CHAPTER ONE

OVERVIEW

ML is a strict higher-order functional programming language with statically checked polymorphic types, garbage collection, and a complete formally defined semantics. *Standard ML of New Jersey* is an optimizing compiler and runtime system for ML. The intermediate representation that SML/NJ uses for optimization and code generation—*continuation-passing style*—is applicable to the compilation of many modern programming languages, not just ML. This book is about compiling with continuation-passing style.

Prior knowledge of ML is helpful, but not necessary, to read this book. Since the Standard ML of New Jersey compiler is itself written in ML, we use ML notation for many of the examples. But will we use only a simple subset of the language for illustrations, and we will explain the notation as we go along. Readers completely unfamiliar with ML should refer to the introduction in Appendix A.

1.1 Continuation-passing style

The beauty of FORTRAN—and the reason it was an improvement over assembly language—is that it relieves the programmer of the obligation to make up names for intermediate results. For example, we write $x = (a + b) * (c + d)$ instead of the assembly language:

$$\begin{aligned} r_1 &\leftarrow a + b \\ r_2 &\leftarrow c + d \\ x &\leftarrow r_1 \times r_2 \end{aligned}$$

The simple, one-line expression is easier to read and understand than the three-line assembly program; the names r_1 and r_2 don't really aid our understanding. Furthermore, the assembly program spells out explicitly the order of evaluation: $a + b$ is computed before $c + d$; this is something we did not really need to know.

The λ-calculus gives us these same advantages for functional values as well. We can write $f(\lambda x.x + 1)$ instead of (as in Pascal):

```
function g(x: integer): integer;
  begin g := x+1 end;
. . . f(g) . . .
```

Here, Pascal forces us to give a name (g) to this function; the name (and the consequent verbosity of the definition) may not help us understand what's going on. The λ-calculus also frees us from needing to know too much about order of evaluation, even across function-call boundaries.

These conveniences are quite appropriate for humans writing programs. But in fact, they may be just the wrong thing for compilers manipulating programs. A compiler might like to give a distinct name to *every* intermediate value of the program, so it can use these names for table lookups, manipulating sets of values, register allocation, scheduling, and so on.

Continuation-passing style (CPS) is a program notation that makes every aspect of control flow and data flow explicit. It also has the advantage that it's closely related to Church's λ-calculus, which has a well-defined and well-understood meaning.

We can illustrate CPS informally with an example. Start with a source program

```
fun prodprimes(n) =
   if n=1
    then 1
    else if isprime(n) then n*prodprimes(n-1)
                       else prodprimes(n-1)
```

This is an ML program that computes the product of all primes less than or equal to a positive integer n. The keyword `fun` introduces a function definition; the expression on the right-hand side of the equal sign is the body of the function. The `if-then-else` and arithmetic expression notation should be familiar to most readers.

Now, there are several points in the control flow of this program that deserve names. For example, when the function `isprime` is called, it will be passed a *return address*. It is helpful to name this return address—let's call it k. And `isprime` will return a Boolean value—call it b. The first call to `prodprimes` (in the `then` clause of the second `if`) will return to a point j with an integer p, but the second call to `prodprimes` (in the `else` clause) will return to a point h with an integer q. The first computation of $n-1$ will be put in a temporary variable m, and the second one in a variable i, and so on.

We should also mention that the function `prodprimes`, when it is called, is handed a return address; we can call that c and treat it as one of the arguments (formal parameters) of the function. Then we can use c when it is time to leave the function.

We can express all of this using *continuations*. A continuation is a function that expresses "what to do next;" for example, we can say that `prodprimes` is given a continuation c as one of its arguments, and when `prodprimes` has computed its result a it will continue by applying c to a. Thus, returning from a function looks just like a function call!

The following program is a continuation-passing-style version of the program, written in ML. For those unfamiliar with ML, it will help to explain that `let` *declaration* `in` *expression* `end` declares some local value (a function, an integer,

etc.) whose scope includes the *expression*; the result of the `let` is just that of the *expression*. A `fun` declaration declares a function (with formal parameters) and a `val` declaration (in this simple example) binds a variable to a value.

```
fun prodprimes(n,c) =
   if n=1
   then c(1)
   else let fun k(b) =
            if b=true
            then let fun j(p)=
                       let val a=n*p
                        in c(a)
                       end
                     val m=n-1
                  in prodprimes(m,j)
                 end
            else let fun h(q)=c(q)
                     val i=n-1
                  in prodprimes(i,h)
                 end
        in isprime(n,k)
       end
```

Observe that all the control points c, k, j, h discussed above are just continuation functions, and all the data labels b, p, a, m, q, i are just variables. In order to use continuation-passing style, we didn't have to change our notation very much; we just used a restricted form of the existing language. A full explanation of CPS will be given in Chapters 2 and 3.

The CPS representation is easy for optimizing compilers to manipulate and transform. For example, we would like to perform *tail-recursion elimination*: If a function f calls a function g as the very last thing it does, then instead of passing g a return address within f, it could pass to g the return address that f was given by f's caller. Then, when g returned, it would return directly to the caller of f.

If we look at the original version of `prodprimes`, we find that one of the calls is tail recursive (the last recursive call to `prodprimes`). In the CPS version of the program, this is manifested in the fact that the continuation function h is trivial: $h(q) = c(q)$. Now, whenever we have a function h that just calls another function c with the same argument, we can say that h is equivalent to c; and we might as well use c wherever h is referred to. So we can perform this simple transformation

```
let fun h(q)=c(q)
    val i=n-1
 in prodprimes(i,h)
end
```

→

```
let val i=n-1
 in prodprimes(i,c)
end
```

and now we have accomplished *tail-recursion elimination* in a clean way.

1.2 Advantages of CPS

There are many reasons to use continuation-passing style as a framework for compiling and optimization. In this discussion we will compare it to several alternatives:[1]

λ The lambda calculus (without explicit continuations); this might seem like an appropriate language for reasoning about languages (such as ML and Scheme) that are "based on λ-calculus."

QUAD Register transfers, or "quadruples," that correspond (approximately) to the instructions of a very simple von Neumann machine.

PDG Program-dependence graphs, that can represent both control flow and data-flow without intertwining them more than necessary.

SSA Static single-assignment form [34], which is particularly designed for the efficient implementation of certain dataflow algorithms. In SSA, each variable is assigned exactly once; when control paths merge, explicit transfer functions are used to maintain the illusion of single assignment. SSA and CPS are similar in important ways, since CPS also has a kind of single-assignment property.

These intermediate representations are designed to facilitate different transformations. Let us consider several different kinds of optimizations, and see how easy they are to perform with each of these representations.

In-line expansion

CPS λ QUAD PDG SSA

The λ-calculus has variable-binding and scope rules particularly designed for β-reduction, or in-line expansion of functions: The body of a function is substituted for the function call, and the actual parameters of the function are substituted for the formal parameters.

But there is a problem with using λ-calculus to express the behavior of *strict* call-by-value languages (such as ML, Pascal, Lisp, Scheme, Smalltalk, etc.). In the programming language, the parameters of a function are supposed to be evaluated before the evaluation of the body begins; but in λ-calculus this is not necessary. The usual method of β-reduction for λ-calculus will just put a copy of the actual parameter at each location where the formal parameter had appeared in the body. This means:

- A program that was supposed to infinite loop (when interpreted *strictly*) may now terminate.

[1] Readers unfamiliar with the literature on compiler optimization might want to skip this section.

- An actual parameter that was evaluated once in the original program may now be evaluated several times (if the formal parameter is used several times in the original program).
- In a language with side effects (ML, Pascal, Lisp, etc.), the side effects of the actual parameter may now occur after *some* of the side effects in the body of the function, or may not occur at all, or may occur more than once.

It is just as easy in CPS to express substitution, but CPS has none of the problems listed in the previous paragraph. *All actual parameters to functions are variables or constants, never nontrivial subexpressions.* Thus the substitution of actuals for formals (and the consequent "moving" of the actual parameters into the body of the function) can't cause a problem.

As for the other representations—QUAD, PDG, SSA—they are primarily concerned with the representation of individual function bodies, not with optimizations across function boundaries. It is still possible to do in-line expansion with these frameworks, if additional mechanisms are added to represent function parameters and calling sequences; but the problems of termination, and out-of-order evaluation of side-effects, must still be solved.

Closure representations

CPS ⌣ λ ⌣ QUAD ⌢ PDG ⌢ SSA ⌢

In languages with "block structure" or "nested functions"—such as Pascal, Scheme, ML—a function f may be nested inside a function g, which itself may be nested inside another function h. Then f may access its own formal parameters and local variables, but it may also access the formals and locals of g and h. One of the tasks of the compiler is to implement this access efficiently. Since the λ-calculus and CPS also have functions with nested scope, it is easier for the compiler to manipulate these functions and their representations in computing efficient access methods for nonlocal variables. The other three representations, since they are primarily concerned with control and dataflow within individual functions, cannot address this problem very easily.

Dataflow analysis

CPS — λ ⌢ QUAD — PDG — SSA ⌣

Dataflow analysis involves the static propagation of values (more precisely, of compile-time tokens standing for runtime values) around a flow graph. It answers questions such as "does this definition of a variable reach that use of the variable?" which is useful when doing certain optimizations. Since the continuation-passing-style representation of a program contains a fairly faithful representation of the control-flow graph, dataflow analysis is as easy in cps as it is in more traditional representations such as QUADruples.

Static single-assignment form is designed to make forward dataflow analysis particularly efficient, since it is easy to identify the definition that reaches any use of a variable—each variable is defined (assigned) exactly once. Continuation-passing style has a property very much like single assignment, as we will discuss

later. On the other hand, λ-calculus does not appear to be well suited for dataflow analysis.

Register allocation

CPS ⌣ λ ⌢ QUAD ⌣ PDG — SSA ⌣

In allocating variables of the program to registers on a machine, it is useful to have a notation that can conveniently represent the lifetime—the creation and destruction—of values. This *liveness analysis* is really a kind of dataflow analysis, and so the observations of the previous paragraph apply equally well here. We note in particular that in certain phases of our CPS-based compiler, the *variables* of a CPS-expression correspond very closely to the *registers* of the target machine.

Vectorizing

CPS — λ ⌢ QUAD — PDG ⌣ SSA —

Program-dependence graphs are particularly designed for such optimizations as the synthesis of vector instructions out of ordinary loops. Such optimizations are still possible in other representations, but may in the end require auxiliary data structures that accomplish much of what is done by PDGs.

Instruction scheduling

CPS — λ ⌢ QUAD — PDG — SSA —

Modern, highly pipelined computers require *instruction scheduling* at the very back end of the compiler to avoid pipeline interlocks at runtime. Instruction scheduling requires the manipulation of individual instructions with detailed knowledge of their sizes, timings, and resource requirements. The representations described in this chapter are probably a bit too abstract for use in the scheduling phase of a compiler.

Conclusion

The intermediate representations described here have many similarities. Static single-assignment form is just a restricted form of quadruples. Continuation-passing style is a restricted form of λ-calculus. And in fact, there are many similarities between SSA and CPS, since CPS variables have a single-binding property. With continuations, we get both the clean substitution operations of λ-calculus and the dataflow and register analyses appropriate for von Neumann machines.

1.3 What is ML?

This book will demonstrate the use of continuations for compilation and optimization in a real compiler. Our compiler—*Standard ML of New Jersey*—compiles *ML*; but continuation-passing style is not tied in any way to ML, and has been used in compilers for several languages [52].

The programming language ML was originally developed in the late 1970s as the **M**eta-**L**anguage of the Edinburgh *Logic for Computable Functions* (LCF) theorem-proving system [42]. In the early 1980s it was recognized as a useful language in its own right (even for people who don't want to prove theorems) and a stand-alone ML system was implemented [26]. Since then, the *Standard ML* language has been defined [64], a formal semantics has been written [65], several compilers have become available [13, 63], and several hundred programmers at scores of locations are actively using the language.

ML has several advantages as a practical programming language:

- ML is *strict*—arguments to a function are evaluated before the function call, as in Pascal, C, Lisp, and Scheme but not as in Miranda or Haskell, which are *lazy*.

- It has *higher-order functions*, meaning that a function can be passed as an argument and returned as the result of another function—as in Scheme, C, and Haskell, but not Pascal and Lisp. But unlike C, ML has *nested functions* (so do Scheme and Haskell), which make the higher-order functions much more useful.

- It has *parametric polymorphic types*, meaning that a function can be applied to arguments of several different types as long as it does exactly the same thing to the argument regardless of the type. Lisp, Scheme, and Haskell also have parametic polymorphism, but Pascal and C do not. Parametric polymorphism is different from *overloading*, with which a function can be applied to arguments of different types only if a different implementation of the function is written for each type.

- Types are *statically checked* at compile time, so there is no need for runtime type checking (and many bugs may be found before running the program). Other statically checked languages are Pascal, C, Ada, and Haskell; dynamically type-checked (at runtime) languages include Lisp, Scheme, and Smalltalk. But ML (like Haskell) has *type inference*, which relieves the programmer of writing down most type declarations; in Pascal, C, Ada, and other languages descended from Algol the programmer must declare explicitly the type of each variable.

- ML has *garbage collection*, which automatically reclaims unreachable pieces of storage; this is typical of functional languages such as Scheme and Haskell, but Lisp and Smalltalk also have garbage collection; Pascal, C, and Ada usually do not.

- Variable bindings are *statically determined*; as in Pascal, C, and Scheme the variable declaration corresponding to any particular use can be determined by lexical scope in the program text. This is in contrast to Smalltalk, which has dynamic binding for functions ("dynamic method lookup").

- ML has *side effects*: input/output, and reference variables with assignment and update. In this respect it is like most languages (such as Pascal, C, Smalltalk, Lisp, and Scheme), but differs from *purely functional* languages like Haskell. In ML, however, the updateable variables and data structures are constrained and statically identifiable by the compile-time type system. In a typical program, the vast majority of variables and data structures are not updateable.

- ML has a *formally defined semantics* [65] that is *complete* in the sense that each legal program has a deterministic result, and all illegal programs are recognizable as such by a compiler. This is in contrast to Ada, for which formal semantics have been written but are not complete, in the sense that some "erroneous" programs must be accepted by compilers; Pascal, which is recognized to have certain "ambiguities and insecurities" [95]; and C, in which it is very easy for programmers to trash arbitrary parts of the runtime environment by careless use of pointers. Lisp and Scheme are not too bad in this respect; in principle, "erroneous" programs are detected either at compile time or at runtime, though certain things (such as order of evaluation of arguments) are left unspecified [69].

From this summary we see that the problems and language features that our compiler must address have much in common with those addressed by other compilers. But there are several ways in which compilers for "modern" languages like ML must differ from compilers for "traditional" languages like C.

The higher-order functions of ML (Scheme, Smalltalk, etc.) require the compiler to introduce runtime data structures to represent the free variables of these functions. And because the lifetimes of these "closures" are not always determinable at compile time, some form of garbage collection is required.

The presence of garbage collection requires that all runtime data structures be in a format that the collector can understand; the machine registers and other temporaries used by the compiled code must also be accessible and understandable to the collector.

Since a "functional" programming style is encouraged—in which old data is rarely updated, but instead new data is produced—the garbage collector must be particularly efficient. In some older Lisp systems, and in some Algol descendents with garbage collection, much less load is placed on the collector because new objects are less frequently allocated.

Most control flow is indicated by source-language function calls (instead of built-in constructs like **while** and **repeat**). So function calls—especially tail-recursive ones—must have very low overhead.

There is no "macro" facility. Macros in C and Lisp are often used for in-line expansion of frequently executed code fragments. A good compiler for a macro-free language should do some in-line expansion of functions—a much safer way to provide this kind of efficiency.

A unique feature of ML, shared by no other commonly used language, is that most data structures are *immutable* once created. That is, once a variable—or a

list cell, or a record on the heap—is created and initialized, it cannot be modified. Of course, a local variable of a function will be *instantiated* each time the function is invoked, with a different value each time; list cells on the heap eventually become garbage (because no local variable points to them anymore) as new cells are created. But the fact that list cells can't be modified means that the *aliasing problem* becomes trivial. In compiling a conventional language, the statements

$$a \leftarrow \#1(p); \quad \#1(q) \leftarrow b$$

(where $\#1(x)$ means the first field of the record that x points to) can't be exchanged, because p and q might be *aliased*—might point to the same object. Similarly,

$$a \leftarrow \#1(p); \quad b \leftarrow f(x)$$

can't be exchanged unless a great deal is known about the behavior of f.

In ML, mutable variables and data structures—those that can be modified (stored into) *after* their creation—have a different static type than immutable ones. That is, they are distinguished by the type-checking phase of the compiler. In a typical program, the vast majority of variables and data structures are immutable. Thus, the aliasing problem mostly disappears: If p is an immutable variable (as is usually the case), the fetch from p commutes with just about anything. Our compiler exploits this property in several ways.

Lazy languages such as Haskell have immutable variables in principle, but in fact the update of a variable to replace a thunk by an evaluated result looks very much like a mutation to some parts of the compiler and runtime system.

Finally, the fact that ML has a rather abstract formal semantics is quite useful in some ways. Any optimization or change in the representation that leads to the same computable function is legal. In C, on the other hand, there is no formal semantics. Even allowing that there is a reasonably good informal understanding of what a C program is supposed to do, this "semantics" is tied to low-level machine representations. And there are many C programs that "break the rules" and are still expected to work in any "reasonable" compiler. In this situation, the compiler has limited freedom to transform the program.

1.4 Compiler organization

Standard ML of New Jersey [13] is a compiler for ML written in ML. It is a multipass compiler which transforms a source program into a machine-language program in a series of phases:

1. Lexical analysis, parsing, type checking, and producing an annotated abstract syntax tree.

2. Translation into a simple, λ-calculus-like representation (described in Chapter 4).

3. Conversion into continuation-passing style (CPS, described in Chapter 5).

4. Optimization of the CPS expression, producing a "better" CPS expression (Chapters 6–9).

5. Closure conversion, producing a CPS expression in which each function is closed, i.e., has no free variables (Chapter 10).

6. Elimination of nested scopes, producing a CPS expression with one global set of mutually recursive, nonnested function definitions (Chapter 10).

7. "Register spilling," producing a CPS expression in which no subexpression has more than n free variables, where n is related to the number of registers on the target machine (Chapter 11).

8. Generation of target-machine "assembly-language" instructions—in abstract, not textual form (Chapter 13).

9. Instruction scheduling, jump-size optimization, backpatching, and the generation of target-machine instructions (Chapter 14).

The rest of this book is organized very much like the compiler, except that the first phase—which is specific to the ML language—is not described. In fact, this "front-end" part of the compiler is much larger than the back-end phases covered in the book. But our focus here is the use of continuations for optimization and code generation, not how to compile ML.