Typed Intermediate Representations for Compiling Object-Oriented Languages

by

Joseph C. Vanderwaart

A Thesis
Submitted in Partial Fulfillment of
the Requirements for the Degree of Bachelor of Arts with Honors
in Computer Science

WILLIAMS COLLEGE
Williamstown, Massachusetts
May 24, 1999

Revised July 9, 1999

Abstract

In the past few years a large body of work has developed on the use of typed intermediate languages in compilers. It has become apparent that the retention of type information in the intermediate representation of a program is useful for ensuring compiler correctness and facilitating optimizations. The use of intermediate languages resembling typed λ-calculus in the compilation of functional languages like ML and Haskell has been particularly successful, but not much has been done on using this kind of intermediate format for non-functional languages.

Meanwhile, TOOPL, TOIL and $\mathcal{LOOM}$ have been developed as object-oriented programming languages with static type-safety and semantic foundations firmly in mind. Encodings for the object and class constructs of these languages in λ-calculus are known, and consideration of these encodings has proved fruitful in their design and implementation.

In this thesis, our intention is to exploit the work on the semantic foundations of object-oriented languages and on typed intermediate languages for compilation, in an attempt to design a compiler for $\mathcal{LOOM}$. Our translation makes use of two intermediate languages, each based on a version of polymorphic typed λ-calculus and similar enough to the intermediate formats of other compilers to make the use of a previously developed back-end possible in principle. We outline the design of our intermediate languages and the translations involved in our proposed compilation strategy, and we discuss our prototype implementation which compiles a subset of $\mathcal{LOOM}$ into λ-calculus for subsequent interpretation.

Contents

List of Figures

Chapter 1

Introduction

1.1 Background

In the Williams College honors thesis that introduced the object-oriented language TOIL in 1993, [vG93], it was observed that many of the features of the language which were very simply expressed in the formal semantics might be difficult to compile efficiently. It was further suggested that the range of complicated features like polymorphism included in the language definition should be limited as much as possible, to minimize the burden placed upon those who were eventually to write the compiler. Nevertheless, the very next year a Williams College senior thesis [Sch84] introduced PolyTOIL, which extended TOIL with explicit parametric polymorphism. Three years later, PolyTOIL became $\mathcal{LOOM}$ with the thesis [Pet96], in which a module system was added to facilitate the writing of large programs. $\mathcal{LOOM}$ did also simplify PolyTOIL by eliminating subtyping and relying only on matching, but by this time it was too late to stop the expansion. The minimalist functional language TOOPL, originally little more than a convenient shorthand for a denotational model of objects and classes, had evolved into an imperative language with bounded polymorphism, hidden mutable instance variables, self types and a full-blown module system.

The growth and development of the $\mathcal{LOOM}$ family of languages has been remarkably fruitful and instructive. Several important safety and decidability results have been proven about the languages, and many insights into difficult language design problems have been gained through continued work with the object-oriented type systems. But in many respects, the progress of research focusing on these languages has been hindered by the lack of a compiler. The interpreters that were implemented in the course of this research, while they did provide constructive proof of the tractability of the type-checking problems and insight by experience into the underlying semantics, were not useful for running large programs. As the focus of the work on $\mathcal{LOOM}$ moved toward module systems and programming in the large, it became more and more clear that the slow and clumsy execution provided by an interpreter written in Standard ML would no longer be sufficient. In order to justify any claims about the convenience of new language constructs in practice, or about modularity or separate compilation, a compiler was needed. But the problem observed by van Gent in 1993 remained: how could a compiler be constructed for such a complicated, and quickly evolving, family of languages?

In this thesis, we will argue that the solution is to exploit the work that has been done on the semantic foundations of $\mathcal{LOOM}$ and its predecessors. We will outline a compilation strategy for $\mathcal{LOOM}$ that makes use of typed intermediate representations based on polymorphic λ-calculus. Because of the tight relationship between the semantics of $\mathcal{LOOM}$ and the known encodings of objects in such a theoretical framework, translation of $\mathcal{LOOM}$ programs into a typed functional intermediate language proves to be remarkably simple compared to the task of directly generating machine code for the complex constructs of the language. We will also discuss related work on typed intermediate languages, and argue that the efficiency penalties paid for translation of imperative programs into a functional intermediate representation are not as large as they seem. Rather, we claim that recent research on compilation of functional languages shows that significant optimizations can be performed on λ-calculus-like languages, resulting in good run-time performance. We have implemented the initial stages of a prototype compiler for a subset of $\mathcal{LOOM}$, based on the translations we specify in this thesis.

In the remainder of this chapter, we will describe some related work on compilation using typed intermediate languages. In Chapter 2 we will provide a crash course on $\mathcal{LOOM}$, giving particular emphasis to those constructs which are handled by our prototype compiler. Chapter 3 gives an overview of our compiler and describes IL, the intermediate language into which $\mathcal{LOOM}$ programs are translated. In Chapter 4 we describe our other intermediate language, SIL, and give the translation algorithm we use to translate between SIL and IL. In Chapter 5 we give the translation of $\mathcal{LOOM}$ into SIL. Chapter 6 summarizes and evaluates our results, gives some future directions for related work, and concludes.

1.2 Related Work

There has been a great deal of work done on the use of typed intermediate languages for compilation, although most of this work has focused on functional, rather than object-oriented, source languages. In this section we will discuss two major Standard ML compilers, the TIL/ML compiler and the FLINT architecture which has been integrated into Standard ML of New Jersey. We will also discuss other work on compilation and optimization of functional languages using functional intermediate representations, and some work on the type system of the Java Virtual Machine.

1.2.1 TIL

The TIL/ML compiler, developed at Carnegie Mellon and Cornell Universities, is a compiler for Standard ML that makes use of a typed intermediate language to deal with some of the difficulties that arise when compiling polymorphic functions [TMC+96]. In contrast to the original Standard ML of New Jersey compiler, in which all types were dropped from expressions after a certain stage of the compilation, TIL preserves typing information all the way through. The reasons for doing this include the possibility of more reasonable representation choices for polymorphic code and the ability to use compile-time type information to make garbage collection more efficient.

TIL Compilation

The first phase of the TIL compiler is the translation of Standard ML programs into a polymorphic, typed, functional intermediate language called λ_i^{ML} or Lmli. Lmli is based on predicative polymorphic lambda-calculus, and contains mechanisms for dealing with polymorphism efficiently, including a `typecase` construct that is used in places where the program must decide at run-time how to handle a value whose type is not statically known. The Lmli representation is then transformed into a subset of Lmli which restricts the form of expressions by requiring all intermediate values in any computation to be named and removing almost all nested `let`-expressions. Because of its simplicity, this subset (called Bform) is more amenable to optimization. After this translation has been performed, the Bform code undergoes various optimizations including dead-code elimination, uncurrying, inlining, constant folding and the elimination of redundant comparisions and switches. The optimizations are applied iteratively several times, and then closure conversion (which we will discuss shortly) is performed.

After escaping functions have been converted to explicit closures, the program is translated into an untyped language. Polymorphic functions and `typecase` expressions are converted into term-level functions and `switch` expressions, and types become terms containing representation information. This information is used at run-time to inform the garbage collector of what kind of data is being stored in any particular variable. The untyped code is then translated into an assembly-language-like representation, after which register allocation is performed and actual assembly code is produced.

Typed Closure Conversion

Closure conversion is the method employed by ML compilers to translate higher-order functions into a form that can be efficiently compiled and executed. In languages where functions cannot be nested, or where they cannot be passed as parameters or returned as results of other functions, it suffices to represent a function at run-time just as a pointer to the start of its code. In languages like ML, however, this is not possible since the source code for a function may contain free variables whose locations cannot be "compiled in" because they may change each time the enclosing block is entered. Furthermore, if a function is to be returned from another function, it cannot count on finding the values it needs in any stack frame, since that frame may disappear by the time the function is called. Closure conversion modifies all nested functions that contain free variables so that they take an extra parameter, an environment record, which will contain the values of the free variables. Once this has been done, the function no longer depends on the context of its enclosing block and can be hoisted to the outermost level of the program. Each time the block that enclosed the function is entered, an environment record can be created with the appropriate values and packed with the code pointer as a *closure*. Any calls to the function are then rewritten to pass the environment parameter in addition to any user-specified parameters for the function.

The early SML/NJ compilers removed all type information from a program starting at the closure-conversion stage [AM87, AM91, App92]; the designers of TIL wanted to avoid this, but there were some complications to proper representation of closures in a typed language, particularly in the presence of polymorphism. The report [MMH95] gives a detailed account of typed closure conversion. In both the monomorphic and polymorphic cases, the process is considered as two stages: the first, abstract closure conversion, takes a program in a lambda-calculus-like source language and converts it into a format in which the operation of creating a closure from the code of a function is explicitly represented. The second stage, closure representation, translates this result into a language without explicit closures, where no functions have free variables. Closures are represented as pairs containing a closed lambda expression (possibly represented in the eventual machine language program as a pointer to the code of the function) and an environment value, which is usually a record whose type depends on what variables appeared free in the original function. To make this translation preserve type soundness, this final target language incorporates existential types, which are used in the representation of closures so that different functions may use different environment types but the types of closures for any functions of the same type in the source language will coincide.

The polymorphic case is similar, but it is complicated by the fact that any function code may contain free type variables as well as free term variables. This is dealt with by packing a "type environment" with each closure, the kind of which is hidden using a higher-order existential. Unfortunately, hiding all of this type information is not necessarily what is desired, since the type of the function being represented may itself depend on one of the free type variables. To fix this, Minamide *et. al.* are forced to use translucent (or "manifest") existential types, so that the information necessary to make uses of the closure well-typed is revealed. Crary, Weirich and Morrisett [CWM98] argue that this is only a problem because of the dynamic type-passing semantics of TIL. In a type-erasure semantics, closure conversion need not be performed for type variables since application of a polymorphic function to a type will have no run-time effects. This fact is also exploited in [MWCG97], which we discuss later.

1.2.2 FLINT

The more recent versions of Standard ML of New Jersey have been modified to replace the original intermediate representation with a new, fully typed intermediate language called FLINT [Sha97a, Sha97b]. Like TIL, the FLINT language is based on predicative polymorphic lambda-calculus. The designers of FLINT, however, present their intermediate format as part of a compiler architecture suitable for compiling any higher-order typed language, including domain-specific application languages and object-oriented languages such as Java in addition to ML and its close relatives [Sha97b]. It is argued that the existence of a universal compiler architecture would improve the interoperability of different programming languages and the portability of programs from one hardware architecture to another. The idea behind the FLINT architecture, then, is to allow front-ends for different languages — ML, Java, Haskell, and so on — to translate their source programs into the FLINT intermediate format, and provide a middle-end optimizer and several back-end code generators (one for each of an unlimited number of target architectures). All these parts are intended to be completely modular, making it a simple matter to retarget a compiler to whatever platform is desired, and a single compilation manager could be used to control the compilation of programs with components written in several different languages. While SML/NJ is the only compiler so far to adopt FLINT, front-ends for other languages (notably Java) are being developed actively by the FLINT group.

The core FLINT language is a straightforward, predicative, higher-order lambda calculus. The full language augments the core with several primitive type constructors such as exceptions, sum types, ML-style recursive datatypes and more. Since FLINT is intended to be useful for all kinds of programming languages and applications, many practical primitive types and constructors are included, including different sizes of integers and floating-point values, mutable reference cells, arrays, vectors, first-class continuations and suspensions for lazy evaluation.

1.2.3 Other Functional Language Compilers

The Church Project has developed a typed intermediate language that integrates control-flow information into the representation of a program [DMTW97]. All arrow types in the language, which is presented as a tool for compiling ML, are annotated with source and sink labels corresponding to all the places (lambdas) where a value of that type might have come from and all the places (application sites) where it might be applied to a value. Each lambda and each function application site in a program is given a unique label to be used for this purpose. The language also contains "virtual" union and intersection types, which function like the conventional sum and product types except that these new types are intended to have no run-time significance. Rather, they are a notation used by the compiler to keep track of the different possible sources and sinks for a functional value and generate the proper code to deal with function calls as efficiently as possible. Some, but not necessarily all, of the virtual tuples and virtual variants in a program may be converted to actual tuples or variants in later stages of the compilation. [DMTW97] gives a detailed account of the several steps involved in translating and optimizing a program using this flow-annotated typed language.

There has also been work on transformation-based optimization of intermediate code in Haskell programs. [Pey98] describes the optimizations performed by the Glasgow Haskell Compiler, all of which are implemented as transformations on the intermediate representation of programs. The optimizations include avoiding the creation of a thunk to suspend a computation that will certainly be performed anyway, rearranging `case` statements so that alternatives which will never be executed can be thrown away, explicit boxing and unboxing of numeric values, and many others. A compiler for a lazy language such as Haskell has a fair amount of freedom in moving code around, since hardly anything has any side-effects and the programmer is not expected to care in what order expressions are evaluated. The Glasgow Haskell Compiler, therefore, has many "code motion" transformations that float chunks of code around in an attempt to minimize the number of times an expression must be evaluated, the size of the thunk allocated for a computation, and so forth. While the lazy evaluation semantics of Haskell makes the optimization segment of a compiler for that language very different from that of an ML compiler, it has also been argued [PLST98] that this is needless and that in fact a common intermediate language can be used for both.

A slightly different approach has been taken to compilation of functional languages by type-soundness-preserving transformations of polymorphic languages in [MWCG97]. In that paper, Morrisett *et. al.* give a formal definition and correctness proof of a transformation from polymorphic lambda-calculus (System F) to a low-level typed language similar to the assembly language of a RISC architecture. The compilation is simply the composition of conversion to continuation-passing style (for more on CPS see also [App92]), closure conversion, hoisting of lambda-expressions to the outermost level, introduction of explicit heap allocation and finally code generation. Because of the nearly imperative style of the closure-converted, continuation passing representation of the program, the code generation is not very complicated. The type-preserving properties and operational correctness of all the steps in this translation are proven. The code written in continuation-passing style does not make explicit use of a stack for activation records; it is possible, however, to force a stack-based discipline upon the code by changing the way in which continuation values are handled [MCGW98].

1.2.4 The Java Virtual Machine Language

The language Java [AG96] is compiled, not into native machine code, but into a bytecode format known as JVML [LY97]. While JVML is not really an intermediate language — nothing comes after it in the compilation process — it is worthy of some attention. In order to make interpretation of compiled Java programs efficient and safe, the JVM bytecode format contains information about the types of variables. Before the code of a class is executed, it must pass the bytecode verifier, the job of which is to ensure that the bytecode is legal and that the types of objects and other values are respected. While this is beneficial in terms of the security issues inherent in the internet-based deployment of Java programs and of the efficiency with which they can be executed once loaded, there are some problems from our point of view. The first is that the type system of JVML very closely mirrors that of Java itself: classes function as the types of objects, all subtypes must be declared as such, method types may never change in subclasses, and there is no polymorphism. These restrictions would make it difficult to use JVML as a target or an intermediate language for more flexible languages like $\mathcal{LOOM}$. In fact, all the work of Burstein [Bur98] and the implementors of other extensions of Java would have to be repeated for $\mathcal{LOOM}$, resulting in a clumsy translation producing inefficient code.

The second reason JVML is not useful for us is that it is difficult to reason about. In particular, the dynamic linking and loading necessary to the uses for which JVML was designed make its type system and semantics very complicated. The ability of one class in a Java program to be rewritten without recompiling other classes that depend on it brings up complex questions of binary compatibility [DWE98]. There are also difficulties having to do with verifying simple method bodies: the rules necessary for guaranteeing that newly-created objects are always initialized before any messages are sent to them [FM98] and that exception handlers do not get confused about the types of local variables or operand stack contents [SA98] are quite complicated, and the standards built into the original JVM verifier are in many cases not strong enough to guarantee safety of Java programs.

1.2.5 Summary

There is a large body of work on the compilation of functional languages, and in particular on the use of typed intermediate languages based on lambda-calculus. Transformations from lambda-calculus to assembly language have been defined rigorously and proven correct. Most of the work on compiling useful source languages has focused on functional languages, notably Standard ML, and in particular two full-scale optimizing compilers, TIL and FLINT, for SML have been produced. Both use a predicative polymorphic lambda-calculus as the core of their intermediate languages. TIL is solely an ML compiler, while FLINT is intended as a universal compiler architecture that can be used to construct production-quality compilers for any higher-order typed language, including object-oriented languages.

Extensive use of a typed intermediate language is also made by the Glasgow Haskell Compiler, and it has been suggested that Haskell and ML compilers might even be able to share a common intermediate language. Finally, we have seen that while the Java Virtual Machine does provide a possibly useful partially-compiled format for object-oriented languages, the JVM is too restrictive and not well enough understood for our purposes in compiling $\mathcal{LOOM}$.

Chapter 2

Features of $\mathcal{LOOM}$

In this chapter, we will give a description of the features of the $\mathcal{LOOM}$ language, focusing primarily on those which have been implemented in the prototype compiler. For more details on $\mathcal{LOOM}$ we refer the reader to the papers [BFP97] and [BFSvG95] and the thesis [Pet96].

2.1 Program Organization and Control Structures

$\mathcal{LOOM}$ is an imperative language, and so most of the work in a $\mathcal{LOOM}$ program is done in lists of statements, which are written separated by semicolons as in Pascal. The most basic $\mathcal{LOOM}$ statements are the assignment and the procedure call; more complex statements are formed by grouping statement lists with the control structures `if`, `while` and `for`. An abbreviated grammar for $\mathcal{LOOM}$ statements is given in Figure 2.1.

The overall layout of a $\mathcal{LOOM}$ program is very Pascal-like. Declarations and statements are arranged into blocks, where a block consists of a list of constant definitions, a list of variable definitions, and a statement list. Each procedure, function and method defined in a program can have a complete block, including its own constant list. Functions and classes are first-class values in $\mathcal{LOOM}$, so they can be (and usually are) defined in the constant section of a block; this means that procedures, functions and methods can define their own local subprograms which are visible only within the scope where their definitions appear. In addition, the last statement in any block used to define a function must be a `return` statement which specifies the value the function should return. The `return` statement may not appear anywhere else in any statement list, and $\mathcal{LOOM}$ does not provide a way to return immediately from the middle of a function's statement list. Finally, at the top level of a program there may be a type definition section, which is where all the object types used in the program are defined. An abbreviated grammar for the block structure of simple programs is given in Figure 2.2.

$$
\begin{aligned}
\text{stmt} &\rightarrow \text{id } \texttt{:=} \text{ expr} \\
&\mid \text{id } \texttt{(} \text{ paramlist } \texttt{)} \\
&\mid \texttt{if} \text{ expr } \texttt{then} \text{ stmts } \texttt{end} \\
&\mid \texttt{if} \text{ expr } \texttt{then} \text{ stmts } \texttt{else} \text{ stmts } \texttt{end} \\
&\mid \texttt{while} \text{ expr } \texttt{do} \text{ stmts } \texttt{end} \\
&\mid \texttt{for} \text{ id } \texttt{:=} \text{ expr } \texttt{to} \text{ expr } \texttt{do} \text{ stmts } \texttt{end} \\
&\mid \texttt{for} \text{ id } \texttt{:=} \text{ expr } \texttt{to} \text{ expr } \texttt{by} \text{ expr } \texttt{do} \text{ stmts } \texttt{end} \\
&\mid \ldots \\
\text{paramlist} &\rightarrow \epsilon \\
&\mid \text{nonemptyparamlist} \\
\text{nonemptyparamlist} &\rightarrow \text{exp} \\
&\mid \text{nonemptyparamlist } \texttt{,} \text{ exp}
\end{aligned}
$$

Figure 2.1: Abbreviated grammar for $\mathcal{LOOM}$ statements.

$$
\begin{aligned}
\text{block} &\rightarrow \text{constlist varlist } \texttt{begin} \text{ stmts } \texttt{end} \\
\text{funcblock} &\rightarrow \text{constlist varlist } \texttt{begin} \text{ stmts } \texttt{;} \ \texttt{return} \text{ expr } \texttt{end} \\
\text{varlist} &\rightarrow \epsilon \\
&\mid \texttt{var} \text{ vardecls} \\
\text{constlist} &\rightarrow \epsilon \\
&\mid \texttt{const} \text{ constdefs} \\
\text{constdef} &\rightarrow \text{id } \texttt{=} \text{ expr } \texttt{:} \text{ typeexp } \texttt{;} \\
&\mid \texttt{procedure} \text{ id } \texttt{(} \text{ formallist } \texttt{)} \text{ block } \texttt{;} \\
&\mid \texttt{function} \text{ id } \texttt{(} \text{ formallist } \texttt{)} \ \texttt{:} \text{ typeexp funcblock } \texttt{;} \\
&\mid \ldots
\end{aligned}
$$

Figure 2.2: Grammar giving block structure of $\mathcal{LOOM}$ programs.

```
type
     Point = ObjectType
                 getX : func():integer;
                 getY : func():integer;
                 norm : func():real;
                 moveTo : procedure( integer, integer );
              end;

const

     procedure transPoint( p : Point,
                           dx : integer, dy : integer )
     var
          oldx, oldy : integer;
     begin
          if p <> nil then
              oldx := p.getX();
              oldy := p.getY();
              p.moveTo( oldx + dx, oldy + dy );
          end;
     end;
```

Figure 2.3: An example of an object type and a message send. Objects of type `Point` have methods called `getX`, `getY`, `norm` and `moveTo`. The procedure `transPoint` accepts a `Point` and two integer values, then uses the object's `getX`, `getY` and `moveTo` methods to translate it by the amount specified.

2.2 Objects

Most of the interesting features of $\mathcal{LOOM}$ are to be found in the way it handles objects. Since the precise definitions of the nature and origin of objects vary from one object-oriented language to the next, this section will explain the meanings of the important object-oriented terminology as it pertains to $\mathcal{LOOM}$. Section 2.3 will describe the specification and creation of objects in $\mathcal{LOOM}$, and those following will discuss some of the more advanced features of the language.

A $\mathcal{LOOM}$ object should be thought of as consisting of two parts: its *state*, which is represented by a collection of mutable *instance variables*, and its *methods*. Methods are procedures and functions associated with an object; their bodies may refer to the mutable instance variables but in all other respects they are fixed for the lifetime of the object.

Unlike many other object-oriented languages, notably Java [AG96] and C++ [ES90], which allow instance variables to be declared "public," $\mathcal{LOOM}$ restricts access to all instance variables to within method bodies, effectively declaring all of them "protected." Thus the visible part of an object in any scope other than a method block is just its collection of methods; in $\mathcal{LOOM}$ this view is captured by the object's type.[1] As the example in Figure 2.3 shows, an object type is essentially nothing more than a list of method signatures, where a method signature is an identifier (the method's name) and a type expression (the method's type), which must be a procedure or function type.

In the example, an object type called `Point` is defined; this might be the type of an object used to store the coordinates of a point on the display in a graphics application. The object type does not allow us to see what type of data is used to represent this information, and so we can only interact with the object through the interface provided by its methods. In the case of the `Point` type, we are provided with `getX` and `getY` methods, which presumably return the x and y coordinates of the point, respectively, a `norm` method which returns the point's distance from the origin $(0, 0)$, and finally a `moveTo` method which changes the point's internal state so that it represents the new location whose coordinates we specify. Insofar as an object type specifies only the names and types of methods, omitting any information about instance variables or the way methods are implemented, it is similar to an "interface" in Java. Unlike Java, however, $\mathcal{LOOM}$ determines equality of types structurally rather than by name. Thus, for example, if the definition in Figure 2.3 were repeated and given a different name (say `Point2`) then the types `Point` and `Point2` would be interchangeable.

The example in Figure 2.3 also defines a procedure called `transPoint` that accepts a `Point` object and translates it by the given amounts. Notice the use of the dot notation, common to many object-oriented languages, for invoking a method of the object `p`. The act of calling an object method is traditionally known as *message-sending*, since by calling the method we are in effect "asking" the object for some information or to perform some action. $\mathcal{LOOM}$ actually treats message sends a little differently from many languages in that the expression `p.getX()` is interpreted as "call the function `p.getX` with the empty parameter list," which means that `p.getX` is just the description of a function. This function – the method that has been "extracted" from the object – can be passed around just like any other value, and it will continue to refer to the same object `p` from which it was taken.

One last thing to note about the `transPoint` example is the test in the function body to assure that `p <> nil`. $\mathcal{LOOM}$, like Java, Eiffel [Mey92], and Object Pascal [Tes85], but unlike C++, uses the rule that all parameters, variables and other expressions declared to have object types really denote *references* – that is, pointers – to the memory where the

[1] In fact it is possible for $\mathcal{LOOM}$ objects to have methods that are marked `hidden`, which is the equivalent of Java's `protected`, but this is not implemented in the prototype compiler and so we will not discuss hidden methods further.

object resides. This makes it easier for objects to refer to each other in the natural way (a GUI component with a `parent` instance variable wants to point to its parent, not contain a copy of its parent) and makes the passing of objects as parameters more efficient and intuitive. As a result, however, one must often be careful of aliasing issues (assigning `o1 := o2` makes o1 and o2 refer to the exact same object, not two identical ones) and of attempting to send messages to an object that is not there. Messages to `nil` cannot be detected by $\mathcal{LOOM}$'s semantic checking, and if one occurs at run-time it results in an error.

2.3 Classes

In the previous section we discussed the types of objects and how to access their methods, but did not address the issue of how the structure and behavior of an object is specified. Also, we explained that a variable of an object type holds a pointer to the space in memory where the object resides, but we did not discuss how this space gets allocated. In a $\mathcal{LOOM}$ program, the detailed description of an object's instance variables and methods is given in a *class*, which is then used to create the object.

Many other well-known object-oriented languages (Java, C++, Eiffel, *etc.*) use classes to specify the instance variables and methods of an object. These languages differ from $\mathcal{LOOM}$ in that the name of the class specifying an object is also used as its type, while in $\mathcal{LOOM}$ this is not allowed. The type of an object is determined instead by taking the names and types of the methods its class defines and putting them together into an `ObjectType`. If we have given a name to this particular combination of method signatures in the type definition section of our program, then we can use that name to refer to the type of objects generated by the class. To see an example of this, consider the code fragment in Figure 2.4.

In the example shown in the figure, a class called `PointClass` is defined. The class definition specifies objects that have two integer instance variables `x` and `y`, and the same four methods as were given in the `Point` example of the last section. In fact, $\mathcal{LOOM}$ infers a type for the objects generated by this class that is identical to the one we have named `Point`. Thus it is legal to assign an object from `PointClass` to a variable declared to have type `Point`, and this is exactly what the program body does. A new object is created by first passing the initialization parameters (in this case, the coordinates for the new point) to the class, and then passing the resulting "instantiated class" to the `new` operator. The result of this operation is the allocation of enough space in memory to store the state of a `PointClass` object, the initialization of its instance variables as specified in the class, and the creation of a *method suite* consisting of the four functions the class defines. A reference to this object is then available to be assigned to the variable `p`.

Note that the method bodies defined in `PointClass` are allowed to refer to the instance variables. In fact, the instance variables of an object are always visible in the scope of any

```
const
     class PointClass( initx : integer, inity : integer )
     var
          x = initx : integer;
          y = inity : integer;
     methods
     visible
          getX = function() : integer
          begin return x; end;

          getY = function() : integer
          begin return y; end;

          norm = function() : real
          begin return sqrt( int2real(x*x + y*y) ); end;

          moveTo = procedure( newx : integer, newy : integer )
          begin
               x := newx;
               y := newy;
          end;
     end;

var
     p : Point;
     r : real;

begin
     p := new( PointClass(3,4) );
     r := p.norm()   -- should be 5.0
end;
```

Figure 2.4: An example of a class and of creating an object. It can be seen by inspection of the types in `PointClass` that it generates objects of the type `Point` defined in Figure 2.3.

of that object's methods. Furthermore, any method in an object can call any other method in the same object, either by just using its name as though it were a normal function or by sending the appropriate message to the pseudovariable `self`. The name `self`, which is only meaningful inside a method body, refers to the entire object executing the method, so it is always the case that `self` is a non-null object reference having all the methods defined in the class where the occurrence is found. As we will see in the next section, however, it is not always true that `self` has *only* the methods defined in the class where it appears; in particular, the method code might be executed on behalf of a subclass which has added new methods. Because of this, the $\mathcal{LOOM}$ type system forces the programmer to be careful what assumptions she makes about the use of `self`.

It is also significant that the name of the type `Point` does not appear anywhere in the definition of `PointClass`. It need not, because the typechecker can tell from the method signature information in the `PointClass` definition that any object generated by that class will have exactly the methods specified by `Point`. The implication of this is that there may be many different classes generating objects of type `Point`. In particular we could define a different class, say `NormedPointClass`, that had an extra instance variable to keep track of its distance from the origin, making the `norm` method more efficient. This class need not be related to `PointClass` in any explicit way, but if it has exactly the same methods with the same signatures as the definition of `Point` it will generate objects of type `Point`. A similar situation is allowed in Java, where a single interface can be implemented by many distinct classes. One difference here, though, is that in $\mathcal{LOOM}$ we can only say `PointClass` generates `Point` because it has *exactly* the same methods, whereas Java would only require that the class have *at least* the methods of the interface. Also, $\mathcal{LOOM}$ does not require that the programmer explicitly state that a class is supposed to generate objects of a particular type.

Finally, notice that in the example code fragment the class `PointClass` is given in the constant definition section of the program. This is because in $\mathcal{LOOM}$, classes are first-class values. They have types of their own and can be passed to procedures, returned from functions or assigned to variables just like any other values in the language. Of course, most programs will not have any reason to do this.[2] Since most classes in practice are defined in the constant section of a block and never passed as parameters or assigned to variables, the syntax for a class definition in the constant section omits any specification of the type of the class, which must then be inferred by the type-checker. The type of a class should not be confused with the type of the objects it generates: a class type must specify the types of instance variables as well as the types of methods, since this type information is needed in order to properly generate new objects from the class.

[2] Actually, the passing of initialization parameters to a class is implemented by wrapping a parameterless class definition in a function that accepts the parameters and returns an instantiated version of the class. While this "syntactic sugar" treatment of class parameters does simplify some aspects, it does have some unfortunate consequences such as rendering it impossible for a class to accept parameters of type `MyType`.

2.4 Inheritance and Matching

Inheritance is a concept fundamental to object-oriented programming. The basic idea is to promote code re-use by allowing the programmer to use a class she has already written to derive another class, called a *subclass*, that is largely the same. When creating a new class in this way, all the programmer has to do is give the changes between the old class and the new one. Only certain kinds of changes are permissible: new instance variables and methods may be added, existing methods may be redefined by giving a new body to replace the old one, and the initial values of existing instance variables may be changed. Even with these restrictions on how a class can be modified when deriving a subclass, major complications are introduced into the type-checking of the language by the fact that method bodies written for one class may end up being used in another. As a result, the rules for type-checking *all* classes must take the possibility of inheritance into account.

An example of inheritance is given in Figure 2.5, which shows a new class, called `ColorPointClass`, being derived from the `PointClass` class of the previous section. A third class, `SpecialPointClass`, is then derived from `ColorPointClass`. Both of these classes generate objects of the type `ColorPoint` defined at the start of the code fragment, but objects produced by `SpecialPointClass` always start out as blue points at the origin and color themselves red every time they are moved.

As the figure illustrates, the definition of a class derived by inheritance looks very much like a normal class definition, except that the *superclass* – the class from which a subclass is being derived – is specified using the `inherit` keyword. If the superclass expects any parameters, their values must be specified; these values may come from the subclass' own parameters, as in `ColorPointClass`, or they may be constants as in `SpecialPointClass` or any other expressions of the proper types. If the new subclass is to have any new instance variables in addition to the ones being inherited from the superclass, these are specified in a `var` section; if there are to be any new method definitions, these go in a `methods` section just like in a regular class definition. Note that because `SpecialPointClass` redefines the `moveTo` method – that is, overrides it by providing a new implementation – it must include the clause `modifying moveTo` in its header. This language feature protects the programmer from accidentally redefining a method he didn't know was in the superclass.

In any new method bodies that are given in a class derived by inheritance, all of the instance variables and methods of the class, even those inherited automatically from the superclass, are visible.[3] Also, if a method that is inherited unchanged ever calls a method that was overridden, the new version of the method should be executed. While this is intuitive and convenient for the programmer, we will see in later chapters that it complicates the formal semantics of the language considerably.

[3]All of the methods in the superclass are visible to subclass method bodies, even those marked `hidden` in the superclass. Thus `hidden` is more like Java's `protected` access specifier than like `private`.

```
type
     Point = ObjectType ...   -- As in previous example
     ColorPoint = ObjectType include Point
                      getcolor : func():string;
                      secolor : proc( string );
                  end;
const
     class PointClass ...   -- As in previous example

     class ColorPointClass( initx : integer, inity : integer,
                            initc : string )
     inherit PointClass( initx,inity )
     var color = initc : string;
     methods
     visible
          getcolor = function() : string
          begin return color; end;

          setcolor = procedure( newcolor : string )
          begin color := newcolor; end;
     end;

     class SpecialPointClass
     inherit ColorPointClass( 0,0,"blue" ) modifying moveTo
     methods
     visible
          moveTo = procedure( newx : integer, newy : integer )
          begin
               x := newx;
               y := newy;
               color := "red" ;
          end;
     end;
```

Figure 2.5: An example of defining classes by inheritance. Note that both `ColorPointClass` and `SpecialPointClass` generate objects of type `ColorPoint`.

We are now in a position to explain a somewhat cryptic comment we made in section 2.3 about the type attributed to the pseudovariable `self`. If, in a method body, the programmer makes some reference to `self`, then she can be certain that she is dealing with a non-null object that has *at least* all the methods with all the same signatures as the class in which that method body resides. However, she cannot correctly infer that `self` is an object of the type generated by the class she is writing: the method that refers to `self` might be inherited by, and executed on behalf of, another class that adds some new methods. It is, therefore, impossible to statically know the exact type of any occurrence of `self`. To deal with this problem, $\mathcal{LOOM}$ gives a special name, `MyType`, to the type of `self`, and the programmer can use `MyType` as the type of any instance variable, method parameter, or method return value in the class. To prevent the programmer from writing any code that might lead to type errors when methods are inherited, the $\mathcal{LOOM}$ type-checker enforces the restriction that any operations performed on `self` or any other value of type `MyType` must not assume any more about the type than that it has at least the methods of the class in which the occurrence appears.

Now we must discuss the type of the objects produced by subclasses. It is easy to determine by inspection that an object generated by `ColorPointClass` will have exactly the same methods as an object of type `Point`, plus the new methods `getcolor` and `setcolor`. This is expressed in our definition of the type `ColorPoint`, where the `include` keyword represents a convenient shorthand. The $\mathcal{LOOM}$ typechecker immediately expands this abbreviation and defines `ColorPoint` to be a full object type containing all six methods explicitly – we could have defined it that way to begin with, but the `include` shorthand makes our intentions clearer and additionally reduces the amount of changes we must make if we wish add a new method to `Point`.

It is clear that the objects generated by any class defined using inheritance will have all the same methods as objects from the superclass, and possibly more. We express this by saying that the type of the subclass objects *matches* that of the superclass objects; in general one object type `T1` matches another object type `T2`, written `T1 <# T2`, if `T1` has all the same methods, with all the same signatures, as `T2`, plus possibly more. This is similar to the notion of *subtyping* that is present in many object-oriented languages, but there is an important difference. In Java, C++ and Eiffel one is allowed to use objects of a subclass as though they were elements of a superclass: that is, assign them to variables declared to have the superclass type, pass them as parameters to functions expecting superclass objects, and return them from functions that are supposed to return superclass objects. In $\mathcal{LOOM}$ this is not allowed. The reason for this has to do with the fact that $\mathcal{LOOM}$ programmers are allowed to use `MyType` as the type of a method parameter. It is possible to write a simple program using `MyType` and this kind of "subsumption" rule in which a message ends up being sent to an object lacking the corresponding method, resulting in a run-time error. For more information on these type-safety issues we refer the reader to [BFP97] and [Bru97]; for the purposes of this chapter it suffices to say that $\mathcal{LOOM}$ abandons the convenience of subsumption in favor of the expressiveness of `MyType`.

2.5 More Advanced Features

The $\mathcal{LOOM}$ language incorporates several advanced features to allow greater flexibility in writing programs involving objects and inheritance. Some of these, like polymorphism and the module system, are fairly standard constructs in object-oriented and ADT-style languages. Others, like #-types and groups, are innovative typing constructs specific to $\mathcal{LOOM}$. In this section we will give a brief overview of these features, most of which are not implemented in our prototype compiler and will not be discussed any further in this thesis.

The first of $\mathcal{LOOM}$'s advanced features is bounded parametric polymorphism, similar to the kind found in Eiffel and several proposed extensions to Java, notably Pizza [OW97], Generic Java [BOSW97] and Rupiah [Bur98]. The idea is to allow a function (or a method, or an entire class) to take a type as a parameter in such a way that the type which is given "specializes" the function to act on values of that type. For example, using parametric polymorphism one can create generic homogeneous data structures: a programmer can write, say, a polymorphic `Stack` class and then specialize it as many times as necessary to create stacks containing different types of values by writing `Stack(Point)`, `Stack(OperandDescriptor)`, `Stack(ActivationRecd)`, and so forth. $\mathcal{LOOM}$ permits procedures, functions, methods and classes to be polymorphic, and also provides a convenient "type function" notation for describing the types of objects produced by a polymorphic class. Our prototype compiler does not handle polymorphism, but we do intend to add support for it in the near future, and so our discussion of the translation of $\mathcal{LOOM}$ will show how polymorphic functions may be translated.

Another construct provided by $\mathcal{LOOM}$ that is related to polymorphism is #-types, pronounced "hash types". Briefly, if a program has defined `Point` to be an object type, then by definition `#Point` is a type containing all objects of type `Point` and all objects of any type whose type matches `Point`. Thus, if a variable is declared to have type `#Point`, then objects from any of the classes in Figure 2.5 may be assigned to that variable. This "unification" of all the types matching a given type is similar to the subtyping behavior exhibited by other object-oriented languages. In fact, the extension to Java proposed in [Bur98] introduces "exact types" which work like normal object types in $\mathcal{LOOM}$, while object types as they would normally be written in Java are treated like #-types. The only difference between Java's object types, which permit subtyping, and $\mathcal{LOOM}$'s #-types, has to do with `MyType`. While most operations on objects of #-types are the same as for exact object types, it is forbidden to invoke a method of such an object if `MyType` occurs as the type of a parameter to the method. A more detailed discussion of the reasons for this can be found in [BFP97]. A translation of #-types into our intermediate languages is known, but we have chosen not to implement it at this time.

The thesis [Pet96] introduced a module system for $\mathcal{LOOM}$, designed to support programming in the large. A $\mathcal{LOOM}$ program may consist of several *compilation units*, one of which is the main program and the rest of which are specifications and implementations of modules. The module system of $\mathcal{LOOM}$ is similar to that of Modula-3 [CDG$^+$88], and supports the definition of constants in addition to abstract types, partially revealed types and transparent type definitions. Each module has exactly one *interface*, which specifies which of the definitions that appear in it are to be visible to clients. The module system was later extended [Van97] to weaken the abstraction barrier surrounding each module and permit the extension of classes defined inside.

Finally, a version of the statically-safe virtual types construct proposed for Java in [BOW98] has been added to $\mathcal{LOOM}$ [BV99]. This construct addresses the problem of creating subclasses of mutually referential classes, where straightforward inheritance often fails to preserve the mutual relationship of the superclasses. It is now possible in $\mathcal{LOOM}$ to define a group of several object types all at once, and associate classes with that group. Then when subclasses are defined, all the cross-references between the types are updated, creating a new group which preserves the mutually referential relationships present in the original one. While these `TypeGroup` and `ClassGroup` constructs may be considered a straightforward extension of $\mathcal{LOOM}$'s original treatment of `MyType` and the matching relation, their semantics are a bit more complicated in that they require mutually recursive types, which we have chosen not to support in this first attempt at a $\mathcal{LOOM}$ compiler.

Chapter 3

Our Intermediate Language IL

In this chapter, we will describe in some detail the important features of two typed intermediate languages, the Lmli language of the TIL compiler [TMC$^+$96] and the intermediate language of the FLINT compiler [Sha97b]. We will outline the steps these compilers take in compiling Standard ML, and discuss our intermediate language design.

3.1 The FLINT and TIL compilers

The typed intermediate languages that most influenced our $\mathcal{LOOM}$ compiler design were the TIL [TMC$^+$96] and FLINT [Sha97a, Sha97b] compilers for Standard ML. These two intermediate languages have much in common: each is based on a predicative version of polymorphic λ-calculus, both support similar base types (integers, floating-point numbers, *etc.*) and primitive type constructors (tuples, variants, functions, polymorphism), and both force terms to be written in A-normal form [SW96] to simplify optimizations.

Both the FLINT and TIL systems follow essentially the same compilation strategy. Each begins by parsing and typechecking the source program; this process, called *elaboration*, is complicated by the fact that ML allows the programmer to declare infix operators and change their precedences at will [AM87]. The TIL compiler borrows the front-end of the ML Kit compiler [BRTT93] for this purpose [TMC$^+$96]. The elaborated ML program is then converted to a typed intermediate language and optimizations are performed. After the optimization passes are finished, closure conversion is performed, followed by low-level optimizations and finally code generation.

It has long been considered a good idea to compile programs written in a functional language by transforming them into continuation-passing style, or CPS [App92]. In this style, programs are rewritten so that every function takes, as an extra parameter, another function to which it should pass its result. This extra function, called a continuation, is then called as the last step performed by the original function body; because it is the last thing the function has to do, it is a tail-call and can be implemented as a simple unconditional branch. Because CPS turns function applications into jumps, it makes code look more linear and enables conventional loop optimizations to be performed.

Neither the FLINT nor the TIL compiler actually uses continuation-passing style as such, however. Instead, both translate the terms in their intermediate language into a style referred to as A-normal form, which has been shown [SW96] to be equivalent to CPS in terms of the optimizations that can be performed. Writing terms in A-normal form essentially means naming every sub-expression of a computation using a `let` construct. This has the effect of turning free-form higher-order function applications and complicated arithmetic expressions into linear sequences of simple operations and `let`-bindings, finally either returning a single named value or making a tail call to another function. The result is that the code looks very linear, and since most of the `let`-bindings in a term will be simple operations, they can usually be compiled into simple machine-language instructions, provided all the previously bound variables are stored in registers or convenient memory temporaries. The only catch is that a λ-expression might be defined which refers to one of these temporary values; the task of saving old bindings is handled by performing closure conversion as described in Section 1.2.1.

3.2 $\mathcal{LOOM}$ Compiler Design: A Narrative

Because of the amount of work involved in writing the optimization and code-generation phases of any real compiler, we decided early on that the best course of action for us would be to write our $\mathcal{LOOM}$ compiler as a front-end for either TIL, FLINT or some similar system. In particular, since FLINT was originally intended to be a universal compiler architecture usable for compiling many different source languages [Sha97b], we planned to implement our $\mathcal{LOOM}$ compiler by writing a front-end to translate $\mathcal{LOOM}$ programs into FLINT, and then using the middle- and back-ends of the FLINT compiler to do optimization and generate code. By proceeding this way, we hoped to minimize the amount of needless labor in the construction of our compiler, while at the same time maximizing the number of platforms we could target and the amount of optimization we could perform.

Unfortunately, there are several impediments to our using FLINT effectively for a $\mathcal{LOOM}$ compiler. First of all, polymorphism in FLINT is predicative; that is, the type system of the FLINT language mirrors that of SML [MTHM97] in not permitting polymorphic functions to be passed as parameters and disallowing polymorphic recursion. $\mathcal{LOOM}$, by contrast, does allow these uses of polymorphic functions, which means that a translation of the full $\mathcal{LOOM}$ language would necessarily involve some inconvenient workarounds.

Of more immediate concern, however, was the fact that the type system of FLINT did not include existential types. Existentials are an integral part of most of the well-known encodings of objects [BCP96], where they serve to hide the types of instance variables.

A translation from an intermediate language with existential types to FLINT could have been implemented, but would have required more work than time allowed. Interprocedural dataflow analysis would have been necessary to determine, for each object type in the program, all the classes that could produce objects of that type and hence all the types with which the variable in each existential could be instantiated; existential types could then be replaced by finite ML-style datatypes, which the FLINT language does support [Sha97b]. This labor-intensive transformation, however, proved to be unnecessary: in personal communication with the FLINT group [Sha98] we discovered that they planned to add existential types to their intermediate language. This was motivated by their efforts to write a Java compiler using FLINT, and they too had discovered that existentials were necessary to encode object types. This addition to FLINT, though, was still in the planning stages, and it was unlikely that a version supporting existentials would be available by the time we needed to finish our work.

The recommendation of Shao was that we design our own intermediate language, as similar to FLINT as possible, and use that language design to implement the initial stages of our compiler. In order to be able to debug our translation, it was suggested that we write an interpreter for this intermediate language so that we could run translated $\mathcal{LOOM}$ programs without a real compiler back-end. When an appropriately extended version of the FLINT back-end became available, a translation from our intermediate language into theirs could be implemented to complete the compiler. We took this suggestion, and designed our intermediate language IL to resemble FLINT in many respects, except for the addition of features like existentials and impredicative polymorphism.

The final obstacle to using the FLINT system to write our compiler was that our preferred encoding of classes [BV99] relied on bounded quantification, which was not present in FLINT.[1] To simplify the task of working around this, we designed a second language called SIL, which incorporated subtyping and bounded quantification, so that the translation of $\mathcal{LOOM}$ to the subtyping-ignorant IL language could be broken up into those parts which dealt with $\mathcal{LOOM}$'s language features and those parts which dealt with the elimination of bounded quantification. The translation from SIL to IL is nontrivial and is discussed in Chapter 4; translation from $\mathcal{LOOM}$ to SIL is discussed in Chapter 5.

3.3 The IL Language

In this section we will discuss the details of our intermediate language IL. Section 3.3.1 gives a grammar for the language and discusses the syntax. Section 3.3.2 gives the typing rules for the language. Finally, Section 3.3.3 discusses the dynamic semantics and interpreter.

[1] The paper [Sha97b] mentions that bounded quantification might appear in a later version of FLINT.

3.3.1 Syntax of IL

The grammar for IL is shown in Figure 3.1. The types are essentially those of the polymorphic λ-calculus F_ω [Gir72, Rey74] extended with tuples, sums and records. Since we allow the definition of type operators, or functions from types to types, we must ensure that the different sorts of type constructors are used properly. To do this we define *kinds*, which will be associated to constructors in the same way types are associated to values in the language. The kind system is very simple, consisting only of the "base kind" $*$ and the kinds that can be build inductively using the "type arrow" constructor $\rightarrow$. The kind $*$ denotes a simple type (which may be a tuple, variant, functional or even polymorphic type), and for kinds κ_1 and κ_2 the kind $\kappa_1 \rightarrow \kappa_2$ denotes type operators which, given a type of kind κ_1, produce a type of kind κ_2. Thus, most types that appear in programs will have kind $*$, since this is the kind of any type that a term or value can take on. Types that have arrow kinds are formed by λ-abstraction as shown in the grammar for types, and kind arrows are eliminated by applying a type function to an appropriate type argument.

The second half of Figure 3.1 shows the different term-level constructs of the IL language. Because the FLINT language requires all terms to be in A-normal form [Sha97b], we enforce the same restriction by requiring every sub-expression in a computation to be named using a `let` construct, and restricting function applications, tuple formation, and other operations to act only on named values that have been previously bound. Because of the similarity between this style of writing terms and continuation-passing style [SW96], most of the desirable properties of CPS hold for expressions in A-normal form. This includes the ability to do various optimizing transformations [App92] and to easily generate efficient, imperative machine code while preserving type information [MWCG97].

3.3.2 IL Typing Rules

In order to discuss the type rules of the IL language, we must first establish some notation that we will be using. In the formal inference rules we will give, $\mathcal{A}$ will denote a set of assumptions of the form $(t :: \kappa)$, where t is a type variable and κ is a kind expression. A judgment of the form $\mathcal{A} \vdash \tau :: \kappa$ will mean that the type constructor τ has the kind κ under the assumptions in $\mathcal{A}$. We will use ST to denote a set of type equivalence assumptions of the form $(t = \tau)$, and $\mathbf{E}$ will denote a set of variable typing assumptions of the form $(x : \tau)$. A judgment $\mathcal{A}, \mathsf{ST}, \mathbf{E} \vdash e : \tau$ will mean that the expression e has type τ under the kind assumptions in $\mathcal{A}$, the type equivalence assumptions in ST, and the term variable typing assumptions in $\mathbf{E}$. Finally, a judgment $\mathcal{A}, \mathsf{ST} \vdash \tau_1 = \tau_2$ will mean that the types τ_1 and τ_2 are equal under the assumptions contained in $\mathcal{A}$ and ST. The inference rules that permit judgments of this form make up the type equivalence checking algorithm; we will not discuss these in any detail.

Figure 3.2 shows the inference rules that determine the kind of a type. These are all

Types

$$
\begin{aligned}
\kappa &::= \ * \mid \kappa_1 \to \kappa_2 \\
\tau &::= \texttt{int} \mid t \mid \tau_1 \to \tau_2 \mid \tau_1 \times \tau_2 \mid \tau_1 + \tau_2 \\
&\quad \mid \forall t :: \kappa.\tau \mid \exists t :: \kappa.\tau \\
&\quad \mid \lambda t :: \kappa.\tau \mid \tau_1[\tau_2] \\
&\quad \mid \texttt{Rec}\ t :: \kappa.\tau \mid \tau\ \texttt{ref} \mid \tau\ \texttt{array}
\end{aligned}
$$

Terms

$$
\begin{aligned}
e &::= x \mid x_1(x_2) \mid x[\tau] \\
&\quad \mid \texttt{let}\ x = v\ \texttt{in}\ e \\
&\quad \mid \texttt{let}\ x_1 = x_2\ \texttt{in}\ e \\
&\quad \mid \texttt{let}\ x_1 = x_2(x_3)\ \texttt{in}\ e \\
&\quad \mid \texttt{let}\ x_1 = x_2[\tau]\ \texttt{in}\ e \\
&\quad \mid \texttt{let}\ x_1 = \langle x_2, x_3 \rangle\ \texttt{in}\ e \\
&\quad \mid \texttt{let}\ x_1 = \#i\ \ x_2\ \texttt{in}\ e \\
&\quad \mid \texttt{let}\ x_1 = \texttt{inj}_i\ x_2\ \texttt{as}\ \tau_1 + \tau_2\ \texttt{in}\ e \\
&\quad \mid \texttt{case}\ x_1\ \texttt{bind}\ x_2\ \texttt{in}\ e_1\texttt{,}\ x_3\ \texttt{in}\ e_2 \\
&\quad \mid \texttt{let}\ x_1 = \texttt{pack}\ [\tau_1; x_2]\ \texttt{as}\ \exists t :: \kappa.\tau_2\ \texttt{in}\ e \\
&\quad \mid \texttt{let}\ [t; x_1] = \texttt{open}\ x_2\ \texttt{in}\ e \\
&\quad \mid \texttt{if}\ x\ \texttt{then}\ e_1\ \texttt{else}\ e_2 \\
&\quad \mid \texttt{letrec}\ b_1, \ldots, b_n\ \texttt{in}\ e \\
&\quad \mid \texttt{abort}\ [\tau] \\
&\quad \mid \ \ldots \\
b &::= x_1 : \tau_1 \to \tau_2 = \lambda x_2.e \\
&\quad \mid x : \forall t :: \kappa.\tau = \Lambda.e
\end{aligned}
$$

Figure 3.1: IL grammar (selected constructs).

$$\mathcal{A} \vdash \texttt{int} :: * \quad (\kappa : \texttt{int}) \qquad\qquad \frac{(t :: \kappa) \in \mathcal{A}}{\mathcal{A} \vdash t :: \kappa}(\kappa : \mathcal{A})$$

$$\frac{\mathcal{A} \vdash \tau_1 :: *, \quad \mathcal{A} \vdash \tau_2 :: *}{\mathcal{A} \vdash \tau_1 \times \tau_2 :: *}(\kappa : \times) \qquad\qquad \frac{\mathcal{A} \cup \{t :: \kappa\} \vdash \tau :: \kappa}{\mathcal{A} \vdash (\texttt{Rec}\ t :: \kappa.\tau) :: \kappa}(\kappa : \texttt{Rec})$$

$$\frac{\mathcal{A} \cup \{t :: \kappa\} \vdash \tau :: *}{\mathcal{A} \vdash (\forall t :: \kappa.\tau) :: *}(\kappa : \forall) \qquad\qquad \frac{\mathcal{A} \cup \{t :: \kappa_1\} \vdash \tau :: \kappa_2}{\mathcal{A} \vdash (\lambda t :: \kappa_1.\tau) :: \kappa_1 \to \kappa_2}(\kappa :\to I)$$

$$\frac{\mathcal{A} \vdash \tau_1 :: \kappa_1 \to \kappa_2, \quad \mathcal{A} \vdash \tau_2 :: \kappa_1}{\mathcal{A} \vdash \tau_1[\tau_2] :: \kappa_2}(\kappa :\to E)$$

Figure 3.2: Rules for determining kinds. The rules governing $+$ and $\to$ are similar to the one for $\times$ and are omitted. The rule for $\exists$ is similar to the one for $\forall$.

fairly straightforward, as any types formed by the constructors $\to$, $\times$, $+$, $\forall$ and $\exists$ will always have type $*$. The only interesting cases are $(\kappa : \texttt{Rec})$ for recursive types, where we must assume that the type being defined has the specified kind and check that the entire expression has the same kind, and the introduction and elimination rules for the kind constructor $\to$, which resemble the corresponding rules for functions in a simply-typed λ-calculus.

Some of the typechecking rules for terms in IL are shown in Figures 3.3 and 3.4. Note that in this figure we drop the requirement that every subexpression be named with a `let` construct; this makes our exposition of the type rules more compact and readable, but there are a few aspects of the A-normal form shown in Figure 3.1 that are worth mentioning. Most important is the fact that in many cases the type of the value being bound to a variable in a `let`-clause can be determined easily if type assignments for the variables in the expression for the value are known. Thus it is not necessary, when writing an IL program, to specify type information at every binding site; we only require a type annotation in those cases where the intended type cannot be immediately deduced. Those cases are injection of a value into a sum type, packing of a value as an existential type, and recursive λ- and Λ-abstractions. The syntax of functions and polymorphism are particularly specialized: in function definitions, we must require the programmer to specify the types of formal parameters in order to typecheck the body easily, but since functions are allowed to be recursive we must also require that the type of the entire function be given. Since the parameter types are really part of the function type, our syntax eliminates the redundancy and only requires that the latter be given explicitly. Similarly, the type of a polymorphic abstraction (Λ-expression) will include a name and kind for the parameter; we require only that this be given as part of the complete type, and it need not be repeated after the Λ.

Because enough type information is given that the intended type of any expression can be easily determined, we do not have to worry very much about making sure that the branches of a `case` expression (Figure 3.3) or an `if` expression (Figure 3.4) have the same type. We can simply determine the type of each branch and then make sure they agree under the assumptions in ST; thus the $(\tau : \texttt{if})$ rule might be more intuitively expressed as follows:

$$\frac{\begin{array}{c}\mathcal{A}, \mathsf{ST}, \mathbf{E} \vdash e_1 : \texttt{bool} \\ \mathcal{A}, \mathsf{ST}, \mathbf{E} \vdash e_2 : \tau_1 \quad \mathcal{A}, \mathsf{ST}, \mathbf{E} \vdash e_3 : \tau_2 \quad \mathcal{A}, \mathsf{ST} \vdash \tau_1 = \tau_2\end{array}}{\mathcal{A}, \mathsf{ST}, \mathbf{E} \vdash (\texttt{if}\ e_1\ \texttt{then}\ e_2\ \texttt{else}\ e_3) : \tau_1}(\tau : \texttt{if}')$$

Because of the rule $(\tau : \mathsf{ST})$, however, this rule is equivalent.[2] In Chapter 4 we will see that this is not so easy in SIL, where subtyping complicates the issue. For the purposes of IL, however, we do not have to worry about the validity of writing the rules the way we have presented them here.

Most of the type rules shown in Figures 3.3 and 3.4 are fairly straightforward, treating the formation and use of the various sorts of values in the usual ways. The treatment of functions and polymorphic abstractions (λ- and Λ-expressions) is a little unusual, because for us all such expressions are implicitly allowed to be recursive. In fact, this is another instance where the format of terms in the typechecking rules we have shown is oversimplified: as the grammar in Figure 3.1 shows, the only place these expressions can occur in the concrete syntax of our language is in a `letrec` binding, and more than one function may be defined in a single `letrec`. The concrete version of this construct thus permits mutual recursion: the body of any function may refer to any other function defined in the same `letrec` expression, regardless of the order in which they appear within the list of definitions. To typecheck this, we extend the rule shown so that the bodies are all typechecked in an environment that has been extended with the bindings $f_i : \tau_i$ for each function f_i defined in the `letrec` expression.

3.3.3 The IL Interpreter

In order to debug our $\mathcal{LOOM}$ compiler in the absence of a fully developed back-end, we have implemented an interpreter for the IL language in Standard ML. Our interpreter

[2]Provided, of course, that the (unshown) equivalence algorithm permits the judgment $\mathcal{A}, \mathsf{ST} \vdash \tau = \tau$ – which it does.

$$\frac{(x:\tau)\in\mathbf{E}}{\mathcal{A},\mathsf{ST},\mathbf{E}\vdash x:\tau}(\tau:\mathbf{E})$$

$$\frac{\mathcal{A},\mathsf{ST},\mathbf{E}\vdash e:\tau}{\mathcal{A},\mathsf{ST},\mathbf{E}\vdash(\texttt{ref}\ e):\tau\ \texttt{ref}}(\tau:\texttt{ref}\ I)$$

$$\frac{\mathcal{A},\mathsf{ST},\mathbf{E}\vdash e:\tau_1\times\tau_2}{\mathcal{A},\mathsf{ST},\mathbf{E}\vdash(\#i\ e):\tau_i}(\tau:\times Ei)$$

$$\frac{\mathcal{A},\mathsf{ST},\mathbf{E}\vdash e:\tau_1\quad\mathcal{A},\mathsf{ST}\vdash\tau_1=\tau_2}{\mathcal{A},\mathsf{ST},\mathbf{E}\vdash e:\tau_2}(\tau:\mathsf{ST})$$

$$\frac{\mathcal{A},\mathsf{ST},\mathbf{E}\vdash e:\tau\ \texttt{ref}}{\mathcal{A},\mathsf{ST},\mathbf{E}\vdash(!e):\tau}(\tau:\texttt{ref}\ E)$$

$$\frac{\mathcal{A},\mathsf{ST},\mathbf{E}\vdash e_1:\tau_1\quad\mathcal{A},\mathsf{ST},\mathbf{E}\vdash e_2:\tau_2}{\mathcal{A},\mathsf{ST},\mathbf{E}\vdash(e_1,e_2):\tau_1\times\tau_2}(\tau:\times I)$$

$$\frac{\mathcal{A},\mathsf{ST},\mathbf{E}\cup\{f:\tau_1\to\tau_2,x:\tau_1\}\vdash e:\tau_2}{\mathcal{A},\mathsf{ST},\mathbf{E}\vdash(\texttt{rec}\ f:\tau_1\to\tau_2=\lambda x.e):\tau_1\to\tau_2}(\tau:\to I)$$

$$\frac{\mathcal{A},\mathsf{ST},\mathbf{E}\vdash e_1:\tau_1\to\tau_2\quad\mathcal{A},\mathsf{ST},\mathbf{E}\vdash e_2:\tau_1}{\mathcal{A},\mathsf{ST},\mathbf{E}\vdash e_1(e_2):\tau_2}(\tau:\to E)$$

$$\frac{\mathcal{A},\mathsf{ST},\mathbf{E}\vdash e:\tau_i\quad\mathcal{A}\vdash\tau_{(2-i)}::*}{\mathcal{A},\mathsf{ST},\mathbf{E}\vdash(\texttt{inj}_i\ e\ \texttt{as}\ \tau_1+\tau_2):\tau_1+\tau_2}(\tau:+Ii)$$

$$\frac{\begin{array}{c}\mathcal{A},\mathsf{ST},\mathbf{E}\vdash e_1:\sigma_1+\sigma_2\\ \mathcal{A},\mathsf{ST},\mathbf{E}\cup\{x_1:\sigma_1\}\vdash e_2:\tau\quad\mathcal{A},\mathsf{ST},\mathbf{E}\cup\{x_2:\sigma_2\}\vdash e_3:\tau\end{array}}{\mathcal{A},\mathsf{ST},\mathbf{E}\vdash(\texttt{case}\ e_1\ \texttt{bind}\ x_1\ \texttt{in}\ e_2\mid x_2\ \texttt{in}\ e_3):\tau}(\tau:+E)$$

Figure 3.3: Selected type rules for IL.

$$\frac{\begin{array}{c}\mathcal{A}, \mathsf{ST}, \mathbf{E} \vdash e_1 : \texttt{bool} \\ \mathcal{A}, \mathsf{ST}, \mathbf{E} \vdash e_2 : \tau \quad \mathcal{A}, \mathsf{ST}, \mathbf{E} \vdash e_3 : \tau\end{array}}{\mathcal{A}, \mathsf{ST}, \mathbf{E} \vdash (\texttt{if } e_1 \texttt{ then } e_2 \texttt{ else } e_3) : \tau}(\tau : \texttt{if})$$

$$\frac{\mathcal{A} \cup \{t :: \kappa\}, \mathsf{ST}, \mathbf{E} \cup \{f : \forall t :: \kappa.\tau\} \vdash e : \tau \quad t \notin \mathsf{ST}}{\mathcal{A}, \mathsf{ST}, \mathbf{E} \vdash (\texttt{rec } f : \forall t :: \kappa.\tau = \Lambda.e) : \forall t :: \kappa.\tau}(\tau : \forall I)$$

$$\frac{\mathcal{A}, \mathsf{ST}, \mathbf{E} \vdash e : \forall t :: \kappa.\tau \quad \mathcal{A} \vdash \sigma :: \kappa}{\mathcal{A}, \mathsf{ST}, \mathbf{E} \vdash e[\sigma] : \tau[\sigma/t]}(\tau : \forall E)$$

$$\frac{\mathcal{A}, \mathsf{ST}, \mathbf{E} \vdash e : \tau[\sigma/t] \quad \mathcal{A} \vdash \sigma :: \kappa}{\mathcal{A}, \mathsf{ST}, \mathbf{E} \vdash (\texttt{pack } [\sigma; e] \texttt{ as } \exists t :: \kappa.\tau) : \exists t :: \kappa.\tau}(\tau : \exists I)$$

$$\frac{\begin{array}{c} t \notin \mathsf{ST},\ t \text{ not free in } \tau \\ \mathcal{A}, \mathsf{ST}, \mathbf{E} \vdash e_1 : \exists t :: \kappa.\sigma \quad \mathcal{A} \cup \{t :: \kappa\}, \mathsf{ST}, \mathbf{E} \cup \{x : \sigma\} \vdash e_2 : \tau,\end{array}}{\mathcal{A}, \mathsf{ST}, \mathbf{E} \vdash (\texttt{let } [t; x] = \texttt{open } \ e_1 \texttt{ in } e_2) : \tau}(\tau : \exists E)$$

Figure 3.4: Selected type rules for IL (continued).

accepts programs written in a syntax very similar to that shown in Figure 3.1, typechecks them using the rules given in the previous section, and runs them if they are found to be type-correct.

Because of the syntactic restriction forcing IL programs to be written in A-normal form, the interpreter is able to execute terms in a fairly imperative style. The interpreter executes each `let`-binding as though it were an assignment statement, fetching the values of the variables on the right-hand side from the environment, performing whatever operation is specified, updating the environment so that the variable named on the left-hand side is bound to the resulting value, and then moving on to execute the remaining bindings with the new environment. This recursive invocation of the evaluation function is a tail call, so excessive stack space is not consumed when evaluating long programs. Furthermore, although we currently use a splay tree (provided by the SML/NJ library) to store the execution environment, it should be possible to design a better representation that allows bindings to be inserted, and values changed, in constant time. Such an implementation improvement would mirror the compilation strategy of treating variable bindings as though they were assignments and storing as many values as possible in registers.

The execution phase of our interpreter does not make very much use of the type information in a program. Any values manipulated by the IL code are represented by a single ML datatype with constructors for tuples, function closures, members of disjoint unions, and so forth. This uniform representation makes it easier to build up values, pass them around, and take them apart, but it does impose some costs in terms of efficiency when we must check that a given value really is of a certain kind. Naturally, we attempt to minimize the number of places where this is necessary, but there seems to be no way around it in the cases of primitive or library function application (the "integer addition" function must extract the integer values from its arguments) and any other operations that need to access the internal structure of some value, most notably tuple component extraction and function (closure) application.

Another noteworthy aspect of our representation of values has to do with garbage collection. The original $\mathcal{LOOM}$ interpreter [Pet96, Van97] worked by directly simulating the operational semantics of the language [BFP97], including the maintenance of a "store," or a mapping from numbers representing memory locations to their contents. Because of that design choice and the representation of object references and pointers that it supports, responsibility for the management of a $\mathcal{LOOM}$ program's memory rests completely on the interpreter. Further, due to ever-present restrictions on the time and person-power that could be applied to the $\mathcal{LOOM}$ project, no garbage collector for the interpreter was ever implemented. In our IL interpreter, we attempt to make this problem less severe by taking advantage of the fact that side-effecting memory operations in IL are restricted to values of `ref` types[3], which behave in exactly the same way as the `ref` types of Standard ML. It is

[3]IL does also have arrays, similar to those in SML, but the same observations apply to those.

therefore possible for us to implement reference values *using* reference values, and in fact the definition of our `ILvalue` datatype contains the constructor `REFERENCE of ILvalue ref` – the code for interpreting reference operations is then very simple indeed, and we can let the SML runtime system take care of at least some of the memory management work for our IL programs.

Finally, in order to make our interpreter as extensible as possible, we have attempted to create and enforce an abstraction boundary between the interpreter code itself and the "run-time library," which contains the functions that perform input/output and other such tasks on behalf of IL programs. We allow the run-time library to define its own data types and values, and delegate the interpretation of expressions involving them to a separate structure which is linked into the interpreter. We also distinguish between library functions and primitive operations like addition, since we want to be able to analyze occurrences of addition in the hope of performing constant-folding or other optimizations.

Chapter 4

The Subtype-enhanced Intermediate Language SIL

In this chapter, we will describe the language SIL into which $\mathcal{LOOM}$ programs are directly translated. Since SIL is an extension of IL, we will discuss only those aspects of its semantics which differ from those of our final intermediate language. We will also describe the translation algorithm by which we turn the SIL representation of a $\mathcal{LOOM}$ program into an IL program for further processing or interpretation.

4.1 The SIL Language

When we designed the IL language, our goals were mainly to make it as simple as possible to type-check and interpret or translate into FLINT. As a result, the design of IL omitted some key features which were necessary for easy compilation of $\mathcal{LOOM}$; our compiler design bridges the gap with SIL, an extension of IL that supports all the necessary constructs for a straightforward $\mathcal{LOOM}$ encoding. In this section, we will discuss the features of SIL that are absent in IL, and in the next we will outline the translation algorithm that produces legal IL programs by eliminating uses of these features.

4.1.1 Record types

The simplest feature that we added was record types. In an effort to simplify the semantic processing of IL programs we omitted records in favor of simpler ordered tuples with unlabeled fields; it did not seem necessary to support full-fledged record types when unlabeled finite products would do just as well. In most encodings for objects, [BCP96], though, record types are used to describe collections of methods or instance variables. At first glance it does not seem that there is a substantial difference between records and tuples for this

$$\begin{aligned}
\tau &\rightarrow \ldots \\
&\mid \{l_1 : \tau_1, \ldots, l_n : \tau_n\} \\
e &\rightarrow \ldots \\
&\mid \texttt{let } x_0 = \{l_1 = x_1, \ldots, l_n = x_n\} \texttt{ in } e \\
&\mid \texttt{let } x_0 = x_1.l \texttt{ in } e
\end{aligned}$$

Figure 4.1: Grammar for records in SIL. We have introduced a new syntactic category of *labels*, denoted by l. Labels can only be used in records and are distinct from variable names.

purpose, but one does arise in the presence of subtyping, which we will discuss in Section 4.1.2.

The particular advantage of record types is that record field labels can be used to discriminate between different fields of the same type. This is significant when applied to languages (like $\mathcal{LOOM}$) where the order in which the components of a record are listed is not significant to its type. If neither the order nor the labelling on the fields is significant, then it is difficult to know which field is desired when the programmer attempts to extract one. Problems can then arise in the presence of structurally-determined subtyping, where it is possible to attempt to extract a component from a record whose exact layout cannot be determined at compile time. Since $\mathcal{LOOM}$ does not pay attention to the ordering of the fields of a record (or object) type and determines subtyping (and matching) structurally with no regard for field order, there is the potential for this kind of ambiguity. Any intermediate language for $\mathcal{LOOM}$ that allows subtyping behavior has to keep field label information around as a hint at which components of a subtype correspond to which ones in the supertype.

The records we added to SIL are similar to records in ML in that they are immutable, behaving essentially like tuples with labeled components. The order in which the fields of a record type are listed in the concrete syntax is unimportant, but the SIL translator immediately sorts all the fields of a record into alphabetical order by their labels, so any two records of the same type will have their parts laid out in memory the same way. The drawback to this is that supertypes and subtypes do not have their fields in the prefix relationship relied on in the translation of C++ programs, since one or more of the fields added in the subtype may lexicographically precede the last field of the supertype. If we did not sort the fields all the time, though, we might sometimes encounter records of the *same* type whose fields were in a different order, which we decided was even less desirable.

The grammar for record types and for the construction and use of record values in SIL is shown in Figure 4.1. The fields of a record may be listed in any order without affecting the meaning of the record or type expression. Extraction of a field, using the standard dot notation, is the only operation that can be performed on a record. In particular, side-effecting assignments to individual fields of an already existing record are not allowed. Of course, mutable records can be simulated by using records of `ref` values, but even this trick produces different run-time values from Pascal's mutable records or C's `struct` types; in these, the components of a record are stored contiguously in memory, while writing a fake mutable record in SIL using reference types produces a contiguous block that only holds pointers to the components, which may be anywhere in memory. Adding mutable contiguous records to the language would have made the translation to eliminate subtyping difficult or impossible. The reasons for this will become clear in Section 4.2.

4.1.2 Subtyping and Bounded Quantification

The most important feature that was added in the creation of SIL was bounded quantification, which in turn requires support for subtyping. Bounded polymorphic functions, in addition to appearing explicitly in the $\mathcal{LOOM}$ source language, are essential to the encoding of $\mathcal{LOOM}$ classes described in [BV99]. They permit the parameterization of method bodies over all possible extensions of the class in which they appear and codify the assumptions that must be made about `MyType` in order to ensure safe use of `self` and binary methods. Bounded existential types, while they will not be used much in this thesis, are important for the encoding of $\mathcal{LOOM}$'s #-types and modules.

The bounds in bounded quantification are expressed in terms of the subtyping relation, which is a partial ordering on types. Intuitively, a type σ is a subtype of τ if a value of type σ could be used anywhere a value of type τ is expected without causing any errors. In most object-oriented languages, notably Java, C++ and Eiffel, this is taken to be automatically true of the types of objects produced by classes that are in an ancestor-descendant relationship. $\mathcal{LOOM}$ does not make this assumption, and in fact does not allow any subtyping behavior explicitly in the source language, but we still needed to add subtyping to our intermediate language in order to translate it using the known encodings.

The rules governing subtyping in SIL are given in Figure 4.2. In these rules, we use $\mathcal{C}$ to denote a set of subtyping assumptions of the form $(t <: \tau)$; the judgment $\mathcal{C}, \mathcal{A} \vdash \sigma <: \tau$ means that the type σ is a subtype of τ under the assumptions contained in $\mathcal{C}$ and $\mathcal{A}$. The rules for the $\times$ and $+$ type constructors are straightforward and intuitive; the rule for $\rightarrow$ is a little different in that the required subtype relation in the parameter types is *contravariant*; that is, opposite in direction to the relation being derived. This is necessary to make the type system sound: if a function of type $\sigma_1 \rightarrow \sigma_2$ is to masquerade as one of type $\tau_1 \rightarrow \tau_2$, then it must be that our function can accept anything that a function of the supertype could accept. When we call a function that is masquerading, we are essentially forcing the argument we send it to masquerade as well: in particular, the type we send it (the parameter type of the function's supertype) must be a subtype of the type it is expecting. A similar contravariant rule could be used for bounded $\forall$-types, but this would make the

$$\mathcal{C}, \mathcal{A} \vdash \tau <: \tau \quad (<:: \mathrm{Id})$$

$$\frac{\mathcal{C} \cup \{t <: \sigma\}, \mathcal{A} \cup \{t :: \kappa\} \vdash \tau_1 <: \tau_2}{\mathcal{C}, \mathcal{A} \vdash \forall t :: \kappa <: \sigma.\tau_1 <: \forall t :: \kappa <: \sigma.\tau_2} (<:: \forall <:)$$

$$\frac{\mathcal{C}, \mathcal{A} \vdash \tau_1 <: \tau_2 \quad \mathcal{C}, \mathcal{A} \vdash \tau_2 <: \tau_3}{\mathcal{C}, \mathcal{A} \vdash \tau_1 <: \tau_3} (<:: \mathrm{Trans})$$

$$\frac{\mathcal{C} \cup \{t <: \sigma\}, \mathcal{A} \cup \{t :: \kappa\} \vdash \tau_1 <: \tau_2}{\mathcal{C}, \mathcal{A} \vdash \exists t :: \kappa <: \sigma.\tau_1 <: \exists t :: \kappa <: \sigma.\tau_2} (<:: \exists <:)$$

$$\frac{\mathcal{C}, \mathcal{A} \vdash \sigma_1 <: \tau_1 \quad \mathcal{C}, \mathcal{A} \vdash \sigma_2 <: \tau_2}{\mathcal{C}, \mathcal{A} \vdash \sigma_1 \times \sigma_2 <: \tau_1 \times \tau_2} (<:: \times)$$

$$\frac{\mathcal{C}, \mathcal{A} \cup \{t :: \kappa\} \vdash \sigma <: \tau \quad t \notin \mathcal{C}}{\mathcal{C}, \mathcal{A} \vdash (\lambda t :: \kappa.\sigma) <: (\lambda t :: \kappa, \tau)} (<:: \lambda)$$

$$\frac{\mathcal{C}, \mathcal{A} \vdash \tau_1 <: \sigma_1 \quad \mathcal{C}, \mathcal{A} \vdash \sigma_2 <: \tau_2}{\mathcal{C}, \mathcal{A} \vdash \sigma_1 \to \sigma_2 <: \tau_1 \to \tau_2} (<::\to)$$

$$\frac{\mathcal{C} \cup \{s <: t\}, \mathcal{A} \cup \{s :: \kappa, t :: \kappa\} \vdash \sigma <: \tau}{\mathcal{C}, \mathcal{A} \vdash \mathtt{Rec}\ s :: \kappa.\sigma <: \mathtt{Rec}\ t :: \kappa.\tau} (<:: \mathtt{Rec})$$

$$\frac{\mathcal{C}, \mathcal{A} \vdash \sigma_l <: \tau_l \quad L \subseteq M}{\mathcal{C}, \mathcal{A} \vdash \{l : \sigma_l\}_{l \in M} <: \{l : \tau_l\}_{l \in L}} (<:: \{\})$$

$$\frac{\mathcal{C}, \mathcal{A} \vdash \tau_1 <: \tau_2 \quad \mathcal{A} \vdash \tau_1 :: \kappa_1 \to \kappa_2 \quad \mathcal{A} \vdash \sigma :: \kappa_1}{\mathcal{C}, \mathcal{A} \vdash \tau_1[\sigma] <: \tau_2[\sigma]} (<:: [\,])$$

Figure 4.2: Rules for subtyping in SIL. Notice the depth subtyping for records and the "kernel" rule for polymorphic functions. The rule for + is similar to the one for $\times$, and the rules for unbounded quantification are similar to the bounded ones.

$$
\begin{aligned}
\tau &\rightarrow \forall t :: \kappa <: \tau_1.\tau_2 \\
&\quad \mid \exists t :: \kappa <: \tau_1.\tau_2 \\
e &\rightarrow \texttt{let } x_1 = \texttt{pack } [\tau_1; x_2] \texttt{ as } \exists t :: \kappa <: \tau_2.\tau_3 \texttt{ in } e \\
b &\rightarrow x : \forall t :: \kappa <: \tau_1.\tau_2 = \Lambda.e
\end{aligned}
$$

Figure 4.3: Grammar for bounded quantification in SIL. These productions should be added to the IL grammar of Figure 3.1; in particular, they do not replace or override the syntactic rules for unbounded quantification.

type system undecidable [Pie92]. Instead, we make do with the "kernel" rule for bounded polymorphic functions, which does not allow changes in the bound.

While the meaning of subtyping for actual types (that is, constructors of kind $*$) is clear, it is not obvious *a priori* what it should mean for constructors of higher kinds. We adopt a "pointwise" subtyping rule, which states that two type operators F and G are in a subtype relation, *i.e.* $F <: G$, if for any type τ, $F[\tau] <: G[\tau]$. Accordingly, we write the rule ($<:\lambda$) to detect pairs of type functions of which this is true and the rule ($<:[\,]$) to apply our knowledge of two functions when considering the types they return. This pointwise subtyping, however, should not be confused with the stronger requirement given for recursive types to be subtypes. In order to conclude that $\texttt{Rec}\ s :: \kappa.\sigma$ is a subtype of $\texttt{Rec}\ t :: \kappa.\tau$, we must be able to derive $\sigma <: \tau$ assuming only that $s <: t$. Assuming that $s = t$ would render the type system unsound [AC93], so we have to be content with the weaker assumption.

Finally, the syntax for bounded polymorphic functions in SIL is given by Figure 4.3. Bounded quantified types are constructed by writing the bound after the kind of the variable, and the term-level constructs for creating values of quantified types are extended in the obvious way. Note that new rules only need to be added for the creation of values with the new types; the `open` and polymorphic function application constructs have the same syntax regardless of whether or not the type in question is bounded.

4.1.3 Type Annotations

We have just explained all of the important differences between IL and SIL: all of the added functionality comes from the new record types and bounded quantifiers. In order to make the language easily type-checkable, however, certain other changes need to be made as well, and we will discuss those here. The basic problem is that, because of subtyping, the type the SIL system deduces for a variable or term may not be exactly the one the programmer intended – it may be a subtype.

In Section 3.3.2 it was noted that, because of our A-normal form syntax, it was very

$$\frac{\mathcal{A}, \mathcal{C}, \mathsf{ST}, \mathbf{E} \vdash e_1 : \mathtt{bool} \quad \mathcal{A}, \mathcal{C}, \mathsf{ST}, \mathbf{E} \vdash e_2 : \tau \quad \mathcal{A}, \mathcal{C}, \mathsf{ST}, \mathbf{E} \vdash e_3 : \tau}{\mathcal{A}, \mathcal{C}, \mathsf{ST}, \mathbf{E} \vdash (\mathtt{if}_\tau\ e_1\ \mathtt{then}\ e_2\ \mathtt{else}\ e_3) : \tau} (\tau : \mathtt{if})$$

Figure 4.4: Type-checking of `if` in SIL. Note the change of syntax. Similar changes have been made to the `case` construct for accessing members of types formed by $+$.

easy to determine the type of any variable or term in an IL program provided all the types of previously defined variables were known. To make this possible, though, it was necessary to require the programmer to insert some type annotations for `let`-bindings where the type of the expression would not be obvious to the type-checker. This is, in a sense, still possible in SIL, but the type that is most easily constructed for a term will be its "minimum" type. The programmer might have intended a different type, of which the found type is a subtype. This becomes a problem in the type-checking of `if` and `case` expressions, where we must compare two terms to ensure that they have the same type.

In the type-checking rule for `if` expressions in IL, we check each branch and then compare their types to make sure they are the same. If SIL used the same syntax and semantics for the `if` construct, it would be possible to write a SIL `if` expression where one branch returned a value of type σ_1 and the other returned a value of type σ_2, where σ_1 and σ_2 were distinct types and neither was a subtype of the other but each was a subtype of a weaker type τ. The `if` expression would then be legal and could have type τ, but designing a type-checker that could automatically find such a τ would be a pain (this is what had to be done for the TOOPLE language [BCM+93]). Instead, we simply require that every `if` expression in SIL be annotated with the type its branches are supposed to return. We can then just check that each branch does have that type, invoking subsumption if necessary. The type rule for `if` in SIL is shown in Figure 4.4; similar changes need to be made for `case`.

In the concrete syntax, the A-normal form allows `if` and `case` expressions only at the "tail" of a `let`-expression; in particular, binding of the result of an `if` or a `case` directly to a variable is not permitted. As a result, it suffices to require that the intended type of every A-normal form term be known in advance. Because the return type of any function expression must be specified as part of the function's type, all that needs to be added to a program in A-normal form is the type that the entire program is intended to have; in our interpreter we have adopted the convention that any whole program will always have type `unit`. Thus the need for extra type annotations is not visible in the concrete syntax.[1]

[1]Actually, the syntax we use does allow the programmer to use a `let`-expression as the right-hand side of a variable binding, thus violating A-normal form. In SIL, such nested `let`'s must be labeled with their intended types.

4.2 Translation to IL

This section will describe in some detail the translation that removes uses of records and subtyping from SIL programs to produce valid IL programs. We will begin in Section 4.2.1 by discussing the ideas and previous work behind our translation, and then Section 4.2.2 we will give a more detailed and formal account of our translation algorithm.

4.2.1 Overview of the Translation

Our translation is very similar to that used in [BTCGS94]; the basic idea is as follows. When we force a value of type σ to masquerade as one of type τ, we are essentially using the fact that any object of type σ "looks like" a value of type τ. What this really means, more precisely, is that for every value v of type σ there is another value, v', of type τ that v looks like. In fact, in any context where a value of type τ is expected, we can use v instead of an object of type τ because any expressions which act upon the value of type τ will treat v as though it were v'. Furthermore, in our SIL language we can actually construct v' given v – that is, given any object of type σ we can actually build the object of type τ that it "looks like." We exploit this property to eliminate uses of subsumption: all we have to do is find places where values are forced to masquerade, and explicitly generate the value of the weaker type that should be used.

A translation as simple as that would be sufficient if all the types that appear anywhere in the program were statically known. In SIL, however, this is not the case because of bounded quantification. If the type of something that needs to be converted is not statically known, it may not be possible for the compiler to generate the code to convert it without some help. What is needed is for every bounded type variable to have associated with it some kind of hint as to how to coerce it up to its bound. The solution, therefore, is to change bounded polymorphic functions so that they must also accept a *coercion function* that can convert values of the parameter type to values of the type by which it is bounded. For bounded existentials, a coercion must be packed with the original value so that when the package is opened, any values of the hidden type may be coerced up as well. This introduction of coercion functions is the heart of the translation.

The final thing we must make clear before giving the details of the algorithm is the way type constructors of higher kinds are handled. Obviously, it is impossible to use conversion functions of the same kind as are used for simple types, as it makes no sense to speak of converting "values" of a function from types to types. Fortunately, this is not what we really need anyway. The subtyping rules for SIL (Figure 4.2) define subtyping of type functions to mean "pointwise" subtyping: thus if we know that a type operator F is bounded above by G, then what we really need is to be able to find coercions from $F[\tau]$ to $G[\tau]$ for any τ. Thus, we define the type of coercions for different kinds of types in Figure 4.5. As that figure illustrates, coercions for higher kinds of type constructors are really polymorphic

$$\text{coerce}_*(\tau_1 <: \tau_2) = |\tau_1| \to |\tau_2|$$

$$\text{coerce}_{\kappa_1 \to \kappa_2}(\tau_1 <: \tau_2) = \forall t :: \kappa_1.(\text{coerce}_{\kappa_2}(\tau_1[t] <: \tau_2[t]))$$

Figure 4.5: Determining the type of a coercion function.

abstractions of coercions for lower kinds. Thus if we have a coercion c for F and G and we want a coercion for $F[\tau]$ and $G[\tau]$, we can just use $c[\tau]$. With this brief presentation of the ideas behind our algorithm complete, we can move on to discuss the details.

4.2.2 The Translation Algorithm

The most straightforward part of the translation algorithm is the mapping of SIL types to IL types, which is shown in Figure 4.6. We use the notation $|\tau|$ to mean the translation of the SIL type τ into IL. Since the kind systems of the two languages are identical, we have omitted the trivial translation of kind expressions and instead treat SIL kinds and IL kinds as interchangeable. The translation mapping $|\cdot|$ is defined recursively on types, and most of the cases are trivial. The only interesting ones are for records, which become tuples, and for bounded quantification. When converting a record into a tuple, we must first sort the labels into ascending lexicographical order (hence the condition that $l_1 < \cdots < l_n$), to ensure that records of the same type become tuples of the same type. In bounded universal quantification, we turn a function that takes just a type parameter into one that also accepts a coercion function of the appropriate type, which should be thought of as a "proof" that the bound is satisfied. In the corresponding rule for existential types, we modify the type so that the coercion witnessing the subtype relation is packed along with the main value. In the term translation, we will take care of generating these coercions and passing them around appropriately.

In order to generate the coercion functions needed for passing to bounded polymorphic functions and packing into existentials, we use an approach similar to that in [BTCGS94]. Following that paper, we note that we can use a derivation of a subtyping judgment to construct a coercion function to a type from one of its subtypes. The rules for doing this are given in Figure 4.7. Here we use the letter $\mathcal{C}$ to denote a set of coercion assumptions of the form $(c : t <: \tau)$, where t is a type variable which represents a subtype of τ, and c is a coercion function from t to τ, whose type depends on the kind of τ. The judgment $\mathcal{A}, \mathcal{C} \vdash c : \tau_1 <: \tau_2$ will mean that under the assumptions in $\mathcal{A}$ and $\mathcal{C}$, the expression c can be used as a coercion from the type τ_1 to the supertype τ_2. Also, here and throughout the rest

$$|\texttt{int}| = \texttt{int} \qquad |\texttt{Rec}\ t :: \kappa.\tau| = \texttt{Rec}\ t :: \kappa.|\tau|$$

$$|t| = t \qquad |\tau_1 \to \tau_2| = |\tau_1| \to |\tau_2|$$

$$|(\lambda t :: \kappa.\tau)| = (\lambda t :: \kappa.|\tau|) \qquad |\tau_1 \times \tau_2| = |\tau_1| \times |\tau_2|$$

$$|\tau_1[\tau_2]| = |\tau_1|[|\tau_2|] \qquad |\tau_1 + \tau_2| = |\tau_1| + \tau_2|$$

$$|\{l_1 : \tau_1, \dots, l_n : \tau_n\}| = |\tau_1| \times \cdots \times |\tau_n|, \text{ if } l_1 < \cdots < l_n$$

$$|\forall t :: \kappa <: \sigma.\tau| = \forall t :: \kappa.(\mathrm{coerce}_\kappa(t <: \sigma)) \to |\tau|$$

$$|\exists t :: \kappa <: \sigma.\tau| = \exists t :: \kappa.(\mathrm{coerce}_\kappa(t <: \sigma)) \times |\tau|$$

Figure 4.6: Translation of SIL types.

$$\mathcal{A}, \mathcal{C} \vdash \mathrm{id}_\tau : \tau <: \tau \quad (c : \mathrm{id})$$

$$\frac{(c : t <: \tau) \in \mathcal{C}}{\mathcal{A}, \mathcal{C} \vdash c : t <: \tau}(c : \mathcal{C})$$

$$\frac{\mathcal{A}, \mathcal{C} \vdash c_1 : \tau_1 <: \sigma_1 \quad \mathcal{A}, \mathcal{C} \vdash c_2 : \sigma_2 <: \tau_2}{\mathcal{A}, \mathcal{C} \vdash (\lambda f : \sigma_1 \to \sigma_2 . c_2 \circ f \circ c_1) : \sigma_1 \to \sigma_2 <: \tau_1 \to \tau_2}(c :\to)$$

$$\frac{\mathcal{A}, \mathcal{C} \vdash c_1 : \sigma_1 <: \tau_1 \quad \mathcal{A}, \mathcal{C} \vdash c_2 : \sigma_2 <: \tau_2}{\mathcal{A}, \mathcal{C} \vdash (\lambda x : \sigma_1 \times \sigma_2 . (c_1(\#0\ x), c_2(\#1\ x))) : \sigma_1 \times \sigma_2 <: \tau_1 \times \tau_2}(c : \times)$$

$$\frac{\mathcal{A} \cup \{t :: \kappa\}, \mathcal{C} \vdash c : \tau_1 <: \tau_2 \quad t \notin \mathcal{C}}{\mathcal{A}, \mathcal{C} \vdash (\Lambda t :: \kappa . c) : (\lambda t :: \kappa . \tau_1) <: (\lambda t :: \kappa . \tau_2)}(c : \lambda)$$

$$\frac{\mathcal{A}, \mathcal{C} \vdash \tau_1 <: \tau_2 \quad \mathcal{A} \vdash \tau_1 :: \kappa_1 \to \kappa_2 \quad \mathcal{A} \vdash \sigma :: \kappa_1}{\mathcal{A}, \mathcal{C} \vdash c[\sigma] : \tau_1[\sigma] <: \tau_2[\sigma]}(c : [\,])$$

$$\frac{\mathcal{A} \cup \{t :: \kappa\}, \mathcal{C} \cup \{c : t <: \sigma\} \vdash c' : \tau_1 <: \tau_2}{\begin{array}{l}\mathcal{A}, \mathcal{C} \vdash (\lambda f : \forall t :: \kappa . \mathrm{coerce}_\kappa(t <: \sigma) \to \tau_1 . \Lambda t :: \kappa . \lambda c : \mathrm{coerce}_\kappa(t <: \sigma) . c'(f[t]c)) \\ \qquad : \forall t :: \kappa <: \sigma . \tau_1 <: \forall t :: \kappa <: \sigma . \tau_2\end{array}}(c : \forall)$$

$$\frac{\mathcal{A} \cup \{t :: \kappa\}, \mathcal{C} \cup \{c : t <: \sigma\} \vdash c' : \tau_1 <: \tau_2}{\begin{array}{l}\mathcal{A}, \mathcal{C} \vdash (\lambda x : \exists t :: \kappa . \mathrm{coerce}_\kappa(t <: \sigma) \times \tau_1 . \mathtt{let}\ [t; (c, x')] =\ \mathtt{open}\ x\ \mathtt{in} \\ \qquad \mathtt{pack}\ [t; (c, c'(x'))]\ \mathtt{as}\ \exists t :: \kappa . \mathrm{coerce}_\kappa(t <: \sigma) \times \tau_2) \\ \qquad : \exists t :: \kappa <: \sigma . \tau_1 <: \exists t :: \kappa <: \sigma . \tau_2\end{array}}(c : \exists)$$

Figure 4.7: Rules for constructing coercion functions. We use the shorthand id_τ to denote $(\mathtt{rec}\ f : \tau \to \tau = \lambda x.x)$, the identity function on τ.

of this chapter we will use non-recursive λ- and Λ-expressions as a shorthand where recursion is not necessary, and we will use the operator $\circ$ as a shorthand for function composition, which of course can always be expressed using λ-abstraction. The rules in Figure 4.7 are exactly analogous to those in 4.2, except that here we are giving coercion expressions that witness each of the subtyping relations we derive.

The fact that the rules for finding a coercion function are so closely parallel to those for deriving a subtyping judgment suggests that we could succinctly write our translation of terms in a fashion parallel to the rules for deriving a typing judgment in SIL. To do this we would write, for each of SIL's type rules yielding a judgment of the form $\mathcal{A}, \mathcal{C}, \mathsf{ST}, \mathbf{E} \vdash e : \tau$, a translation rule yielding a judgment $\mathcal{A}, \mathcal{C}, \mathsf{ST}, \mathbf{E} \vdash e : \tau \mapsto e'$. Such a judgment would mean that under the assumptions in $\mathcal{A}$, $\mathcal{C}$, ST and $\mathbf{E}$ the expression e has type τ and could be translated into the IL term e'. We would then simply have to show that if this latter judgment were derivable then we could also derive the former, and that the IL type rules would permit us to derive $\mathcal{A}, \mathsf{ST}, (|\mathbf{E}| \cup |\mathcal{C}|) \vdash e' : |\tau|$. The only tricky cases in constructing these rules would be the translation rules corresponding to the subsumption rule (which states that e has type τ' if e has type τ and $\tau <: \tau'$) and the rules for bounded quantifier introduction and elimination. The subsumption rule in SIL is as follows:

$$\frac{\mathcal{A}, \mathcal{C}, \mathsf{ST}, \mathbf{E} \vdash e : \tau \quad \mathcal{A}, \mathcal{C} \vdash \tau <: \tau'}{\mathcal{A}, \mathcal{C}, \mathsf{ST}, \mathbf{E} \vdash e : \tau'}(\tau :<:)$$

The corresponding translation rule would just have to find and apply the appropriate coercion:

$$\frac{\mathcal{A}, \mathcal{C}, \mathsf{ST}, \mathbf{E} \vdash e : \tau \mapsto e' \quad \mathcal{A}, \mathcal{C} \vdash c : \tau <: \tau'}{\mathcal{A}, \mathcal{C}, \mathsf{ST}, \mathbf{E} \vdash e : \tau' \mapsto c(e')}(\mapsto:<:)$$

Similarly, consider the bounded universal introduction rule that provides for type-checking of bounded Λ-expressions:

$$\frac{\mathcal{A}, \mathcal{C} \cup \{t <: \sigma\}, \mathsf{ST}, \mathbf{E} \cup \{f : \forall t :: \kappa <: \sigma.\tau\} \vdash e : \tau \quad t \notin \mathsf{ST}}{\mathcal{A}, \mathcal{C}, \mathsf{ST}, \mathbf{E} \vdash (\mathtt{rec}\ f : \forall t :: \kappa <: \sigma.\tau = \Lambda.e) : \forall t :: \kappa <: \sigma.\tau}(\tau : \forall <: I)$$

This would become a translation rule that inserts a space for the proper coercion to be provided along with the type parameter:

$$\frac{\mathcal{A}, \mathcal{C} \cup \{c : t <: \sigma\}, \mathsf{ST}, \mathbf{E} \cup \{f : \forall t :: \kappa <: \sigma.\tau\} \vdash e : \tau \mapsto e' \quad t \notin \mathsf{ST} \quad c \notin \mathbf{E}}{\begin{array}{l}\mathcal{A}, \mathcal{C}, \mathsf{ST}, \mathbf{E} \vdash (\mathtt{rec}\ f : \forall t :: \kappa <: \sigma.\tau = \Lambda.e) : \forall t :: \kappa <: \sigma.\tau \\ \quad \mapsto (\mathtt{rec}\ f : \forall t :: \kappa.(\mathrm{coerce}_\kappa(t <: \sigma)) \to \tau \\ \quad = \Lambda.\lambda c : \mathrm{coerce}_\kappa(t <: \sigma).e')\end{array}}(\mapsto: \forall <: I)$$

Finally, to translate applications of bounded polymorphic values to types, we would use the translation rule:

$$\frac{\mathcal{A}, \mathcal{C}, \mathsf{ST}, \mathbf{E} \vdash e : \forall t :: \kappa <: \tau_1.\tau_2 \mapsto e' \quad \mathcal{A}, \mathcal{C} \vdash c : \sigma <: \tau_1}{\mathcal{A}, \mathcal{C}, \mathsf{ST}, \mathbf{E} \vdash e[\sigma] : \tau_2[\sigma/t] \mapsto e'[|\sigma|](c)}(\mapsto: \forall <: E)$$

Writing translation rules in this way would produce a translation mechanism similar to that described in [BTCGS94]; in that paper, a translation is defined on entire derivation trees for typing judgments. The problem with expressing a translation this way is that it does not immediately suggest an algorithm for constructing the translated version of a given program. Worse, the relation $\mapsto$ that we have defined does not even determine a single-valued function from SIL terms to IL terms, since any given SIL term may have many different possible types, and translating it at each one may produce a different result.

In order to make our translation more algorithmic, we introduce the concept of a term's *minimum type*. Given contexts $\mathcal{A}$, $\mathcal{C}$, ST and $\mathbf{E}$, the minimum type of a term e will be the type τ such that $\mathcal{A}, \mathcal{C}, \mathsf{ST}, \mathbf{E} \vdash e : \tau$ and if for any τ' we can derive $\mathcal{A}, \mathcal{C}, \mathsf{ST}, \mathbf{E} \vdash e : \tau'$ then we can also derive $\mathcal{A}, \mathcal{C} \vdash \tau <: \tau'$. In other words, τ is the minimal type we can assign to e. We will also define *accessibility* of a type: we say a type τ is accessible if the normal form of τ is *not* of the form t, $\sigma_1[\sigma_2]$ or $\mathtt{Rec}\ t :: \kappa.\sigma$. Basically, we want accessible types to be those whose shape (*i.e.*, function, record, *etc.*) is visible. We will use the phrase *minimum accessible type* to refer to an accessible type that can be assigned to a term and which is a subtype of any other accessible type that can be assigned to the term.

Our goal now is to produce inference rules that define a new translation relation, which we will call $\leadsto$, corresponding to the image of a term under $\mapsto$ at its minimum type. More accurately, we will be interested in deriving judgments of the form $\mathcal{A}, \mathcal{C}, \mathsf{ST}, \mathbf{E} \vdash e : \tau \leadsto e'$; this judgment asserts that τ is the minimum type of e and that translating e gives us the term e' of type $|\tau|$. Our algorithmic rules will introduce as few coercion applications as possible, so the SIL type assigned to a given term will be minimal, but we will sometimes need an expression to have an accessible type. To make this happen, we will also define a relation $\overset{\text{acc}}{\leadsto}$ which will translate a term at its minimum accessible type. That is, we will give rules for deriving judgments of the form $\mathcal{A}, \mathcal{C}, \mathsf{ST}, \mathbf{E} \vdash e : \tau \overset{\text{acc}}{\leadsto} e'$, meaning that τ is the minimum accessible type of e and the translation of e at τ gives e' of type $|\tau|$.

Since the definition of the $\overset{\text{acc}}{\leadsto}$ relation is the most complicated part of our algorithmic presentation, we will discuss it first. The rules for finding minimum accessible types are shown in Figure 4.8. We use two auxiliary functions, which are also shown in the figure. The "one" function gives a coercion to the minimum type of which the given type is a nontrivial subtype, by looking identifiers up in the subtyping context and instantiating coercions for type functions with the appropriate parameters. The "expand" function gives a coercion to the minimum accessible type of which the given type is a subtype. If the type being expanded is accessible, the identity coercion is returned. Otherwise, the "one" function is used to find the next higher type, which is then expanded recursively. Note that, although the "expand" function does not converge for all types ($\mathtt{Rec}\ t.t$ is a good example where it does not), we will never apply it to anything that cannot be the type of a term. Finally, in order to translate a SIL term at its minimum accessible type, the $(\overset{\text{acc}}{\leadsto})$ rule first translates the term at its minimum type using the $\leadsto$ rules we will give shortly, expands that minimum type to its minimum accessible supertype, and applies the appropriate coercion

$$\frac{(c : t <: \tau) \in \mathcal{C}}{\mathcal{C} \vdash \mathrm{one}(t) = (c, \tau)}(\mathrm{one} : \mathcal{C})$$

$$\frac{\mathcal{C} \vdash \mathrm{one}(\tau) = (c, \tau')}{\mathcal{C} \vdash \mathrm{one}(\tau[\sigma]) = (c[\sigma], \tau[\sigma])}(\mathrm{one} : [\,])$$

$$\mathcal{C} \vdash \mathrm{one}(\texttt{Rec}\ t :: \kappa.\tau) = (\mathrm{id}_{\texttt{Rec}\ t::\kappa.\tau}, \tau[(\texttt{Rec}\ t :: \kappa.\tau)/t]) \quad (\mathrm{one} : \texttt{Rec})$$

$$\frac{\tau \text{ accessible}}{\mathcal{A}, \mathcal{C} \vdash \mathrm{expand}(\tau) = (\mathrm{id}_\tau, \tau)}(\mathrm{expand} : \mathrm{Acc})$$

$$\frac{\begin{array}{c}\mathcal{A} \vdash \tau :: * \quad t \text{ not accessible} \\ \mathcal{C} \vdash \mathrm{one}(\tau) = (c, \tau') \quad \mathcal{A}, \mathcal{C} \vdash \mathrm{expand}(\tau') = (c', \tau'')\end{array}}{\mathcal{A}, \mathcal{C} \vdash \mathrm{expand}(\tau) = (c' \circ c, \tau'')}(\mathrm{expand} : \mathrm{NAcc})$$

$$\frac{\mathcal{A}, \mathcal{C}, \mathsf{ST}, \mathbf{E} \vdash e : \tau \rightsquigarrow e' \quad \mathcal{A}, \mathcal{C} \vdash \mathrm{expand}(\tau) = (c, \tau')}{\mathcal{A}, \mathcal{C}, \mathsf{ST}, \mathbf{E} \vdash e : \tau' \overset{\mathrm{acc}}{\rightsquigarrow} c(e')}(\overset{\mathrm{acc}}{\rightsquigarrow})$$

Figure 4.8: Finding minimum accessible types.

to the minimum-type translation of the term.

The complete list of rules defining the $\leadsto$ relation is rather long, and so it will not be given here. All of the rules do appear in Appendix A. In what follows we will describe a few of the cases of the translation, to give an idea of how it works in general. To begin with, note that the type assignment rule for a λ-expression in SIL is the same as the one in IL except that a subtyping context is required:

$$\frac{\mathcal{A}, \mathcal{C}, \mathsf{ST}, \mathbf{E} \cup \{f : \tau_1 \to \tau_2, x : \tau_1\} \vdash e : \tau_2}{\mathcal{A}, \mathcal{C}, \mathsf{ST}, \mathbf{E} \vdash (\texttt{rec}\ f : \tau_1 \to \tau_2 = \lambda x.e) : \tau_1 \to \tau_2}(\tau :\to I)$$

Of course, it may turn out that the minimum type of the return expression e is not exactly τ_2 but some subtype of τ_2. An application of the subsumption rule would be needed in this case in order to derive $e : \tau_2$. We can make this explicit by rewriting this rule as follows:

$$\frac{\mathcal{A}, \mathcal{C}, \mathsf{ST}, \mathbf{E} \cup \{f : \tau_1 \to \tau_2, x : \tau_1\} \vdash e : \tau_2' \quad \mathcal{A}, \mathcal{C} \vdash \tau_2' <: \tau_2}{\mathcal{A}, \mathcal{C}, \mathsf{ST}, \mathbf{E} \vdash (\texttt{rec}\ f : \tau_1 \to \tau_2 = \lambda x.e) : \tau_1 \to \tau_2}(\tau :\to I')$$

This form of the type assignment rule can now be used to devise a rule for translating λ-expressions at their minimum type. Since the minimum type of a λ-expression is the function type whose parameter and return types are exactly as given in the expression, we can create an IL version of the function provided we can coerce the return value from its minimum type to the declared type. The rule that specifies this is as follows:

$$\frac{\mathcal{A}, \mathcal{C}, \mathsf{ST}, \mathbf{E} \cup \{f : \tau_1 \to \tau_2, x : \tau_1\}, \mathsf{ST} \vdash e : \tau_2' \leadsto e' \quad \mathcal{A}, \mathcal{C} \vdash c : \tau_2' <: \tau_2 \quad f, x \notin \mathcal{C}}{\mathcal{A}, \mathcal{C}, \mathsf{ST}, \mathbf{E} \vdash (\texttt{rec}\ f : \tau_1 \to \tau_2 = \lambda x.e) : \tau_1 \to \tau_2 \leadsto (\texttt{rec}\ f : |\tau_1| \to |\tau_2| = \lambda x.c(e'))}(\leadsto, \to I)$$

Function applications are a little different; in this case, we need to expand the type of the function itself in order to be able to tell what type of argument we need to pass to it. For this reason it will be necessary to translate the function being applied, not at its minimum type, but at its minimum *accessible* type. If, as before, we take the SIL type assignment rule for function applications and make the application of subsumption explicit, we get the following rule:

$$\frac{\mathcal{A}, \mathcal{C}, \mathsf{ST}, \mathbf{E} \vdash e_1 : \tau_1 \to \tau_2 \quad \mathcal{A}, \mathcal{C}, \mathsf{ST}, \mathbf{E} \vdash e_2 : \tau_1' \quad \mathcal{A}, \mathcal{C} \vdash \tau_1' <: \tau_1}{\mathcal{A}, \mathcal{C}, \mathsf{ST}, \mathbf{E} \vdash e_1(e_2) : \tau_2}(\tau :\to E')$$

Since an explicit arrow type appears in the first judgment in the hypothesis of this rule, we need to find an accessible type for e_1 and translate it at that type. We will then get a function that accepts a parameter of type τ_1. If we can coerce the argument we have up to that type, then we can apply the function to it. Thus the translation rule for function applications is:

$$\frac{\mathcal{A}, \mathcal{C}, \mathsf{ST}, \mathbf{E} \vdash e_1 : \tau_1 \to \tau_2 \overset{\text{acc}}{\leadsto} e_1' \quad \mathcal{A}, \mathcal{C}, \mathsf{ST}, \mathbf{E} \vdash e_2 : \tau_1' \leadsto e_2' \quad \mathcal{A}, \mathcal{C} \vdash c : \tau_1' <: \tau_1}{\mathcal{A}, \mathcal{C}, \mathsf{ST}, \mathbf{E} \vdash e_1(e_2) : \tau_2 \leadsto e_1'(c(e_2'))}(\leadsto, \to E)$$

The rules for translating bounded polymorphism can be found by a similar analogy with the type assignment rules in SIL. Since in the hypothesis of the $(\tau : \forall <: I)$ rule (given above) we add to the subtyping context an assumption that the parameter satisfies the bound before type-checking the body, we must do the same with a coercion. This coercion must be obtained from the caller of the function, since only at the site where that type is given will it be known how to coerce it; thus we must wrap the body of the bounded Λ-expression in a λ-expression which will accept the coercion function. We also must allow for the fact that the body may have a minimum type different from the specified return type, as we did for ordinary functions. The rule that does all this can be written as follows:

$$\frac{\begin{array}{c} t \notin \mathsf{ST} \quad c \notin \mathbf{E} \quad f \notin \mathcal{C} \\ \mathcal{A} \cup \{t :: \kappa\}, \mathcal{C} \cup \{c : t <: \sigma\}, \mathsf{ST}, \mathbf{E} \cup \{f : \forall t :: \kappa <: \sigma.\tau\} \vdash e : \tau' \leadsto e' \quad \mathcal{A}, \mathcal{C} \vdash c' : \tau' <: \tau \end{array}}{\begin{array}{l} \mathcal{A}, \mathcal{C}, \mathsf{ST}, \mathbf{E} \vdash (\texttt{rec } f : \forall t :: \kappa <: \sigma.\tau = \Lambda.e) : \forall t <: \sigma.\tau \\ \qquad \leadsto (\texttt{rec } f : \forall t :: \kappa.(\mathrm{coerce}_\kappa(t <: \sigma)) \to |\tau| = \Lambda.\lambda c : (\mathrm{coerce}_\kappa(t <: \sigma)).c'(e')) \end{array}} (\leadsto, \forall I)$$

Fortunately, the rule for applications of bounded polymorphic functions is much simpler. All we have to do for that case is make sure the type of the function is accessible, and provide a coercion up to the bound from the type being passed:

$$\frac{\mathcal{A}, \mathcal{C}, \mathsf{ST}, \mathbf{E} \vdash e : \forall t <: \sigma'.\tau \overset{\mathrm{acc}}{\leadsto} e' \quad \mathcal{A}, \mathcal{C} \vdash c : \sigma <: \sigma'}{\mathcal{A}, \mathcal{C}, \mathsf{ST}, \mathbf{E} \vdash e[\sigma] : \tau[\sigma/t] \leadsto e'[|\sigma|](c)} (\leadsto, \forall E)$$

The rules for bounded existentials are similar to those for universals, except that in that case coercions must be created when the existential values are created, rather than when they are used. These rules can be found in Appendix A along with a complete list of the translation rules.

Chapter 5

Translating $\mathcal{LOOM}$ into SIL

In this chapter we give an overview of the translation our compiler performs to map a subset of $\mathcal{LOOM}$ into the SIL intermediate language.

5.1 Early Stages of Compilation

For the syntactic analysis and type-checking phases of our compiler, we use a very slightly modified version of the front end of the existing $\mathcal{LOOM}$ interpreter [Pet96, Van97]. These segments of the compiler transform the text of one or more $\mathcal{LOOM}$ compilation units into abstract syntax trees, and then check those trees for validity according to the type rules of the language. The type-checker also decorates some parts of the syntax trees with type information that it infers or deduces about the variables and constants in the program. It is after this phase that our new code takes over and transforms the $\mathcal{LOOM}$ program into a corresponding term in SIL.

The language represented by a $\mathcal{LOOM}$ syntax tree differs from the concrete $\mathcal{LOOM}$ language in a few ways. Firstly, in the internal version of the language it is illegal to assign to any value that is not of an explicitly mutable type. Before type-checking can be performed, therefore, it is necessary to make an extra pass over the tree to change all global variables, local variables and instance variables into references similar to the `ref`'s in SIL, IL or ML. All occurrences of a variable on the right-hand side of an assignment, or as a parameter to a procedure call or message send, must be changed to explicitly dereference the variable.

All classes with no explicit superclass are rewritten to automatically inherit from a root class whose definition is prepended to the program. Also, every object type in the program is rewritten so that it includes all the public methods of the top class. Thus every object type in any $\mathcal{LOOM}$ program will match the type `TopType`, which includes methods for cloning an object. The `clone` operation, which makes a new distinct copy of an object by replicating all of its instance variables, is a special case in $\mathcal{LOOM}$, since it is difficult to write a typable `clone` method within the framework of our class encoding. Because it requires such special treatment, we have not dealt with `clone` at all in our prototype compiler.

Finally, the parser rewrites the definition of any class that accepts parameters with which to initialize objects. To simplify the semantics, classes in the core $\mathcal{LOOM}$ language do not take parameters, and so if any parameterized classes appear in a $\mathcal{LOOM}$ program they must be immediately translated away. This is done by wrapping the class in a function that accepts the parameters and returns a new copy of the class pre-instantiated with the parameter values. When a new object is to be created, the routine doing the creation passes the parameters to this function and creates a single object from the resulting class, then throws the class away. The appropriate rewriting of parameterized class definitions is performed by the parser.[1]

Once all the syntactic sugar in a program has been removed, it is type-checked. The type-checker rewrites some portions of the syntax tree, adding explicit type information in many places where the programmer was allowed to leave it out. In particular, procedures, functions and classes declared in the `const` section of a block are decorated with their types, as are the method definitions in a class body. In order to be able to generate all the required type annotations for the SIL representation of a program, we have found it necessary to modify the type-checker to record some type information that was discarded in the original $\mathcal{LOOM}$ type-checker. Specifically, our treatment of objects requires the type of the target of any assignment statement to be known, as well as the type of the receiver of any message send and the function in any function call.

5.2 Translation into SIL

In this section we will give the details of our translation from a subset of $\mathcal{LOOM}$ into SIL.

5.2.1 Translating Types

The simple type constructors of $\mathcal{LOOM}$ are directly paralleled in SIL; more complicated types like objects and classes require a nontrivial encoding.

An object type in our $\mathcal{LOOM}$ abstract syntax has the form $\mathtt{OT}(mt)\{m_1 : \tau_1, \ldots, m_n : \tau_n\}$, where mt is this particular object type's name for itself. Object types as they appear in an actual $\mathcal{LOOM}$ program are always of the form $\mathtt{OT}(\mathtt{MyType})\{\ldots\}$, but the name may be

[1]In fact, while this does reflect the "official" semantics given to classes in the definition of $\mathcal{LOOM}$, it has some unfortunate and unintuitive consequences. In particular, classes are not allowed to accept parameters of type `MyType` because that name is not bound outside the class and hence cannot be the type of a parameter to the wrapper function. Some work has been done on finding an alternative class encoding that would build in parameterization so that `MyType` would be allowed, but our prototype compiler keeps the original semantics.

α-converted by the typechecker to avoid being captured when one object type is substituted into another. In any case, $m_1, \ldots, m_n$ are the names of the object's methods, and $\tau_1, \ldots, \tau_n$ are their types, which may involve mt. We can therefore capture all the essential information about this particular object type with the operator defined by

$$M :: * \to * = \lambda mt.\{m_1 : \tau_1^*, \ldots, m_n : \tau_n^*\},$$

where for each i, τ_i^* is the appropriately translated type of the ith method. Most importantly, we can use this function to define the matching relation using higher-order subtyping [AC95]. If $\texttt{Obj}[M]$ is the object type with method type function M, we can say that $\texttt{Obj}[M] <\# \texttt{Obj}[M']$ (where $<\#$ denotes the matching relation) if and only if $M <: M'$. This is how we translate bounded polymorphism. A $\mathcal{LOOM}$ function of the type $\texttt{func}(T <\# \sigma) : \tau$ will become a SIL function of the type $\forall T <: M_\sigma.\tau^*$, where $\texttt{Obj}[M_\sigma]$ is the object type σ.

Because of this use of type functions for parametric polymorphism, it will not do to simply translate object types the same way any time they appear. We actually have to deal with two representations of each object type, namely the type operator we use to express matching and the type of the objects themselves. The `Obj` operator can be expressed in the type system of SIL[2], so if we have the operator version of an object type we can always find the object version. We cannot go in the other direction, however, and so we must make certain that we always have the operator version of an object type around when we need it. This is accomplished by translating definitions of object types into definitions of the corresponding type operators, and passing the operators to polymorphic functions instead of the actual object types they represent.

Thus in order to make our translation work we need two different ways of translating types. The first, which we will denote by $\mathcal{TT}$, will be translation of a type in the context of a type definition or a type parameter; the second, written $\mathcal{TV}$, will be translation of an expression meant to denote the type of a term or variable. The two will agree in most cases; the only time they will not is when the type to which they are applied is an object type. So that these translation functions can work properly, we must provide each one with a set of all the currently visible type names that stand for object types. We will usually call this $\mathcal{O}$, and we will write the translation of a $\mathcal{LOOM}$ type τ relative to $\mathcal{O}$ as either $\mathcal{TT}_\mathcal{O}[\![\tau]\!]$ or $\mathcal{TV}_\mathcal{O}[\![\tau]\!]$. When we encounter a declaration of a type variable or parameter, we must add it to $\mathcal{O}$ if it stands for an object type.[3] The rules for translating types are shown in Figure 5.1; we will now discuss the interesting cases.

[2] At least, the `Obj` operator for our chosen object encoding can. This was a factor in our choice of encoding, as we might have made some things more efficient by choosing one that was less easily expressible within SIL.

[3] Of course, nearly all uses of type variables in the $\mathcal{LOOM}$ syntax *do* stand for object types, so the only time we encounter a type variable declaration and leave $\mathcal{O}$ unchanged will be in Section 5.2.3, Figure 5.6.

$$
\begin{aligned}
&\mathcal{TV}_{\mathcal{O}}[\![\texttt{integer}]\!] = \texttt{int}\\
&\mathcal{TV}_{\mathcal{O}}[\![\texttt{func}(\sigma) : \tau]\!] = \mathcal{TV}_{\mathcal{O}}[\![\sigma]\!] \to \mathcal{TV}_{\mathcal{O}}[\![\tau]\!]\\
&\mathcal{TV}_{\mathcal{O}}[\![\texttt{func}(T <\# \ \sigma) : \tau]\!] = \forall T :: * \to * <: \mathcal{TT}_{\mathcal{O}\cup\{T\}}[\![\sigma]\!].\mathcal{TV}_{\mathcal{O}\cup\{T\}}[\![\tau]\!]\\
&\mathcal{TV}_{\mathcal{O}}[\![\texttt{ref}\ \tau]\!] = \texttt{ref}\ \mathcal{TV}_{\mathcal{O}}[\![\tau]\!]\\
&\mathcal{TV}_{\mathcal{O}}[\![\texttt{CT}(mt)(\{x_1 : \sigma_1, \ldots, x_k : \sigma_k\}, m_1 : \tau_1, \ldots, m_n : \tau_n)]\!] =\\
&\qquad \text{Class}[(\lambda mt :: *.\{x_1 : \mathcal{TV}_{\mathcal{O}-\{mt\}}[\![\sigma_1]\!], \ldots, x_k : \mathcal{TV}_{\mathcal{O}-\{mt\}}[\![\sigma_k]\!]\}),\\
&\qquad\qquad (\lambda mt :: *.\{m_1 : \mathcal{TV}_{\mathcal{O}-\{mt\}}[\![\tau_1]\!], \ldots, m_n : \mathcal{TV}_{\mathcal{O}-\{mt\}}[\![\tau_n]\!])]\\
&\mathcal{TV}_{\mathcal{O}}[\![\texttt{OT}(mt)(meth)]\!] = \texttt{Obj}[\mathcal{TT}_{\mathcal{O}}[\![\texttt{OT}(mt)(meth)]\!]]\\
&\mathcal{TV}_{\mathcal{O}}[\![T]\!] = \begin{cases} T & \text{if } T \notin \mathcal{O}\\ \texttt{Obj}[T] & \text{if } T \in \mathcal{O}\end{cases}
\end{aligned}
$$

$$
\begin{aligned}
&\mathcal{TT}_{\mathcal{O}}[\![T]\!] = T\\
&\mathcal{TT}_{\mathcal{O}}[\![\texttt{OT}(mt)\{m_1 : \tau_1, \ldots, m_n : \tau_n\}]\!] =\\
&\qquad \lambda mt :: *.\{m_1 : \mathcal{TV}_{\mathcal{O}-\{mt\}}[\![\tau_1]\!], \ldots, m_n : \mathcal{TV}_{\mathcal{O}-\{mt\}}[\![\tau_n]\!]\}\\
&\mathcal{TT}_{\mathcal{O}}[\![\tau]\!] = \mathcal{TV}_{\mathcal{O}}[\![\tau]\!], \qquad \tau \text{ not a variable or object type}
\end{aligned}
$$

The standard preamble defines

$$\texttt{Obj}[M] = \texttt{unit} + \exists Y :: *.(Y \times (Y \to M[\texttt{Obj}[M]]))$$

Figure 5.1: Translating type expressions. Note that while Obj can be expressed directly in the SIL type system, "Class" cannot. Thus class types must be directly expanded by the translator when they are encountered.

The interesting parts of the translation of types are those that involve our object encoding. In particular, when translating the type of an object or a variable of an object type, we use the type operator `Obj`. As the figure shows, we define $\texttt{Obj}[M]$ to be the sum of the unit type and a complicated existential type. The need for the sum arises because object-typed variables in $\mathcal{LOOM}$ really hold references to objects which may be null. We have chosen to represent the null reference of an object type to be a unit value which is injected into the sum. For non-null objects, we use a package of the existential type. The hidden type in the existential is that of the object's record of mutable instance variables. One of the components of the tuple is this instance variable record. The other component of the tuple is a function which, given the instance variable record, returns the method suite as a record of closures. From this record we can then extract any method we wish to invoke. This representation of objects is essentially the ORE encoding from [BCP96].

Figure 5.1 is also missing the definition of the Class operator. This function takes two functions (each of which is a record type parameterized by `MyType`, one for the instance variables and one for the methods) and returns a type of classes that produce the given instance variables and methods. It cannot be defined directly in the type language of SIL, so we really are only using it as a shorthand which is immediately expanded by our translator. It is defined as follows:

$$\mathrm{Class}[I, M] = \forall I' <: I^{\mathrm{ref}}, M' <: M.(I[\texttt{Obj}[M']] \times (\texttt{VisObj}[I', M'] \rightarrow M[\texttt{Obj}[M']]))$$

where $\texttt{VisObj}[I', M']$ is the type of the "internal view" of an object produced by the class. This *visible object type* is essentially the same as the non-null case of the regular object type, except there is no existential and everything is visible. In fact, we can define `VisObj` in SIL as follows:

$$\texttt{VisObj}[I, M] = I[\texttt{Obj}[M]] \times (I[\texttt{Obj}[M]] \rightarrow M[\texttt{Obj}[M]])$$

We also use the notation I^{ref} to mean the type function that returns the type of a record of *references* to the types of the fields of body of I. Thus I^{ref} is the type of the record of mutable instance variables actually handled by an object.[4] The parts of a class, then, are a record containing the initial values of all the instance variables (having type $I[\texttt{Obj}[M']]$) and a function which will return a fully-instantiated method suite for an object given all the internal information about that object (a function of type $\texttt{VisObj}[I', M'] \rightarrow M[\texttt{Obj}[M']]$). All of this is parameterized over all possible extensions of the instance variable and method records, which allows the definitions in the class to be re-used when a subclass is derived. We will see how this works when we discuss the translation of class expressions.

$$\mathcal{E}_{\mathcal{O}}[\![v]\!] = v$$
$$\mathcal{E}_{\mathcal{O}}[\![x]\!] = x$$
$$\mathcal{E}_{\mathcal{O}}[\![\texttt{val}\ e]\!] = !(\mathcal{E}_{\mathcal{O}}[\![e]\!])$$
$$\mathcal{E}_{\mathcal{O}}[\![e_1(e_2)]\!] = (\mathcal{E}_{\mathcal{O}}[\![e_1]\!])(\mathcal{E}_{\mathcal{O}}[\![e_2]\!])$$
$$\mathcal{E}_{\mathcal{O}}[\![e(\tau)]\!] = (\mathcal{E}_{\mathcal{O}}[\![e]\!])[\mathcal{TT}_{\mathcal{O}}[\![\tau]\!]]$$
$$\mathcal{E}_{\mathcal{O}}[\![\texttt{function}(x : \sigma) : \tau\ \mathit{block}]\!] = (\lambda x : \mathcal{TV}_{\mathcal{O}}[\![\sigma]\!].\mathcal{B}_{\mathcal{O}}[\![\mathit{block}]\!])$$
$$\mathcal{E}_{\mathcal{O}}[\![\texttt{function}(T <\#\ \sigma) : \tau\ \mathit{block}]\!] = (\Lambda T <: \mathcal{TT}_{\mathcal{O}\cup\{T\}}[\![\sigma]\!].\mathcal{B}_{\mathcal{O}\cup\{T\}}[\![\mathit{block}]\!])$$
$$\mathcal{E}_{\mathcal{O}}[\![\texttt{nil}_\tau]\!] = \texttt{inj}\ 0\ ()\ \texttt{as}\ \mathcal{TV}_{\mathcal{O}}[\![\tau]\!]$$
$$\mathcal{E}_{\mathcal{O}}[\![(o.m) : \tau]\!] = \begin{cases} \mathit{methfun}(\texttt{inst}).m & \text{if } o \text{ is } \texttt{self} \\ \texttt{inst}.m & \text{if } o \text{ is } \texttt{inst} \\ \begin{array}{l} \texttt{case}\ (\mathcal{E}_{\mathcal{O}}[\![o]\!])\ \texttt{bind}\ n\ \texttt{in abort}\ [\mathcal{TV}_{\mathcal{O}}[\![\tau]\!]] \\ \mid y\ \texttt{in let}\ [IV; (iv, \mathit{meth})] =\ \texttt{open}\ y\ \texttt{in}\ (\mathit{meth}(iv)).m \end{array} & \text{otherwise} \end{cases}$$

Figure 5.2: Translation of simple expressions, including message sends.

5.2.2 Translating Expressions

Most simple expressions are fairly easy to translate, as Figure 5.2 shows. We denote by $\mathcal{E}$ the function that translates expressions; here again we must provide a set $\mathcal{O}$ of type names referring to object types so that polymorphic functions and their applications may be translated properly. Variables, dereferencing operations, function expressions and function applications all translate straightforwardly into their SIL conterparts. To create a null object reference of type τ, denoted by $\texttt{nil}_\tau$, we simply inject a unit expression into the left-hand side of the proper object type. To send a message to an object, we examine it to see whether it is a unit (*i.e.*, a null reference) or an actual object. If it is null, we use SIL's `abort` primitive to make the program die as gracefully as possible. If it is not null, we unpack the object's components, instantiate the method suite with the instance variables, and select the appropriate method.

When we are translating the body of a method in a class definition, we may encounter some references to parts of the object executing the method. In particular, the programmer may use the pseudovariable `self` to refer to this object directly, or she may simply use the name of a method or an instance variable without qualifying it in any way. The type-checker will detect these references and resolve them, so that in the abstract syntax self-inflicted messages always look like message sends with `self` as the receiver and references to instance variables look like selections from a record named `inst`. Since `self` and `inst`

[4] The use of I^{ref} is in fact the reason why the Class operator cannot be written in SIL – the type system does not provide a way to apply the `ref` constructor to each element of a record

are not explicitly defined by the programmer, our translation must carefully ensure that references to them are treated properly. Further, message sends to `self` should be treated differently from other uses of `self`, since in the former case we can use our knowledge that `self` is not a null reference, while if we wish to do anything else with `self` we must treat it as we would any other object reference of type `MyType`.[5]

The proper treatment of `self` and `inst` is accomplished in the translation by examining every expression that looks like a message send or record extraction and checking whether the target is either of these two identifiers. As the figure shows, we translate extractions from the pseudo-record `inst` as extractions from some SIL record of the same name. Messages to `self` are translated by applying a function called *methfun* to the `inst` record and extracting the appropriate field from the record that results. If the name `self` occurs in any other context, it will be translated as itself. Our next task is to ensure that the translated version of the code of any method appears in a context where the names `self`, *methfun* and `inst` are defined and have the proper types. This is done by the translation of classes, which is shown in Figure 5.3.

Recall from Section 5.2.1 that the translation of a class is supposed to be a polymorphic function which ranges over possible extensions of the class and returns two values. The first is a record containing initial values for all the instance variables of the class. The second, which will eventually become the method suite, is a function which, given the "internal view" of an object from the class, returns the record of methods for that object. In order to translate a class, therefore, we need to create such a bounded polymorphic function, using translated versions of the instance variables and methods specified in the class definitions to form the parts of the return value. As Figure 5.3 illustrates, generating the record of initial instance variable values is simple: each initial value expression is translated in a context where `MyType` denotes an object type, and the results are collected in a record with the instance variable names as labels.

The function that produces an object's method suite given the internal view of the object is written as a λ-expression that takes the "visible object" *visself* and returns the record of translated methods. In order for the translated method bodies to make sense, though, there must be definitions of `self`, `inst` and *methfun* visible in the scope where the methods are defined. These definitions are provided by the `let`-bindings surrounding the record of translated method expressions. The record `inst` of mutable instance variables is simply extracted from the visible object, as is the *methfun* function. That function can later be applied to `inst` to get a record containing the methods of `self`. Finally, the visible object *visself* is packed and injected so that it has the type `Obj`[`MyType`], so that the resulting object, called `self`, will be available within the methods in case it must be returned or passed to another function.

[5]In fact there are other reasons to separate these uses of `self`, the most important being that invocations of a hidden method of `self` are allowed.

$$
\begin{array}{l}
\mathcal{E}_{\mathcal{O}}[\![\texttt{class}(\{x_1 = e_1, \ldots, x_k = e_k\},\\
\qquad\qquad \{m_1 = f_1, \ldots, m_n = f_n\}) : \texttt{ClassType}(mt)(iv, m)]\!] =\\
\quad \Lambda I' <: I^{\text{ref}}, \texttt{MyType} <: M.\\
\quad (\{x_1 = \mathcal{E}_{\mathcal{O}\cup\{\texttt{MyType}\}}[\![e_1]\!], \ldots, x_k = \mathcal{E}_{\mathcal{O}\cup\{\texttt{MyType}\}}[\![e_k]\!]\},\\
\quad \lambda \mathit{visself} : \texttt{VisObj}[I', \texttt{MyType}].\\
\qquad \texttt{let inst} = \#0\ \mathit{visself}\ \texttt{in}\\
\qquad \texttt{let}\ \mathit{methfun} = \#1\ \mathit{visself}\ \texttt{in}\\
\qquad \texttt{let}\ \mathit{packedself} = \texttt{pack}\ [I'[\texttt{Obj}[\texttt{MyType}]]; \mathit{visself}]\\
\qquad\quad \texttt{as}\ \exists Y.Y \times (Y \rightarrow \texttt{MyType}[\texttt{Obj}[\texttt{MyType}]])\ \texttt{in}\\
\qquad \texttt{let self} = \texttt{inj}\ 1\ \mathit{packedself}\ \texttt{as}\ \texttt{Obj}[\texttt{MyType}]\ \texttt{in}\\
\qquad\quad \{m_1 = \mathcal{E}_{\mathcal{O}\cup\{\texttt{MyType}\}}[\![f_1]\!], \ldots, m_n = \mathcal{E}_{\mathcal{O}\cup\{\texttt{MyType}\}}[\![f_n]\!]\})
\end{array}
$$

$$
\begin{array}{l}
\mathcal{E}_{\mathcal{O}}[\![\texttt{inherit}(sc, \{x_1 = e_1, \ldots, x_k = e_k\},\\
\qquad\qquad\qquad \{m_1 = f_1, \ldots, m_n = f_n\}) : \texttt{ClassType}(mt)(iv, m)]\!] =\\
\quad \Lambda I' <: I^{\text{ref}}, \texttt{MyType} <: M.\\
\quad \texttt{let}\ \mathit{superparts} = (\mathcal{E}_{\mathcal{O}}[\![sc]\!])[I', \texttt{MyType}]\ \texttt{in}\\
\quad \texttt{let}\ \mathit{superivinit} = \#0\ \mathit{superparts}\ \texttt{in}\\
\quad \texttt{let}\ \mathit{superfun} = \#1\ \mathit{superparts}\ \texttt{in}\\
\quad (\mathit{superivinit} \oplus \{x_1 = \mathcal{E}_{\mathcal{O}\cup\{\texttt{MyType}\}}[\![e_1]\!], \ldots, x_k = \mathcal{E}_{\mathcal{O}\cup\{\texttt{MyType}\}}[\![e_k]\!]\},\\
\quad \lambda \mathit{visself} : \texttt{VisObj}[I', \texttt{MyType}].\\
\qquad \texttt{let}\ \mathit{supermeths} = \mathit{superfun}(\mathit{visself})\ \texttt{in}\\
\qquad \texttt{let inst} = \#0\ \mathit{visself}\ \texttt{in}\\
\qquad \texttt{let}\ \mathit{methfun} = \#1\ \mathit{visself}\ \texttt{in}\\
\qquad \texttt{let}\ \mathit{packedself} = \texttt{pack}\ [I'[\texttt{Obj}[\texttt{MyType}]]; \mathit{visself}]\\
\qquad\quad \texttt{as}\ \exists Y.Y \times (Y \rightarrow \texttt{MyType}[\texttt{Obj}[\texttt{MyType}]])\ \texttt{in}\\
\qquad \texttt{let self} = \texttt{inj}\ 1\ \mathit{packedself}\ \texttt{as}\ \texttt{Obj}[\texttt{MyType}]\ \texttt{in}\\
\qquad\quad \mathit{supermeths} \oplus \{m_1 = \mathcal{E}_{\mathcal{O}\cup\{\texttt{MyType}\}}[\![f_1]\!], \ldots, m_n = \mathcal{E}_{\mathcal{O}\cup\{\texttt{MyType}\}}[\![f_n]\!]\})
\end{array}
$$

where in both cases $\mathcal{TV}_{\mathcal{O}}[\![\texttt{ClassType}(mt)(iv, m)]\!] = \text{Class}[I, M]$.

Figure 5.3: Translating classes and inheritance. The right-dominant record combination operator $\oplus$ does not exist in SIL; we use it as a shorthand, which can be expanded by the translator since all the names and types of the fields are known.

$$
\begin{array}{l}
\mathcal{E}_{\mathcal{O}}[\![\texttt{new}\ (c : \texttt{ClassType}(mt)(iv, m))]\!] = \\
\quad \texttt{let}\ \mathit{classparts} = \mathcal{E}_{\mathcal{O}}[\![c]\!][I^{\mathrm{ref}}, M]\ \texttt{in} \\
\quad \texttt{let}\ \mathit{ivinit} = \#0\ \mathit{classparts}\ \texttt{in} \\
\quad \texttt{let}\ \mathit{premethfun} = \#1\ \mathit{classparts}\ \texttt{in} \\
\quad \texttt{let}\ \mathit{ivs} = \{x_1 = \texttt{ref}\ (\mathit{ivinit}.x_1), \ldots, x_k = \texttt{ref}\ (\mathit{ivinit}.x_k)\}\ \texttt{in} \\
\quad \texttt{letrec}\ \mathit{methfun} : I^{\mathrm{ref}}[\texttt{Obj}[M]] \to M[\texttt{Obj}[M]] = \lambda\ \mathit{ivs}. \\
\qquad \texttt{let}\ \mathit{visself} = (\mathit{ivs}, \mathit{methfun})\ \texttt{in}\ \mathit{premethfun}(\mathit{visself}) \\
\quad \texttt{in let}\ \mathit{visself} = (\mathit{ivs}, \mathit{methfun})\ \texttt{in} \\
\quad \texttt{let}\ \mathit{packedself} = \texttt{pack}\ [I^{\mathrm{ref}}[\texttt{Obj}[M]]; \mathit{visself}] \\
\qquad \texttt{as}\ \exists Y.Y \times (Y \to M[\texttt{Obj}[M]])\ \texttt{in} \\
\qquad \texttt{inj}\ 1\mathit{packedself}\ \texttt{as Obj}\ [M]
\end{array}
$$

Where: $iv = \{x_1 : \sigma_1, \ldots, x_k : \sigma_k\}$, and
$\mathcal{TV}_{\mathcal{O}}[\![\texttt{ClassType}(mt)(iv, m)]\!] = \mathrm{Class}[I, M]$

Figure 5.4: Translating creation of new objects.

The translation of a class produced by inheritance is not too much more complicated than that of a root class. In the body of the polymorphic function generated by the second part of Figure 5.3, the superclass *sc* of the class being defined is evaluated and instantiated with the proper values of I' and `MyType`. The parts of the superclass – *i.e.*, its initial instance variable values and its method-producing function – are then extracted and named *superivinit* and *superfun*, respectively. The rest of the translated subclass is exactly analogous to the root class case, except that the instance variable and method records returned are those formed by combining the components specified by the subclass with those obtained from the superclass using the right-dominant combination operator $\oplus$. The $\oplus$ operator stands for the merging of two records, where if there is a field label that is used in both, the value from the right-hand side should be used – that is, if a method is defined in both the superclass and the subclass, then the new version should override the old. This kind of record combination is not available directly in SIL, but since the names and types of all the fields in the records involved are known, the translator can expand the shorthand without difficulty.

The final type of expressions for which we must describe the translation is the application of the `new` operator, which takes a class and constructs a new object from that class. The translation is shown in Figure 5.4. The first step in creating an object is to evaluate the class itself, and instantiate it with the actual types its instance variables and methods will have. The class will return a pair consisting of the initial values for its instance variables and a function that will give its methods. The next step is to create `ref` values for each of the instance variables, and pack them into a record which is named *ivs*. This record will become the hidden instance variable component of the object; to get the methods, we must construct a function that takes this instance variable record and produces the method suite. The definition of such a function, which we call *methfun*, appears after the instance variable record. Once the instance variable record and method suite function for the object have been defined, they can be put together into a visible object, then packed to hide the instance variables and injected into the object type to complete the operation.

A few remarks need to be made regarding the *methfun* function. The first is that it is recursive, in that it passes itself to the function provided by the class. In the translation of classes, we carefully avoided making the method suite function recursive, so this recursion in *methfun* is responsible for any recursive behavior in or between the methods of an object. There is a good reason for keeping recursion out of the class itself, and introducing it only when an object is created. Because self-reference is not introduced until object creation time, any method that calls another method will be sure to call the right one, even if the method it is invoking has been overridden. The other observation about *methfun* is that it does not make any direct reference to the instance variable record that has just been created. This means that the code for *methfun* is the same in every instance of object creation for a given class. In fact, it would theoretically be possible to make all objects of a given class share the same *methfun* code, if that function were somehow made an integral part of the class and passed around with the other components. We have chosen not to do this in our prototype compiler, but it is an enhancement we are considering for the future.

5.2.3 Translation of Block Structures

Since the imperative features of $\mathcal{LOOM}$ are uncomplicated, their translation into the SIL intermediate language is straightforward. In this section we will describe the implementation of statements and blocks, which will complete our discussion of $\mathcal{LOOM}$ compilation.

Statements

There are three kinds of statements supported in our prototype compiler: assignments, conditionals, `while` loops, and procedure calls. Each of these can be expressed with a sequence of `let`-bindings in SIL, so a statement list becomes a long sequence of bindings, followed by a simple unit expression. This is a little tricky to express in our translation, since it means that the translation of a single statement is not a complete SIL term; we get around this by having our statement translation function, which we denote by $\mathcal{S}$, return a function from SIL terms to SIL terms. Intuitively, the translated version $\mathcal{S}_{\mathcal{O}}[\![s]\!]$ of a $\mathcal{LOOM}$ statement s will be a function which, given the translation of the rest of the statement list in which s appeared, will prepend the bindings corresponding to s. Thus the SIL term it returns will be the translation of the sequence consisting of s and all the statements that follow it.

$$
\begin{array}{l}
\mathcal{S}_{\mathcal{O}}[\![e_1 \ := \ e_2]\!](k) = \texttt{let } \mathcal{E}_{\mathcal{O}}[\![e_1]\!] := \mathcal{E}_{\mathcal{O}}[\![e_2]\!] \texttt{ in } k \\
\mathcal{S}_{\mathcal{O}}[\![e_1(e_2)]\!](k) = \texttt{let } \mathit{temp} = (\mathcal{E}_{\mathcal{O}}[\![e_1]\!])(\mathcal{E}_{\mathcal{O}}[\![e_2]\!]) \texttt{ in } k \\
\mathcal{S}_{\mathcal{O}}[\![e(\tau)]\!](k) = \texttt{let } \mathit{temp} = (\mathcal{E}_{\mathcal{O}}[\![e]\!])[\mathcal{TT}_{\mathcal{O}}[\![\tau]\!]] \texttt{ in } k \\
\mathcal{S}_{\mathcal{O}}[\![\texttt{if } e \texttt{ then } \mathit{stmts1} \texttt{ else } \mathit{stmts2} \texttt{ end}]\!](k) = \\
\quad \texttt{let } \mathit{dummy} = \texttt{if } \mathcal{E}_{\mathcal{O}}[\![e]\!] \texttt{ then } \mathcal{SS}_{\mathcal{O}}[\![\mathit{stmts1}]\!](()) \texttt{ else } \mathcal{SS}_{\mathcal{O}}[\![\mathit{stmts2}]\!](()) \\
\quad \texttt{in } k \\
\mathcal{S}_{\mathcal{O}}[\![\texttt{while } e \texttt{ do } \mathit{stmts} \texttt{ end}]\!](k) = \\
\quad \texttt{letrec } \mathit{loop} : \texttt{unit} \to \texttt{unit } = \lambda \mathit{dummy}. \\
\qquad \texttt{if } \mathcal{E}_{\mathcal{O}}[\![e]\!] \texttt{ then } \mathcal{SS}_{\mathcal{O}}[\![\mathit{stmts}]\!](\mathit{loop}()) \texttt{ else } () \\
\quad \texttt{in let } \mathit{dummy} = \mathit{loop}() \texttt{ in } k \\
\\
\mathcal{SS}_{\mathcal{O}}[\![\epsilon]\!](k) = k \\
\mathcal{SS}_{\mathcal{O}}[\![s; \mathit{stmts}]\!](k) = \mathcal{S}_{\mathcal{O}}[\![s]\!](\mathcal{SS}_{\mathcal{O}}[\![\mathit{stmts}]\!](k))
\end{array}
$$

Figure 5.5: Translation of statements and statement lists.

The rules for translating statements are given in Figure 5.5. The case for assignment statements is the simplest: it just translates both sides of the assignment and generates the corresponding SIL assignment expression. The cases for procedure calls are very similar to the parts of the expression translation that handle function calls. Procedures are translated as functions that return `unit`, so the translated version of a procedure call must assign the `unit` value that is returned to a dummy variable. The conditional construct of $\mathcal{LOOM}$ is translated directly into the `if-then-else` construct of SIL. The only nontrivial case of statement translation is the `while` loop. Since SIL is a functional language, there is no built-in iteration construct and so we have to use recursion. The translation of a loop, therefore, builds a parameterless function called *loop* which tests the condition and, if it is true, executes the loop body and calls itself recursively. When the intermediate program is written in A-normal form, this recursive call will be in the tail position, so the recursion will be translated as iteration when the IL program is compiled. The loop is executed by calling the *loop* function once after it is defined.

Figure 5.5 also shows the way in which statements are strung together to form statement lists. Notice that the $\mathcal{SS}$ function also returns a function from SIL terms to SIL terms; this provides the flexibility necessary to allow the same statement-list translation function to be used in all of the contexts where statement lists appear: conditionals, loops and blocks.

$$\mathcal{D}_{\mathcal{O}}[\![\texttt{type } t = \tau]\!](k) = \texttt{lettype } t = (\texttt{Rec } t.\mathcal{TT}_{\mathcal{O}}[\![\tau]\!]) \texttt{ in } k$$
$$\mathcal{D}_{\mathcal{O}}[\![\texttt{var } x : \tau]\!](k) = \texttt{let } x = \texttt{ref DV}(\tau) \texttt{ in } k$$
$$\mathcal{D}_{\mathcal{O}}[\![\texttt{const } x = e]\!](k) = \texttt{let } x = \mathcal{E}_{\mathcal{O}}[\![e]\!] \texttt{ in } k$$

$$\mathcal{B}_{\mathcal{O}}[\![\texttt{type } t = \tau; \textit{typedefs constdefs vardecls stmts } \texttt{return } e \texttt{ end }]\!] =$$
$$\quad \mathcal{D}_{\mathcal{O}}[\![\texttt{type } t = \tau]\!](\mathcal{B}_{\mathcal{O}}[\![\textit{typedefs constdefs vardecls stmts } \texttt{return } e \texttt{ end }]\!]),$$
$$\qquad \text{if } \tau \text{ is not an object type.}$$
$$\mathcal{B}_{\mathcal{O}}[\![\texttt{type } t = \tau; \textit{typedefs constdefs vardecls stmts } \texttt{return } e \texttt{ end }]\!] =$$
$$\quad \mathcal{D}_{\mathcal{O}}[\![\texttt{type } t = \tau]\!](\mathcal{B}_{\mathcal{O}\cup\{t\}}[\![\textit{typedefs constdefs vardecls stmts } \texttt{return } e \texttt{ end }]\!]),$$
$$\qquad \text{if } \tau \text{ is an object type.}$$
$$\mathcal{B}_{\mathcal{O}}[\![\texttt{const } x = f; \textit{constdefs vardecls stmts } \texttt{return } e \texttt{ end }]\!] =$$
$$\quad \mathcal{D}_{\mathcal{O}}[\![\texttt{const } x = f]\!](\mathcal{B}_{\mathcal{O}}[\![\textit{constdefs vardecls stmts } \texttt{return } e \texttt{ end }]\!])$$
$$\mathcal{B}_{\mathcal{O}}[\![\texttt{var } x : \tau; \textit{vardecls stmts } \texttt{return } e \texttt{ end }]\!] =$$
$$\quad \mathcal{D}_{\mathcal{O}}[\![\texttt{var } x : \tau]\!](\mathcal{B}_{\mathcal{O}}[\![\textit{vardecls stmts } \texttt{return } e \texttt{ end }]\!]$$
$$\mathcal{B}_{\mathcal{O}}[\![\textit{stmts } \texttt{return } e \texttt{ end }]\!] =$$
$$\quad \mathcal{SS}_{\mathcal{O}}[\![\textit{stmts}]\!](\mathcal{E}_{\mathcal{O}}[\![e]\!])$$

Figure 5.6: Translation of Blocks

Blocks

The translation of a $\mathcal{LOOM}$ program block, containing various kinds of declarations, is given in Figure 5.6. As discussed in Chapter 2, a block that appears as the body of a procedure, function or method may contain local constant and variable declarations but not type definitions, while a the block containing the body of a whole program may define types. Also, the block for a function must contain a `return` expression, while a block for a program or procedure cannot. We ignore these distinctions in our presentation of the translation. To make our figures more compact, we will treat all blocks as possibly containing local type definitions, and we will assume the parser transforms blocks with no return value into blocks that return `unit`.

As the figure illustrates, a block is processed by translating each of its declarations and definitions, followed by its statement list, and ending with its return expression. We use $\mathcal{D}$ to represent the translation function for all kinds of declarations, and the function $\mathcal{SS}$, defined previously, for lists of statements. The function $\mathcal{B}$, which translates blocks, is defined recursively so that it effectively traverses the entire block, turning definitions and statements into `let` expressions as it goes. Type definitions are processed first, followed by constant definitions, variable declarations and finally the statement list. When types are translated, they are wrapped with a `Rec` so that recursive types will work. Of course, this will fail in the presence of mutually recursive types or types used before they are declared; eventually we will have to extend the compiler to handle mutually recursive types, but for now we make do with single recursion only. Note that the subset of $\mathcal{LOOM}$ accepted by our prototype compiler does not allow mutual recursion of constants either. SIL does allow mutual recursion of functions (and, of course, $\mathcal{LOOM}$ classes become functions in SIL), so extending the compiler to properly handle mutual recursion would be straightforward and uninteresting.[6] We intend to implement these extensions in the near future.

The other thing to notice about the translation of blocks is the use of the function DV in the translation of a variable declaration. Since SIL's primitive for creating `ref` values requires an initial value for the contents of the reference cell, we are forced to explicitly initialize all variables declared in a $\mathcal{LOOM}$ program. To do this, we define the auxiliary function DV to give the "default value" for any type τ. For primitive types, choosing this default value is simple – DV(`integer`) is 0, DV(`bool`) is `false`, and so on. If a $\mathcal{LOOM}$ program ever contains a variable that is of a function type, our translator will generate a default function which, when called, prints out a message complaining that an uninitialized variable has been used and halts the program. Finally, the default value of any object type is the `nil` reference of that type. Of course, in order to be able to compute DV(τ) we have to be able to expand the type τ enough to determine its shape, so in fact our translation must carry around more context information than we have been showing.

[6]The only interesting part would be that the definition of $\mathcal{LOOM}$ might have to be revised, so that illegal recursions (ones involving non-functional constants) are ruled out successfully.

Chapter 6

Conclusion

In this chapter we will summarize the foregoing chapters and evaluate the results of our work. Finally, we will conclude by discussing some possibilities for future related work.

6.1 Summary

In this thesis, we have described a compilation strategy for a statically-typed object-oriented language, $\mathcal{LOOM}$. In particular, we have outlined a translation algorithm transforming an abstract core of the $\mathcal{LOOM}$ language into SIL, a typed intermediate language based on $\mathsf{F}_\omega^{<:}$, the higher-order polymorphic λ-calculus with subtyping and extended with tuples, variants, records and several primitive types. We have also given a translation from SIL into IL, a second typed intermediate language based on F_ω and similar to the intermediate languages of other recently-developed compilers. We have discussed the type rules and semantics of both our intermediate languages, and have claimed that our translations preserve both type-soundness and meaning in the programs to which they are applied.

We have used these intermediate languages and translation mechanisms to implement a partial prototype compiler for $\mathcal{LOOM}$, based on the front-end (syntactic analysis and type-checking) of the original $\mathcal{LOOM}$ interpreter. While we have neither discussed nor implemented any optimizations or code generation for our compiler, we believe that the similarity between our intermediate language IL and the intermediate languages of the TIL and FLINT compilers will eventually allow us to create a full-fledged compiler by providing a translation from IL to one of those languages. The success of such an endeavor would depend on the extension of the TIL or FLINT intermediate language to support impredicative polymorphism and existential types; the FLINT group has stated [Sha98] that they plan to add existentials in the near future. For the meantime, we have implemented an interpreter for IL, which we can use to run partially-compiled $\mathcal{LOOM}$ programs and debug our compiler. For simple programs, the execution speed of the IL interpreter on a translated $\mathcal{LOOM}$ program seems comparable to that of the original $\mathcal{LOOM}$ interpreter.

We believe that our two-step compilation strategy using λ-calculus-like typed intermediate languages is appropriate for $\mathcal{LOOM}$. The original ancestor of $\mathcal{LOOM}$, TOOPL [Bru93], was designed with an encoding in λ-calculus firmly in mind. Formal semantics for TOOPLE, a slightly extended version, were discussed in in [BCK93]. The operational and denotational semantics for the imperative language TOIL were given in the thesis [vG93], and a formal description of $\mathcal{LOOM}$ has also been given. Finally, the type rules and semantics for an extension of $\mathcal{LOOM}$ were specified in terms of a λ-calculus encoding in [BV99]. Thus, we feel that the history of $\mathcal{LOOM}$ and its predecessors justifies the use of a compilation strategy based on its embedding in λ-calculus. Furthermore, F_ω is a sufficiently powerful type system that just about any foreseeable extension to $\mathcal{LOOM}$ that may be derived in the future will probably also be representable in SIL and IL. This should make the compiler easily extensible to keep up with new revisions to the language.

While we firmly believe the use of a functional intermediate language is appropriate for the $\mathcal{LOOM}$ language, it does appear to be very inefficient compared to more direct compilation strategies. In particular, the act of sending a message to an object, which in many object-oriented language implementations is as simple as looking up a code pointer in a method table and performing a single function call with the object as an extra parameter, becomes very complicated when translated into a language without explicit objects. When a $\mathcal{LOOM}$ program sends a message, it must unpack the object and pass the instance variables into the method suite. The function to which the instance variable record is passed, moreover, must pass a closure of itself into another function, which then creates closures for all the methods of the object, packs them together into a record, and returns them. The sender of the message then selects the single closure in which it is interested, and invokes it – when everything is taken into account, the process is much more complicated in the SIL version of a $\mathcal{LOOM}$ program than it would be if the message-send were translated directly to machine code. Worse, if the type of the object is not statically known then the IL version of the program will have to instantiate and apply a polymorphic coercion function.

The semantics of message sending are very complicated in our system, but it is possible that the optimization modules of a well-designed back-end would lessen the amount of work that must be done. For example, multiple message sends to the same object could be coalesced to share a single copy of the method suite, so the business of creating the record of closures would not have to be repeated every time. Furthermore, in cases where the class of an object could be statically determined using dataflow analysis, the tedious lookup mechanism could probably be avoided altogether. Finally, clever representation decisions concerning continuations and closure environments could reduce the need for re-evaluating all the methods of an object each time a message is sent. Experimentation with standard optimizations will likely have to wait until we can connect our compiler to a back-end developed elsewhere, but there is reason to hope that a clever optimizer could greatly reduce the inefficiency of the code produced by our translation. Even if it should turn out that well-known optimizations do not have much effect on the efficiency of our code, we believe that the efficiency penalties are worth paying in exchange for the convenience of semantics-directed compilation.

6.2 Future Work

In this section, we will discuss potential directions for future work on this compiler, the translations underlying it, and $\mathcal{LOOM}$ in general.

6.2.1 Extensions to the Compiler

Since our prototype compiler only handles a subset of the $\mathcal{LOOM}$ language, a substantial amount of work must still be done to expand our translation to cover the entire source language. We have mentioned some of the unimplemented features in our discussion of the translation, specifically the inability to support mutual recursion of types or of constant definitions. There may be some complications in implementing these. In the case of types, the SIL and IL languages lack the capacity to explicitly express mutual recursion, so an extension of the intermediate languages will be necessary in order to compile mutually recursive types properly. The appropriate extension is probably to allow the definition of *sequences* of types – that is, to add to the kind system of the language a "type tuple" constructor which would stand for a group of several types. The existing `Rec` constructor could then be used to create a recursive group of types. This kind of construction would be essential if we were to attempt an implementation of the `TypeGroup` and `ClassGroup` constructs of $\mathcal{LOOM}$.

In the case of mutually recursive functions and/or classes, the problem does not lie with SIL or IL, but with $\mathcal{LOOM}$ itself. The constant definitions in a $\mathcal{LOOM}$ block may appear in any order, which means that numerical, boolean or string constants may be intermixed with functional and class constants. Furthermore, arbitrary expressions may be used as the right-hand side of a constant binding, which means that it is not necessarily obvious until type-checking time which constants are functions or classes and which ones are primitive values. Since so much freedom is granted to the programmer in $\mathcal{LOOM}$, but only λ- and Λ-expressions may be recursive in SIL, the task of determining how best to group a block's constants into `letrec` definitions is nontrivial. It may turn out that the best solution is to place new restrictions on the order in which constants may be defined in $\mathcal{LOOM}$. In any case, extension of the compiler to support mutual recursion in constants is left as future work.

While in our presentation of the translation of $\mathcal{LOOM}$ into SIL we showed how polymorphic function types, expressions and applications are to be represented, this has not actually been implemented yet in the compiler. The reason for this omission was simply that there was not enough time. Even though functions in $\mathcal{LOOM}$ may take multiple parameters, of which some are types and some are values [Van97], all that has to be done to translate polymorphic functions is to separate the parameters to each $\mathcal{LOOM}$ function into the type parameters and the value parameters. All the type parameters may be moved out to the front of the list, and the entire function translated as a bounded Λ-expression whose body is a λ-expression accepting the value parameters. Polymorphic procedure and function applications can be translated analogously; we anticipate no difficulty in performing this extension to the compiler.

There are also several aspects of $\mathcal{LOOM}$'s object-oriented features which we have not addressed at all in this thesis. We believe most of them can be implemented in our compiler by extensions to the object and class encodings. To begin with, we have ignored the fact that any $\mathcal{LOOM}$ class is allowed to have "hidden" methods, which are only visible inside the methods of the class and its descendants. Any class method is also allowed to access the method implementations of the superclass by sending messages to the pseudovariable `super`. Finally, it is possible in $\mathcal{LOOM}$ to compare two object references to see if they are "equal" – that is, point to the identically same object in memory.

This last is particularly annoying to implement in SIL, since testing reference equality is not supported at all, except for the fact that values of `ref` types may be compared to see if they are the same. Furthermore, because of our coercion-based implementation of subtyping in the translation to IL, an object may be disassembled and reassembled several times, which means that two pointers to the same object may in fact point to separate copies of the same instance variable references. The only way we know of to implement equality testing in SIL and IL is to augment every object's instance variable record with an extra member, its "identity." The existential in the object encoding could then be replaced by a bounded existential revealing the presence of the "identity" field, and two non-null objects could be tested for equality by comparing their identities. In order to ensure that we never run out of unique identities for objects, this field could be of a `ref` type; this way a unique one could be assigned to every live object, but when an object was garbage-collected the address its identity pointed to would become available once more. We have not implemented this, but intend to do so in the future.

Another object-oriented feature of $\mathcal{LOOM}$ which will be tricky to encode in SIL is the `clone` method of the standard global base class. The intended behavior of this method is to create a new, separate copy of the object in which it is called, replicating all the instance variable values. It is not possible to write this method in a type-safe way within the $\mathcal{LOOM}$ language, so it must be provided as a primitive; however, we hope to be able to implement it in SIL anyway. The tricky part is to ensure that the `clone` method is always automatically updated when a class is inherited, so the version of it that is called for a given object knows about all that object's instance variables. The design and implementation of a `clone` feature for $\mathcal{LOOM}$ objects in SIL and IL is left as future work.

Finally, all the advanced features of $\mathcal{LOOM}$ discussed in Section 2.5 remain to be implemented in the compiler. These are, in particular, #-types, modules and groups. We are confident that the known encoding of #-types using bounded existentials will work; the only difficulty will be in dealing with (positive) occurrences of `MyType` in the type of a message sent to an object of a #-type. That case will require the application of some ugly coercions or wrapper functions. The original module system as described in [Pet96] seems to have a straightforward encoding using bounded existentials, and the extended module system described in [Van97] should not be too much more difficult. Lastly, `TypeGroup`'s and `ClassGroup`'s are fairly well understood, and a candidate encoding for them is given in [BV99]. We believe that this encoding will work, once it is extended to account for all the technical considerations left out of that fairly abstract paper. Of course, the implementation of the `TypeGroup` construct will rely heavily on mutual recursion between types, so these fancy features cannot be implemented until the issues involving recursion are resolved.

6.2.2 Theorems to Prove

The formal specification of the translation algorithms we employ is useless unless it can be applied to proving theorems about our compiler. In particular, we would like to prove that our complete translation always produces a type-correct IL program given a type-correct $\mathcal{LOOM}$ program, and that the computation performed by the target program is exactly that which was intended by the source program. Since our translation is performed in two separate stages, and a total of three different languages are involved, there are several parts to this task. None of the proofs of these theorems have yet been written up, but some are closely related to published work.

First, there are several theorems that need to be proved about the languages involved, beginning with $\mathcal{LOOM}$ itself. None of the proofs for $\mathcal{LOOM}$ has been published as yet, but they are similar to the corresponding ones for TOIL and PolyTOIL, which in turn are based on the theoretical work on TOOPLE [BCM+93]. In particular, proofs need to be given that the type rules of $\mathcal{LOOM}$ are sound with respect to the operational semantics that have been given, and that the typability of a $\mathcal{LOOM}$ program is decidable – in other words, that a sound and complete typechecking algorithm exists. Once the soundness of $\mathcal{LOOM}$ has been established, we must do the same for the SIL and IL intermediate languages. This should be a straightforward extension of the well-known proofs of soundness and decidability for F_ω and $\mathsf{F}_\omega^{<:}$.

Once the desired properties of all three of the type systems involved have been proven, the next step is to show that if a $\mathcal{LOOM}$ program is type-correct then our translation algorithm will map it to a type-correct IL program. Naturally, this would be accomplished by first showing that the $\mathcal{LOOM}$-to-SIL translation preserves type soundness, and then showing that the SIL-to-IL translation does as well. It will follow immediately that the composition of these two mappings on programs will be type-soundness-preserving. The proofs that the translations preserve types will probably be by straightforward, but very tedious, induction on the structure of the terms being translated. The final step in proving that our compiler works properly is to show that the translations preserve the "meaning" of the terms upon which they act. This, too, would be accomplished by considering each stage of the translation separately, but the proofs would be much more complicated than those for type-safety. One source of difficulty would be the presence of mutable variables and side-effects in all three languages; while the paper [BTCGS94] gives a proof that a coercion-based translation of subtyping preserves the meaning of terms, it does so for purely functional source and target languages, so it is not clear whether that proof would be applicable to our situation. Nevertheless, we are confident that an appropriate proof of correctness is possible and that our translation represents a demonstration of the applicability of typed intermediate represenations to the compilation of object-oriented languages.

Bibliography

[AC93] Roberto Amadio and Luca Cardelli. Subtyping recursive types. *ACM Transactions on Programming Languages and Systems*, 15(4):575–631, 1993.

[AC95] Martin Abadi and Luca Cardelli. On subtyping and matching. In *Proceedings of the 1995 European Conference on Object-Oriented Programming*, pages 145–167, 1995.

[AG96] Ken Arnold and James Gosling. *The Java Programming language.* Addison-Wesley, 1996.

[AM87] Andrew W. Appel and David B. MacQueen. A Standard ML Compiler. In *Third Conf. on Functional Programming Languages and Computer Architecture*, pages 301–324. Springer-Verlag LNCS 274, 1987.

[AM91] Andrew W. Appel and David B. MacQueen. Standard ML of New Jersey. In *Third Int'l Symposium on Programming Language Implementation and Logic Programming*, pages 1–13. Springer-Verlag LNCS 528, 1991.

[App92] Andrew W. Appel. *Compiling With Continuations.* Cambridge University Press, 1992.

[BCK93] Kim B. Bruce, Jonathan Crabtree, and Gerald Kanapathy. An operational semantics for TOOPLE: A statically-typed object-oriented programming language. In *Proceedings of MFPS '93, LNCS 802*, pages 603–626. Springer-Verlag, 1993.

[BCM+93] Kim B. Bruce, Jon Crabtree, Thomas P. Murtagh, Robert van Gent, Allyn Dimock, and Robert Muller. Safe and decidable type checking in an object-oriented language. In *1993 ACM Symposium on Object-Oriented Programming: Systems, Languages and Applications*, 1993.

[BCP96] Kim B. Bruce, Luca Cardelli, and Benjamin Pierce. Comparing object encodings. In *Proceedings of the Third Workshop on Foundations of Object Oriented Languages*, 1996.

[BFP97] Kim Bruce, Adrian Fiech, and Leaf Petersen. Subtyping is not a good "match" for object-oriented languages. In *Proceedings of the 1997 European Conference on Object-Oriented Programming*, pages 104–127. Springer LNCS 1241, 1997.

[BFSvG95] Kim B. Bruce, Adrian Fiech, Angela Schuett, and Robert van Gent. PolyTOIL: A type-safe polymorphic object-oriented language. In *Proceedings of the 1995 European Conference on Object-Oriented Programming*, pages 27–51. Springer LNCS 952, 1995.

[BOSW97] Gilad Bracha, Martin Odersky, David Stoutamire, and Philip Wadler. Generic Java: Java with type parameters. Javasoft, Bell Labs, and University of South Australia, November 1997.

[BOW98] Kim Bruce, Martin Odersky, and Philip Wadler. A statically safe alternative to virtual types. In *Proceedings of the 1998 European Conference on Object-Oriented Programming*, pages 523–549. Springer, 1998.

[BRTT93] Lars Birkedal, Nick Rothwell, Mads Tofte, and David N. Turner. The ML Kit, Version 1. Technical Report 93/14, DIKU, 1993.

[Bru93] Kim B. Bruce. Safe type checking in a statically typed object-oriented programming language. In *Proceedings of the 1993 ACM Symposium on Principles of Programming Languages*, pages 285–298, 1993.

[Bru97] Kim B. Bruce. Typing in object-oriented languages: Achieving expressiveness and safety. Technical report, Williams College, 1997.

[BTCGS94] Val Breazu-Tannen, Thierry Coquand, Carl A. Gunter, and Andre Scedrov. Inheritance as implicit coercion. In Carl A. Gunter and John C. Mitchell, editors, *Theoretical Aspects of Object-Oriented Programming: Types, Semantics, and Language Design*, Foundations of Computing Series. The MIT Press, 1994.

[Bur98] Jonathan L. Burstein. *Rupiah: An extension to Java supporting match-bounded parametric polymorphism, ThisType, and exact typing.* Williams College Senior Honors Thesis, 1998.

[BV99] Kim B. Bruce and Joseph C. Vanderwaart. Semantics-driven language design: Statically type-safe virtual types in object-oriented languages. In *Preliminary Proceedings of the Fifteenth Conference on the Mathematical Foundations of Programming Semantics*, pages 79–102, 1999.

[CDG+88] L. Cardelli, J. Donahue, L. Galssman, M. Jordan, B. Kalsow, and G. Nelson. Modula-3 report. Technical Report SRC-31, DEC systems Research Center, 1988.

[CWM98] Karl Crary, Stephanie Weirich, and Greg Morrisett. Intensional polymorphism in type-erasure semantics. In *1998 International Conference on Functional Programming*, 1998.

[DMTW97] Allyn Dimock, Robert Muller, Franklyn Turbak, and J. B. Wells. Strongly typed flow-directed representation transformations (extended abstract). In *1997 ACM SIGPLAN International Conference on Functional Programming*, 1997.

[DWE98] Sophia Drossopoulou, David Wragg, and Susan Eisenbach. What *is* Java binary compatibility? In *1998 ACM Symposium on Object-Oriented Programming: Systems, Languages and Applications*, 1998.

[ES90] Margaret A. Ellis and Bjarne Stroustrop. *The Annotated C++ Reference Manual.* Addison-Wesley, 1990.

[FM98] Stephen N. Freund and John C. Mitchell. A type system for object initialization in the Java bytecode language. In *1998 ACM Symposium on Object-Oriented Programming: Systems, Languages and Applications*, 1998.

[Gir72] Jean-Yves Girard. *Interprétation Fonctionnelle et Elimination des Coupures dans l'Arithmétique d'Ordre Supérieur.* PhD thesis, University of Paris VII, 1972.

[LY97] Tim Lindholm and Frank Yellin. *The Java Virtual Machine Specification.* Addison-Wesley, 1997.

[MCGW98] Greg Morrisett, Karl Crary, Neal Glew, and David Walker. Stack-Based Typed Assembly Language. In *1998 ACM Workshop on Types in Compilation*, 1998.

[Mey92] Bertrand Meyer. *Eiffel: the language.* Prentice-Hall, 1992.

[MMH95] Yasuhiko Minamide, Greg Morrisett, and Robert Harper. Typed closure conversion. Technical Report CMU-CS-95-171, Carnegie Mellon University, 1995.

[MTHM97] Robin Milner, Mads Tofte, Robert Harper, and David MacQueen. *The Definition of Standard ML (Revised).* The MIT Press, 1997.

[MWCG97] Greg Morrisett, David Walker, Karl Crary, and Neal Glew. From System F to Typed Assembly Language. Technical Report TR97-1651, Cornell University, 1997.

[OW97] Martin Odersky and Philip Wadler. Pizza into Java: Translating theory into practice. In *24th ACM Symposium on the Principles of Programming Languages*, pages 146–159, 1997.

[Pet96] Leaf E. Petersen. *A Module System for LOOM*. Williams College Senior Honors Thesis, 1996.

[Pey98] Simon L. Peyton Jones. A transformation-based optimiser for Haskell. *Science of Computer Programming*, 32(1-3):3–47, September 1998.

[Pie92] Benjamin C. Pierce. Bounded quantification is undecidable. In *Proceedings of the Nineteenth ACM Symposium on Principles of Programming Languages*, pages 305–315, 1992.

[PLST98] Simon Peyton Jones, John Launchbury, Mark Shields, and Andrew Tolmach. Bridging the gulf: a common intermediate language for ML and Haskell. In *1998 ACM Symposium on Principles of Programming Languages*, 1998.

[Rey74] John C. Reynolds. Towards a theory of type structure. In *Proceedings of the Colloque sur la Programmation*, pages 408–425. Springer-Verlag, 1974.

[SA98] Raymie Stata and Martín Abadi. A type system for Java bytecode subroutines. In *1998 ACM Symposium on Principles of Programming Languages*, 1998.

[Sch84] Angela M. Schuett. *Parametric Polymorphism in a Type-Safe, Object-Oriented Programming Langage*. Williams College Senior Honors Thesis, 1984.

[Sha97a] Zhong Shao. An overview of the FLINT/ML compiler. In *1997 ACM SIGPLAN Workshop on Types in Compilation*, 1997.

[Sha97b] Zhong Shao. Typed common intermediate format. In *1997 USENIX Conference on Domain-Specific Languages*, 1997.

[Sha98] Zhong Shao. Discussion of FLINT group objectives. Personal communication, 1998.

[SW96] Amr Sabry and Philip Wadler. A reflection on call-by-value. In *Proceedings of the 1996 ACM SIGPLAN International Conference on Functional Programming*, 1996.

[Tes85] L. Tesler. Object Pascal report. Technical Report 1, Apple Computer, 1985.

[TMC+96] D. Tarditi, G. Morrisett, P. Cheng, C. Stone, R. Harper, and P. Lee. TIL: A type-directed optimizing compiler for ML. In *1996 ACM SIGPLAN Conference on Programming Language Design and Implementation*, 1996.

[Van97] Joseph C. Vanderwaart. The new guide to LOOM: Draft. Technical report, Williams College, 1997.

[vG93] Robert van Gent. *TOIL: An Imperative Type-Safe Object-Oriented Language.* Williams College Senior Honors Thesis, 1993.

Appendix A

Complete SIL→IL Translation

In this appendix, we reproduce all the rules necessary to perform the translation from SIL into IL. Recall the following conventions:

- $\mathcal{A}$ refers to a set of kind assumptions of the form $(t :: \kappa)$
- $\mathcal{C}$ refers to a set of coercion assumptions of the form $(c : t <: \tau)$
- ST refers to a set of type equality assumptions of the form $t = \tau$
- $\mathbf{E}$ refers to a set of variable typing assumptions of the form $x : \tau$.

Types of Coercions

$$\text{coerce}_*(\tau_1 <: \tau_2) = |\tau_1| \to |\tau_2|$$

$$\text{coerce}_{\kappa_1 \to \kappa_2}(\tau_1 <: \tau_2) = \forall t :: \kappa_1.(\text{coerce}_{\kappa_2}(\tau_1[t] <: \tau_2[t]))$$

Translating Types: $|\cdot|$

$$|\mathtt{int}| = \mathtt{int}$$

$$|(\lambda t :: \kappa.\tau)| = (\lambda t :: \kappa.|\tau|)$$

$$|t| = t$$

$$|\tau_1[\tau_2]| = |\tau_1|[|\tau_2|]$$

$$|\texttt{Rec}\ t :: \kappa.\tau| = \texttt{Rec}\ t :: \kappa.|\tau| \qquad |\tau_1 \times \tau_2| = |\tau_1| \times |\tau_2|$$

$$|\tau_1 \to \tau_2| = |\tau_1| \to |\tau_2| \qquad |\tau_1 + \tau_2| = |\tau_1| + \tau_2|$$

$$|\{l_1 : \tau_1, \ldots, l_n : \tau_n\}| = |\tau_1| \times \cdots \times |\tau_n|, \text{ if } l_1 < \cdots < l_n$$

$$|\forall t :: \kappa <: \sigma.\tau| = \forall t :: \kappa.(\mathrm{coerce}_\kappa(t <: \sigma)) \to |\tau|$$

$$|\exists t :: \kappa <: \sigma.\tau| = \exists t :: \kappa.(\mathrm{coerce}_\kappa(t <: \sigma)) \times |\tau|$$

Rules for Constructing Coercions

$$\mathcal{A}, \mathcal{C} \vdash \mathrm{id}_\tau : \tau <: \tau \quad (c : \mathrm{id})$$

$$\frac{(c : t <: \tau) \in \mathcal{C}}{\mathcal{A}, \mathcal{C} \vdash c : t <: \tau}(c : \mathcal{C})$$

$$\frac{\mathcal{A}, \mathcal{C} \vdash c_1 : \tau_1 <: \sigma_1 \quad \mathcal{A}, \mathcal{C} \vdash c_2 : \sigma_2 <: \tau_2}{\mathcal{A}, \mathcal{C} \vdash (\lambda f : \sigma_1 \to \sigma_2.c_2 \circ f \circ c_1) : \sigma_1 \to \sigma_2 <: \tau_1 \to \tau_2}(c :\to)$$

$$\frac{\mathcal{A}, \mathcal{C} \vdash c_1 : \sigma_1 <: \tau_1 \quad \mathcal{A}, \mathcal{C} \vdash c_2 : \sigma_2 <: \tau_2}{\mathcal{A}, \mathcal{C} \vdash (\lambda x : \sigma_1 \times \sigma_2.(c_1(\#0\ x), c_2(\#1\ x))) : \sigma_1 \times \sigma_2 <: \tau_1 \times \tau_2}(c : \times)$$

$$\frac{\mathcal{A} \cup \{t :: \kappa\}, \mathcal{C} \vdash c : \tau_1 <: \tau_2 \quad t \notin \mathcal{C}}{\mathcal{A}, \mathcal{C} \vdash (\Lambda t :: \kappa.c) : (\lambda t :: \kappa.\tau_1) <: (\lambda t :: \kappa.\tau_2)}(c : \lambda)$$

$$\frac{\mathcal{A}, \mathcal{C} \vdash \tau_1 <: \tau_2 \quad \mathcal{A} \vdash \tau_1 :: \kappa_1 \to \kappa_2 \quad \mathcal{A} \vdash \sigma :: \kappa_1}{\mathcal{A}, \mathcal{C} \vdash c[\sigma] : \tau_1[\sigma] <: \tau_2[\sigma]}(c : [\,])$$

$$\frac{\mathcal{A} \cup \{t :: \kappa\}, \mathcal{C} \cup \{c : t <: \sigma\} \vdash c' : \tau_1 <: \tau_2}{\begin{array}{l}\mathcal{A}, \mathcal{C} \vdash (\lambda f : \forall t :: \kappa.\mathrm{coerce}_\kappa(t <: \sigma) \to \tau_1.\Lambda t :: \kappa.\lambda c : \mathrm{coerce}_\kappa(t <: \sigma).c'(f[t]c)) \\ \quad : \forall t :: \kappa <: \sigma.\tau_1 <: \forall t :: \kappa <: \sigma.\tau_2\end{array}}(c : \forall)$$

$$\frac{\mathcal{A} \cup \{t :: \kappa\}, \mathcal{C} \cup \{c : t <: \sigma\} \vdash c' : \tau_1 <: \tau_2}{\begin{array}{l}\mathcal{A}, \mathcal{C} \vdash (\lambda x : \exists t :: \kappa.\mathrm{coerce}_\kappa(t <: \sigma) \times \tau_1.\mathtt{let}\ [t; (c, x')] = \ \mathtt{open}\ x\ \mathtt{in} \\ \quad \mathtt{pack}\ [t; (c, c'(x'))]\ \mathtt{as}\ \exists t :: \kappa.\mathrm{coerce}_\kappa(t <: \sigma) \times \tau_2) \\ \quad : \exists t :: \kappa <: \sigma.\tau_1 <: \exists t :: \kappa <: \sigma.\tau_2\end{array}}(c : \exists)$$

Finding Minimum Accessible Types: $\overset{\text{acc}}{\leadsto}$

$$\frac{(c : t <: \tau) \in \mathcal{C}}{\mathcal{C} \vdash \mathrm{one}(t) = (c, \tau)}(\mathrm{one} : \mathcal{C})$$

$$\frac{\mathcal{C} \vdash \mathrm{one}(\tau) = (c, \tau')}{\mathcal{C} \vdash \mathrm{one}(\tau[\sigma]) = (c[\sigma], \tau[\sigma])}(\mathrm{one} : [\,])$$

$$\mathcal{C} \vdash \mathrm{one}(\mathtt{Rec}\ t :: \kappa.\tau) = (\mathrm{id}_{\mathtt{Rec}\ t::\kappa.\tau}, \tau[(\mathtt{Rec}\ t :: \kappa.\tau)/t]) \quad (\mathrm{one} : \mathtt{Rec})$$

$$\frac{\tau\ \text{accessible}}{\mathcal{A}, \mathcal{C} \vdash \mathrm{expand}(\tau) = (\mathrm{id}_\tau, \tau)}(\mathrm{expand} : \mathrm{Acc})$$

$$\frac{\begin{array}{c}\mathcal{A} \vdash \tau :: * \quad t\ \text{not accessible} \\ \mathcal{C} \vdash \mathrm{one}(\tau) = (c, \tau') \quad \mathcal{A}, \mathcal{C} \vdash \mathrm{expand}(\tau') = (c', \tau'')\end{array}}{\mathcal{A}, \mathcal{C} \vdash \mathrm{expand}(\tau) = (c' \circ c, \tau'')}(\mathrm{expand} : \mathrm{NAcc})$$

$$\frac{\mathcal{A}, \mathcal{C}, \mathsf{ST}, \mathbf{E} \vdash e : \tau \leadsto e' \quad \mathcal{A}, \mathcal{C} \vdash \mathrm{expand}(\tau) = (c, \tau')}{\mathcal{A}, \mathcal{C}, \mathsf{ST}, \mathbf{E} \vdash e : \tau' \overset{\text{acc}}{\leadsto} c(e')}(\overset{\text{acc}}{\leadsto})$$

Translating Terms: $\leadsto$

$$\frac{\mathbf{E} \vdash x : \tau}{\mathcal{A}, \mathcal{C}, \mathsf{ST}, \mathbf{E} \vdash x : \tau \leadsto x}(\leadsto, x)$$

$$\frac{\vdash v : \tau}{\mathcal{A}, \mathcal{C}, \mathsf{ST}, \mathbf{E} \vdash v : \tau \leadsto v}(\leadsto, v)$$

$$\frac{\mathcal{A}, \mathcal{C}, \mathsf{ST}, \mathbf{E} \cup \{f : \tau_1 \to \tau_2, x : \tau_1\}, \mathsf{ST} \vdash e : \tau_2' \leadsto e' \quad \mathcal{A}, \mathcal{C} \vdash c : \tau_2' <: \tau_2 \quad f, x \notin \mathcal{C}}{\mathcal{A}, \mathcal{C}, \mathsf{ST}, \mathbf{E} \vdash (\texttt{rec}\ f : \tau_1 \to \tau_2 = \lambda x.e) : \tau_1 \to \tau_2 \leadsto (\texttt{rec}\ f : |\tau_1| \to |\tau_2| = \lambda x.c(e'))}(\leadsto, \to I)$$

$$\frac{\mathcal{A}, \mathcal{C}, \mathsf{ST}, \mathbf{E} \vdash e_1 : \tau_1 \to \tau_2 \overset{\text{acc}}{\leadsto} e_1' \quad \mathcal{A}, \mathcal{C}, \mathsf{ST}, \mathbf{E} \vdash e_2 : \tau_1' \leadsto e_2' \quad \mathcal{A}, \mathcal{C} \vdash c : \tau_1' <: \tau_1}{\mathcal{A}, \mathcal{C}, \mathsf{ST}, \mathbf{E} \vdash e_1(e_2) : \tau_2 \leadsto e_1'(c(e_2'))}(\leadsto, \to E)$$

$$\frac{\mathcal{A}, \mathcal{C}, \mathsf{ST}, \mathbf{E} \vdash e_1 : \tau_1 \leadsto e_1' \quad \mathcal{A}, \mathcal{C}, \mathsf{ST}, \mathbf{E} \vdash e_2 : \tau_2 \leadsto e_2'}{\mathcal{A}, \mathcal{C}, \mathsf{ST}, \mathbf{E} \vdash \langle e_1, e_2 \rangle : \tau_1 \times \tau_2 \leadsto \langle e_1', e_2' \rangle}(\leadsto, \times I)$$

$$\frac{\mathcal{A}, \mathcal{C}, \mathsf{ST}, \mathbf{E} \vdash e : \tau_1 \times \tau_2 \overset{\text{acc}}{\leadsto} e'}{\mathcal{A}, \mathcal{C}, \mathsf{ST}, \mathbf{E} \vdash (\#i\ e) : \tau_i \leadsto (\#i\ e')}(\leadsto, \times E)$$

$$\frac{\mathcal{A}, \mathcal{C}, \mathsf{ST}, \mathbf{E} \vdash e : \tau' \leadsto e' \quad \mathcal{A}, \mathcal{C} \vdash c : \tau' <: \tau_i}{\mathcal{A}, \mathcal{C}, \mathsf{ST}, \mathbf{E} \vdash \mathrm{in}_i\ e \text{ as } \tau_1 + \tau_2 : \tau_1 + \tau_2 \leadsto \mathrm{in}_i\ c(e') \text{ as } |\tau_1| + |\tau_2|}(\leadsto, +I)$$

$$\frac{\mathcal{A}, \mathcal{C}, \mathsf{ST}, \mathbf{E} \vdash e_i : \tau_i \leadsto e_i' \quad l_0 < \cdots < l_n}{\mathcal{A}, \mathcal{C}, \mathsf{ST}, \mathbf{E} \vdash \{l_0 = e_0, \ldots, l_n = e_n\} : \{l_0 : \tau_0, \ldots, l_n : \tau_n\} \leadsto (e_0, \ldots, e_n)}(\leadsto, \{\}I)$$

$$\frac{\mathcal{A}, \mathcal{C}, \mathsf{ST}, \mathbf{E} \vdash e : \{l_0 : \tau_0, \ldots, l_n : \tau_n\} \overset{\text{acc}}{\leadsto} e' \quad l_0 < \cdots < l_n}{\mathcal{A}, \mathcal{C}, \mathsf{ST}, \mathbf{E} \vdash e.l_k : \tau_k \leadsto (\#k\ e')}(\leadsto, \{\}E)$$

$$\frac{\begin{array}{c} t \notin \mathsf{ST} \quad c \notin \mathbf{E} \quad f \notin \mathcal{C} \\ \mathcal{A} \cup \{t :: \kappa\}, \mathcal{C} \cup \{c : t <: \sigma\}, \mathsf{ST}, \mathbf{E} \cup \{f : \forall t :: \kappa <: \sigma.\tau\} \vdash e : \tau' \leadsto e' \quad \mathcal{A}, \mathcal{C} \vdash c' : \tau' <: \tau \end{array}}{\begin{array}{ll} \mathcal{A}, \mathcal{C}, \mathsf{ST}, \mathbf{E} \vdash & (\texttt{rec}\ f : \forall t :: \kappa <: \sigma.\tau = \Lambda.e) : \forall t <: \sigma.\tau \\ & \leadsto (\texttt{rec}\ f : \forall t :: \kappa.(\mathrm{coerce}_\kappa(t <: \sigma)) \to |\tau| = \Lambda.\lambda c : (\mathrm{coerce}_\kappa(t <: \sigma)).c'(e')) \end{array}}(\leadsto, \forall I)$$

$$\frac{\mathcal{A},\mathcal{C},\mathsf{ST},\mathbf{E} \vdash e : \forall t <: \sigma'.\tau \overset{\text{acc}}{\rightsquigarrow} e' \quad \mathcal{A},\mathcal{C} \vdash c : \sigma <: \sigma'}{\mathcal{A},\mathcal{C},\mathsf{ST},\mathbf{E} \vdash e[\sigma] : \tau[\sigma/t] \rightsquigarrow e'[|\sigma|](c)} (\rightsquigarrow, \forall E)$$

$$\frac{\begin{array}{c}\mathcal{A},\mathcal{C},\mathsf{ST},\mathbf{E} \vdash e : \tau' \rightsquigarrow e' \quad \mathcal{A},\mathcal{C} \vdash c_1 : \tau' <: \tau[\sigma_1/t] \\ \mathcal{A} \vdash \sigma_1 :: \kappa \quad \mathcal{A},\mathcal{C} \vdash c_2 : \sigma_1 <: \sigma_2\end{array}}{\begin{array}{ll}\mathcal{A},\mathcal{C},\mathsf{ST},\mathbf{E} \vdash & \texttt{pack}\ [\sigma_1; e]\ \texttt{as}\ \exists t :: \kappa <: \sigma_2.\tau : \exists t :: \kappa <: \sigma_2.\tau \\ & \rightsquigarrow \texttt{pack}\ [|\sigma_1|; (c_2, c_1(e'))]\ \texttt{as}\ \exists t :: \kappa.(\mathrm{coerce}_\kappa(t <: \sigma_2)) \times |\tau|\end{array}} (\rightsquigarrow, \exists I)$$

Appendix B

Using the $\mathcal{LOOM}$ Compiler

B.1 Directory Organization

All of the $\mathcal{LOOM}$-related tools reside in the same directory on the Williams CS Department system. Since it may be moved around, we'll refer to it simply as `LOOM/`. This directory contains the following subdirectories:

- `LOOM/src` contains the source code for all the tools. It has the following subdirectories:
  - `LOOM/src/LOOM` contains the source code of the $\mathcal{LOOM}$ parser and type-checker, including the definitions of syntax trees.
  - `LOOM/src/LOOMint` contains the source code of the remaining parts of the $\mathcal{LOOM}$ interpreter.
  - `LOOM/src/LOOMtrans` contains the code for the translation from $\mathcal{LOOM}$ to SIL and the top-level module of the $\mathcal{LOOM}$ compiler.
  - `LOOM/src/ILint` contains everything pertaining to the definition of IL, including the syntax trees, parser, type-checker, interpreter and pretty-printer.
  - `LOOM/src/SIL` contains code pertaining to the definition of SIL, including syntax trees, parser, pretty-printer and translator into IL.
- `LOOM/bin` contains the heap images of the tools, as produced by SML/NJ. It also contains one-line shell scripts for invoking each program using the SML runtime.
- `LOOM/iltest` contains some IL programs for testing the IL interpreter.
- `LOOM/siltest` contains SIL programs for testing the SIL translator.
- `LOOM/loomtest` contains $\mathcal{LOOM}$ programs for testing the $\mathcal{LOOM}$ compiler.

B.2 Compiling the Tools

The $\mathcal{LOOM}$ tools rely on the SML/NJ compilation manager; each subdirectory of `LOOM/src` contains a `sources.cm` file which can be used to compile the corresponding application. Also, the top-level structure of each application contains a function for creating an "executable" heap image. To automate this task, each directory also contains a file `make`*xxx*`.sml` which performs the necessary steps to create an up-to-date heap image for the application whose code resides there. So, to build the $\mathcal{LOOM}$-to-SIL compiler, use the commands:

```
-> cd LOOM/src/LOOMtrans
-> sml < makeloomc.sml
```

Similarly, the SIL-to-IL translator can be built by running SML on `makesilt.sml` from `LOOM/src/SIL`, and the IL interpreter can be built using `makelili.sml` in `LOOM/src/ILint`.

B.3 Running the Tools

If the $\mathcal{LOOM}$ tools have been compiled as specified above, then their heap images should reside in `LOOM/bin`, along with some shell scripts to load them with. Each one accepts a very simple set of command-line arguments, as we will now describe.

The $\mathcal{LOOM}$ compiler, `loomc`, is invoked as follows:

`->` `loomc` *program.loom* `-s` *output.sil*

where *program.loom* is the source of the $\mathcal{LOOM}$ program to be translated, and *output.sil* is the name of the file to which the SIL output should be written. To translate this SIL version of the program into IL, use the translator called `silt` as follows:

`->` `silt` *silprogram.sil* *output.il*

where *output.il* is the file that will receive the IL program. Finally, interpret an IL program using the interpreter `lili`:

`->` `lili` *ilprogram.il*

B.4 Obtaining $\mathcal{LOOM}$

The source code for the $\mathcal{LOOM}$ compiler and other tools is available via the World Wide Web at:

`http://www.cs.williams.edu/~99jcv/LOOM`

If this link ever disappears, send email to `kim@cs.williams.edu` for information on $\mathcal{LOOM}$.