Compiling Standard ML to Java

by

Simon Gammage

An essay
presented to the University of Waterloo
in fulfilment of the
essay requirement for the degree of
Master of Mathematics
in
Computer Science

Waterloo, Ontario, Canada, 1997

I hereby declare that I am the sole author of this essay.

I authorize the University of Waterloo to lend this essay to other institutions or individuals for the purpose of scholarly research.

I further authorize the University of Waterloo to reproduce this essay by photocopying or by other means, in total or in part, at the request of other institutions or individuals for the purpose of scholarly research.

The University of Waterloo requires the signatures of all persons using or photocopying this essay. Please sign below, and give address and date.

Abstract

The phenomenal recent success of the Java programming language stands in stark contrast to the reluctance with which other new languages have been greeted in industry. Java's popularity can be attributed to both its superficial similarity to C++, and, perhaps more importantly, the massive proliferation of the Java Virtual Machine (JVM), to which Java is most often compiled. The Java language is not, however, a significant improvement over other conventional languages like C and C++; Standard ML (SML) incorporates many advanced features not offered by Java (such as parametric polymorphism, an advanced module system, type inference, and algebraic datatypes), and therefore offers a potentially more attractive model for Internet programming. This document describes a compiler which dispenses with the Java language, and instead compiles SML to the Java Virtual Machine. Both the wide availability of the JVM, and the many powerful features of SML, are thereby exploited. Compilation from SML to the JVM runs into many of the traditional problems of compiling functional languages; to deal with these, several advanced implementation techniques are employed, including representation analysis, A-normalization, and closure conversion, in the framework a type-directed compilation strategy. The result is a compiler which demonstrates both the feasibility of compiling high-level typed languages to the JVM, and the utility of advanced compilation techniques.

Acknowledgements

I would like to thank my family, without whose constant moral (not to mention financial) support in the face of my frequently changing plans, the last seven years would have not been possible. I would also like to thank Professor Dominic Duggan for giving me the opportunity to complete my MMath at Waterloo. My thanks also to Professor Peter Buhr for acting as my supervisor after Dominic's departure and providing valuable feedback on the essay, and to Professors Bruce Simpson and Peter Forsyth for their excellent instruction and encouragement outside the classroom. Finally, I would like to thank my friends, whose friendship over the last seven years has made my stay in Waterloo worthwhile, and who have taught me more of value than the sum of my academic courses.

Contents

List of Figures

Chapter 1

Introduction

Programming language research has received considerable recent public attention with the emergence of the Java programming language [19]. In the past, new programming languages have been only reluctantly (if at all) adopted by the computing community at large. Despite fifty years of programming language research, and the design and development of many hundreds of languages, only about ten are in widespread use today — Sammet lists Ada, APT, C, C++, COBOL, Common Lisp, FORTRAN, Pascal, Prolog and Smalltalk [37]. Conspicuously absent from this list are the many advanced languages, like Standard ML, Modula-3 and Haskell, which have periodically appeared, and almost always failed to achieve success outside the academic community. By contrast, Java has been embraced by programmers in academia and industry alike in a very short space of time.

Java has two key features that make it attractive:

1. It is syntactically and semantically similar to C++ [41], meaning that experienced C++ programmers can learn Java quickly. Furthermore, Java addresses some of the more serious shortcomings and insecurities of C++, making it more appealing for use in environments requiring robust and secure programs (such as the World Wide Web).

2. Java is closely associated with a virtual machine model (the Java Virtual Machine (JVM) [28]), to which it is most often compiled. JVM interpreters have been very widely disseminated, most often as components of web browsers. This feature is crucial — it means that programmers can write applications in Java which can be run on millions of machines around the world without regard for architecture, memory model, operating system, or any other of the other miscellaneous barriers to portability. The JVM is as close to a 'universal' machine as has been seen in recent years.

Despite its sudden popularity, the Java language, while a tidy subset of C++, is only a small step forward in language design, and deliberately so. Languages like Standard ML (SML) [30, 36, 49] incorporate advanced features absent from Java, and thus offer a more attractive programming model, perhaps better suited to the task of Internet programming.

If the real innovation of Java is the massive proliferation of the JVM, one is tempted to dispense with the Java language altogether, and instead use the JVM as a universal machine code.

This document describes the design and implementation of a compiler which adopts precisely this strategy, compiling a subset of the SML to the JVM. While the idea of translating non-Java languages to the JVM is by no means novel — see, for example, Intermetrics' AppletMagic Ada95-to-JVM compiler [42], or the Kawa Scheme-to-Java compiler [12] — some of SML's features make the translation particularly challenging. To deal with these issues, a number of advanced compilation techniques are employed, including A-normalization, representation analysis, and closure conversion. The compiler described is therefore both a demonstration of the feasibility of compiling advanced languages to the JVM, and an exploration of implementation techniques for those languages.

Chapter 2

Overview

2.1 Introduction

This chapter provides brief summaries of the languages involved in the translation (Standard ML and Java), and presents an overview of the compiler's architecture.

2.2 Standard ML

The Standard ML programming language (SML) can be succinctly characterized as a strongly, implicitly and polymorphically typed, impure, and strict functional language. SML is first and foremost a functional language, and encourages a functional style of programming. It is not purely functional, however, as it includes support for traditional imperative features (although they are not always convenient to use, and cause some trouble for the type system [46, 53]). SML incorporates many of the advanced features characteristic of languages developed in the last twenty years, including first-class functions, exception handling, algebraic data types, abstract types, and an advanced module system, including facilities for parameterized modules (functors); it is not, however, nor does it claim to be, 'object-oriented'.

One interesting feature of SML is that it possesses a formal definition of its static and dynamic semantics, in the form of the Definition of Standard ML [30] and accompanying Commentary [29]. The Definition has recently been amended to simplify some of the more awkward features of the original language [31] — the revised language is referred to as SML97. These documents provide the compiler implementor with a precise specification for both compile-time analysis (static semantics), and run-time behaviour (dynamic semantics) of SML programs.

There are several guides to programming in SML at a range of skill levels, to which the reader is referred for a comprehensive language overview [49, 36]. Broad familiarity with SML is assumed in this document.

2.3 Java

The Java programming language is to C++ what Scheme is to Lisp: a cleaned-up variant, retaining the flavour of the original language, while adding a few features.

The most obvious difference between C++ and Java is that Java does not possess pointer types; in Java, all instances of classes are heap-allocated, whereas all instances of primitive types (`int`, `double`, and so on) are stack-allocated. Java also incorporates automatic storage management (garbage collection), a feature notably absent from C++.

Java's object model is also slightly different from that of C++:

- In addition to traditional classes, Java introduces the notion of an *interface*, which is essentially a class signature.
- Whereas C++ permits multiple inheritance, Java allows to single inheritance only (although classes may implement multiple interfaces, and an interface may be an extension of multiple interfaces).
- In Java, all functions must be methods of classes, and the only user-definable types are classes and interfaces; Java therefore restricts the programmer to a more purely object-oriented mode of operation than C++.

Although it does not possess a formal semantics in the manner of SML's Definition, Java's static and dynamic semantics are reasonably well-defined by the Java Language Specification [19]. Again, there are a wide variety of Java programming guides available, to which the reader is referred for a language overview (see [9], for example); familiarity with Java is assumed in this document.

2.4 Java and the JVM

Compilers have traditionally translated the source language into assembly code for the target platform (*native-code* compilers). This approach is attractive from the standpoint of performance, but can result in considerable development effort if the source language is relatively high-level, and means that the compiler must be customized for each target architecture.

Another approach is to translate the source language into another high-level language for which a compiler already exists. This approach sacrifices some performance for reduced development time and enhanced portability, especially if a widely used language (like C or C++) is used as the target. Several major projects have adopted this approach, including compilers for FORTRAN [15], Pascal [17], Standard ML [45], Scheme [10], and Haskell [23], all of which use C as the target language.

Compilation to the JVM does not fall neatly into either category, however, since the JVM, although nominally a machine code, is relatively high-level. In particular, it includes some constructs not normally found in assembly code (such as representation of classes, interfaces and exceptions), and lacks others (such as non-local gotos). The JVM also includes different operations for different operand types, giving it somewhat the flavour of a typed language, as distinct from traditional assembly code, which is more or less untyped. Compilation to the JVM therefore takes much more the character of a source-to-source translation (where in particular type information must be preserved) than a traditional source-to-assembly translation.

Further inspection of the JVM reveals that it is very strongly tied to the Java language. Many Java language constructs (including classes, interfaces and exceptions) map directly to JVM constructs, making the task of compiling to the JVM almost identical to that of compiling to the Java language. In the interests of simplicity and ease of development, then, it was decided to compile SML to Java and then use an existing Java compiler to compile to the JVM. It is gratifying to note that other projects have independently adopted this strategy, for much the same reasons [35].

2.5 Compiler Architecture

In common with most compilers, the SML-to-Java compiler is structured as a pipeline of stages, each stage passing its results to the next for further transformation. The SML-to-Java compiler comprises six stages, as illustrated in Figure 2.1.

The first stage of compilation is the traditional 'front-end' of the compiler, performing lexical analysis, parsing, type checking (elaboration), and translation to an intermediate language `Lambda` (a typed λ-calculus). This stage of translation is described in Chapter 3.

The second stage translates from `Lambda` to another (somewhat different) λ-calculus, λ_i^{ML}, which is used as the basis for further intermediate form transformations. This translation is also described in Chapter 3.

The next three stages perform transformations on the λ_i^{ML} intermediate form. The first such transformation is *representation analysis* (discussed in Chapter 4), which deals with issues arising from SML's polymorphic type system. The next transformation is *A-normalization* (discussed in Chapter 5), where a distinction between expressions and statements is introduced. The final transformation is *closure conversion* (discussed in Chapter 6), where SML's first-class functions and nested scope are dealt with.

The final stage of the compiler translates the (suitably transformed) λ_i^{ML} intermediate form to Java code. This process is described in Chapter 7. The Java code generated by the compiler can then be compiled and run by an existing Java compiler and interpreter (such as Sun's Java Development Kit (JDK), or Microsoft's Visual J++).

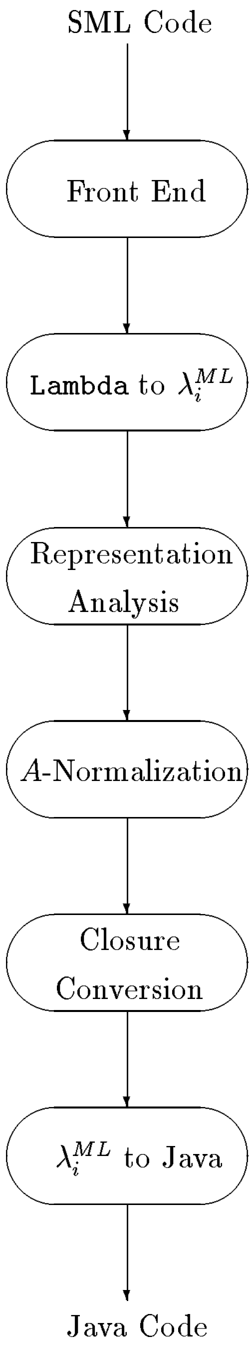

Figure 2.1: Architecture of the compiler

Chapter 3

The Front End

3.1 Introduction

Compilers are traditionally separated into front- and back-ends: the front-end performs lexical and syntactic analysis, and semantic analysis such as type-checking, whereas the back-end is responsible for code generation. This chapter discusses the front-end of the SML-to-Java compiler.

3.2 The ML Kit

Lexical and syntactic analysis of SML has been an area of significant complexity for SML compilers [5, 8, 11, 14], due in no small part to the ambiguous grammar presented in the Definition. The static semantics, by contrast, are well understood and easily translated into practice; indeed, the ML Kit system [11, 47] includes an almost literal implementation of the static semantics of the Definition in SML, as well as a full lexer and parser[1]. Rather than revisit problems solved by the Kit's implementors, it seems sensible to make use of the Kit's front-end, and simply implement a new back-end for the translation to Java. This approach has been adopted by a number of other projects [43, 14].

The SML-to-Java compiler thus makes use of the front-end provided by the ML Kit version 2 to perform lexical analysis, syntactic analysis, and elaboration. As a further benefit, the Kit version 2 includes a compiler to a typed λ-calculus `Lambda`, which performs pattern-matching compilation. With the issues of type-checking and pattern-matching resolved, the development of the SML-to-Java compiler can focus on the specific issues related to translation to Java.

[1]Version 1 of the Kit also includes a full interpreter for SML, based on the dynamic semantics of the Definition, and a small compiler. Version 2 of the Kit (with region inference) includes a full region-based compiler.

3.2.1 Lambda

The ML Kit's Lambda language is essentially a simplified form of the abstract syntax for SML presented in the Definition, with pattern-matching reduced to simpler constructs. The abstract syntax for Lambda is shown in Figures 3.1 and 3.2. Here i represents an integer, r a real number, s a string, t a type name, c a datatype constructor name, ex an exception constructor name, x an expression variable, and α a type variable. Each kind of variable exists in its own name space.

A Lambda *monotype* τ can be a type variable α, a function type $(\tau_1, \ldots, \tau_n) \rightarrow \tau$, an instance of a datatype $(\tau_1, \ldots, \tau_n)\ t$, or a record type $\langle \tau_1 \times \cdots \times \tau_n \rangle$. A *polytype* (or *type scheme*) σ is a universally quantified monotype. A type *instance* ρ is a lists of types, used to annotate term variable occurrences, and in construction of datatypes. A *datatype* corresponds to an SML datatype declaration, and represents the sum type of constructors c_i, each possibly carrying a type τ_i, abstracted over by a set of type variables $\alpha_1, \ldots, \alpha_n$. The *arity* of a datatype is defined as the number of abstracted type variables. Types int, real, string, exn, instream and outstream are treated as arity-0 datatypes with no constructors (essentially abstract types), bool is treated as an arity-0 datatype with constructors true and false, and list is treated as an arity-1 datatype with constructors nil and :: (cons). A *dgroup* defines a group of mutually recursive datatype declarations. Finally, a *datbind* represents the datatype declarations for a Lambda program.

Lambda *primitives* include injection into and selection from datatypes (con and decon) at a particular type instance ρ, injection into and selection from exception packets (excon and deexcon), construction of and projection from records (record and π_i), dereference, construction and assignment operations on reference types (annotated by the referred-to type), equality and inequality testing operations, arithmetic and comparison operations, some of which are overloaded for integer and real types (shown by subscripts int and real respectively), operations on strings (including concatenation and length computation), and I/O operations, including the print operation overloaded for integers, strings, booleans, and reals.

A Lambda *expression* (or *term*) can be:

- a variable annotated with its instantiating type,
- an integer, string or real constant.
- a λ-abstraction (i.e. anonymous function),
- a let-binding,
- a recursive function definition (fix),
- a function application,
- a primitive application (note that primitives are always fully applied),

Monotypes	τ	$::=$	α
		$\mid$	$(\tau_1, \ldots, \tau_n) \to \tau$
		$\mid$	$(\tau_1, \ldots, \tau_n)\ t$
		$\mid$	$\langle \tau_1 \times \cdots \times \tau_n \rangle$
Polytypes	σ	$::=$	$\forall(\alpha_1, \ldots, \alpha_n).\tau$
Instances	ρ	$::=$	$(\tau_1,\ \ldots,\ \tau_n)$
Datatypes	d	$::=$	$t = \Lambda(\alpha_1, \ldots, \alpha_m).c_1[:\tau_1] + \cdots + c_n[:\tau_n]$
Dgroups	dg	$::=$	$(d_1, \ldots, d_n)$
Datbinds	dbs	$::=$	$(dg_1, \ldots, dg_n)$
Primitives	p	$::=$	$\mathsf{con}_\rho\ c \mid \mathsf{decon}_\rho\ c \mid \mathsf{excon}\ ex \mid \mathsf{deexcon}\ ex \mid \mathsf{record} \mid \pi_i \mid$
			$!_\tau \mid \mathsf{ref}_\tau \mid\ :=_\tau\ \mid\ =_\tau\ \mid\ \neq_\tau\ \mid$
			$\mathsf{not} \mid \sim_{\mathsf{int}} \mid \sim_{\mathsf{real}} \mid \mathsf{abs}_{\mathsf{int}} \mid \mathsf{abs}_{\mathsf{real}} \mid \mathsf{floor} \mid \mathsf{real} \mid$
			$\mathsf{exp} \mid \mathsf{ln} \mid \mathsf{sqrt} \mid \mathsf{sin} \mid \mathsf{cos} \mid \mathsf{arctan} \mid$
			$\div_{\mathsf{int}} \mid \div_{\mathsf{real}} \mid \mathsf{mod} \mid \times_{\mathsf{int}} \mid \times_{\mathsf{real}} \mid +_{\mathsf{int}} \mid +_{\mathsf{real}} \mid -_{\mathsf{int}} \mid -_{\mathsf{real}} \mid$
			$<_{\mathsf{int}} \mid <_{\mathsf{real}} \mid >_{\mathsf{int}} \mid >_{\mathsf{real}} \mid \leq_{\mathsf{int}} \mid \leq_{\mathsf{real}} \mid \geq_{\mathsf{int}} \mid \geq_{\mathsf{real}} \mid$
			$\wedge \mid \mathsf{size} \mid \mathsf{chr} \mid \mathsf{ord} \mid \mathsf{explode} \mid \mathsf{implode} \mid$
			$\mathsf{open\_out} \mid \mathsf{close\_out} \mid \mathsf{std\_out} \mid \mathsf{output} \mid$
			$\mathsf{open\_in} \mid \mathsf{close\_in} \mid \mathsf{std\_in} \mid \mathsf{input} \mid$
			$\mathsf{flush\_out} \mid \mathsf{lookahead} \mid \mathsf{end\_of\_stream} \mid$
			$\mathsf{print}_{\mathsf{int}} \mid \mathsf{print}_{\mathsf{real}} \mid \mathsf{print}_{\mathsf{bool}} \mid \mathsf{print}_{\mathsf{string}}$

Figure 3.1: Lambda: Types and primitives

$$
\begin{array}{llcl}
\textit{Expressions} & e & ::= & x_\rho \\
 & & | & i \\
 & & | & s \\
 & & | & r \\
 & & | & \lambda(x_1 : \tau_1, \ldots, x_n : \tau_n).e \\
 & & | & \mathsf{let}\ x : \sigma = e_1\ \mathsf{in}\ e_2 \\
 & & | & \mathsf{fix}\ f_1 : \sigma_1 = e_1,\ \ldots,\ f_n : \sigma_n = e_n\ \mathsf{in}\ e \\
 & & | & e\ (e_1, \ldots, e_n) \\
 & & | & p\ (e_1, \ldots, e_n) \\
 & & | & \mathsf{exception}\ ex[: \tau]\ \mathsf{in}\ e \\
 & & | & \mathsf{raise}_\tau\ e \\
 & & | & e_1\ \mathsf{handle}\ e_2 \\
 & & | & \mathsf{switch}_i\ e\ \mathsf{of}\ i_1 : e_1,\ \ldots,\ i_n : e_n,\ [\mathsf{default} : e_d] \\
 & & | & \mathsf{switch}_s\ e\ \mathsf{of}\ s_1 : e_1,\ \ldots,\ s_n : e_n,\ [\mathsf{default} : e_d] \\
 & & | & \mathsf{switch}_r\ e\ \mathsf{of}\ r_1 : e_1,\ \ldots,\ r_n : e_n,\ [\mathsf{default} : e_d] \\
 & & | & \mathsf{switch}_c\ e\ \mathsf{of}\ c_1 : e_1,\ \ldots,\ c_n : e_n,\ [\mathsf{default} : e_d] \\
 & & | & \mathsf{switch}_e\ e\ \mathsf{of}\ ex_1 : e_1,\ \ldots,\ ex_n : e_n,\ [\mathsf{default} : e_d] \\
 & & | & \mathsf{frame} \\
 & & & \quad \mathsf{values}\ =\ x_1 : \sigma_1,\ \ldots,\ x_n : \sigma_n \\
 & & & \quad \mathsf{excons}\ =\ ex_1[: \tau_1],\ \ldots,\ ex_m[: \tau_m] \\
 & & & \\
\textit{Programs} & pgm & ::= & \mathsf{datbinds} : dbs \\
 & & & \mathsf{body} : e
\end{array}
$$

Figure 3.2: Lambda: Expressions and programs

- an exception declaration (as induced by an `exception` declaration in SML),
- an exception raised at a type,
- an expression with associated exception handler,
- a switch over integers, strings, reals, data constructors, or exception constructors, each with optional default clause, or
- a frame, corresponding to an SML structure definition.

A `Lambda` *program* consists of a set of datatype binding groups, each of which defines a set of mutually recursive datatypes, followed by an expression representing the body of the program.

The static semantics and dynamic semantics for `Lambda` are standard.

As noted above, translation from SML source to `Lambda` is performed by the Kit system. For example, the SML program:

```
datatype 'a A = A1 of 'a | A2 of 'a * 'a

fun fact 0 = 1
  | fact x = x * fact (x - 1)

fun foo x =
  case x
   of A1 y => y
    | A2 (y1, y2) => y1

fun bar x =
  let
    exception baz of int
  in
    if x < 0 then
      raise (baz x)
    else
      x
  end

val a = fact 5
val b = foo (A2 (1.2, 3.4))
```

is translated by the Kit into the following `Lambda` program:

$$
\begin{array}{l}
\mathsf{datbinds:} \\
(A = \Lambda\ (\alpha)\ .\ \mathit{A1} : \alpha + \mathit{A2} : \langle \alpha \times \alpha \rangle) \\
\mathsf{body:} \\
\mathsf{fix}\ \mathit{fact} : \mathsf{int} \to \mathsf{int} = \\
\quad \lambda\ (\mathit{v1} : \mathsf{int})\ . \\
\quad\quad \mathsf{switch}_i\ (\mathit{v1})\ \mathsf{of} \\
\quad\quad\quad 0 : 1 \\
\quad\quad\quad \mathsf{default:}\ \times_{\mathsf{int}}(\mathit{v1},\ \mathit{fact}\ (-_{\mathsf{int}}(\mathit{v1},\ 1))) \\
\mathsf{in\ fix}\ \mathit{foo} : \forall\ (\alpha)\ .\ \alpha\ A \to \alpha = \\
\quad \lambda\ (x : \alpha\ A)\ . \\
\quad\quad \mathsf{switch}_c\ (x)\ \ \mathsf{of} \\
\quad\quad\quad \mathit{A1}\!:\ \mathsf{let}\ y : \alpha = (\mathsf{decon}_\alpha\ \mathit{A1})\ (x)\ \mathsf{in}\ y \\
\quad\quad\quad \mathit{A2}\!:\ \mathsf{let}\ \mathit{v2} : \langle \alpha \times \alpha \rangle = (\mathsf{decon}_\alpha\ \mathit{A2})\ (x) \\
\quad\quad\quad\quad \mathsf{in\ let}\ \mathit{y1} : \alpha = \pi_1\ (\mathit{v2}) \\
\quad\quad\quad\quad \mathsf{in\ let}\ \mathit{y2} : \alpha = \pi_2\ (\mathit{v2}) \\
\quad\quad\quad\quad \mathsf{in}\ \mathit{y1} \\
\mathsf{in\ fix}\ \mathit{bar} : \mathsf{int}\!\to \mathsf{int} = \\
\quad \lambda\ (\mathit{v3} : \mathsf{int})\ . \\
\quad\quad \mathsf{exception}\ \mathit{baz} : \mathsf{int} \\
\quad\quad \mathsf{in\ switch}_c\ (<_{\mathsf{int}}(\mathit{v3},\ 0))\ \ \mathsf{of} \\
\quad\quad\quad \mathsf{true:}\ \mathsf{raise}_{\mathsf{int}}\ \ ((\mathsf{excon}\ \mathit{baz})\ (\mathit{v3})) \\
\quad\quad\quad \mathsf{false:}\ \mathit{v3} \\
\mathsf{in\ let}\ a : \mathsf{int} = \mathit{fact}\ (5) \\
\mathsf{in\ let}\ b : \mathsf{real} = \mathit{foo}_{\mathsf{real}}\ ((\mathsf{con}_{\mathsf{real}}\ \mathit{A2})\ (\mathsf{record}(1.2,\ 3.4))) \\
\mathsf{in\ frame\ values} = \mathit{fact} : \mathsf{int}\!\to \mathsf{int},\ \mathit{foo} : \forall(\alpha).\alpha\ A \to \alpha,\ a : \mathsf{int},\ b : \mathsf{real}
\end{array}
$$

3.3 Typed Intermediate Languages

So far the problem of compilation from SML to Java has been reduced to translating the typed λ-calculus `Lambda` to Java. Here the *type-directed compilation* approach of Morrisett et al. [33, 44, 43] is employed, using a series of *typed intermediate languages* as the basis for compilation.

Type-directed compilation is a fairly recent innovation in programming language research, but has already been incorporated in a number of advanced compiler development projects [40, 39, 43, 48]. Although the idea is simple — retain type information for as long as possible through compilation, rather than discarding it early on, as in many conventional compilers — the potential benefits of this technique are seen most clearly when applied to languages with advanced type systems, such as SML. Moreover, in translating *between* strongly typed languages (as between SML and Java), as opposed to conventional compilation from a strongly typed language to untyped assembly, it seems natural that types in the source language should be preserved in the translation.

The first stage of a type-directed compiler is conventionally to translate the source language into a second-order typed λ-calculus, such as Harper and Mitchell's XML [20], Harper and Morisett's λ_i^{ML} [43, 33, 44], Shao's FLINT [39], or Harper and Stone's Internal Language [21], all of which are essentially predicative variants of Girard's system **F** [18]. Examples of such translations are given by Harper and Mitchell (transforming Core-ML to Core-XML) [20], and Harper and Stone (transforming (most of) SML97 (including the module system) to their Internal Language) [21].

3.3.1 λ_i^{ML}

Rather than invent a new calculus, a group of variants of Harper and Morisett's λ_i^{ML} are used as the intermediate forms of the type-directed translation. The first variant used is λ_i^{ML}-Rep, to which the ML Kit's `Lambda` language is translated. Later transformations add wrapping and unwrapping operations, introduce the distinction between statements and exceptions, and add constructs for closure representation.

λ_i^{ML}-Rep comprises *kinds*, *constructors*, *types* and *expressions*, as shown in Figures 3.3 to 3.6. λ-abstraction occurs at both the expression and constructor levels; the language of constructors is hence a first-order calculus, whose 'types' are the *kinds* of λ_i^{ML}-Rep.

λ_i^{ML}-Rep *kinds* κ include:

- the 'ground' or 'base' kind Ω, representing basic types (such as integers, reals, functions, records, sums, exceptions, and so on),
- the kind of constructor functions $(\kappa_1, \ldots, \kappa_n) \rightarrow \kappa$, and
- the kind of constructor tuples $\langle \kappa_1 \times \cdots \times \kappa_n \rangle$.

λ_i^{ML}-Rep *constructors* μ include:

- constructor variables (α) and module projections ($m.l$),
- integer, real, string, enumeration and exception constructors (Int, Real, String, $\mathsf{Enum}\ i$, and Exn),
- the array constructor Array (note that reference types can be simulated using arrays, and hence there is no distinct reference constructor),
- function, disjoint sum and tuple (record) constructors ($(\mu_1, \ldots, \mu_n) \rightarrow \mu$, $(\mu_{11}, \ldots, \mu_{1n_1}) + \cdots + (\mu_{m1}, \ldots, \mu_{mn_m})$, and $\langle \mu_1 \times \cdots \times \mu_n \rangle$, respectively),
- Excon and $\mathsf{Deexcon}$ constructors, values of which type are used to introduce and eliminate exception packets, as discussed below,

- constructor tuples $\langle \mu_1, \ldots, \mu_n \rangle$ (note carefully the distinction between a constructor tuples, and tuple constructors) and projection π_i,
- constructor abstraction $\lambda(\alpha_1 :: \kappa_1, \ldots, \alpha_n :: \kappa_n).\mu$, and application $\mu\ (\mu_1, \ldots, \mu_n)$,
- recursive constructor definition $\mathsf{Rec}\ \alpha_1 = \mu_1, \ldots, \alpha_n = \mu_n\ \mathsf{In}\ \mu$ (note that in λ_i^{ML}-Rep, as in Morrisett and Harper's calculus, the isomorphism between a recursive type and its unrolling is not implicit; a value must be 'rolled' or 'unrolled' at the term level, as discussed below), and
- constructor let-binding.

Types σ include constructors μ (constrained to be of kind Ω), polymorphic types $\forall(\alpha_1 :: \kappa_1, \ldots, \alpha_n :: \kappa_n).\sigma$, and Export types, representing module signatures.

λ_i^{ML}-Rep *expressions* include:

- variables (x) and module projections ($m.l$),
- tuples ($\langle e_1, \ldots, e_n \rangle$),
- injections into the i^{th} variant of sum type μ ($\mathsf{inject}_i^{\mu}\ (e_1, \ldots, e_n)$),
- integer, real, enumeration, and string constants (i, r, $\mathsf{enum}_{\mu}\ i$, and s respectively),
- let declarations, binding:
  - simple values ($x : \sigma = e$),
  - simple constructors ($\alpha :: \kappa = \mu$),
  - groups of mutually recursive term functions (corresponding to λ-abstractions), or
  - groups of mutually recursive type functions (corresponding to Λ-abstractions),
- λ-abstractions (representing anonymous functions over terms) and applications,
- Λ-abstractions (representing anonymous functions over types) and applications,
- coercion operations, including:
  - unroll, coercing a recursive type to its unrolled form — the unrolled form of

    $$\mathsf{Rec}\ \alpha_1 = \mu_1, \ldots, \alpha_n = \mu_n\ \mathsf{In}\ \mu$$

    is defined as

    $$\{\mu_1'/\alpha_1, \ldots, \mu_n'/\alpha_n\}\mu,$$

$$
\begin{array}{llll}
\textit{Kinds} & \kappa & ::= & \Omega \\
& & | & (\kappa_1, \ldots, \kappa_n) \to \kappa \\
& & | & \langle \kappa_1 \times \cdots \times \kappa_n \rangle \\
\\
\textit{Constructors} & \mu & ::= & \alpha \\
& & | & m.l \\
& & | & \mathsf{Int} \\
& & | & \mathsf{Real} \\
& & | & \mathsf{String} \\
& & | & \mathsf{Exn} \\
& & | & \mathsf{Enum}\ i \\
& & | & \mathsf{Array}\ \mu \\
& & | & (\mu_1, \ldots, \mu_n) \to \mu \\
& & | & (\mu_{11}, \ldots, \mu_{1n_1}) + \cdots + (\mu_{m1}, \ldots, \mu_{mn_m}) \\
& & | & \langle \mu_1 \times \cdots \times \mu_n \rangle \\
& & | & \mathsf{Excon}\ \mu \\
& & | & \mathsf{Deexcon}\ \mu \\
& & | & \langle \mu_1, \ldots, \mu_n \rangle \\
& & | & \pi_i\ \mu \\
& & | & \lambda(\alpha_1 :: \kappa_1, \ldots, \alpha_n :: \kappa_n).\mu \\
& & | & \mu\ (\mu_1, \ldots, \mu_n) \\
& & | & \mathsf{Rec}\ \alpha_1 = \mu_1, \ldots, \alpha_n = \mu_n\ \mathsf{In}\ \mu \\
& & | & \mathsf{Let}\ \alpha :: \kappa = \mu_1\ \mathsf{In}\ \mu_2 \\
\\
\textit{Types} & \sigma & ::= & \mu \\
& & | & \forall(\alpha_1 :: \kappa_1, \ldots, \alpha_n :: \kappa_n).\sigma \\
& & | & \mathsf{Export} \\
& & & \mathsf{Types}\ :\ l_{11} :: \kappa_1, \ldots, l_{1n} :: \kappa_n \\
& & & \mathsf{Values}\ :\ l_{21} : \sigma_1, \ldots, l_{2m} : \sigma_m
\end{array}
$$

Figure 3.3: λ_i^{ML}-Rep: Kinds, constructors and types

$$
\begin{array}{llll}
\textit{Unary ops} & op_1 & ::= & \mathsf{real} \mid \mathsf{not} \mid \mathsf{floor} \mid \mathsf{sqrt} \mid \mathsf{sin} \mid \mathsf{cos} \mid \mathsf{arctan} \mid \\
 & & & \mathsf{exp} \mid \mathsf{ln} \mid \mathsf{size} \mid \pi_i \mid - \mid \mathsf{abs} \mid \mathsf{strlen} \\
 & & & \\
\textit{Binary ops} & op_2 & ::= & \div \mid \times \mid + \mid - \mid \mathsf{mod} \mid \\
 & & & = \mid < \mid > \mid \leq \mid \geq \mid \mathsf{or} \mid \mathsf{and} \mid \mathsf{xor} \mid << \mid >> \mid \\
 & & & \mathsf{alloc} \mid \mathsf{sub} \mid \mathsf{excon} \mid \mathsf{deexcon} \mid \mathsf{strcat} \\
 & & & \\
\textit{Coercion ops} & c & ::= & \mathsf{roll}_\sigma \\
 & & \mid & \mathsf{unroll} \\
 & & \mid & \mathsf{enum\_to\_int} \\
 & & \mid & \mathsf{int\_to\_enum}_\sigma
\end{array}
$$

Figure 3.4: λ_i^{ML}-Rep: Unary operators, binary operators and coercions

$$
\begin{array}{llll}
\textit{Declarations} & d & ::= & x : \sigma = e \\
 & & \mid & \alpha :: \kappa = \mu \\
 & & \mid & \mathsf{fix} \\
 & & & f_1 : \sigma_1 = \lambda(x_{11} : \mu_{11}, \ldots, x_{1n_1} : \mu_{1n_1}).e_1 \\
 & & & \vdots \\
 & & & f_m : \sigma_m = \lambda(x_{m1} : \mu_{m1}, \ldots, x_{mn_m} : \mu_{mn_m}).e_m \\
 & & \mid & \mathsf{fixtype} \\
 & & & f_1 : \sigma_1 = \Lambda(\alpha_{11} :: \kappa_{11}, \ldots, \alpha_{1n_1} :: \kappa_{1n_1}).e_1 \\
 & & & \vdots \\
 & & & f_m : \sigma_m = \Lambda(\alpha_{m1} :: \kappa_{m1}, \ldots, \alpha_{mn_m} :: \kappa_{mn_m}).e_m \\
 & & & \\
\textit{Misc ops} & m & ::= & e_1[e_2] \ := \ e_3 \\
 & & \mid & \mathsf{extern}\ s : \sigma \\
 & & \mid & \mathsf{newexn}\ \mu \\
 & & \mid & = \ (\mu, e_1, e_2) \\
 & & \mid & \neq \ (\mu, e_1, e_2)
\end{array}
$$

Figure 3.5: λ_i^{ML}-Rep: Declarations and misc ops

$$
\begin{array}{llll}
\textit{Expressions} & e & ::= & x \\
 & & | & m.l \\
 & & | & \langle e_1, \ldots, e_n \rangle \\
 & & | & \mathsf{inject}_i^{\mu}\ (e_1, \ldots, e_n) \\
 & & | & i \\
 & & | & r \\
 & & | & \mathsf{enum}_{\mu}\ i \\
 & & | & s \\
 & & | & \mathsf{let}\ d\ \mathsf{in}\ e \\
 & & | & \lambda(x_1 : \mu_1, \ldots, x_n : \mu_n).e \\
 & & | & \Lambda(\alpha_1 :: \kappa_1, \alpha_n :: \kappa_n).e \\
 & & | & e\ (e_1, \ldots, e_n) \\
 & & | & e\ (\mu_1, \ldots, \mu_n) \\
 & & | & c\ e \\
 & & | & op_1\ e \\
 & & | & op_2\ (e_1, e_2) \\
 & & | & m \\
 & & | & \mathsf{switch}\ e\ \mathsf{of} \\
 & & & i_1 : \lambda(x_{11} : \mu_{11}, \ldots, x_{1n_1} : \mu_{1n_1}).e_1, \\
 & & & \vdots, \\
 & & & i_m : \lambda(x_{m1} : \mu_{m1}, \ldots, x_{mn_m} : \mu_{mn_m}).e_m, \\
 & & & [\mathsf{default} : e] \\
 & & | & \mathsf{raise}_{\sigma}\ e \\
 & & | & e_1\ \mathsf{handle}(x : \mathsf{Exn})\ e_2 \\
 & & | & \mathsf{export} \\
 & & & \mathsf{types}:\ l_{11} : \mu_1, \ldots, l_{1n} : \mu_n \\
 & & & \mathsf{values}:\ l_{21} : e_1, \ldots, l_{2m} : e_m
\end{array}
$$

Figure 3.6: λ_i^{ML}-Rep: Expressions

where

$$\mu'_i \equiv \{\mu''_1/\alpha_1, \ldots, \mu''_n/\alpha_n\}\mu_i$$

and

$$\mu''_i \equiv \mathsf{Rec}\ \alpha_1 = \mu_1, \ldots, \alpha_n = \mu_n\ \mathsf{In}\ \alpha_i,$$

so that, for example, the unrolled form of type

$$\mathsf{Rec}\ l = () + \langle \mathsf{Int} \times l \rangle\ \mathsf{In}\ l$$

is

$$() + \langle \mathsf{Int} \times \mathsf{Rec}\ l = () + \langle \mathsf{Int} \times l \rangle\ \mathsf{In}\ l \rangle$$

  - roll_σ, coercing a recursive type to its rolled form (note that there are many possible rolled forms for a particular type, and hence the explicit annotation σ of the rolled type is required),
  - enum_to_int, coercing an enumerated type to an integer, or
  - $\mathsf{int\_to\_enum}_\sigma$, coercing an integer type to an enumeration (with the attendant range check)

- unary operations, including:
  - conversions for widening an integer to a real, and truncating a real to an integer, (real and floor),
  - bitwise negation (not) over integers,
  - square-root taking (sqrt) over reals,
  - trigonometric operations (sin, cos, and arctan) over reals,
  - hyperbolic operations (exp and ln) over reals,
  - array size computation (size),
  - projection of the i^{th} field from a record (π_i),
  - unary negation ($-$) and absolute value (abs) operations, for integer or real types, and
  - string length computation (strlen),

- binary operations, including:
  - arithmetic operations for division ($\div$), multiplication ($\times$), addition ($+$) and subtraction ($-$) over integers and reals, and modulus (mod) over integers,
  - comparison operations ($<, >, \leq, \geq$) over integers and reals, and (in)equality comparisons ($=$ and $\neq$) over integers, reals and strings,

- bitwise and, or and xor operations over integers,
- left and right shift operation (<< and >>) over integers,
- array allocation (alloc) and subscripting (sub) operations,
- exception packet introduction (excon) and elimination (deexcon); the former takes arguments of type Excon μ and μ, and creates an exception packet of type Exn carrying a value of type μ; the latter takes arguments of type Deexcon μ and Exn, and returns a variant sum $() + \mu$, where the first variant is returned if the exception packet is not of the type created by the Deexcon's Excon counterpart, and the second is returned otherwise, and
- string concatenation (strcat),

- miscellaneous operations, including:
  - array update ($e_1[e_2] := e_3$),
  - external (i.e. native Java) reference,
  - exception construction, which creates a pair of type $\langle \mathsf{Excon}\ \mu, \mathsf{Deexcon}\ \mu \rangle$ whose components are used to introduce and eliminate exception packets (as described above),
  - polymorphic equality and inequality testing at type μ,

- switches over integer, enumerated or sum types; the switch arms deconstruct sum types, so that, for example, a switch over sum type $(\mu_1) + \cdots + (\mu_n)$ would appear as

  switch e of
  1: $\lambda(x_1 : \mu_1)\ .\ e_1$
  $\vdots$
  n: $\lambda(x_n : \mu_n)\ .\ e_n$

  where if e is the i^{th} variant of the sum, then the i^{th} arm of the switch is selected, and x_i bound in e_i to the value carried by e,

- exceptions raised at type σ ($\mathsf{raise}_\sigma\ e$),

- guarded expressions with an exception handler (e_1 handle(x : Exn) e_2), where x is bound in e_2 to any exception thrown in e_1, and can subsequently be decomposed using deexcon operations as described above, and

- module definitions (export), specifying types and values exported by the module.

3.3.2 Lambda to λ_i^{ML}-Rep

As will be observed from the above description, there are two main differences between Lambda and λ_i^{ML}-Rep:

1. λ_i^{ML}-Rep does not include an explicit `datatype` type; instead, datatypes are represented by more basic constructors, such as enumerations or recursive sums, and

2. types are passed explicitly in λ_i^{ML}-Rep.

Translation from Lambda to λ_i^{ML}-Rep deals with these issues in a two-step transformation: first, representations for the datatypes of the Lambda program in λ_i^{ML}-Rep types are selected; second, the expression component of the Lambda program is translated to an λ_i^{ML}-Rep expression. These two components are dealt with in turn.

Datatype translation

Datatype translation roughly follows the scheme suggested by Morrisett [33]. A datatype is translated more or less as follows:

- If the datatype is recursive, then it is represented as a Rec constructor, whose body is a sum (+) constructor. For example, the SML datatype

```
datatype 'a list =
  nil
| cons of 'a * 'a list
```

  is translated into the Lambda datatype

  $$list = \Lambda(\alpha).nil + cons : \langle \alpha \times \alpha\ list \rangle$$

  which in turn is represented by the λ_i^{ML}-Rep type

  $$\lambda(\alpha :: \Omega).\mathsf{Rec}\ list = () + (\langle \alpha \times list \rangle)\ \mathsf{In}\ list$$

  Note that the constructor names have been erased in the translation. Note also the use of constructor-level λ-abstraction to capture the polymorphic nature of the datatype. Instances of the list type (such as `int list`) then appear as applications of the constructor function in λ_i^{ML}-Rep.

- If none of the constructors carry a value, the datatype is represented as an enumerated type. For example, the SML datatype

```
datatype degree = BMath | MMath | PhD
```

  is translated to the Lambda datatype

$$degree = BMath + MMath + PhD$$

which in turn is represented by the λ_i^{ML}-Rep constructor

$$\mathsf{Enum}\ 3$$

- If there is only one constructor, and it carries a value, then the datatype is represented directly by the value type. For example, the SML datatype

  ```
  datatype foo = BAR of int * int
  ```

  is translated to the Lambda datatype

  $$foo = BAR : \langle \mathsf{int} \times \mathsf{int} \rangle$$

  which in turn is represented by the λ_i^{ML}-Rep constructor

  $$\langle \mathsf{Int} \times \mathsf{Int} \rangle$$

- Otherwise (i.e. the type is not recursive and there are at least two constructors, at least one of which carries a value), the Lambda datatype is represented by a sum constructor. For example, the SML datatype

  ```
  datatype A = A1 of int | A2 of real | A3 of string
  ```

  is translated to the Lambda datatype

  $$A = A1 : \mathsf{int} + A2 : \mathsf{real} + A3 : \mathsf{string}$$

  which in turn is represented by the λ_i^{ML}-Rep constructor

  $$\mathsf{Int} + \mathsf{Real} + \mathsf{String}$$

Note that in any of the above cases, if the datatype is polymorphic, then the representation is a constructor function in λ_i^{ML}-Rep. In his translation, Morrisett also employs a specialized enumorrec representation for datatypes where only one of the constructors carries a value, and enumorsum for datatypes where some constructors carry values and some do not [33]. The enumorsum and enumorrec representations, however, are specifically designed for efficient assembly-code representation; there is no benefit to their use in translating to Java, and hence the additional complication that they introduce is avoided by not including them in λ_i^{ML}-Rep.

Expression translation

Translation of expressions from Lambda to λ_i^{ML}-Rep proceeds much in the manner described by Harper and Mitchell [20], Tolmach [48], and Morrisett et al. [43, 33] for compiling an implicitly typed ML-like language to a second-order λ-calculus.

The main feature of this translation is to insert type abstraction (Λ-abstraction in λ_i^{ML}-Rep) at the points in the code where type abstraction occurs (namely, in the generalization of let- and fix-bound variables), and type application at the points of mention of polymorphic variables.

This translation is not as straight-forward as it may seem, however. In particular, as Tolmach observes, since evaluation does not proceed under Λ-abstraction, any computation performed by a let-bound expression will not be performed at the binding point; instead, it will be repeated at each point of instantiation of the bound variable [48]. This is potentially inefficient, and more to the point semantically incorrect if the let-bound expression causes side-effects. For this reason, Wright's *value restriction*, which restricts polymorphic abstraction to values (i.e. constants and λ-abstractions, which are guaranteed not to cause side-effects) [53], is adopted. Fortunately, SML97 adopts precisely this restriction [31].

Translation of expressions proceeds as follows:

- A Lambda variable x_ρ is translated as an application of x to the types ρ. If ρ is empty, then no application is performed.

- Lambda constants i, s and r are translated directly to their λ_i^{ML}-Rep counterparts.

- A Lambda λ-abstraction is translated as a λ_i^{ML}-Rep λ-abstraction.

- A Lambda let expression

  $$\mathsf{let}\ x : \sigma = e_1\ \mathsf{in}\ e_2$$

  is translated to a let binding in λ_i^{ML}-Rep. If x is polymorphic, it is bound to a Λ-abstraction, abstracting over the free type variables of x; if not, then no abstraction is introduced.

- Translation of the Lambda fix construct (binding mutually recursive functions) is complicated by the fact that the fix expression may introduce type abstraction. It is assumed that the abstracted type variables for all bound functions are the same. If there are no abstracted type variables, the Lambda fix is translated to the λ_i^{ML}-Rep fix construct. (Note that although it is not enforced at a syntactic level, the expressions $e_1, \ldots, e_n$ in a fix binding are always λ-abstractions in Lambda.) If the set of abstracted type variables is non-empty, then an λ_i^{ML}-Rep fixtype declaration is introduced, where each of the bound variables $f_1, \ldots, f_n$ is bound to a curried function abstracting first over type variables, and second over term variables. Appearances of $f_1, \ldots, f_n$ in the bound expressions $e_1, \ldots, e_n$ are then translated as type application to the abstracted type variables. That is to say, if the binding of f_1 is $f_1 : \forall(\alpha :: \kappa). \ldots = \Lambda(\alpha :: \kappa).e_1$, then an appearance of f_1 in e_1 is be translated as an application of f_1 to α. As a (fairly contrived) example, the SML code

```
fun f (x, y, z) =
  if x = 0 then
    y
  else
    f (x - 1, z, y)
...
```

where `f` has type $\forall\alpha.\texttt{int} \times \alpha \times \alpha \rightarrow \alpha$, would be translated into `Lambda` as

$$
\begin{array}{l}
\textsf{fix}\ f : \forall\alpha\ .\ (\textsf{int},\ \alpha,\ \alpha) \rightarrow \alpha = \\
\quad \lambda\ (x : \textsf{int},\ y : \alpha,\ z : \alpha)\ . \\
\qquad \textsf{switch}_i\ (x)\ \textsf{of} \\
\qquad\quad 0\!:\ y \\
\qquad\quad \textsf{default:}\ f\ (-_{\textsf{int}}(x,\ 1),\ z,\ y) \\
\textsf{in} \ldots
\end{array}
$$

and in turn translated to the λ_i^{ML}-Rep code

$$
\begin{array}{l}
\textsf{let fixtype}\ f : \forall\ (\alpha :: \Omega)\ .\ (\textsf{Int},\ \alpha,\ \alpha) \rightarrow \alpha = \\
\quad \Lambda\ (\alpha :: \Omega)\ . \\
\qquad \lambda\ (x : \textsf{Int},\ y : \alpha,\ z : \alpha)\ . \\
\qquad\quad \textsf{switch}\ x\ \textsf{of} \\
\qquad\qquad 0\!:\ y \\
\qquad\qquad \textsf{default:}\ (f\ (\alpha))\ (\text{-}\ (x,\ 1),\ z,\ y) \\
\textsf{in} \ldots
\end{array}
$$

- Application in `Lambda` is translated directly as application in λ_i^{ML}-Rep.
- An `exception` expression in `Lambda` for exception constructor ex gives rise to a binding for an λ_i^{ML}-Rep variable ex of type $\langle \textsf{Excon}\ \tau, \textsf{Deexcon}\ \tau \rangle$ in λ_i^{ML}-Rep, corresponding to a (constructor, deconstructor) pair for the exception. A `Lambda` `excon` primitive at exception constructor ex (i.e. an expression of form $(\textsf{excon}\ ex)\ (e)$) is then translated to an λ_i^{ML}-Rep `excon` operation taking $\pi_1\ ex$ and e as arguments (i.e. an expression of form $\textsf{excon}\ (\pi_1\ ex, e)$). Deconstruction is handled similarly, where the sum type resulting from the `deexcon` is decomposed with the aid of a `switch` construct.
- A `Lambda` `raise` expression is translated directly as an λ_i^{ML}-Rep `raise` expression.
- The `handle` expression of `Lambda` is translated to the corresponding `handle` construct in λ_i^{ML}-Rep. Note that in λ_i^{ML}-Rep, the handler is a function whose parameter is bound

to the thrown exception, whereas in Lambda, the handler is constrained to be a function over exceptions (but not necessarily a λ-abstraction per se); in the translation from Lambda to λ_i^{ML}-Rep, then, the λ_i^{ML}-Rep translation for the Lambda handler function is applied to the variable bound to the caught exception.

- The Lambda switch construct is compiled differently for each of the different types of index:

  - The integer variant switch_i is straight-forward, and maps directly to an λ_i^{ML}-Rep switch expression, where the arm functions take no parameters.

  - The real and string variants switch_r and switch_s have no direct counterpart in SML (indeed, type `real` is no longer considered an equality type in the SML97, so the value of the switch_r construct is questionable). The switch is translated to a series of comparisons in λ_i^{ML}-Rep (which are themselves two-case switch statements ranging over Enum 2 (i.e. boolean) types).

  - The datatype switch variant switch_c presents some difficulty in translation. In λ_i^{ML}-Rep, sum-indexed switch arms decompose the switch argument implicitly, whereas there is an explicit decon primitive to achieve this effect in Lambda. Proper translation of the switch therefore depends on the type of the argument to the switch. As noted above, a Lambda datatype may be represented by any of a number of λ_i^{ML}-Rep types, and so translation of the switch_c depends on the λ_i^{ML}-Rep representation of the argument argument type.

    In the case of a recursive datatype (represented as a recursive sum type in λ_i^{ML}-Rep), the argument is unrolled by the unroll coercion, and the switch arms are handled as in the case of a sum datatype.

    In the case of a sum datatype, fresh variables are generated for each case in the sum, binding the decomposed sum type. The name of this binding is stored in the translation environment, so that, as an optimization, if a decon is encountered in an arm decomposing the switch argument, it can be replaced by the bound name.

    In the case of an enumeration datatype (where there is more than one constructor, and no constructor carries a type), the switch is translated in the same manner as an integer-indexed switch_i.

  - The switch_{ex} construct is translated similarly to the switch_c described above.

- The con primitive has no direct counterpart in λ_i^{ML}-Rep; the code generated in λ_i^{ML}-Rep depends on the representation of the Lambda datatype being constructed.

  - In the case of a sum representation, an appropriate inject expression is generated.

  - In the case of a recursive sum representation, the inject expression is also 'rolled' via the roll coercion.

  - For an Enum i representation, the tag for the constructor is introduced as an enum constant expression.
  - For a single-constructor datatype, represented as itself, the argument to the con is translated directly.

- The decon primitive in Lambda again has no direct counterpart in λ_i^{ML}-Rep. Translation proceeds as described in the case of the switch_c above, save that here there are only two arms: one for the constructor being deconstructed, and another default case which raises a bind exception.

- The Lambda excon primitive is translated to an excon operation in λ_i^{ML}-Rep, taking the first component of the $\langle \mathsf{Excon}\ \mu, \mathsf{Deexcon}\ \mu \rangle$ pair (to which the ex will have been translated) as its Excon argument.

- The deexcon primitive is translated as a λ_i^{ML}-Rep switch operation (as in the switch_{ex} above) operating on an deexcon expression.

- Other Lambda primitives are translated in the obvious manner; in some cases, it is necessary to extract the type of the argument to properly annotate the λ_i^{ML}-Rep operation (as for the '=' operator).

3.3.3 Optimization

In order to prevent unnecessary effort in subsequent stages of the compiler, a series of simple optimizations on the λ_i^{ML}-Rep form are performed, roughly following those suggested by Appel and Jim [6, 7, 4].

In the discussion that follows, an expression e is defined to be *small* if and only if e is a variable, or an integer, real or enumeration constant. An expression e is defined to be *safe* if its execution cannot cause side-effects (such as updating an array, or input/output) or cause an exception to be raised. As a conservative approximation to the safeness property, it is assumed all expressions are safe except function application, certain unary and binary operations (which either cause updates to the store, or can cause exceptions to be raised), raise expressions, and expressions whose sub-expressions are not themselves safe.

The optimizations performed include:

- Constructs of form

  $$(\lambda(x_1 : \mu_1, \ldots, x_n : \mu_n).e\ (x_1, \ldots, x_n))\ (e_1, \ldots, e_n)$$

  are optimized to

  $$e\ (e_1, \ldots, e_n)$$

This construct is generated quite frequently by the ML Kit in its pattern-matching compiler, and so optimization is highly beneficial.

- Constructs of form

  $$(\lambda(x_1 : \mu_1, \ldots, x_n : \mu_n).e)\ (e_1, \ldots, e_n)$$

  are optimised to

  $$\begin{array}{l} \mathsf{let}\ x_1\ :\ \mu_1\ =\ e_1 \\ \vdots \\ \mathsf{in}\ \mathsf{let}\ x_n\ :\ \mu_n\ =\ e_n \\ \mathsf{in}\ e \end{array}$$

  This optimization helps to prevent unnecessary closure allocation.

- If e_1 is small, or if e_1 is safe and x occurs free exactly *once* in e_2, then

  $$\mathsf{let}\ x\ :\ \sigma = e_1\ \mathsf{in}\ e_2$$

  is optimized to

  $$\{e_1/x\}\ e_2$$

- If e_1 is safe and x does not occur free in e_2, then

  $$\mathsf{let}\ x\ :\ \sigma = e_1\ \mathsf{in}\ e_2$$

  is optimized to

  $$e_2$$

  This optimization amounts to dead-variable elimination, and is particularly useful in conjunction with the following optimization.

- If $e_1, \ldots, e_n$ are all small, then

  $$\mathsf{let}\ x\ :\ \langle\mu_1 \times \cdots \times \mu_n\rangle = \langle e_1, \ldots, e_n\rangle\ \mathsf{in}\ e$$

  is optimized to

  $$\mathsf{let}\ x\ :\ \langle\mu_1 \times \cdots \times \mu_n\rangle = \langle e_1, \ldots, e_n\rangle\ \mathsf{in}\ e'$$

  where

  $$e' \equiv \{e_1/(\pi_1\ x),\ \ldots,\ e_n/(\pi_n\ x)\}\ e$$

- Simple constant-folding of integer arithmetic operations which do not cause overflow is performed.

Chapter 4

Representation Analysis

4.1 Introduction

From a software engineering standpoint, one of the most attractive features of SML is that it possesses a *polymorphic* type system. Polymorphism can be broadly defined as the property that some values and variables of a program may have more than one type [3, 13]. While many languages have this property in their type systems to some degree (for example, operators such as '+' are often overloaded to perform similar operations on different types), SML's type system permits variables and values to be defined over a range of types — usually referred to as *parametric polymorphism* (as distinct from *ad-hoc* and *subtype* (or *inclusion*) polymorphism). Parametric polymorphism allows the SML programmer to write generic routines which operate uniformly over a range of types. While C++ and Ada address the issue of generic code by providing template and generic and facilities respectively [41, 52], these are not type-checked or compiled to object code until instantiation, and hence amount to little more than macro substitution.

The flexibility and power afforded the programmer by a polymorphic type system is not without penalty, however. An immediate consequence of the polymorphic type system is that the actual type of an object may not be known until run-time. Compiling polymorphic code therefore requires either that all types have the same representation, or that the code has some mechanism for discovering types and their representations at run-time.

Consider as an example the operation of pairing. In SML, a polymorphic pairing function could be written as

```
type 'a pair = {fst : 'a, snd : 'a}
fun make_pair (x : 'a) : 'a pair = {fst = x, snd = x}
val y : string pair = make_pair "hello"
val z : string = #fst y
```

where `make_pair` has type $\forall\alpha.\alpha \rightarrow \alpha$ `pair`, `y` has type `string pair`, and `z` has type `string`. Consider now an attempt to translate this into Java. Begin by defining a pair type:

```
class pair {
    Object _fst;
    Object _snd;
    pair(Object fst, Object snd) { _fst = fst; _snd = snd; }
};
```

The idea here is that a pair can consist of any two objects; note that property that they must be the *same* type has been lost, replaced by the weaker property that they must both be subtypes of `Object`. Note also that a number of basic types have been excluded from pairing, since in Java the 'primitive' types (`int`, `double`, etc) are not reference types, and so not subtypes of `Object`, as will be explored further below. Fortunately, the example pairs strings, and `String` is indeed a subtype of `Object`. The pairing operation would then look something like

```
pair make_pair(Object x) {
    return new pair(x, x);
}
```

and a pair of strings can be created by

```
pair y = make_pair("hello");
```

The difficulty now arises in selecting the first field from `y`. The naïve approach

```
String z = y._fst;
```

does not work, because `Object` is not a subtype of `String`. Instead, the result of `y._fst` must be cast in order to perform the assignment, at the cost of a run-time type check, and rather messy code

```
String z = (String) y._fst;
```

As noted above, Java makes a distinction between primitive types (`int`, `double`, `boolean`, etc), which are always stack-allocated, and reference types (classes and interfaces), which are always heap allocated. That is to say, the primitive types are not reference types, and so not subclasses of `Object`. The example code above can only construct pairs of reference types, so the call

```
pair y = make_pair(3);
```

is illegal since `int` (the type of 3) is not a subtype of `Object`. Instead, a heap-allocated copy of 3 must be made using the Java-provided 'wrapper' class `Integer`:

```
pair y = make_pair(new Integer(3));
```

Extraction of the first field from the pair now becomes the cumbersome

```
int z = ((Integer) y._fst).intValue();
```

where the result of the field selection `y._fst` must be cast an `Integer`, and then the `int` extracted by method invocation.

Java's use of name-equivalence for types makes translation of polymorphism even more difficult. Consider for example a pair of `string` values (of SML type `string` $\times$ `string`); the natural representation of string pairs would in Java is

```
class StringPair {
    String _fst;
    String _snd;
    StringPair(String fst, String snd) {
        _fst = fst;
        _snd = snd;
    }
};
```

One would then expect to be able to use objects of type `StringPair` anywhere the more general type `Pair` is employed (in particular, so that they could be passed to polymorphic functions expecting arguments of type $\alpha \times \alpha$). Indeed, it seems sensible that if type A is a subtype of A' ($A <: A'$), then it should follow that $A \times A <: A' \times A'$. However, this is not the case in Java, where the subtyping relation must be explicitly indicated — so that, for example, the definition of `StringPair` would have to be changed to include an `extends` clause, indicating that it is a subtype of `Pair`. Unfortunately, this is not good enough, since objects of type `string` $\times$ `string` could also be passed to functions taking arguments of type $\alpha \times$ `string`, `string` $\times \alpha$, or α, so that `StringPair` would have to be a subclass of three distinct types (which are not themselves subclasses of one another) — clearly impractical. There is therefore a choice between either representing string pairs (and indeed all pairs) as instances of `Pair` at all times, or introducing a coercion from `StringPair` to `Pair` when moving from monomorphic to polymorphic contexts — both approaches are investigated below.

It is clear then that naïve translation of SML code to Java is not feasible; a more sophisticated scheme to deal with the awkwardnesses imposed by Java's type system must be adopted. In the sections that follow, a number of approaches are described.

It is worth noting that these problems are not unique to translating to Java. Much the same problems arise in compilation to assembly code if different types occupy different amounts of space, or are passed by different conventions. A common example is double-precision floating point numbers, which are often larger than pointers and integers (64-bit versus 32-bit) and are often stored in their own special-purpose floating-point registers. A polymorphic pairing function which operates equally well on floating-point numbers and other types, then, must be passed a 'boxed' floating-point number in order to be able to operate correctly.

4.2 Traditional Approaches

The problem with polymorphism is essentially that a polymorphic variable must have a *uniform* representation regardless of its actual type. One obvious way of achieving this property is to represent all values uniformly at all times, as is done in untyped languages like Lisp and Scheme. The most sensible choice for a uniform representation is a single word pointer, corresponding to reference types in Java. In this scheme, values of primitive types such as `int` and `double` are 'boxed' in their heap-allocated counterparts `Integer` and `Double` at all times, only being 'unboxed' when passed to native functions (such as for addition, subtraction, and so on). Similarly, pair types like `string` $\times$ `string` adopt the uniform representation for all pair types, such as the `Pair` type above, rather than using more specialized types such as `StringPair`. This approach is clearly unattractive from a performance standpoint, since each time an integer is passed to a function, or placed in a data structure, an `Integer` object is allocated, taking both time and space; unnecessary boxing and unboxing can also occur if the compiler is not careful [22]. Casts (and hence run-time type checks) to extract fields from records like `Pair` are also necessary. Finally, interfacing to lower-level languages (for example, invoking Java API functions from an SML program) becomes difficult, since the lower-level languages generally use 'unboxed' representations at all times.

A somewhat more attractive solution is to create specific instances of polymorphic code for each instantiation in the program (what Morrison et al. call 'textual polymorphism' [34]), much in the way that templates and generics in C++ and Ada are implemented. Since there are only finitely many different instantiations of a polymorphic function in a program, it is possible to generate different code for each such instantiation, specialized to the particular instantiating types. In the example above, specialized versions of the pairing function which operated on strings and integers would be created. In this approach, boxing and unboxing are unnecessary, since effectively all of the polymorphism has been removed from the program. There is, however, potential for considerable code duplication. Furthermore, separate compilation is impossible, since if a function in a module is compiled separately from its application, it is not known at compile-time what all the possible instantiations of the function will be. This approach is therefore space-inefficient and non-general, and hence unattractive.

The two approaches suggested in this section roughly correspond to the 'homogeneous' and 'heterogeneous' translations presented by Odersky and Walder for their Pizza language, which extends Java with parametric polymorphism, algebraic datatypes, and high-order functions [35]. In the heterogeneous translation, each instantiation of a polymorphic class in the source gives rise to a different class definition in the target — this corresponds to the specialization approach. In the homogeneous translation, type variables in polymorphic datatypes and functions are replaced by type `Object`, much as was done in the pairing example above. This approach roughly corresponds to the uniform representation scheme suggested above, with the drawback that less efficient 'boxed' representations are used for some types even when the type is not used polymorphically. For example, in Pizza the pair type could be declared as

```
class Pair<T> {
    T _fst;
    T _snd;
    Pair(T fst, T snd) {
        _fst = fst;
        _snd = snd;
    }
}
```

and the make_pair function as

```
<T> Pair<T> make_pair(T x) {
    return new Pair(x, x);
}
```

The pair type would then be translated into Java almost identically to the Pair object above (i.e., with two Object fields). Thus, field extraction from a Pair object always involves a cast, even if the Pair is not being used in a polymorphic context. Furthermore, creating pairs of non-reference objects (such as pairs of int) requires the 'wrapping' procedure described above. The result is a rather inefficient representation.

4.3 Coercion-Based Approaches

A novel *coercion*-based approach to the problem of polymorphic code has been suggested by Leroy [26], building on his own earlier work [25] and complementing the work of Peyton Jones [22]. Peyton Jones suggested introducing boxing and unboxing operations explicitly into the intermediate language (by way of an algebraic data type), exposing them for conventional optimisation. He further introduced the key restriction that polymorphic functions should only range over boxed types in order to address what Shao later called the *vararg* problem [38], which will be addressed in greater detail below. Leroy's contribution was to show a transformation from the unrestricted polymorphism of SML to Peyton Jones' restricted system.

Leroy's idea is to keep objects in their natural (unboxed) representation whenever possible; objects are *coerced* into a uniform 'wrapped' representation when passed to polymorphic functions, and coerced back to their natural representation when returned from polymorphic functions (much as primitive values must be coerced to and from their boxed forms when passed to native functions in the traditional untyped approach). The 'wrapped' representation would normally be the 'boxed' form of the value (where the 'boxed' form of a value is one which fits in a single-word register), although it may be more elaborate than this, as will be discussed below.

Coercions are introduced so as to preserve the invariant that objects whose type is a type variable (i.e., unknown at compile-time) have a uniform representation — the coercion `wrap`(τ) coerces from the natural (specialized) representation of a value of type τ to its uniform representation, and `unwrap`(τ) coerces from uniform representation to the specialized representation for type τ. Polymorphic values in polymorphic code are thus guaranteed to always be in uniform representation, while monomorphic code (which Leroy and others have claimed is the majority of 'realistic' SML code) leaves values in their natural (and more efficient) representation.

The key to Leroy's approach is to introduce the coercions to and from uniform wrapped representation at the appropriate points in the code. SML has the convenient property that the only polymorphic values in a program are `let`-bound variables. As a consequence, in SML, the appropriate points at which coercions should be introduced are where a let-bound variables are applied to their arguments (where their type schemes are instantiated).

Leroy's scheme works adequately for simple types like as tuples and functions, but has difficulty with mutable and recursive types. The difficulty with coercion in these cases is that it involves creating a wrapped *copy* of the coerced object. Reference types cannot be coerced by copying, since to do so would lose the update semantics of the type, resulting in an unsound translation. Although recursive values can be coerced, doing so would require time and space proportional to the size of the structure being coerced, which is clearly impractical. Leroy's solution is to leave reference and recursive types wrapped at all times, even when not being used in polymorphic code. The representation of recursive types (widely used in SML in the form of lists) is therefore relatively inefficient.

As Leroy points out, however, it is not enough to require that recursive types be wrapped at all times. Consider, for example, a list 'unzipping' function having type $\forall(\alpha,\beta).(\alpha \times \beta)\ \texttt{list} \rightarrow (\alpha\ \texttt{list} \times \beta\ \texttt{list})$. This function could be called to operate on lists of any possible type of pair — for example, lists of type $(\texttt{int} \times \texttt{int})\ \texttt{list}$, $(\texttt{int} \times (\texttt{int} \times \texttt{int}))\ \texttt{list}$, and so on. The representation of each of these types must therefore be compatible with all the others. Since it is not known beforehand to what functions a value may be passed, the worst must be assumed, and a representation for lists employed that allows the list to be passed into contexts requiring an $\alpha\ \texttt{list}$, $(\alpha \times \beta)\ \texttt{list}$, $(\texttt{int} \times \alpha)\ \texttt{list}$, and so on. The restriction that the components of a wrapped object must themselves be wrapped must therefore be introduced. This 'recursive' wrapping (or *full boxing*, as Shao calls it [38]) detracts considerably from the simplicity of Leroy's scheme.

Interestingly, Leroy has recently suggested that it may be more effective to leave objects in their wrapped state at all times, and employ local optimizations to reduce the cost of wrapping, rather than to employ a full-fledged coercion-based approach [27].

4.4 Type Passing Approaches

Another approach suggested by Morrisett et al. is to pass types at run-time [33, 43]. In such an approach, a program can discover an object's representation by inspecting the run-time representation of the type. As a consequence, objects can remain in their unboxed state at all times.

This method is attractive in that it can cope properly (and efficiently) with recursive and mutable types, leaving them in their natural state, rather than arbitrarily requiring that they be (recursively) boxed at all times. The disadvantage of this approach is that additional time and space is required for constructing, passing and analyzing type descriptors at run-time. Shao and Leroy have claimed that the cost of this 'heavy-weight' run-time type analysis is considerable [38, 27].

A more serious problem is that it may be impossible to call functions with different argument types in the first place — Shao's *vararg* problem [38]. In Java, for example, a function which takes an `int` argument can *only* be passed an `int` — the virtual machine does not permit an object with any other type to be passed, regardless of whether if it is the same size as `int` or what additional type information accompanies it. The only occasion where this is an issue for Morrisett is in the passing of floating-point values, which do not fit in general-purpose registers; Morrisett deals with this in a rather ad-hoc way by wrapping all floating-point values, except where they occur in arrays [33]. The problem is more serious in Java, where all primitive types have this problem, and effectively precludes a type-passing approach.

4.5 Hybrid Approaches

Shao has recently suggested a *flexible representation* approach, which basically amounts to a hybrid where aspects of both coercion and type-passing approaches are employed [38]. Shao seeks to employ the coercion-based approach where possible, and resort to type-passing only where necessary; as a by-product, the vararg problem can be dealt with reasonably neatly.

Shao's approach first introduces a type function Wrap which maps a type to its boxed form, and two value functions wrap, which maps a value to its boxed form, and unwrap, which performs the inverse transformation from boxed form to natural form. That is to say, wrap has type $\forall(\alpha :: \Omega).\alpha \rightarrow \mathsf{Wrap}\ \alpha$, and unwrap has type $\forall(\alpha :: \Omega).\mathsf{Wrap}\ \alpha \rightarrow \alpha$. Note that what is meant by the wrapped form of a type has deliberately not been defined — one of the strengths of Shao's technique is that it works equally well for any definition of wrapping and unwrapping which satisfy certain basic criteria.

Wrapping and unwrapping coercions are introduced in the source code more or less in the manner of Leroy's coercion-based approach. For a second-order, explicitly typed λ-calculus, coercions must be introduced at the point of type application, much as they are at the point of mention of a polymorphic variable in Leroy's technique.

The novelty of Shao's work is that the definition of wrap and unwrap depend on the extent of boxing desired by the programmer. Shao presents three schemes which define differing levels of boxing for types:

- In *full boxing*, types are boxed recursively. This is the scheme used in Leroy's coercion-based approach.
- In *simple boxing*, only the top layer of a data structure is boxed.
- *Partial boxing* is similar to simple boxing, save that all function arguments and results are also boxed.

The attraction of simple boxing (as opposed to full boxing) is that less coercions are required; however, in certain circumstances, run-time analysis of types may be necessary. Partial boxing eliminates some (but not all) of the cases where run-time type analysis is necessary.

The run-time work of Shao's approach is performed by the wrap and unwrap coercions. Effectively, then, Shao's approach is two-level method: first, wrap and unwrap coercions are introduced, following the method of Leroy; second, wrap and unwrap are defined in terms of boxing and unboxing operations.

4.6 Implementation

The SML-to-Java compiler essentially employs Shao's approach of a two-phase translation, in order to achieve a fully-boxed representation. The fully-boxed representation was chosen because it is straight-forward to implement, avoiding altogether the need for run-time type analysis.

The SML-to-Java compiler takes advantage of the two-stage nature of Shao's representation analysis to break the analysis into two phases. In the first stage, the wrap and unwrap coercions and the Wrap constructor are added to λ_i^{ML}-Rep, but their definition is deferred to the final stage of compilation (translation to Java). This permits flexibility in choosing representation strategy, and allows the use Java's subclassing and method override facilities to implement wrapping and unwrapping, without the need for introducing an elaborate typecase or typerec operation in the intermediate form. Note that no commitment to a fully-boxed representation has been made at this stage. The translation of the wrap and unwrap operations will be discussed in detail in Chapter 7.

At this point of the compiler, then, only modest revision to the λ_i^{ML}-Rep syntax is required. In particular, a constructor Wrap μ, denoting the wrapped form of constructor μ, is added, and two new term-level operations, wrap(μ, e) and unwrap(μ, e), which coerce terms of type μ and Wrap μ to types Wrap μ and μ respectively, are introduced.

4.6.1 Type Translation

Following Shao, two transformations on types are defined: σ^u, translating a type in λ_i^{ML}-Rep to its unwrapped form, and σ^w, translating a type to its wrapped form. σ^u is in most cases an identity operation; however, for recursive and array types, the unwrapped representation must be identical to the wrapped form, so that no copying coercions occur. As a consequence, the unwrapped form of constructor $\mathsf{Array}\ \mu$ is $\mathsf{Array}\ (\mathsf{Wrap}\ \mu)$.

Recursive types are rather more troublesome to translate. Consider two lists types: a monomorphic integer list type, and the usual polymorphic list type. These would normally be declared in SML as

```
datatype ilist =
  inil
| icons of int * ilist

datatype 'a list =
  nil
| cons of 'a * 'a list
```

The equivalent λ_i^{ML}-Rep types are

$$\mathit{ilist} \equiv \mathsf{Rec}\ \mathit{il} = () + (\langle \mathsf{Int} \times \mathit{il} \rangle)\ \mathsf{In}\ \mathit{il}$$

and

$$\mathit{list} \equiv \lambda(\alpha :: \Omega).\mathsf{Rec}\ l = () + (\langle \alpha \times l \rangle)\ \mathsf{In}\ l$$

respectively. Now, `ilist` can only be an instantiation of α; in particular, polymorphic code always treat the Int field of type *ilist* as an Int. The same is not true of *list*, however, where a polymorphic function of type $\forall(\alpha :: \Omega).\mathit{list}\ \alpha \to \alpha$, for example, could extract the first element from a list, and treat it as a polymorphic type. The unwrapped form of *ilist* therefore does not require that the Int field be wrapped, whereas the α component of *list* must be wrapped, so that no coercion is required to pass the list into a context requiring an object of type *list* α.

The difficulty arises when considering an SML value of type `int list`; this has type *list* Int in λ_i^{ML}-Rep, which, after type normalization, is indistinguishable from *ilist*. Once types are normalized, then, it is impossible to determine which fields of a datatype are instantiations of type variables (such as α in *list*) or are monomorphic (such as Int in *ilist*). It is therefore not possible to determine by inspection what form the unwrapped type should take. This is not a problem in a true type-passing approach, such as that employed by Morrisett, since both the instantiation of the polymorphic type and the monomorphic type have the same run-time representation. However, since a type-passing approach is not being used, it is necessary to distinguish the two cases.

The root of the problem is really that the λ_i^{ML}-Rep calculus is overly eager in doing away with datatypes and type-level application. In retrospect, may have been more sensible to employ a simpler calculus which retained a notion of polymorphic datatypes, rather than dispensing with them early in the compilation. A type-passing variant of `Lambda`, for example, may have been more suitable. As a work-around, however, it is sufficient to avoid type normalization until after the coercions have been introduced, and to treat type-level λ-abstraction as if it defined a polymorphic datatype. This makes it possible to determine which fields in the unwrapped representation of recursive datatypes should be wrapped — namely, those which are abstracted over.

4.6.2 Term Translation

Following Shao, coercions are introduced at the point of type application (i.e., at a term $e\ (\mu_1, \ldots, \mu_n))$, so that in the translated form, e is only applied to wrapped types. The coercion for most types is relatively straight-forward, and will not be described in detail here.

There are two other points where it is necessary to introduce coercions, corresponding to the introduction and elimination forms for incoercible types:

- roll and unroll coercions can be thought of as introduction and elimination forms for recursive types. Since certain fields of recursive types are boxed at all times (i.e., even in the 'unwrapped' state), it is necessary to introduce coercions at roll and unroll operations, so that the unwrapped, unrolled form is properly converted to the wrapped, rolled form, and vice versa. It is again worth noting that λ_i^{ML}-Rep's type system makes the task of deciding which fields are to be rolled more difficult than necessary.

- The array introduction and elimination forms (alloc and sub) also require coercion, since array elements are kept wrapped at all times.

Chapter 5

A-Normalization

5.1 Introduction

A distinction is often made in programming languages between *expressions*, which compute values, and *statements*, operate on values, but which are evaluated for their side effects only. Many languages restrict the type of computations which are permitted as expressions. In traditional imperative languages, expressions are limited to arithmetic and logical operations; constructs for selection and iteration are not generally permitted[1]. In the extreme case of assembly code, only numeric constants and variables are allowed as expressions. In most functional languages, by contrast, all computations belong to a universal class of expressions. One of the tasks in compiling SML to Java is therefore translating from a view of computations as expressions to the more conventional view of computations as statements operating on expressions.

As an example, consider the use of a case expression as the argument to a function in SML:

```
fun foo (x : int) = ... (* Some SML code *) ...
...
val y = 3
val z = foo (case y of 0 => 1 | 1 => 2 | _ => y - 1)
```

The obvious translation into Java or C does not work:

```
int foo(int arg) { ... /* Some Java or C code */ ... }
...
int y;
int z;
```

[1]Although it is interesting to note that ALGOL 68 made no distinction between statements and expressions [50, 51].

```
y = 3;
z = foo(switch (y) (1: 1; break; 2: 2; break; default: y - 1; break))
```

This code is not syntactically valid, because `switch` is a statement, not an expression, and, in Java and C, arguments to methods must be expressions. The solution is to name the result of the switch with a temporary, and pass the temporary to the function:

```
int foo(int arg) { ... }
int y = ...;
int temp;
int z;

y = 3;
switch (y) {
    1: temp = 1; break;
    2: temp = 2; break;
    default: temp = y - 1; break;
}
z = foo(temp);
```

The difficulty can be attributed to the fact that functional languages allow the results of all computations (including conditionals) to remain anonymous, while conventional imperative languages require that some computations (namely, those which take the form of statements) must modify store in order to be useful — essentially, that their results must be named. As Appel observes, this is analogous to the way in which the early imperative languages allowed arithmetic and logical relations to be anonymous in expressions, by contrast with assembly code, where all intermediate arithmetic results had to be named [4]. The problem is then that some operations which are explicit in conventional languages (such as storing the result of a statement in a variable) are implicit in functional ones.

5.2 Continuation Passing Style and A-Normal Form

A technique often employed to deal with this problem is to transform the source program into Continuation Passing Style (CPS) [4, 6]. The idea of a CPS transformation is to make control- and data-flow explicit, and to name all intermediate results, resulting in a form which is amenable to translation to machine code.

CPS achieves this by transforming the program so that all control flow takes the form of a tail-call. Every function is passed an additional argument (its *continuation* — 'what to do next' in the computation), which it invokes instead of returning. Another useful feature of a CPS transformation is that all intermediate results are named, so that a program in CPS operates only on named values and constants, much as assembly code. CPS is then basically a formalized assembly code, and is straight-forward to translate to machine code.

While CPS intermediate forms have been fruitfully employed in a number of compilers for functional languages [24, 8], CPS has two significant drawbacks:

- Since CPS functions do not return, CPS function invocations must be translated to assembly code in such a way that they do not cause stack growth — the target language (to which the CPS is being translated) must therefore provide a form of global jump or `goto`.
- Naïve CPS translation introduces many 'administrative' λ-abstractions, which must be eliminated by optimization (for example by β-reduction).

In the particular case of translation to the Java, the first point above is a critical problem. For reasons of security, the JVM does not permit jumps out of a method; as a consequence, the Java language does not provide a `goto` construct. The only calling mechanism available in Java is therefore method invocation, which causes stack growth, and is hence unusable for CPS tail calls. This restriction seems to rule out a CPS transformation.

The second problem is of more general concern, and is addressed by Flanagan et al. [16]. Their observation is that in performing a CPS transformation, the optimization phase required to eliminate unnecessary administrative λ-abstractions is essentially an undoing of the original CPS transformation. That is to say, a CPS compiler must go through the stages of:

1. CPS transformation
2. β-normalization
3. "un"-CPS transformation

in order to produce efficient code. As a consequence, it is suggested that these three stages be replaced by a single-step A-transformation to A-normal form.

Indeed, A-normal form is attractive for an SML-to-Java compiler. The A-reductions lift redexes out of evaluation contexts (as the `switch` was lifted out of the invocation of `foo`), and names intermediate results, while not introducing unnecessary λ-abstractions that need to be optimized away. For this reason, a transformation in the flavor of A-normalization was adopted for the SML-to-Java compiler.

5.3 λ_i^{ML}-Norm

The implementation of A-normalization in the SML-to-Java compiler employs modified version of λ_i^{ML}-Rep, λ_i^{ML}-Norm, which distinguishes between expressions and statements, as shown in Figures 5.1 and 5.2. Kinds, constructors, and types are as before, and expressions are separated into two categories, based on Java's separation of expressions and statements. Note that traditional *A*-normalization is considerably more restrictive in what is left as an expression — typically only names and constants [33, 44, 16]. However, the fact that Java permits some intermediate results (namely arithmetic and logical expressions) to remain unnamed can be exploited to simplify the translation. If the target of the translation were the JVM instead of Java, a more aggressive normalization would have to be adopted.

Statements are only allowed in a few places: in the bodies of functions and let-declarations, as the arms of a switch, as the guarded block of a handle, and importantly in a let assignment. This last feature allows the results of arbitrary statements to be named, and provides a way of translating a statement into a variable expression representing the result of the statement. return sub-statements in let bindings should be interpreted as assignment to the nearest enclosing let-bound variable.

Translation from λ_i^{ML}-Rep to λ_i^{ML}-Norm takes the form of a continuation-passing-style algorithm, in the manner of the linear-time algorithm presented by Flanagan et al. [16]. The idea is to define a function *normalize* taking an λ_i^{ML}-Rep expression, and a continuation k mapping expressions to statements. λ_i^{ML}-Rep expressions which map to λ_i^{ML}-Norm expressions are then translated by applying the continuation to the expression, to produce an λ_i^{ML}-Norm statement. λ_i^{ML}-Rep expressions which map to λ_i^{ML}-Norm statements are translated by generating a temporary label for the statement, applying the continuation to the label, and then enclosing the resulting statement in a let statement, binding the variable to the result of the statement.

The interesting cases in this translation arise in the translation of switch expressions in λ_i^{ML}-Rep. For example, the λ_i^{ML}-Rep code:

let foo : (Int) $\rightarrow$ Int = ...
in let x : Int = ...
in foo (switch (x) of 1: 2, 2: 3, default: x - 1)

is translated as

let foo : (Int) $\rightarrow$ Int = ...
in let x : Int = ...
in let $temp$: Int = switch (x) of 1: return 2, 2: return 3, default: return (x - 1)
in foo ($temp$)

Note that the result of the switch has been bound to $temp$.

The handle expression in λ_i^{ML}-Rep becomes a statement in λ_i^{ML}-Norm by a similar process. For example, the λ_i^{ML}-Rep code:

let foo : (Int) $\rightarrow$ Int =
in let bar : (Int) $\rightarrow$ Int = ...
in let x : Int = ...
in foo (bar (45) handle(x : Exn) . 21)

$$
\begin{array}{llcl}
\textit{Declarations} & d & ::= & x : \sigma = t \\
 & & | & \alpha : \kappa = \mu \\
 & & | & \textsf{fix} \\
 & & & f_1 : \sigma_1 = \lambda(x_{11} : \mu_{11}, \ldots, x_{1n_1} : \mu_{1n_1}).t_1 \\
 & & & \vdots \\
 & & & f_m : \sigma_m = \lambda(x_{m1} : \mu_{m1}, \ldots, x_{mn_m} : \mu_{mn_m}).t_m \\
 & & | & \textsf{fixtype} \\
 & & & f_1 : \sigma_1 = \Lambda(\alpha_{11} :: \kappa_{11}, \ldots, \alpha_{1n_1} :: \kappa_{1n_1}).t_1 \\
 & & & \vdots \\
 & & & f_m : \sigma_m = \Lambda(\alpha_{m1} :: \kappa_{m1}, \ldots, \alpha_{mn_1} :: \kappa_{mn_m}).t_m \\
\\
\textit{Expressions} & e & ::= & x \\
 & & | & m.l \\
 & & | & \langle e_1, \ldots, e_n \rangle \\
 & & | & \textsf{inject}_i^{\mu}\ (e_1, \ldots, e_n) \\
 & & | & i \\
 & & | & r \\
 & & | & \textsf{enum}_{\mu}\ i \\
 & & | & s \\
 & & | & \lambda(x_1 : \mu_1, \ldots, x_n : \mu_n).t \\
 & & | & \Lambda(\alpha_1 :: \kappa_1, \ldots, \alpha_n :: \kappa_n).t \\
 & & | & c\ e \\
 & & | & op_1\ e \\
 & & | & op_2\ (e_1, e_2) \\
 & & | & m
\end{array}
$$

Figure 5.1: λ_i^{ML}-Norm: Declarations and expressions

$$
\begin{array}{llcl}
\textit{Misc ops} & m & ::= & e_1[e_2] \ := \ e_3 \\
 & & | & \mathsf{extern}\ s : \sigma \\
 & & | & \mathsf{newexn}\ \mu \\
 & & | & = \ (\mu, e_1, e_2) \\
 & & | & \neq \ (\mu, e_1, e_2) \\
 & & | & \mathsf{wrap}\ (\mu, e) \\
 & & | & \mathsf{unwrap}\ (\mu, e) \\
 & & & \\
\textit{Statements} & t & ::= & \mathsf{return}\ e \\
 & & | & \mathsf{let}\ d\ \mathsf{in}\ t \\
 & & | & \mathsf{switch}\ e\ \mathsf{of} \\
 & & & i_1 : \lambda(x_{11} : \mu_{11}, \ldots, x_{1n_1} : \mu_{1n_1}).t_1, \\
 & & & \vdots, \\
 & & & i_m : \lambda(x_{m1} : \mu_{m1}, \ldots, x_{mn_m} : \mu_{mn_m}).t_m, \\
 & & & [\mathsf{default} : t] \\
 & & | & \mathsf{raise}\ e \\
 & & | & t_1\ \mathsf{handle}(x : \mathsf{Exn})\ t_2
\end{array}
$$

Figure 5.2: λ_i^{ML}-Norm: Statements and misc ops

becomes

```
let foo : (Int) → Int = ...
in let bar : (Int) → Int = ...
in let x : Int = ...
in let temp : Int = (return (bar (45))) handle(x : Exn) . return 21
in foo (temp)
```

Putting the examples together, consider the λ_i^{ML}-Rep code:

```
let foo : (Int, Int) → Int = ...
in let bar : (Int) → Int = ...
in let x : Int = ...
in foo (switch (x) of 1: 2, 2: 3, default: x - 1,
        (bar (45)) handle(x : Exn) . 21)
```

becomes

```
let foo : (Int, Int) → Int = ...
in let bar : (Int) → Int = ...
in let x : Int = ...
in let temp1 : Int = switch (x) of 1: return 2, 2: return 3, default: return (x - 1),
in let temp2 : Int = (bar (45)) handle(x : Exn) . 21
in foo (temp1, temp2)
```

Note here that the order of evaluation of the switch and handle are preserved.

Chapter 6

Closure Conversion

6.1 Introduction

A common feature of modern programming languages (including SML) is the provision of facilities for treating functions as *first-class objects* — meaning that they can be stored in data structures, passed as parameters, and so on, just as any other object in a program. Facilities for nesting of scope (functions within functions) are also often provided. While conventional languages sometimes provide either or both of these (for example, C (almost) provides the former but not the latter, Ada provides the latter but not the former, and Java provides neither), their combination in languages like SML causes difficulty in compilation, both to machine code, and to other more conventional languages.

Restrictions on functions are included in conventional languages in order to preclude the computation of *closures*. A closure is a pairing of a piece of code with an *environment*, mapping its free variables to values. Closures arise because the combination of first-class functions and nested scope can create situations where variables out-live their lexical scope. Consider the following SML program, for example:

```
val f = fn (x : int) =>
  let val g = fn (y : int) => x + y
  in g
val add1 = f 1
val add2 = f 2
val a = add1 2
val b = add2 2
```

Even when `f` returns in these cases, the variable `x` is still needed by `g`; `x` has outlived its scope. To deal with this, the invocation `f 1` returns a closure consisting of the code for `g`, together with a binding of `f`'s free variable `x` to 1, and the invocation `f 2` returns a closure consisting of the code for `g`, together with a binding of `f`'s free variable `x` to 2. The resulting functions `add1` and `add2` may be applied as if they had been declared by

```
val add1 =
  let val g = fn (y : int) => 1 + y
  in g
```

and

```
val add2 =
  let val g = fn (y : int) => 2 + y
  in g
```

`add1` and `add2` cannot simply be treated as pointers to code, as is the case in C (where there are no free variables to consider).

A key phase in the translation of high-level languages to lower-level languages or assembly, then, is that of *closure conversion*, where the representation of closures in the target language is chosen. In particular, the free variable environment for a function is made explicit (often as an additional parameter) and references to free variables are replaced by indices into the environment. A closure is then the result of applying a function to its environment, and is hence a pairing of code with data. This is similar to the concept of an *object*, and indeed the implementation of closures in Java is by an object, as described in Chapter 7.

6.2 Typed Closure Conversion

Typed closure conversion is a particular form of closure conversion in which types are preserved in the transformation [32]. This approach is particularly useful in the context of an SML-to-Java compiler, where type information must be preserved throughout compilation.

Minamide, Harper and Morrisett give a detailed account of typed closure conversion for both first- and second-order typed λ-calculi [32]. Their approach is employed in performing closure conversion on λ_i^{ML}, which is after all an enriched second-order λ-calculus. Since λ_i^{ML} is a second-order calculus, closure conversion must account for free *type* variables as well as free term variables; both type- and value-environments, mapping type- and term-level variables to types and values respectively, must be dealt with.

Closure conversion is implemented as a transformation from λ_i^{ML}-Norm to λ_i^{ML}-Close, in which closures, environments, and environment projection are introduced as primitive constructs of the calculus, as shown in Figures 6.1 to 6.3.

At the term level, λ_i^{ML}-Close possesses two forms for function code, vcode and tcode, corresponding to λ- and Λ-abstraction in the unconverted code, which abstract over the type and value environments. An expression

$$\mathsf{vcode}([\kappa_1, \ldots, \kappa_m], [\sigma_1, \ldots, \sigma_n], (x_1 : \mu_1, \ldots, x_o : \mu_o).t)$$

defines a function with constructor environment entries of kind κ_i, value environment entries of type σ_1, and ranging over variables x_i of type μ_i. The tcode expression is similar, save that it abstracts over type variables. An expression of form

$$\langle\!\langle e, [\mu_1, \ldots, \mu_n], [e_1, \ldots, e_m] \rangle\!\rangle$$

represents a closure over code expression e, partially applying it to its constructor environment $[\mu_1, \ldots, \mu_n]$ and value environment $[e_1, \ldots, e_m]$. An expression $\#\ i$ denotes the projection of the i^{th} component of the value environment. The let binding fixcode replaces the fix and fixtype constructs of λ_i^{ML}-Norm, allowing arbitrary expressions (and in particular, closure expressions) to be recursively bound to identifiers.

At the type level, types Vcode and Tcode are introduced, denoting the types of the vcode and tcode expressions. At the constructor level, forms for closed code (Code, abstracting over a constructor environment), projection from constructor environment ($\#\ i$), and constructor closure ($\langle\!\langle \mu, [\mu_1, \ldots, \mu_n] \rangle\!\rangle$, representing partial application of constructor code to its environment), are also introduced. The kind Code classifies Code constructors.

Note that at the expression level, a closure expression either has arrow or polymorphic types ($(\mu_1, \ldots, \mu_n) \to \mu$ or $\forall(\alpha_1 :: \kappa_1, \ldots, \alpha_n :: \kappa_n).\sigma$), depending on whether it closes over a value abstraction (an object of type Vcode) or a type abstraction (an object of type Tcode). Similarly, at the constructor level, a closure constructor has an arrow kind ($(\kappa_1, \ldots, \kappa_n) \to \kappa$).

The closure conversion algorithm (translating from λ_i^{ML}-Norm to λ_i^{ML}-Close) essentially follows that of Minamide, Harper and Morrisett. A recursive traversal of the λ_i^{ML}-Norm form is performed, collecting bindings for variables, much in the manner of a type checker. Four environments are maintained:

1. A mapping of non-local type variables currently in scope to their kinds $\Delta_{\mathbf{env}}$.

2. A mapping of local type variables currently in scope to their kinds $\Delta_{\mathbf{arg}}$.

3. A mapping of non-local term variables currently in scope to their types $\Gamma_{\mathbf{env}}$.

4. A mapping of local term variables currently in scope to their types $\Gamma_{\mathbf{arg}}$.

When a λ- or Λ-abstraction is encountered, it is replaced by a closure expression ($\langle\!\langle\cdot\rangle\!\rangle$), comprising a tcode or vcode expression (using the current mapping of type variables to kinds and term variables to types to abstract over type and value environments), and references to the current environment values for the environment component. Upon entry to a λ- or Λ-abstraction, the entries of the local environments are shifted into the non-local environments, and are replaced in the local environments by the formal parameters to the abstraction. As an optimization, and at the cost of some compile-time efficiency, the compiler only abstracts over variables which are free in the λ- or Λ-abstraction.

When a term or type variable is encountered, it is looked up first in the local term (resp. type) environment; if it is found, it is left unchanged. If it is not found, it is looked up in the non-local term (resp. type) environment, and replaced by a projection from that environment.

$$
\begin{array}{llcl}
\textit{Kinds} & \kappa & ::= & \Omega \\
& & \mid & (\kappa_1, \ldots, \kappa_n) \to \kappa \\
& & \mid & \langle \kappa_1 \times \cdots \times \kappa_n \rangle \\
& & \mid & \mathsf{Code}([\kappa'_1, \ldots, \kappa'_m], (\kappa_1, \ldots, \kappa_n) \to \kappa) \\
\\
\textit{Constructors} & \mu & ::= & \alpha \\
& & \mid & m.l \\
& & \mid & \mathsf{Int} \\
& & \mid & \mathsf{Real} \\
& & \mid & \mathsf{String} \\
& & \mid & \mathsf{Exn} \\
& & \mid & \mathsf{Enum}\ i \\
& & \mid & \mathsf{Array}\ \mu \\
& & \mid & (\mu_1, \ldots, \mu_n) \to \mu \\
& & \mid & (\mu_{11}, \ldots, \mu_{1n_1}) + \cdots + (\mu_{m1}, \ldots, \mu_{mn_m}) \\
& & \mid & \langle \mu_1 \times \cdots \times \mu_n \rangle \\
& & \mid & \mathsf{Excon}\ \mu \\
& & \mid & \mathsf{Deexcon}\ \mu \\
& & \mid & \mathsf{Wrap}\ \mu \\
& & \mid & \langle \mu_1, \ldots, \mu_n \rangle \\
& & \mid & \pi_i\ \mu \\
& & \mid & \mu\ (\mu_1, \ldots, \mu_n) \\
& & \mid & \mathsf{Rec}\ \alpha_1 = \mu_1, \ldots, \alpha_n = \mu_n\ \mathsf{In}\ \mu \\
& & \mid & \mathsf{Let}\ \alpha :: \kappa = \mu_1\ \mathsf{In}\ \mu_2 \\
& & \mid & \mathsf{Code}([\kappa'_1, \ldots, \kappa'_m], (\alpha_1 :: \kappa_1, \ldots, \alpha_m :: \kappa_n).\mu) \\
& & \mid & \#\ i \\
& & \mid & \langle\!\langle \mu, [\mu_1, \ldots, \mu_n] \rangle\!\rangle \\
\\
\textit{Types} & \sigma & ::= & \mu \\
& & \mid & \mathsf{Vcode}([\kappa_1, \ldots, \kappa_m], [\sigma_1, \ldots, \sigma_n], (\mu_1, \ldots, \mu_o) \to \mu) \\
& & \mid & \mathsf{Tcode}([\kappa'_1, \ldots, \kappa'_m], [\sigma_1, \ldots, \sigma_n], (\alpha_1 :: \kappa_1, \ldots, \alpha_o :: \kappa_o).\sigma) \\
& & \mid & \forall(\alpha_1 :: \kappa_1, \ldots, \alpha_n :: \kappa_n).\sigma \\
& & \mid & \mathsf{Export} \\
& & & \mathsf{Types}\ :\ l_{11} :: \kappa_1, \ldots, l_{1n} :: \kappa_n \\
& & & \mathsf{Values}\ :\ l_{21} : \sigma_1, \ldots, l_{2m} : \sigma_m
\end{array}
$$

Figure 6.1: λ_i^{ML}-Close: Kinds, constructors and types

$$
\begin{array}{llcl}
\textit{Declarations} & d & ::= & x : \sigma = e \\
 & & | & \alpha :: \kappa = \mu \\
 & & | & \mathsf{fixcode} \\
 & & & f_1 : \sigma_1 = e_1 \\
 & & & \vdots \\
 & & & f_n : \sigma_n = e_n \\
 \\
\textit{Misc ops} & m & ::= & e_1[e_2] \ := \ e_3 \\
 & & | & \mathsf{extern}\ s : \sigma \\
 & & | & \mathsf{newexn}\ \mu \\
 & & | & = \ (\mu, e_1, e_2) \\
 & & | & \neq \ (\mu, e_1, e_2) \\
 & & | & \mathsf{wrap}\ (\mu, e) \\
 & & | & \mathsf{unwrap}\ (\mu, e)
\end{array}
$$

Figure 6.2: λ_i^{ML}-Close: Declarations and misc ops

Note that no effort is made to perform further transformations in the manner of Minamide, Harper and Morrisett's 'closure representation' analysis. It is relatively straightforward to encode the primitive closure operations efficiently in Java (indeed, more straightforward than encoding the existential types used by Minamide, Harper and Morrisett in their closure representation), as described in Chapter 7.

As an example, consider the following SML code:

```
val f = fn (x : int) => fn y => x + y
val swap (x, y) = (y, x)
val inc = f 1
val x = inc 3
...
```

This is translated to the following λ_i^{ML}-Rep program

$$
\begin{array}{l}
\mathsf{let}\ f : (\mathsf{Int}) \to ((\mathsf{Int}) \to \mathsf{Int}) = \\
\quad \lambda(x : \mathsf{Int})\ .\ \lambda(y : \mathsf{Int})\ .\ +\ (x,\ y) \\
\mathsf{in\ let}\ \mathit{swap} : \forall(\alpha :: \Omega, \beta :: \Omega)\ .\ (\langle \alpha \times \beta \rangle) \to \langle \beta \times \alpha \rangle = \\
\quad \Lambda(\alpha :: \Omega, \beta :: \Omega)\ .\ \lambda(\mathit{v1} : \langle \alpha \times \beta \rangle)\ .\ \langle \pi_1\ \mathit{v1},\ \pi_0\ \mathit{v1} \rangle \\
\mathsf{in\ let}\ \mathit{inc} : (\mathsf{Int}) \to \mathsf{Int} = f\ (1) \\
\mathsf{in\ let}\ x : \mathsf{Int} = \mathit{inc}\ (3) \\
\mathsf{in}\ \ldots
\end{array}
$$

$$
\begin{array}{llll}
\textit{Expressions} & e & ::= & x \\
& & | & m.l \\
& & | & \langle e_1, \ldots, e_n \rangle \\
& & | & \mathsf{inject}_i^{\mu}\ (e_1, \ldots, e_n) \\
& & | & i \\
& & | & f \\
& & | & \mathsf{enum}_{\mu}\ i \\
& & | & s \\
& & | & \mathsf{vcode}([\kappa_1, \ldots, \kappa_m], [\sigma_1, \ldots, \sigma_n], (x_1 : \mu_1, \ldots, x_o : \mu_o).t) \\
& & | & \mathsf{tcode}([\kappa'_1, \ldots, \kappa'_m], [\sigma_1, \ldots, \sigma_n], (\alpha_1 :: \kappa_1, \ldots, \alpha_o :: \kappa_o).t) \\
& & | & e\ (e_1, \ldots, e_n) \\
& & | & e\ (\mu_1, \ldots, \mu_n) \\
& & | & \#\ i \\
& & | & \langle\!\langle e, [\mu_1, \ldots, \mu_m], [e_1, \ldots, e_n] \rangle\!\rangle \\
& & | & c\ e \\
& & | & op_1\ e \\
& & | & op_2\ (e_1, e_2) \\
& & | & m \\
\\
\textit{Statements} & t & ::= & \mathsf{return}\ e \\
& & | & \mathsf{let}\ d\ \mathsf{in}\ t \\
& & | & \mathsf{switch}\ e\ \mathsf{of} \\
& & & i_1 : \lambda(x_{11} : \mu_{11}, \ldots, x_{1n_1} : \mu_{1n_1}).t_1, \\
& & & \vdots \\
& & & i_m : \lambda(x_{m1} : \mu_{m1}, \ldots, x_{mn_m} : \mu_{mn_m}).t_m \\
& & & [\mathsf{default} :\ t] \\
& & | & \mathsf{raise}\ e \\
& & | & t_1\ \mathsf{handle}(x : \mathsf{Exn})\ t_2 \\
& & | & \mathsf{export} \\
& & & \mathsf{types} :\ l_{11} : \mu_1, \ldots, l_{1n} : \mu_n \\
& & & \mathsf{values} :\ l_{21} : e_1, \ldots, l_{2m} : e_m
\end{array}
$$

Figure 6.3: λ_i^{ML}-Close: Expressions and statements

After closure conversion (removing the wrapping and unwrapping coercions introduced by representation analysis for clarity), this becomes:

$\mathsf{let}\ f : (\mathsf{Int}) \to ((\mathsf{Int}) \to \mathsf{Int}) =$
$\quad \langle\!\langle \mathsf{vcode}([], [], \lambda(x : \mathsf{Int})\ .\ \langle\!\langle \mathsf{vcode}([], [\mathsf{Int}], \lambda(y : \mathsf{Int})\ .\ +(\#\ 0, y)), [], [x]\rangle\!\rangle),$
$\quad\quad [], []\rangle\!\rangle$
$\mathsf{in\ let}\ swap : \forall(\alpha :: \Omega, \beta :: \Omega)\ .\ (\langle \alpha \times \beta\rangle) \to \langle \beta \times \alpha\rangle =$
$\quad \langle\!\langle \mathsf{tcode}([], [], \Lambda(\alpha :: \Omega, \beta :: \Omega)\ .$
$\quad\quad \langle\!\langle \mathsf{vcode}([\Omega, \Omega], [], \lambda(v1 : \langle \#\ 0 \times \#\ 1\rangle)\ .\ \langle \pi_1\ v1, \pi_0\ v1\rangle), [\alpha, \beta], []\rangle\!\rangle),$
$\quad\quad [], []\rangle\!\rangle$
$\mathsf{in\ let}\ inc : (\mathsf{Int}) \to \mathsf{Int} = f\ (1)$
$\mathsf{in\ let}\ x : \mathsf{Int} = inc\ (3)$
$\mathsf{in}\ \ldots$

6.3 Hoisting

Having performed closure-conversion, the λ_i^{ML}-Close code has the convenient property that each piece of code is self-contained, independent of its enclosing scope. Code may therefore be *hoisted* to the top level of the program from its lexical position, its place being taken by a variable of the appropriate type. This transformation is useful in translating to languages like Java, since they often, in the first place, require that all code be named, and in the second, require that all code be declared at the top-level scope.

Hoisting is a relatively simple transformation. All vcode and tcode expressions are identified and named; they are then moved to the top-level of the program, where they are bound to their name in let declarations, and replaced in their position of appearance by the generated name.

To continue the example above, after closure hoisting the code would become

$\mathsf{let}\ code1 : \mathsf{Vcode}([\Omega, \Omega], [], (\langle \#\ 0 \times \#\ 1\rangle) \to \langle \#\ 1 \times \#\ 0\rangle) =$
$\quad \mathsf{vcode}([\Omega, \Omega], [], \lambda(v1 : \langle \#\ 0 \times \#\ 1\rangle).\ \langle\ \pi_1\ v1, \pi_0\ v0\ \rangle)$
$\mathsf{in\ let}\ code2 : \mathsf{Tcode}([], [], \forall(\alpha :: \Omega, \beta :: \Omega)\ .\ (\langle \alpha \times \beta\rangle) \to \langle \alpha \times \beta\rangle) =$
$\quad \mathsf{tcode}([], [], \Lambda(\alpha :: \Omega, \beta :: \Omega)\ .\ \langle\!\langle code1, [\alpha, \beta], []\rangle\!\rangle)$
$\mathsf{in\ let}\ code3 : \mathsf{Vcode}([], [\mathsf{Int}], (\mathsf{Int}) \to \mathsf{Int}) =$
$\quad \mathsf{vcode}([], [\mathsf{Int}], \lambda(y : \mathsf{Int}).+(\#\ 0, y))$
$\mathsf{in\ let}\ code4 : \mathsf{Vcode}([], [], (\mathsf{Int}) \to ((\mathsf{Int}) \to \mathsf{Int})) =$
$\quad \mathsf{vcode}([], [], \lambda(x : \mathsf{Int})\ .\ \langle\!\langle code3, [], [x]\rangle\!\rangle$
$\mathsf{in\ let}\ f : (\mathsf{Int}) \to ((\mathsf{Int}) \to \mathsf{Int}) = \langle\!\langle code4, [], []\rangle\!\rangle$
$\mathsf{in\ let}\ swap : \forall(\alpha :: \Omega, \beta :: \Omega)\ .\ (\langle \alpha \times \beta\rangle) \to \langle \beta \times \alpha\rangle = \langle\!\langle code2, [], []\rangle\!\rangle$
$\mathsf{in\ let}\ inc : (\mathsf{Int}) \to \mathsf{Int} = f\ (1)$
$\mathsf{in\ let}\ x : \mathsf{Int} = inc\ (3)$
$\mathsf{in}\ \ldots$

Chapter 7

Translation to Java

7.1 Introduction

Having dealt with the issues of representation analysis, A-normalization, and closure conversion, actual Java code can be generated from the λ_i^{ML}-Close intermediate representation.

The principal difference between λ_i^{ML}-Close and Java at this stage is the restrictive nature of Java's type system. While λ_i^{ML}-Close permits anonymous user-defined types, and admits structural equality of types, Java rigidly requires that all user-defined types be named (and moreover that all user-defined types be classes or interfaces), and that types are name-equivalent (with the usual subtyping rules). Java's type system makes life somewhat simpler in as much as all class definitions are visible to one another (that is, a Java compilation unit can be thought of as a set of mutually recursive class type definitions), so that definition of recursive types is straight-forward, and the order in which type definitions appear is immaterial.

7.2 Type Erasing

The first step in translating λ_i^{ML}-Close to Java is to perform a transformation to λ_i^{ML}-Box form. λ_i^{ML}-Box is a variant of λ_i^{ML}-Close with a considerably simplified type system. In particular, the distinction between constructors and types made in λ_i^{ML}-Close is eliminated; furthermore, since there is no type-based dispatch in λ_i^{ML}-Close (that is to say, no typecase or typerec construct in the style of Morrisett), type functions at the term level may be eliminated without changing the dynamic semantics of the program, and so tcode constructs are absent from λ_i^{ML}-Box.

Type abbreviations (introduced, for example, by Let and Rec constructors) and type variables (introduced by constructor-level λ abstraction, type-level $\forall$ abstraction, and term-level Λ abstraction (or rather, their equivalents after closure-conversion), which are essentially place-holders for unknown types), are also distinguished at this point. The representation of the former is known at compile-time, whereas a uniform representation must be adopted for the latter. As a consequence, all instances of type variables are replaced by a Box constructor (although perhaps Any may be more suggestive of the meaning).

As a final simplification, the representation of recursive types (introduced by the Rec constructor) is changed to be an SML reference to a type. As a consequence, an λ_i^{ML}-Close type such as

$$list \equiv \lambda(\alpha :: \Omega)\ .\ \mathsf{Rec}\ l = () + (\langle \alpha \times l \rangle)\ \mathsf{In}\ l$$

will be replaced by

$$\texttt{list} \equiv \texttt{ref}\ (() + (\langle \mathsf{Box} \times \texttt{list} \rangle))$$

in λ_i^{ML}-Box. This makes translation of recursive types considerably simpler.

7.3 Type Translation

Translation of λ_i^{ML}-Box to Java begins with a translation of the simplified λ_i^{ML}-Box types to Java types. The first phase of this translation creates a table of distinct types which appear in the λ_i^{ML}-Box program, and each non-trivial type is given a Java type name. In the second phase, Java code for each type is emitted, using the names assigned in the first phase. Each λ_i^{ML}-Box type is considered in turn.

7.3.1 Basic Types

The basic λ_i^{ML}-Box types (Int, Real, String) are translated to their direct counterparts in Java, namely `int`, `double`, and `String`.

7.3.2 Enumerated Types

The lack of subrange types in Java makes the translation of enumerated types somewhat less that optimal. The λ_i^{ML}-Box type Enum i is either translated as a Java `byte` type if $i > 2$, or as a Java `boolean` type if $i = 2$. Java `byte` types are rather inconvenient to use since there are no `byte` literals, and `byte` values can only be introduced by way of a cast from an `int`.

7.3.3 Record Types

Record types in λ_i^{ML}-Box (of form $\langle \mu_1, \ldots, \mu_n \rangle$) are translated as a Java class, comprising fields `_f₁` through `_fₙ` of the appropriate types, and a constructor.

For example, the λ_i^{ML}-Box type $\langle \mathsf{Int} \times \mathsf{Int} \rangle$ is translated to the following Java code:

```
class _record$_int_int$ {
    int _f1;
    int _f2;

    _record$_int_int$(int f1, int f2) {
        _f1 = f1;
        _f2 = f2;
    }
}
```

7.3.4 Sum Types

Sum types in λ_i^{ML}-Box of form $(\mu_{11}, \ldots, \mu_{1n_1}) + \cdots + (\mu_{m1}, \ldots, \mu_{mn_m})$ are compiled into $m+1$ classes: an abstract base class, comprising an **int _tag** field; and m sub-classes of the base classes, one for each variant of the sum. Each variant class's constructor sets the **_tag** field appropriately, and defines fields $\mathtt{\_f_1}$ through $\mathtt{\_f_{n_j}}$ of the appropriate types for variant j.

For example, the λ_i^{ML}-Box type $(\mathsf{Int}) + (\langle \mathsf{Int} \times \mathsf{Int} \rangle)$ would be translated to the following Java code (assuming that type $\langle \mathsf{Int} \times \mathsf{Int} \rangle$ was given the Java class name **_record$_int_int$**):

```
abstract class _sum$_int$$_record$_int_int$$ {
    int _tag;
}

class _sum$_int$$_record$_int_int$$_1
    extends _sum$_int$$_record$_int_int$$ {
    int _f1;

    _sum$_int$$_record$_int_int$$_0(int f1) {
        _tag = 1;
        _f1 = f1;
    }
}

class _sum$_int$$_record$_int_int$$_2
    extends _sum$_int$$_record$_int_int$$ {
    _record$_int_int$ _f1;

    _sum$_int$$_record$_int_int$$_0(_record$_int_int$ f1) {
        _tag = 2;
        _f1 = f1;
    }
}
```

7.3.5 Arrow Types

An arrow type in λ_i^{ML}-Box of form $(\mu_1, \ldots, \mu_n) \rightarrow \mu$ is represented in Java as an abstract class, defining the abstract method `invoke` with the appropriate parameter and return types. Code instances of a particular type are defined as subclasses of the abstract arrow type, as will be seen below.

For example, the λ_i^{ML}-Box type $(\mathsf{Int}, \mathsf{String}) \rightarrow \mathsf{Int}$ would be translated to the following Java code:

```
abstract class _arrow$_int_string$_int {
    abstract int invoke(int _arg1, String _arg2);
}
```

7.3.6 Type Variables

Type variables (i.e., types which are unknown at compile-type), which appear as Box in λ_i^{ML}-Box, are represented by type `SMLBox` in Java. Any type which could be an instantiation of a type variable is therefore constrained to be a subclass of `SMLBox`. Note that the coercions introduced by the representation analysis phase of compilation will guarantee that type variables will only be instantiated with boxed types (subtypes of `SMLBox`).

It may seem more sensible to use the predefined type `Object` to represent type variables; however, as will be seen, it was desirable to add a number of abstract methods to the `SMLBox` class.

For reasons which are made clear in section 7.3.7, `SMLBox` is defined as a subclass of `Exception`.

7.3.7 Exception Types

An exception declaration in SML gives rise to four types in λ_i^{ML}-Box: the exception packet type, the exception constructor type, the exception deconstructor type, and a constructor and deconstructor pair. The translation from λ_i^{ML}-Box to Java must account for all of these types, in such a way as to allow the Java exception system to be used.

The exception packet type seems as though it should be simply a subclass of `Exception`, so that they may be thrown by Java programs. However, since exception packets can be instantiations of type variables, they must be subclasses of `SMLBox` (see section 7.3.6 above). Exception packets must therefore be subclasses of both `SMLBox` and of `Exception`. Note however that Java only permits single inheritance, so either `SMLBox` must be as a subclass of `Exception`, or vice versa.

Since `Exception` is a 'core' Java class, its position in the class hierarchy cannot be changed — in particular, it cannot be made a subclass of `SMLBox`. Thus, the only alternative is to make `SMLBox` a subclass of `Exception`, and all exception packets subclasses of `SMLBox`. In this way, exceptions can at once be instantiations of type variables, and throwable exceptions. The only peculiarity of this approach is that any SML type is throwable.

The treatment of exception constructor and deconstructor types is somewhat tricky. First, `excon_interface` and `deexcon_interface` interfaces defining methods `_excon`, and `_deexcon` respectively are defined. A record type with fields `_f1` of type `excon_interface`, and `_f2` of type `deexcon_interface`, and which implements the `excon_interface` and `deexcon_interface` interfaces (i.e., which defines the `_excon` and `_deexcon` methods), is then defined. The constructor for the record assigns the `_f1` and `_f2` fields to itself, thereby creating in a single allocation a structure which is the pair of constructor and deconstructor, *and* the constructor and deconstructor themselves.

The excon and deexcon operations are then translated simply as invocation of the `_excon` and `_deexcon` methods of the record respectively.

For example, the λ_i^{ML}-Box code:

$$\mathsf{let}\ x : \langle \mathsf{Excon\ Int} \times \mathsf{Deexcon\ Int} \rangle = \mathsf{newexn\ Int}$$
$$\mathsf{in}\ldots$$

generates the following Java code (assuming that `_sum$$_int$` is the Java name for λ_i^{ML}-Box type $() + (\mathsf{Int})$):

```
interface _excon_int {
    abstract SMLExn excon(int data);
}

interface _deexcon_int {
    abstract _sum$$_int$ deexcon(SMLExn exn);
}

class _exn_x extends SMLExn {
    int _data;

    _exn_x(int data) {
        _data = data;
    }
}

class _con_x implements _excon_int, _deexcon_int {
    _excon_int _f1;
    _deexcon_int _f2;

    SMLExn excon(int data) {
        return new _exn_x(data);
```

```
        }

        _sum$$_int$ deexcon(SMLExn exn) {
            if (exn instanceof _exn_x) {
                return new _sum$$_int$_2(((_exn_x) exn)._data);
            } else {
                return new _sum$$_int$_1();
            }
        }

        _con_x() {
            _f1 = this;
            _f2 = this;
        }
    }

    _con_x x = new _con_x();
```

7.3.8 Recursive Types

As noted above, recursive types are represented in λ_i^{ML}-Box by an SML reference to an λ_i^{ML}-Box type. Translation for a reference type therefore simply requires translation of the referred-to type (although care must be taken to avoid infinitely translating referred-to types).

7.4 Code Translation

In the closure conversion phase of the compilation (Chapter 6), all code was

- Closed with respect to its free variables, and
- Hoisted to the top-level and named.

Code emission in Java then involves identifying the top-level declarations which correspond to code segments, and emitting a closure class. The closure class consists of fields `_ei` to for each free variable in the enclosing environment, and a single `invoke` method. The class is defined as a subclass of the appropriate abstract arrow type (section 7.3.5 above).

Individual expressions can be translated almost directly from λ_i^{ML}-Box, thanks to the control flow analysis performed earlier:

- λ_i^{ML}-Box variable references (x) are translated to the corresponding Java variable. λ_i^{ML}-Box variable names are suitably altered so that they represent valid Java identifiers.

- Injection into sum types ($\mathsf{inject}_i^{\mu}\ (e_1, \ldots, e_n)$) is translated as a class instance allocation for variant i of the sum type.

- Integer, real, and string expressions are translated to integer, double, and string literals in Java.

- Enumerated constants of type Enum i are translated as either Java byte constants (which must unfortunately be introduced indirectly, by a cast from an integer literal), or as boolean constants if $i = 2$.

- Record creation ($\langle e_1, \ldots, e_n \rangle$) is translated as a class instance allocation for the appropriate record class type.

- Function invocation ($e\ (e_1, \ldots, e_n)$) is translated as a invocation of the `invoke` method of the operand with the corresponding arguments. If e is an external function (i.e., a Java API function) it is invoked directly instead of through the `invoke` method.

- Unary and binary expressions are translated to their direct equivalents in Java where they exist; some give rise to calls to methods in the `java.math` library, and certain others had to be hand-coded in Java in order to detect overflow and other exceptional conditions, as required by SML. Record selection is translated as accessing the appropriate field of the record object.

- Environment projection is translated as a field access to the appropriate environment field.

- λ_i^{ML}-Box's closure expression ($\langle\!\langle \cdot \rangle\!\rangle$) gives rise to a class instance allocation of the closure class referenced in the operation, with the appropriate environment arguments to the constructor. Note that hoisting guarantees that the code segment named in the closure operation will be named.

Translation of statements is similarly straight-forward:

- Translation of the λ_i^{ML}-Box return statement depends on the current context. When translating a let binding, the return is translated as assignment to the bound variable. When translating a function, the return is translated as a Java `return` statement.

- Let declaration translation is somewhat involved, particularly in the case of recursive function definitions. The problem that arises is essentially that a recursive environment structure must be built, so that all of the functions in the fix can refer to each other. This is achieved by first allocating the closures for each of the functions in the fix, with dummy (`null`) arguments for the environment entries of functions in the fix. Then, when all of the closures have been allocated, they are updated by assigning the allocated closures to the environment slots for each defined function. The example at the end of this section clarifies this procedure.

- Translation of the switch construct in λ_i^{ML}-Box to Java's `switch` statement causes essentially the same difficulty as translation from the Lambda switch construct to the λ_i^{ML}-Box switch save in reverse: that is, λ_i^{ML}-Box's switch construct implicitly decomposes its argument when switching on sum types, whereas neither Java nor Lambda do so.

  Translation of integer-indexed switches is straight-forward, since no decomposition is performed.

  Translation of enumeration-indexed switches depends on the type of the argument. If the argument is of type Enum 2, the switch is operating over a `boolean` type, and so is translated as an `if-then-else` construct in Java. Otherwise, the switch is operating over a Java `byte` type. In this case, Java's lack of enumerated and subrange types hinders the translation again, by requiring that the switch include a default case even though the switch may be provably exhaustive.

  Translation of sum-indexed switches requires that decomposition of the argument be performed. The switch arm is selected by inspection of the `_tag` field of the sum type. Each arm is then translated with a preamble that assigns variables to the fields of the appropriate sum arm. Again, since Java does not possess enumeration or subrange types, a default case must always be included, which simply invokes the `fatal` method of the pre-defined `General` class, signaling an internal compiler error.

- The raise statement of λ_i^{ML}-Box is translated as a `throw` statement in Java.

- The handle statement of λ_i^{ML}-Box is straight-forward to translate, since Java provides a mechanism for binding the handled exception to an identifier, in the guise of the `try-catch` construct, much in the manner of λ_i^{ML}-Box.

The following SML code illustrates more clearly how mutually recursive functions are compiled to Java:

```
fun even 0 = true
  | even x = odd (x - 1)
and odd 0 = false
  | odd x = even (x - 1);
val x = even 43;
```

The Java translation would look as follows:

```
abstract class _arrow$_int$_bool {
    abstract boolean invoke(int arg);
}

class even_closure extends _arrow$_int$_boolean {
```

```
    _arrow$_int$_boolean _even;
    _arrow$_int$_boolean _odd;

    fact_closure(_arrow$_int$_boolean even, _arrow$_int$_boolean odd) {
        _even = even;
        _odd = odd;
    }

    int invoke(int x) {
        switch (x) {
            case 0:
                return true;
            default:
                return _odd.invoke(x - 1);
        }
    }
}

class odd_closure extends _arrow$_int$_boolean {
    _arrow$_int$_boolean _even;
    _arrow$_int$_boolean _odd;

    fact_closure(_arrow$_int$_boolean even, _arrow$_int$_boolean odd) {
        _even = even;
        _odd = odd;
    }

    int invoke(int x) {
        switch (x) {
            case 0:
                return false;
            default:
                return _even.invoke(x - 1);
        }
    }
}

class main {
    public static void Main(String args[]) {
        even = new even_closure(null, null);
        odd = new odd_closure(null, null);
```

```
            even._even = even;
            even._odd = odd;
            odd._even = even;
            odd._odd = odd;
            boolean x;

            x = even.invoke(43);
        }
    }
```

7.5 Polymorphic Equality

One issue that has proved troublesome for past implementations of SML is that of polymorphic equality [5]. SML allows certain non-trivial types to be compared for equality — in particular, datatypes and record types whose fields are themselves equality types may be compared, just like the basic types `int`, `real`[1], and `string`. Adding to the complexity, objects of unknown type (that is, whose type is a type variable), but which are known to be equality types, may also be compared.

A number of possible approaches exist to deal with this problem. That adopted by the ML Kit is to generate `Lambda` code for comparing each equality type defined by the user [11]. This code must then of course be translated into Java in later stages, at a considerable cost both to the efficiency of the compiler, and the legibility of the intermediate code. In keeping with the type-passing approach, Morrisett suggests a type-passing equality function (essentially a function eq of type $\forall(\alpha :: \kappa).(\alpha, \alpha) \rightarrow \mathsf{bool}$) which inspects its type parameter at run-time to dispatch to the appropriate code to perform the actual (possibly recursive) comparison [33]. This approach is attractive, but requires more machinery in the intermediate form (in particular, a `typecase` or `typerec` construct) than was provided. Instead, the issue of polymorphic equality was dealt with at the Java translation stage of compilation.

The idea essentially is that every class representing a SML type (i.e., every subclass of `SMLBox`) must define an `equals` method, taking a single `SMLBox` argument; this is enforced by making `equals` an abstract method of `SMLBox`. The SML-to-Java compiler guarantees that the argument to the `equals` method will always be the same type as the object on which the method was invoked; however, it is a classic deficiency of object-oriented type systems lacking self-types that this restriction cannot be specified in Java [1, 2]. The penalty for the lack of this feature is that a run-time cast will be necessary to get an object of the appropriate type for comparison.

For record types, the `equals` method is recursively invoked for each field of the record. For sum types, the `equals` method checks by way of an `instanceOf` test whether the parameter

[1] Although it is not clear whether it is desirable to compare `real` values for equality.

passed is the same variant as the method's class — if it is, then a further recursive field-by-field analysis is performed, otherwise `false` can be returned immediately. Exception and function types in SML do not admit equality, and so their `equals` methods should never be invoked — as a safeguard, they will always invoke the `fatal` method of the `General` class.

While it is perhaps desirable to move the code for polymorphic equality to an earlier stage of compilation, so that for example it may be exposed for optimization, it was felt that the additional machinery required to do so would have added considerable complexity to the compiler for comparatively little gain.

7.6 Wrapping and Unwrapping

As noted in Chapter 4, representation analysis in the SML-to-Java compiler is dealt with in two phases. In the first phase, `wrap` and `unwrap` coercions are inserted into the code at the appropriate points; in the second, `wrap` and `unwrap` are elaborated into their actual definition, in terms of boxing and unboxing.

In his presentation, Shao makes use of a `typecase` construct to define the wrapping and unwrapping transformations for each of his boxing schemes (full boxing, partial boxing, and simple boxing) [38]. Since λ_i^{ML} does not include a `typecase` construct, definition of the wrapping and unwrapping transformations is deferred until the point of translation to Java. At that stage, virtual method invocation can be used to achieve essentially the same effect.

For reasons of simplicity, full boxing approach was implemented. Although partial boxing would perhaps be more efficient in execution time, it requires that wrapping and unwrapping be defined in terms of '(un)boxing' and '(un)covering' operations. Full boxing by contrast defines the (un)wrapping transformation by the recursive application of '(un)boxing' only. To some extent then it is unclear whether the additional benefit of partial boxing is merited, given that it may require addition virtual method invocations on each wrapping operation.

Full boxing is implemented in Java by adding two methods to each subclass of `SMLBox`, namely the method `wrap`, which transforms an object (by recursive invocation of `wrap` if necessary) into its fully boxed form, and the static method `unwrap`, which unwraps an `SMLBox` object into the type at which the method is defined (and which is therefore basically a specialized Java constructor for the type).

Since the fully boxed form of a class is of a different type from the class itself, at type extraction time a 'boxed' form for each type must also be defined. This is done by replacing each field of a record, sum or exception packet type with `SMLBox`, so that each component of the type is represented by a heap-allocated object.

The wrapping operation performs recursive full boxing on the object, creating an instance of its 'boxed' counterpart. Each field of a record, sum or exception packet will be recursively wrapped (by invocation of the 'wrap' method) and passed to the constructor of the 'boxed' type. Unwrapping reverses this procedure; type casts are required to extract the actual types from the `SMLBox` types to which they are coerced by wrapping.

The full Java translation of the λ_i^{ML}-Box type $\langle \mathsf{Int} \times \mathsf{Int} \rangle$ is therefore as follows:

```
class _record$_box_box$ {
    SMLBox _f1;
    SMLBox _f2;

    _record$_box_box$(SMLBox f1, SMLBox f2) {
        _f1 = f1;
        _f2 = f2;
    }

    SMLBox wrap() {
        return this;
    }

    static _record$_box_box$ unwrap(SMLBox x) {
        return (_record$_box_box$) x;
    }

    boolean equals(SMLBox x) {
        _record$_box_box$ x_cast = (_record$_box_box$) x;
        return _f1.equals(x_cast._f1) && _f2.equals(x_cast._f2);
    }
}

class _record$_int_int$ {
    int _f1;
    int _f2;

    _record$_int_int$(int f1, int f2) {
        _f1 = f1;
        _f2 = f2;
    }

    SMLBox wrap() {
        return new _record$_box_box$(new SMLBoxedInt(_f1),
                                     new SMLBoxedInt(_f2));
    }

    static _record$_int_int$ unwrap(SMLBox x) {
        _record$_box_box$ x_cast = (_record$_box_box$) x;
```

```
        return new _record$_int_int$(SMLInt.unwrap(x_cast._f1),
                                     SMLInt.unwrap(x_cast._f2));
    }

    boolean equals(SMLBox x) {
        _record$_int_int$ x_cast = (_record$_int_int$) x;

        return (_f1 == x_cast._f1) && (_f2 == x_cast._f2);
    }
```

Chapter 8

Summary, Future Work and Conclusion

8.1 Summary

The emergence of the Java language and Virtual Machine have created a unique opportunity for the development of truly portable code. Exploiting the potential of these technologies for the moment depends on use of the Java language, unfortunately ignoring the great wealth of programming language research of the last twenty years. This document presents an alternative, where the Java language and JVM are used a vehicle for the deployment of Standard ML — a potentially superior platform for Internet and web development.

This compiler also demonstrates how some of the recent developments in the functional language community, such as type-directed compilation, representation analysis, *A*-normalization, and typed closure conversion, can be fruitfully applied to realistic compiler implementation.

8.2 Future Work

The compiler described in this document can, however, only be viewed as a first step in the development of a true SML-to-JVM compiler. In particular, a number of points require further attention:

1. Most importantly, the compiler described implements only the core-SML language, and does not address the compilation of modules. Module compilation to the JVM raises a number of important issues, especially where the naming of types is concerned. The development of a proper module compiler is also probably a necessary first step to developing a type-safe interface to Java APIs (so that, for example, signatures for Java packages can be defined).

2. The issue of tail-recursion elimination has not been addressed. There has been some suggestion that future versions of the JVM will include constructs for proper tail-recursion (i.e., recursion without stack growth) but since languages like C and Java typically do not encourage a recursive programming style, it seems unlikely that there will be much pressure from industry to incorporate this feature. It seems sensible therefore to investigate other strategies, using the resources available — for example, the Kawa Scheme-to-JVM compiler uses the JVM local `goto` instruction to perform limited tail-recursion elimination [12]; this strategy could perhaps be employed in the SML-to-Java compiler, although it would require re-targeting the compiler to emit JVM bytecode rather than Java. Other less elegant and efficient solutions which do not involve the JVM (for example, using labelled loops) are also conceivable.

3. The SML-to-Java compiler currently performs only a few modest optimizations, and hence suffers from fairly poor performance. Efficient compilation of SML requires considerably more aggressive optimization to be competitive with other languages; and there is now a large body of knowledge on SML optimization (see for example Tarditi's thesis [44]) from which to draw.

8.3 Conclusion

Experience with Java as an intermediate form seems to suggest that although typed languages offer safety and reliability, they present considerable, and to some extent artificial, obstacles to compilation. In particular, the compilation of polymorphism to Java was highly troublesome, due largely to the restrictive nature of Java's type system. The lack of features like enumerated types and variable-argument functions also made compilation to Java more difficult than necessary.

Despite its limitations, and despite the difficulties posed by Java, the SML-to-Java compiler serves to demonstrate at once the feasibility of compiling high-level languages to the JVM, and the utility of advanced compilation techniques.

Bibliography

[1] Martín Abadi and Luca Cardelli. *A Theory of Objects.* Monographs in Computer Science. Springer-Verlag, 1996.

[2] Martín Abadi and Luca Cardelli. A theory of primitive objects: Untyped and first-order systems. *Information and Computation*, 125(2):78 – 102, March 1996.

[3] Alfred V. Aho, Ravi Sethi, and Jeffrey D. Ullman. *Compilers: Principless, Techniques, and Tools.* Addison-Wesley, 1988.

[4] Andrew W. Appel. *Compiling with Continuations.* Cambridge University Press, 1992.

[5] Andrew W. Appel. A critique of Standard ML. *Journal of Functional Programming*, 3(4):391 – 429, October 1993.

[6] Andrew W. Appel and Trevor Jim. Continuation-passing, closure-passing style. In *Sixteenth ACM Symposium on Principles of Programming Languages*, January 1989.

[7] Andrew W. Appel and Trevor Jim. Making lambda calculus smaller, faster. Technical Report CS-TR-477-94, Princeton University, November 1994.

[8] Andrew W. Appel and David B. MacQueen. Standard ML of New Jersey. Technical Report CS-TR-329-91, Princeton University, June 1991.

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

[10] Joel F. Bartlett. Scheme → C a portable Scheme-to-C compiler. Technical Report WRL Research Report 89/1, Digital Western Research Laboratory, January 1989.

[11] Lars Birkedal, Nick Rothwell, Mads Tofte, and David N. Turner. The ML Kit version 1. Technical Report DIKU-TR-93-14, University of Copenhagen, March 1993.

[12] Per Bothner. Kawa: Compiling Scheme to Java. available at http://www.cygnus.com/~bothner/kawa.html.

[13] Luca Cardelli and Peter Wegner. On understanding types, data abstraction, and polymorphism. *ACM Computing Surveys*, 17(4):471–522, 1985.

[14] Martin Elsman. A portable Standard ML implementation. Master's thesis, The Technical University of Denmark, 1994.

[15] S. I. Feldman, David M. Gay, Mark W. Maimone, and N. L. Schryer. A Fortran-to-C converter. Technical Report Computing Science Technical Report No. 149, AT&T Bell Laboratories, March 1995.

[16] Cormac Flanagan, Amr Sabry, Bruce F. Duba, and Matthias Felleisen. The essence of compiling with continuations. In *ACM SIGPLAN Conference on Programming Language Design and Implementation*, pages 237 – 247, June 1993.

[17] Dave Gillespie. p2c: the Pascal to C translator, version 1.20. available at ftp://csvax.cs.caltech.edu/pub/p2c-1.20.tar.Z.

[18] Jean-Yves Girard, Yves Lafont, and Paul Taylor. *Proofs and Types.* Number 7 in Cambridge Tracts in Theoretical Computer Science. Cambridge University Press, 1989.

[19] James Gosling, Bill Joy, and Guy Steele. *The Java Language Specification.* Addison-Wesley, 1996.

[20] Robert Harper and John C. Mitchell. On the type structure of Standard ML. *ACM Transactions on Programming Languages and Systems*, 15(2):211–252, November 1993.

[21] Robert Harper and Christopher Stone. An interpretation of Standard ML in type theory. Technical Report CMU-CS-97-147, Carnegie Mellon University, June 1997.

[22] S. L. Peyton Jones and J. Launchbury. *Unboxed values as first class citizens*, pages 636 – 666. Number 523 in Lecture Notes in Computer Science. Springer Verlag, September 1991.

[23] Simon L. Peyton Jones, Cordy Hall, Kevin Hammond, Will Partain, and Phil Walder. The Glasgow Haskell compiler: a technical overview. In *Proceedings of the UK Joint Framework for Information Technology (JFIT) Technical Conference*, 1993.

[24] David Kranz, Richard Kelsey, Jonathan Rees, Paul Hudak, James Philbin, and Norman Adams. ORBIT: An optimizing compiler for Scheme. In *Proceedings of the ACM SIGPLAN '86 Symposium on Compiler Construction*, pages 219–233, June 1986.

[25] Xavier Leroy. Efficient data representation in polymorphic languages. Research report 1264, INRIA, 1990.

[26] Xavier Leroy. Unboxed objects and polymorphic typing. In *Proceedings 19th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, pages 177–188, January 1992.

[27] Xavier Leroy. The effectiveness of type-based unboxing. In *Proceedings of Workshop on Types in Compilation*, June 1997.

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

[29] Robin Milner and Mads Tofte. *Commentary on Standard ML.* MIT Press, 1990.

[30] Robin Milner, Mads Tofte, and Robert Harper. *The Definition of Standard ML.* MIT Press, 1990.

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

[32] Yasuhiko Minamide, Greg Morrisett, and Robert Harper. Typed closure conversion. In *Proceedings 23rd ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, January 1996.

[33] Greg Morrisett. *Compiling with Types.* PhD thesis, Carnegie Mellon University, 1995.

[34] R. Morrison, A. Dearle, R. C. H. Connor, and A. L. Brown. An ad hoc approach to the implementation of polymorphism. *ACM Transactions on Programming Languages and Systems*, 13(3):342 – 371, July 1991.

[35] Martin Odersky and Philip Walder. Pizza into Java: Translating theory into practice. In *Proceedings 24th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, pages 146 – 159, January 1997.

[36] L. C. Paulson. *ML for the Working Programmer.* Cambridge University Press, second edition, 1996.

[37] Jean E. Sammet. From HOPL to HOPL-II (1978-1993): 15 years of programming language development. In Thomas J. Bergin and Richard G. Gibson, editors, *History of Programming Languages*, pages 16 – 23. Addison-Wesley, 1996.

[38] Zhong Shao. Flexible representation analysis. In *Proceedings of the 1997 ACM SIGPLAN Conference on Functional Programming*, 1997.

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

[40] Zhong Shao and Andrew W. Appel. A type-based compiler for Standard ML. In *Proceedings of the 1995 ACM SIGPLAN Conference on Programming Language Design and Implementation*, pages 116–129, June 1995.

[41] Bjarne Stroustrup. *The C++ Programming Language.* Addison Wesley, second edition, 1994.

[42] S. Tucker Taft. *Programming the Internet in Ada 95.* Intermetrics, Inc., March 1996.

[43] D. Tarditi, G. Morrisett, C. Stone, R. Harper, and P. Lee. TIL: A type-directed optimizing compiler for ML. In *Proceedings of the 1996 ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI)*, pages 181–192, May 1996.

[44] David Tarditi. *Design and Implementation of Code Optimizations for a Type-Directed Compiler for Standard ML.* PhD thesis, Carnegie Mellon University, 1996.

[45] David Tarditi, Anurag Acharya, and Peter Lee. No assembly required: Compiling Standard ML to C. Technical Report CMU-CS-90-187, Carnegie Mellon University, November 1990.

[46] Mads Tofte. Type inference for polymorphic references. *Information and Computation*, 89(1):1 – 34, November 1990.

[47] Mads Tofte, Lars Birkedal, Martin Elsman, Niels Hallenberg, Tommy Højfeld Olesen, Peter Sestoft, and Peter Bertelsen. Programming with regions in the ML Kit. Technical Report DIKU-TR-97-12, University of Copenhagen, April 1997.

[48] Andrew Tolmach. Tag-free garbage collection using explicit type parameters. In *ACM Conference on Lisp and Functional Programming*, pages 1–11, June 1994.

[49] Jeffrey D. Ullman. *Elements of ML Programming.* Prentice Hall, 1994.

[50] A. van Wijngaarden, B. J. Mailloux, J. E. L. Peck, and C. H. A Koster. *Report on the Algorithmic Language ALGOL 68.* Mathematisch Centrum, 1969.

[51] A. van Wijngaarden, B. J. Mailloux, J. E. L. Peck, C. H. A. Koster, M. Sintzoff, C. H. Lindsey, L. G. T. L. Meertens, and R. G. Fisker, editors. *Revised Report on the Algorithmic Language ALGOL 68.* Number 50 in Mathematical Centre Tracts. Mathematisch Centrum, 1976.

[52] David A. Watt, Brian A. Wichmann, and William Findlay. *ADA Language and Methodology.* Prentice-Hall, 1987.

[53] Andrew K. Wright. Polymorphism for imperative languages without imperative types. Technical Report TR94-200, Rice University, February 1993.