35)$$

$$\frac{C(longcon) \succ \tau^{(k)}t}{C \vdash longcon : (\{\}, \{\}, \tau^{(k)}t)} \quad (36)$$

$$\frac{C(longexcon) = \texttt{exn}}{C \vdash longcon : (\{\}, \{\}, \texttt{exn})} \quad (37)$$

$$\frac{\langle C \vdash patrow : (VE, SM, \varrho) \rangle}{C \vdash \texttt{\{}\langle patrow \rangle\texttt{\}} : (\{\}\langle + VE \rangle, \{\}\langle + SM \rangle, \{\}\langle + \varrho \rangle \text{ in Type})} \quad (38)$$

$$\frac{C \vdash pat : (VE, SM, \tau)}{C \vdash (pat) : (VE, SM, \tau)} \quad (39)$$

Pattern Rows $\boxed{C \vdash patrow : (VE, SM, \varrho)}$

$$\frac{}{C \vdash \ldots : (\{\}, \{\}, \varrho)} \tag{40}$$

$$\frac{\begin{array}{c} C \vdash pat : (VE, SM, \tau) \\ \langle C \vdash patrow : (VE', SM', \varrho) \quad patrow \notin \mathrm{Dom}\ \varrho \rangle \end{array}}{\begin{array}{c} C \vdash pat\ \texttt{=}\ pat \langle \texttt{,} patrow \rangle : \\ (VE \langle + VE' \rangle, SM \langle + SM' \rangle, \{lab \mapsto \tau\} \langle + \varrho \rangle) \end{array}} \tag{41}$$

Patterns $\boxed{C \vdash pat : (VE, SM, \tau)}$

$$\frac{C \vdash atpat : (VE, SM, \tau)}{C \vdash atpat : (VE, SM, \tau)} \tag{42}$$

$$\frac{C(longcon) \succ \tau' \to \tau \quad C \vdash atpat : (VE, SM, \tau')}{C \vdash longcon\ atpat : (VE, SM, \tau)} \tag{43}$$

$$\frac{C(longexcon) = \tau \to \texttt{exn} \quad C \vdash atpat : (VE, SM, \tau)}{C \vdash longexcon\ atpat : (VE, SM, \texttt{exn})} \tag{44}$$

$$\frac{C \vdash pat : (VE, SM, \tau) \quad C \vdash ty : \tau}{C \vdash pat\ \texttt{:}\ ty : (VE, SM, \tau)} \tag{45}$$

$$\frac{C \vdash var : (VE, SM, \tau) \quad \langle C \vdash ty : \tau \rangle \quad C \vdash pat : (VE', SM', \tau)}{C \vdash var \langle \texttt{:}\ ty \rangle\ \texttt{as}\ pat : (VE + VE', SM + SM', \tau)} \tag{46}$$

Type Expressions $\boxed{C \vdash ty : \tau}$

$$\frac{tyvar = \alpha}{C \vdash tyvar : \alpha} \tag{47}$$

$$\frac{\langle C \vdash tyrow : \varrho \rangle}{C \vdash \texttt{\{} \langle tyrow \rangle \texttt{\}} : \{\} \langle + \varrho \rangle \text{ in Type}} \tag{48}$$

$$\frac{\begin{array}{c} tyseq = ty_1 \ldots ty_k \quad C \vdash ty_i : \tau_i\ (1 \le i \le k) \\ C(longtycon) = (\theta, CE) \end{array}}{C \vdash tyseq\ longtycon : \tau^{(k)} \theta} \tag{49}$$

$$\frac{C \vdash ty : \tau \quad C \vdash ty' : \tau'}{C \vdash ty\ \texttt{->}\ ty' : \tau \to \tau'} \tag{50}$$

$$\frac{C \vdash ty : \tau}{C \vdash (ty) : \tau} \tag{51}$$

$$\frac{}{C \vdash \texttt{genericty} : \tau} \tag{51.1}$$

Type-expression Rows $\boxed{C \vdash tyrow : \varrho}$

$$\frac{C \vdash ty : \tau \quad \langle C \vdash tyrow : \varrho \rangle}{C \vdash lab\ :\ ty\langle\texttt{,}\ tyrow\rangle : \{lab \mapsto \tau\}\langle + \varrho\rangle} \tag{52}$$

A.3 Dynamic Semantics

Reduced Syntax

Since types are, for the most part, dealt with in the static semantics, the dynamic semantics is defined over a reduced syntax that eliminates most of the type information. However, the reduced syntax for HOL-SML varies significantly from that of Core SML in that datatype bindings (DatBind) and constructor bindings (ConBind) are not eliminated. This allows constructor names (see below) to be generated and placed into environments. The syntax for HOL-SML is reduced by the following transformations:

- All explicit type ascriptions "$: ty$" are omitted, and qualifications "`of` ty" are omitted from exception and constructor bindings.
- Any declaration of the form "`type` *typbind*" is replaced by the empty declaration.
- The phrase classes TypBind, Ty, and TyRow are omitted.

The function "trans" executes this transformation.

Semantic Objects

Below we show only the parts of the semantic objects for HOL-SML that are different from that for Core SML. The basic values are

```
size chr ord explode implode abs div mod neg
    =int =str =bool <>int <>str <>bool
        times plus minus < > <= >=
```

Basic values such as `open_in` have been eliminated because input and output are not being considered in this project. Because equality types have been eliminated, the functions `=`

and `<>` apply only to the base types `int`, `string`, and `bool`. In order to avoid resolving problems of overloading, there are separate values for each type.

The basic exceptions are

```
Abs Div Mod Prod Neg Sum Diff Ord Chr Match Bind
```

The basic exceptions corresponding to operations on real numbers or input/output have been removed.

There is one additional simple semantic object, the ConName.

$$cn \in \text{ConName} \quad \text{constructor names}$$

ConName is integrated into the values as follows:

$$\begin{aligned} v &\in \text{Val} = \{\texttt{:=}\} \bigcup \text{SVal} \bigcup \text{BasVal} \bigcup \text{ConVal} \bigcup \\ &\qquad \text{ExVal} \bigcup \text{Record} \bigcup \text{Addr} \bigcup \text{Closure} \\ cv &\in \text{ConVal} = \text{ConName} \bigcup (\text{ConName} \times \text{Val}) \\ cns &\in \text{ConNameSet} = \text{Fin}(\text{ConName}) \\ (mem,\ ens,\ cns) \text{ or } s &\in \text{State} = \text{Mem} \times \text{ExNameSet} \times \text{ConNameSet} \end{aligned}$$

Changes in the Semantics

One of the most glaring errors in the Definition is that the `ref` constructor is used explicitly in several of the evaluation rules for expressions and patterns (the Definition's Rules 154, 155, 158, 112, and 114) to manipulate the state, while in the static semantics nothing prevents the `ref` from being used as a constructor for a user-declared datatype. This obviously breaks type preservation: the following program

```
datatype foo = ref of int; val x = ref 5
```

will elaborate to the type `foo`, but will evaluate to an address, which can only have type τ `ref`, for some τ. There are several suggestions given in [Kahrs93] for fixing this, including making `ref` a reserved word. The solution that seemed to involve the minimal change from the semantics in the Definition was to retain datatype definitions in the reduced syntax used for evaluation and to keep information on constructors in the variable environment. This requires the use of a semantic object called a ConName (for constructor name). ConName is treated similarly to ExName: the names are generated by constructor bindings and would be stored in variable environments for evaluation as the values to which constructors are

bound. This would allow constructors for new datatypes to be distinguished from the `ref` constructor, as well as from each other. Then constructors no longer evaluate to themselves, but must be looked up in the environment, as in our Rule 105 below.

In the rules for the dynamic semantics in the Definition, the presence of state in rules is omitted except when explicitly involved in the rule. Mention of state is also omitted from the boxed equations representing the general form of judgements for each phrase class. This leads to some confusion. For example, does the evaluation of patterns return a new state? It is a simple proof to show that pattern evaluation cannot change the state, yet Rule 158 suggests that patterns do return a state, but that this state must be the same as the initial state. To avoid this confusion, we include the state in the general form of the judgement.

Rules 114 (for creating a new reference cell) and 138 (for declaring a new exception constructor) in the Definition allow a semantic object to be chosen indeterminately. The way we encoded the rules, however, makes the choice of next ExName and Addr deterministic. In HOL-SML there is a linear order on these semantic objects, making them isomorphic to the natural numbers, and we choose the smallest unused one. This is reflected in our version of those rules below. A similar approach is taken with respect to choosing ConNames in Rule 137.2. This simplifies the proof of determinism significantly, since we do not have to allow for the renaming of addresses, exception names, or constructor names in the evaluation results.

Rules 147, 151, and 152 below differ slightly from those in the Definition in stating explicitly that the result of looking an item (an exception constructor or a label) in an environment or record succeeds. This was needed because, due to the details of our definitions of how items are looked up, unless it is specifically stated in the premises, we could not prove that the lookup succeeded if we only knew that the conclusion of the rule held.

Atomic Expressions $\boxed{s, E \vdash atexp \Downarrow v/p, s'}$

$$\frac{}{E, v \vdash scon \Downarrow \mathrm{val}(scon)} \tag{103}$$

$$\frac{E(longvar) = v}{E \vdash longvar \Downarrow v} \tag{104}$$

$$\frac{E(longcon) = cn}{E \vdash longcon \Downarrow cn} \tag{105}$$

$$\frac{E(longexcon) = en}{E \vdash longexcon \Downarrow en} \tag{106}$$

$$\frac{\langle E \vdash exprow \Downarrow r \rangle}{E \vdash \texttt{\{}\langle exprow \rangle\texttt{\}} \Downarrow \{\}\langle +r \rangle \text{ in Val}} \tag{107}$$

$$\frac{E \vdash dec \Downarrow E' \quad E + E' \vdash exp \Downarrow v}{E \vdash \texttt{let } dec \texttt{ in } exp \texttt{ end} \Downarrow v} \tag{108}$$

$$\frac{E \vdash exp \Downarrow v}{E \vdash \texttt{(}exp\texttt{)} \Downarrow v} \tag{109}$$

Expression Rows $\boxed{s, E \vdash exprow \Downarrow r/p, s'}$

$$\frac{E \vdash exp \Downarrow v \quad \langle E \vdash exprow \Downarrow r \rangle}{E \vdash lab \texttt{ = } pat\langle \texttt{, } exprow \rangle \Downarrow \{pat \mapsto v\}\langle + r \rangle} \tag{110}$$

Expressions $\boxed{s, E \vdash exp \Downarrow v/p, s'}$

$$\frac{E \vdash atexp \Downarrow v}{E \vdash atexp \Downarrow v} \tag{111}$$

$$\frac{E \vdash exp \Downarrow cn \quad cn \neq \text{ref\_cn} \quad E \vdash atexp \Downarrow v}{E \vdash exp\ atexp \Downarrow (cn, v)} \tag{112}$$

$$\frac{E \vdash exp \Downarrow en \quad E \vdash atexp \Downarrow v}{E \vdash exp\ atexp \Downarrow (en, v)} \tag{113}$$

$$\frac{s, E \vdash exp \Downarrow \text{ref\_cn}, s' \quad s', E \vdash atexp \Downarrow v, s'' \quad a = \text{next\_addrs}''}{s, E \vdash exp\ atexp \Downarrow a, s'' + a \mapsto v} \tag{114}$$

$$\frac{s, E \vdash exp \Downarrow \texttt{:=}, s' \quad s', E \vdash atexp \Downarrow \{\texttt{1} \mapsto a, \texttt{2} \mapsto v\}, s''}{s, E \vdash exp\ atexp \Downarrow a, s'' + a \mapsto v} \tag{115}$$

$$\frac{E \vdash exp \Downarrow b \quad E \vdash atexp \Downarrow v \quad \text{APPLY}(b, v) = v'/p}{E \vdash exp\ atexp \Downarrow v'/p} \tag{116}$$

$$\frac{\begin{array}{c} E \vdash exp \Downarrow (match, E', VE) \quad E \vdash atexp \Downarrow v \\ E + \text{Rec}\, VE, v \vdash match \Downarrow v' \end{array}}{E \vdash exp\ atexp \Downarrow v'} \tag{117}$$

$$\frac{E \vdash exp \Downarrow (match, E', VE) \quad E \vdash atexp \Downarrow v \quad E' + \mathrm{Rec}\, VE, v \vdash match \Downarrow \mathrm{FAIL}}{E \vdash exp\ atexp \Downarrow [\texttt{Match}]} \tag{118}$$

$$\frac{E \vdash exp \Downarrow v}{E \vdash exp\ \texttt{handle}\ match \Downarrow v'/p} \tag{119}$$

$$\frac{E \vdash exp \Downarrow [e] \quad E, e \vdash match \Downarrow v}{E \vdash exp\ \texttt{handle}\ match \Downarrow v} \tag{120}$$

$$\frac{E \vdash exp \Downarrow [e] \quad E, e \vdash match \Downarrow \mathrm{FAIL}}{E \vdash exp\ \texttt{handle}\ match \Downarrow [e]} \tag{121}$$

$$\frac{E \vdash exp \Downarrow e}{E \vdash \texttt{raise}\ exp \Downarrow [e]} \tag{122}$$

$$\frac{}{E \vdash \texttt{fn}\ match \Downarrow (match, E, \{\})} \tag{123}$$

Matches $\boxed{s, E, v \vdash match \Downarrow v'/p/\mathrm{FAIL}, s'}$

$$\frac{E, v \vdash mrule \Downarrow v'}{E, v \vdash mrule\langle \texttt{|}\ match\rangle \Downarrow v'} \tag{124}$$

$$\frac{E \vdash mrule \Downarrow \mathrm{FAIL}}{E \vdash mrule \Downarrow \mathrm{FAIL}} \tag{125}$$

$$\frac{E, v \vdash mrule \Downarrow \mathrm{FAIL} \quad E, v \vdash match \Downarrow v'/\mathrm{FAIL}}{E, v \vdash mrule\ \texttt{|}\ match \Downarrow v'/\mathrm{FAIL}} \tag{126}$$

Match Rules $\boxed{s, E, v \vdash mrule \Downarrow v'/p/\mathrm{FAIL}, s'}$

$$\frac{E, v \vdash pat \Downarrow VE \quad E + VE \vdash exp \Downarrow v'}{E, v \vdash pat\ \texttt{=>}\ exp \Downarrow v'} \tag{127}$$

$$\frac{E, v \vdash pat \Downarrow \mathrm{FAIL}}{E, v \vdash pat\ \texttt{=>}\ exp \Downarrow \mathrm{FAIL}} \tag{128}$$

Declarations $\boxed{s, E \vdash dec \Downarrow E'/p, s'}$

$$\frac{E \vdash \mathit{valbind} \Downarrow \mathit{VE}}{E \vdash \texttt{val}\ \mathit{valbind} \Downarrow \mathit{VE}\ \text{in Env}} \tag{129}$$

$$\frac{E \vdash \mathit{datbind} \Downarrow \mathit{VE}}{E \vdash \texttt{datatype}\ \mathit{datbind} \Downarrow \mathit{VE}\ \text{in Env}} \tag{129.5}$$

$$\frac{E \vdash \mathit{datbind} \Downarrow \mathit{VE} \quad E + \mathit{VE} \vdash \mathit{dec} \Downarrow E'}{E \vdash \texttt{abstype}\ \mathit{datbind}\ \texttt{with}\ \mathit{dec}\ \texttt{end} \Downarrow E'} \tag{129.6}$$

$$\frac{E \vdash \mathit{exbind} \Downarrow \mathit{EE} \quad \mathit{VE} = \mathit{EE}}{E \vdash \texttt{exception}\ \mathit{exbind} \Downarrow (\mathit{VE}, \mathit{EE})\ \text{in Env}} \tag{130}$$

$$\frac{E \vdash \mathit{dec}_1 \Downarrow E_1 \quad E + E_1 \vdash \mathit{dec}_2 \Downarrow E_2}{E \vdash \texttt{local}\ \mathit{dec}_1\ \texttt{in}\ \mathit{dec}_2\ \texttt{end} \Downarrow E_2} \tag{131}$$

$$\frac{E(\mathit{longstrid}_1) = E_1 \quad \ldots \quad E(\mathit{longstrid}_n) = E_n}{E \vdash \texttt{open}\ \mathit{longstrid}_1 \ldots \mathit{longstrid}_n \Downarrow E_1 + \ldots + E_n} \tag{132}$$

$$\frac{}{E \vdash \quad \Downarrow \{\}\ \text{in Env}} \tag{133}$$

$$\frac{E \vdash \mathit{dec}_1 \Downarrow E_1 \quad E + E_1 \vdash \mathit{dec}_2 \Downarrow E_2}{E \vdash \mathit{dec}_1 \langle ; \rangle \mathit{dec}_2 \Downarrow E_1 + E_2} \tag{134}$$

Value Bindings $\boxed{s, E \vdash \mathit{valbind} \Downarrow \mathit{VE}/p, s'}$

$$\frac{E \vdash \mathit{exp} \Downarrow v \quad E, v \vdash \mathit{pat} \Downarrow \mathit{VE} \quad \langle E \vdash \mathit{valbind} \Downarrow \mathit{VE}' \rangle}{E \vdash \mathit{pat}\ \texttt{=}\ \mathit{exp} \langle \texttt{and}\ \mathit{valbind} \rangle \Downarrow \mathit{VE} \langle + \mathit{VE}' \rangle} \tag{135}$$

$$\frac{E \vdash \mathit{exp} \Downarrow v \quad E, v \vdash \mathit{pat} \Downarrow \text{FAIL}}{E \vdash \mathit{pat}\ \texttt{=}\ \mathit{exp} \langle \texttt{and}\ \mathit{valbind} \rangle \Downarrow [\texttt{Bind}]} \tag{136}$$

$$\frac{E \vdash \mathit{valbind} \Downarrow \mathit{VE}}{E \vdash \texttt{rec}\ \mathit{valbind} \Downarrow \text{Rec}\, \mathit{VE}} \tag{137}$$

Datatype Bindings $\boxed{s \vdash \mathit{datbind} \Downarrow \mathit{VE}, s'}$

$$\frac{\vdash \mathit{conbind} \Downarrow \mathit{VE} \quad \langle \vdash \mathit{conbind} \Downarrow \mathit{VE}' \rangle}{\vdash \mathit{conbind}\ \langle \texttt{and}\ \mathit{datbind} \rangle \Downarrow \mathit{VE} \langle + \mathit{VE}' \rangle} \tag{137.1}$$

Constructor Bindings $\boxed{s \vdash \mathit{conbind} \Downarrow \mathit{VE}, s'}$

$$\frac{cn = \text{next\_conname}\ (cns \text{ of } s) \quad s' = s + \{cn\} \quad \langle s' \vdash conbind \Downarrow VE, s'' \rangle}{s \vdash con\ \langle \texttt{|}\ conbind \rangle \Downarrow \{con\ \mapsto cn\}\langle + VE \rangle, s'\langle' \rangle} \tag{137.2}$$

Exception Bindings $\boxed{s, E \vdash exbind \Downarrow EE/p, s'}$

$$\frac{en = \text{next\_exname}\ (ens \text{ of } s) \quad s' = s + \{en\} \quad \langle s', E \vdash exbind \Downarrow EE, s'' \rangle}{s, E \vdash excon\ \langle \texttt{and}\ exbind \rangle \Downarrow \{excon\ \mapsto en\}\langle + EE \rangle, s'\langle' \rangle} \tag{138}$$

$$\frac{E(longexcon) = en \quad \langle E \vdash exbind \Downarrow EE \rangle}{E \vdash excon\ \texttt{=}\ longexcon\ \langle \texttt{and}\ exbind \rangle \Downarrow \{excon\ \mapsto en\}\langle + EE \rangle} \tag{139}$$

Atomic Patterns $\boxed{s, E, v \vdash atpat \Downarrow VE/\text{FAIL}, s'}$

$$\frac{}{E, v \vdash \_ \Downarrow \{\}} \tag{140}$$

$$\frac{v = \text{val}\ (scon)}{E, v \vdash scon \Downarrow \{\}} \tag{141}$$

$$\frac{v \neq \text{val}\ (scon)}{E, v \vdash scon \Downarrow \text{FAIL}} \tag{142}$$

$$\frac{}{E, v \vdash var \Downarrow \{var \mapsto v\}} \tag{143}$$

$$\frac{E(longcon) = v}{E, v \vdash longcon \Downarrow \{\}} \tag{144}$$

$$\frac{E(longcon) = cn \quad v \neq cn}{E, v \vdash longcon \Downarrow \text{FAIL}} \tag{145}$$

$$\frac{E(longexcon) = v}{E, v \vdash longexcon \Downarrow \{\}} \tag{146}$$

$$\frac{E(longexcon) = en \quad v \neq en}{E, v \vdash longcon \Downarrow \text{FAIL}} \tag{147}$$

$$\frac{v = \{\}\langle + r \rangle \text{ in Val} \quad \langle E, r \vdash patrow \Downarrow VE/\text{FAIL} \rangle}{E, v \vdash \texttt{\{}\langle patrow \rangle\texttt{\}} \Downarrow \{\}\langle + VE/\text{FAIL} \rangle} \tag{148}$$

$$\frac{E, v \vdash pat \Downarrow VE/\text{FAIL}}{E, v \vdash \texttt{(}pat\texttt{)} \Downarrow VE/\text{FAIL}} \tag{149}$$

Pattern Rows $\boxed{s, E, v \vdash patrow \Downarrow VE/\text{FAIL}, s'}$

$$\frac{}{E, r \vdash \ldots \Downarrow \{\}} \tag{150}$$

$$\frac{r(lab) = v \quad E, v \vdash pat \Downarrow \text{FAIL}}{E, r \vdash lab\ \texttt{=}\ pat\ \langle\texttt{,}\ patrow\rangle \Downarrow \text{FAIL}} \tag{151}$$

$$\frac{r(lab) = v \quad E, v \vdash pat \Downarrow VE \quad \langle E, r \vdash patrow \Downarrow VE'/\text{FAIL}\rangle}{E, r \vdash lab\ \texttt{=}\ pat\ \langle\texttt{,}\ patrow\rangle \Downarrow VE\ \langle + VE'/\text{FAIL}\rangle} \tag{152}$$

Patterns $\boxed{s, E, v \vdash pat \Downarrow VE/\text{FAIL}, s'}$

$$\frac{E, v \vdash atpat \Downarrow VE/\text{FAIL}}{E, v \vdash atpat \Downarrow VE/\text{FAIL}} \tag{153}$$

$$\frac{\begin{array}{c}E(longcon) = cn \quad cn \neq \text{ref\_cn} \quad v = (cn, v') \\ E, v' \vdash atpat \Downarrow VE/\text{FAIL}\end{array}}{E, v \vdash longcon\ atpat \Downarrow VE/\text{FAIL}} \tag{154}$$

$$\frac{E(longcon) = cn \quad cn \neq \text{ref\_cn} \quad v \notin \{cn\} \times \text{Val}}{E, v \vdash longcon\ atpat \Downarrow VE/\text{FAIL}} \tag{155}$$

$$\frac{\begin{array}{c}E(longexcon) = en \quad v = (en, v') \\ E, v' \vdash atpat \Downarrow VE/\text{FAIL}\end{array}}{E, v \vdash longexcon\ atpat \Downarrow VE/\text{FAIL}} \tag{156}$$

$$\frac{E(longexcon) = en \quad v \notin \{en\} \times \text{Val}}{E, v \vdash longexcon\ atpat \Downarrow \text{FAIL}} \tag{157}$$

$$\frac{\begin{array}{c}E(longcon) = \text{ref\_cn} \quad s(a) = v \\ s, E, v \vdash atpat \Downarrow VE/\text{FAIL}, s\end{array}}{s, E, v \vdash longcon\ pat \Downarrow VE/\text{FAIL}} \tag{158}$$

$$\frac{E, v \vdash pat \Downarrow VE/\text{FAIL}}{E, v \vdash var\ \texttt{as}\ pat \Downarrow \{var \mapsto v\} + VE/\text{FAIL}} \tag{159}$$

Bibliography

[ACPP89] Martin Abadi, Luca Cardelli, Benjamin Pierce, and Gordon Plotkin. Dynamic Typing in a Statically Typed Language. Digital Systems Research Center Technical report No. 47, 1992.

[Appel92] A. W. Appel. A Critique of Standard ML. Unpublished manuscript. February 1992.

[AM91] A. W. Appel and D. B. MacQueen. Standard ML of New Jersey. In *Programming Language Implementation and Logic Programming*, Vol. 528 of *Lecture Notes in Computer Science*, pages 1–26, Springer-Verlag, August 1991.

[BM79] Robert Boyer and J. Strother Moore. *A Computational Logic*. Academic Press, 1979.

[CM92] Juanito Camilleri and Tom Melham. Reasoning with Inductively Defined Relations in the HOL Theorem Prover. Technical Report No. 265, University of Cambridge Computer Laboratory, August 1992.

[CW85] Luca Cardelli and Peter Wegner. On Understanding Types, Data Abstraction, and Polymorphism. *Computing Surveys*, 17(4):471–522.

[Chirimar95] Jawahar Lal Chirimar. *Proof Theoretic Approach to Specification Languages*, PhD Thesis, University of Pennsylvania, 1995.

[CDDK86] Dóminique Clément, Joëlle Despeyroux, Thierry Despeyroux, Gilles Kahn. A Simple Applicative Language: Mini-ML. In *Proceedings of ACM Symposium on LISP and Functional Programming*, pages 13–27, 1986.

[Cohen89] Richard Cohen. Proving Gypsy Programs. Computational Logic Incorporated Technical Report No. 4, February 1989.

[FF86] M. Felleisen and D. Friedman. Control Operators, the SECD Machine and the λ-Calculus. In *Formal Description of Programming Concepts III*, pages 131–141, North-Holland, 1986.

[Gödel83] Kurt Gödel. Russell's Mathematical Logic. In *Philosophy of Mathematics: Selected Readings*, Benacerraf and Putnam (eds.), pages 447–469, Cambridge University Press, 1983.

[GSY90] Donald Good, Ann Siebert, and William Young. Middle Gypsy 2.05 Definition. Computational Logic Incorporated Technical Report No. 59, May 1990.

[GM93] M. J. C. Gordon and T. F. Melham. *Introduction to HOL: A theorem proving environment for higher order logic*. Cambridge University Press, Cambridge, UK, 1993.

[GGM93] C. A. Gunter, E. L. Gunter and D. B. MacQueen. Using abstract interpretation to compute ML equality kinds. *Information and Computation*, 107:303–323, 1993.

[GR93] Carl Gunter and Didier Rémy. A Proof-Theoretic Assessment of Runtime Type Errors. AT&T Bell Laboratories Technical Memo 11261-921230-43TM, November 1993.

[HDM93] Robert Harper, Bruce Duba, and David MacQueen. Typing First-Class Continuations in ML. *Journal of Functional Programming* 3(4):465–484, October 1993.

[HMV92] My Hoang, John Mitchell, Remesh Viswanathan. Standard ML weak polymorphism and imperative constructs. Unpublished manuscript. December 1992.

[HFWHD91] Gerard Huet, Amy Felty, B. Werner, H Herbelin, Gilles Dowek. Presenting the system Coq, version 5.6. In *Proceedings of the Second Workshop on Logical Frameworks*, Huet, Plotkin, and Jones (eds.), Edinburgh 1991.

[Kahrs93] Stefan Kahrs. Mistakes and Ambiguities in the Definition of Standard ML, University of Edinburgh, Laboratory for Foundations of Computer Science Technical Report ECS-LFCS-93-257, April 1993.

[Kahrs94] Stefan Kahrs. Mistakes and Ambiguities in the Definition of Standard ML, Addenda. Unpublished manuscript, June 1994.

[KR78] B. Kernighan and D. M. Ritchie. *The C Programming Language*, Prentice Hall, 1978.

[Knuth67] Donald Knuth. The Remaining Trouble Spots in ALGOL 60. *Communications of the ACM* 10(10):611–618, October 1967

[Leroy93] Xavier Leroy. Polymorphism by name for references and continuations. *Proceedings of 20th Principles of Programming Languages*, pages 220–231, January 1993.

[MG93] Savitry Maharaj and Elsa Gunter. Studying the ML Module System in HOL. To appear in *Proceedings of Higher Order Logic Theorem Proving and Its Applications*, Sept 1994.

[M-L84] Per Martin-Löf. *Intuitionistic type theory.* Bibliopolis, Napoli, 1984.

[McCarthy] John McCarthy. Recursive Functions of Symbolic Expressions. *Communications of the ACM*, 3(4):184–195, April 1960.

[McMillan92] K.L. McMillan. *Symbolic Model Checking: An Approach to the State Explosion Problem*, PhD Thesis, Carnegie Mellon University, 1992.

[Melham88] T. Melham. Automating Recursive Type Definitions in Higher Order Logic. University of Cambridge Computer Laboratory Technical Report No. 146, September 1988.

[Melham93] T. Melham. *Higher Order Logic and Hardware Verification.* Cambridge Tracts in Theoretical Computer Science Volume 31, 1993.

[Milner78] Robin Milner. A Theory of Type Polymorphism in Programming. *Journal of Computer and System Sciences* 17:348–375, 1978

[MT91] Robin Milner, Mads Tofte. *Commentary on Standard ML*, The MIT Press, Cambridge, Mass, 1991.

[MTH90] Robin Milner, Mads Tofte, Robert Harper. *The Definition of Standard ML*, The MIT Press, Cambridge, Mass, 1990.

[NM88] Gopalan Nadathur and Dale Miller. An Overview of λProlog. In *Fifth International Logic Programming Conference*, pages 810–827, Seattle, Washington, August 1988. MIT press.

[Naur60] P. Naur (Ed.) Report on the Algorithmic Language ALGOL 60. *Communications of the ACM* 3(5):299–314, May 1960

[Pfenning91] Frank Pfenning. Logic programming in the LF logical framework. In *Logical Frameworks*, pages 149–181. Gerard Huet and Gordon Plotkin, editors, Cambridge University Press, 1991.

[RC86] Jonathan Rees and William Clinger (Eds.). Revised^3 Report on the Algorithmic Language Scheme. *SIGPLAN Notices* 21(12):37–79 December 1986.

[Reppy91] John H. Reppy. An Operational Semantics of First-class Synchronous Operations. Cornell Tech Report TR91-1232, August 1991.

[Syme93] Donald Syme. Reasoning with the Formal Definition of Standard ML in HOL. *Higher Order Logic Theorem Proving and Its Applications*, Springer-Verlag Lecture Notes in Computer Science, Vol. 780, August 1993, pages 43–60.

[Tofte90] Mads Tofte. Type Inference and Polymorphic References. *Information and Computation* 89:1–34, 1990.

[Tofte96] Mads Tofte, ed. Detailed list of differences between *Not the Definition of Standard ML, Version 7.3.3, and The Definitions of Standard ML*. Unpublished manuscript, March 1996.

[vW60] A. van Wijngaarden. Revised Report on the Algorithmic Language Algol 68. *Acta Informatica* 5:1–236, 1975.

[Wirth74] Niklaus Wirth. On the Design of Programming Languages. *Proc. IFIP Congress 74*, pages 386–393, North-Holland, Amsterdam, 1974.

[Wright93] Andrew Wright. Polymorphism for Imperative Languages without Imperative Types. Rice University Technical Report TR93-200, February 1993.

[WF92] Andrew Wright and Matthias Felleisen. A Syntactic Approach to Type Soundness. Rice University Department of Computer Science Technical Report TR91-160, June 1992.

[Young89] William Young. A Mechanically Verified Code Generator. *Journal of Automated Reasoning.* 5(4):493–518, December 1989.