UNIVERSITY
of
GLASGOW

Department of
Computing Science

Compiler Architecture using a Portable Intermediate Language

Fermín Javier Reig Galilea

Submitted for the degree of Doctor of Philosophy in Computing Science at the University of Glasgow

March 2002

Abstract

The back end of a compiler performs machine-dependent tasks and low-level optimisations that are laborious to implement and difficult to debug. In addition, in languages that require run-time services such as garbage collection, the back end must interface with the run-time system to provide those services. The net result is that building a compiler back end entails a high implementation cost.

In this dissertation I describe reusable code generation infrastructure that enables the construction of a complete programming language implementation (compiler and run-time system) with reduced effort. The infrastructure consists of a portable intermediate language, a compiler for this language and a low-level run-time system. I provide an implementation of this system and I show that it can support a variety of source programming languages, it reduces the overall effort required to implement a programming language, it can capture and retain information necessary to support run-time services and optimisations, and it produces efficient code.

Acknowledgements

I am very grateful to my supervisor David Watt for his support and encouragement throughout this time. His thorough reviews and insightful comments have been essential to turn the drafts of my chapters into a dissertation.

Thanks to my second supervisor Peter Dickman, who provided additional guidance and helped me to clarify and give structure to my ideas. Simon Peyton Jones supervised me during my first months in Glasgow and has provided kind advice on several occasions afterwards. My thesis examiners Richard Jones and Tony Printezis provided many suggestions that improved the dissertation.

Many thanks to Lal George, David Hanson and Chris Fraser for giving me the opportunity to work with them during two summer internships. I have been fortunate to learn from their vast experience.

Lal George and Allen Leung are the authors of MLRISC, an excellent software package on which I have based my compiler. They provided timely improvements to MLRISC when I needed them.

Special thanks go to my parents for their unconditional support always. And to Johanna, for sharing so many things.

I have been supported financially by a doctoral scholarship from Departamento de Educación y Cultura del Gobierno de Navarra.

This dissertation has been written using the TEX document preparation system, the LATEX macro package and a handful of other macro packages. All of these have been contributed by their authors free of charge. I am grateful to them.

Contents

List of Figures

List of Tables

Chapter 1

Introduction

Building a state-of-the-art code generator that targets several architectures entails a high implementation effort. Optimising code generators are large, complex systems that are laborious to write and difficult to debug. For this reason, the reuse of code generation technology is a perpetual goal of compiler implementors. Fortunately, there is much potential for code reuse in a code generator. Large portions are independent of the source language being translated and independent of the instruction set architecture being targeted.

In this dissertation I describe a *generic code generation infrastructure* that can be used to build a complete programming language implementation (compiler and run-time system) with substantially less work.

The infrastructure consists of a portable compiler intermediate language (`C--`), a compiler for this language and a low-level run-time system. I have implemented this system and I have evaluated it with compilers for C and the functional language Caml (Leroy et al. 2002). I have retrofitted a C compiler to target the intermediate language; and I have retrofitted a compiler for Caml and its run-time system to target the intermediate language and the low-level run-time system, respectively. These are the first compiler implementations to make use of the `C--` infrastructure.

I show that this code generation infrastructure (1) can be used to compile very diverse source programming languages; (2) reduces the overall effort required to implement a programming language; (3) can capture and retain high-level information useful for low-level optimisations; (4) can capture and retain high-level information necessary to support run-time services such as garbage collection and exception handling; (5) does not increase the

complexity of the compiler's front end, the run-time system, or the interface between the two; and (6) produces code competitive to that generated by monolithic compilers.

In the following sections I briefly describe what code generators do, what it means for a code generator to be reusable, and what is needed to achieve reusability.

1.1 The Functions of a Code Generator

At the very least, the code generator of a compiler translates the intermediate representation of the source program to the target instruction set and emits these instructions as machine code or as assembly language.

In practice a code generator also performs a series of optimisations on the program with the goal of reducing its execution time, memory usage, or both. A list of such optimisations includes the following (Muchnick 1997; Bacon et al. 1994):

- constant folding
- algebraic simplifications
- unreachable code elimination
- elimination of branch chains
- alignment of branch targets
- dead code elimination
- common subexpression elimination (local and across basic blocks)
- copy propagation
- constant propagation
- partial redundancy elimination
- tail-call optimisation and tail-recursion elimination
- induction variable optimisations
- strength reduction

- loop reorganisations (unrolling, pipelining, blocking, interchange)
- advanced instruction selection (use of SIMD instructions if supported by the target)
- instruction scheduling (within basic blocks, trace scheduling across basic blocks)
- inlining of function calls
- register allocation
- code and data prefetching
- code and data placement (to avoid or reduce cache conflicts)
- peephole optimisations

It is worth pointing out that even if only a fraction of these optimisations are provided by a particular code generator, the implementation effort may still be substantial. For instance, a sophisticated graph-colouring register allocator performs liveness analysis, live range identification, interference-graph construction and simplification, coalescing of live ranges, calculation of spill costs, live range splitting, and insertion of spill code.

In addition to generating and optimising code, in some language implementations, it is the task of the code generator to compute and emit information that will be consulted by the run-time system during program execution. This includes pointer maps for garbage collection (Diwan et al. 1992) and program counter tables for exception handling (Koenig and Stroustrup 1990; Chase 1994).

In summary, there is a substantial implementation investment behind a state-of-the-art code generator. Reuse of this infrastructure substantially reduces the costs of building compilers.

1.2 Reusable Code Generation

The construction of a reusable code generator presents a number of challenges. First, the code generator should be *generic*, meaning that it can be easily targeted from a variety of source programming languages, regardless of the programming paradigm (procedural, functional, logic, object-oriented).

Second, it should be possible for the front end to communicate high-level information that the code generator might find useful to generate better code. Third, the division of the compiler into front end and reusable code generator should not hinder the implementation of high-level run-time services such as automatic memory reclamation, exception handling, and others. Fourth, the interface that the code generator presents should not be one that forces the front end to be implemented in a specific programming language. Fifth, the code generator should provide all the optimisations that advanced compilers routinely implement. Finally, it should be available for a reasonable number of target instruction set architectures.

A number of code generators have been developed that meet some of the goals stated above. MLRISC (George and Leung 2000), VPO (Benitez and Davidson 1988) and the back end of the GNU Compiler Collection (GCC) (Stallman 2001) are all examples of freely-available, reusable code generators. But none of these systems provides all the capabilities that are demanded. For instance, if one uses VPO or GCC, it is not possible to generate the information needed to implement some run-time services efficiently. Or if MLRISC is used, the front end must be written in the SML language, since this is the only interface provided by MLRISC.

1.3 Requirements for Reusability

Reusable code generators must be portable and generic, and must provide support to implement run-time services efficiently.

Portability

Compilers represent programs during translation using *intermediate representations* (IRs). An IR is thus the interface that a code generator exports to its client front ends. In order for a code generator to be reusable, its interface IR needs to provide abstractions that hide the details of the target: instruction set architecture, number of registers available, etc. Target-independence of the IR ensures *portability* of the client front ends, since they need not be changed when the code generator is ported to a new target.

Genericness

At the same time, the IR must be *generic*, so that it can encode the constructs of different programming languages. The translation into the generic IR must occur without loss of semantic information. This is essential to guarantee correctness of the generated code, but is also important to be able to generate efficient code. In order to be considered truly generic, an IR must not impose any particular programming model on its clients. JVM[1] code (Lindholm and Yellin 1999) has been used as the target IR in implementations of languages other than Java, but only by using unnatural encodings of the source constructs into objects, methods and other constructs of the JVM. Such cumbersome translations have resulted in disappointing performance (Benton et al. 1998; Bothner 1998; Wakeling 1999; Schinz and Odersky 2001). Unfortunately, performance is not the only stumbling block: sometimes the semantic model of the source language cannot be accommodated to the constructs of the JVM language (Kahrs 2001). Thus, while JVM code can serve as a portable intermediate representation (in the sense that the Java virtual machine has been ported to many different systems), it is not generic (in the sense that it does not support well programming models too different from that of Java).

The C programming language has been used as a portable assembly language in compilers for Prolog (Codognet and Diaz 1995), Mercury (Henderson et al. 1995), Scheme (Bartlett 1989), SML (Tarditi et al. 1992), Erlang (Hausman 1994), APL (Otto 2001) and many more. C provides portability and reasonable performance, since good C compilers exist for virtually every existing architecture. Also, since C is a fairly low-level language, it is possible to map into it most of the high-level constructs of modern languages. In addition, the GCC compiler supports extensions to ANSI C (indirect branches and global variables in registers) that enable better code to be generated. Unfortunately, some language features cannot be implemented efficiently in C (Peyton Jones et al. 1999). For instance, in most cases it is impossible for a C compiler to perform tail-call optimisation, which is an essential optimisation for declarative languages. Also, C does not have exceptions itself, and provides only non-local jumps to encode them, but with a performance cost compared to native implementations.

[1] Java Virtual Machine.

Run-Time Support

In addition to translating the source code into target machine code, many programming language implementations include a *run-time system*. The run-time system consists of support code necessary to implement certain features provided by the programming language. For example, in languages that support concurrency, the thread scheduler and the code that implements the synchronisation primitives supported by the language are part of the run-time system. In languages with automatic memory reclamation, the run-time system includes a garbage collector (Jones 1996).

Some run-time services require information about the program that is only available during compilation. For instance, a garbage collector must find, and perhaps modify, all pointers to dynamically-allocated data. The garbage collector needs to identify unambiguously the location and liveness of all such pointers. It is the compiler that knows which variables contain pointers. It is also the compiler that knows about the liveness of variables and allocates them to specific locations (registers and memory). This information must be made available to the garbage collector. Hence, the compiler and run-time system must cooperate to provide some services. This cooperation takes the form of data that is emitted in addition to code as part of compilation and that is consulted by the run-time system during execution of the program. The mechanism to exchange this data has to be designed carefully in order to incur the minimum possible cost in execution time and space.

One difficulty is that this data may depend both on information discovered early, by the front end (for instance, the type of a variable), and also on information that is computed late, by the code generator (for instance, what register or memory location holds the value of a variable). If we want to build our compiler using an off-the-shelf code generator, the front end and the code generator need to interface separately with the run-time system. This separation complicates the interface between compiler and run-time system.

One possible solution is that the code generator supplies its own implementation of the run-time services. This is the approach used by front ends that use JVM code as a portable intermediate language. Any existing implementation of the Java virtual machine can run the generated code and provide services like garbage collection and exception handling. The prob-

lem with this approach is that there are many possible implementations of run-time services, each with its own engineering tradeoffs. For instance, Wakeling (1999) reports that the cost of memory allocation in Haskell programs translated to JVM code is an order of magnitude higher than in an interpreter for Haskell. Functional languages allocate memory at higher rates (Diwan et al. (1995) report that SML programs on average allocate one word per 4–10 machine instructions), and Java implementations are typically not optimised for this case.

Language implementations that use C as a target language cannot be efficient and use accurate garbage collection at the same time. A widely-used solution is *conservative* garbage collection (Boehm and Weiser 1988).[2] This approach can often deliver good performance, at the cost of increased space consumption (Wentworth 1990; Agesen et al. 1998), which can be unbounded in the worst case (Boehm 2002). In addition, some compiler optimisations can "hide" actual pointers from a conservative collector (Diwan et al. 1992), leading to program failure. In most cases, the only solution is to disable optimisations for the particular program that fails. A recently-proposed alternative is to implement (in the generated C code) a shadow stack that holds all the pointers (Henderson 2002). This makes accurate collection possible, but it inhibits certain compiler optimisations and it has a cost in performance of the resulting code.

1.4 UNCOLs

The idea of using a universal IR for multiple compilers dates to the late 1950s, when the term UNCOL (UNiversal Computer Oriented Language) was coined (Conway 1958; Strong et al. 1958; Steel 1961). In the early 1990's the ANDF initiative (Architectural Neutral Distribution Format) (Benitez et al. 1991) was launched with the intention of providing an intermediate language to encode the constructs of C (and later Ada). However, neither project has met its initial goals. Ironically, the language that has come nearest to succeeding as an UNCOL has been C, not so much because it is well suited to the task, but because C compilers that generate efficient code are available for most existing architectures.

[2] A conservative collector considers all values that *appear to be* pointers as pointers. One disadvantage of conservative collection is that it is not safe to relocate all objects, which limits the choice of garbage collection algorithms (Jones 1996).

1.5 Outline of Dissertation

The remainder of this dissertation consists of the following chapters.

Chapter 2: Compiler Architecture Using `C--`. This contains a brief description of the compiler architecture and introduces the `C--` intermediate language.

Chapter 3: High-level Annotations for Intermediate Languages. To obtain high-quality object code, it is important that the front end of the compiler shares with the code generator any high-level information about the program that can be used to perform aggressive low-level optimisations. In this chapter, I identify a number of high-level program properties that can be easily expressed in the intermediate language of a reusable code generator. The focus is on static program knowledge that is readily available at the front end, but can be expensive or even impossible for the code generator to rediscover.

Chapter 4: `C--` Support for Run-time Services. A low-level run-time system for `C--` was first proposed by Peyton Jones et al. (1999). Here I shed light on some limitations of that system. These limitations hinder the implementation of certain services of the high-level run-time system, such as generational stack collection (Cheng et al. 1998), stack walking in the presence of foreign calls and callbacks, and efficient identification of pointers for accurate garbage collection. I propose changes and extensions to the `C--` run-time system so that these services can be implemented efficiently.

Chapter 5: Compiling `C--`. This describes the salient aspects of compiling `C--`, including tail-call optimisation, representing `C--` continuations, and optimised parameter passing. A novel method of implementing dedicated machine registers is also proposed.

Chapter 6: Targeting `C--`. This describes the translation of C and Caml into `C--`. To generate efficient code for C's `switch` statement, I extend the `C--` language with indirect branches, which are missing from the original `C--` proposal. I show that the translation of Caml exceptions to `C--` results in code less efficient than the Caml compiler generates. I describe the changes needed in the Caml run-time system to use `C--`.

Chapter 7: Evaluation. This presents an evaluation of the C compiler and the Caml compiler that were re-engineered to use the `C--` code generation infrastructure.

Chapter 8: Proposed Extensions to `C--`. This suggests further extensions to the syntax of `C--`. The first extension is necessary to express contexts in which it is illegal to perform certain low-level optimisations. The remaining extensions are useful to generate more efficient machine code.

Chapter 9: Conclusions. This summarises the dissertation and suggests directions for further work.

1.6 Terminology

In this dissertation I use the term *front end* to refer to those parts of a compiler that are dependent on the source language. The front end includes the lexer, the parser, the static analyser (scope and type checker) and also all language-dependent optimisations (for instance method dispatch optimisations in object-oriented languages). The rest of the compiler will be referred to as the *back end* or the *code generator*, interchangeably. The optimisations performed here are independent of the source language, but may be target-specific (for instance peephole optimisations specific to one instruction set).

Chapter 2

Compiler Architecture Using C--

This chapter provides a bird's-eye view of the compiler architecture proposed in this dissertation. It also contains a brief introduction to C--, excluding its low-level run-time system, which is described in Chapter 4.

2.1 Compiled Programming Languages

Figure 2.1 shows a conceptual view of the implementation of a compiled programming language. (In the figure, dashed arrows depict interactions that occur at run-time.) The language implementor provides a compiler and a run-time system. The compiler generates object code and data, which are linked together with the run-time system to form the executable program. The generated code requests services from the run-time system, such as thread creation, synchronisation and garbage collection. To provide some of these services the run-time system needs to consult the data emitted by the compiler.

In the compiler architecture described in this dissertation, the front end and the run-time system use the services of a reusable code generation infrastructure. This infrastructure consists of a back end and a low-level run-time system. The interface between the front end and the back end is C--, a generic intermediate language. The run-time system accesses the data emitted by the back end through the low-level run-time system. Figure 2.2 shows a conceptual view of this architecture.

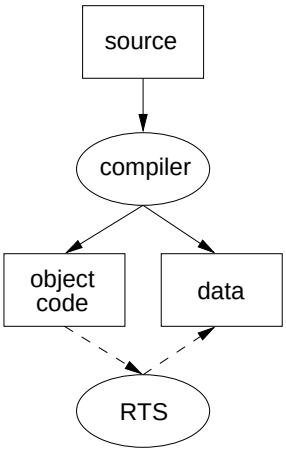


Figure 2.1: Language implementation consisting of a compiler and a run-time system.

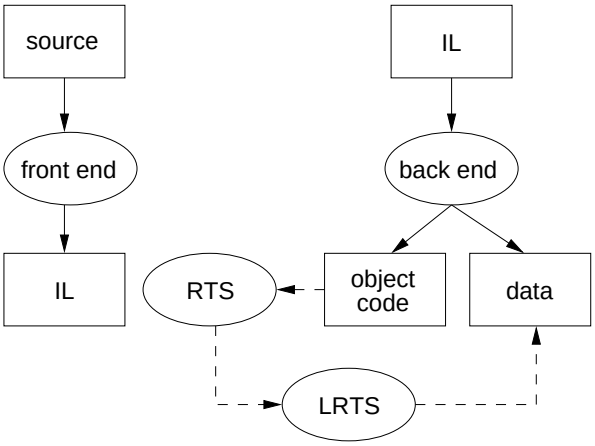


Figure 2.2: Language implementation using a reusable back end and a low-level run-time system.

Terminology

In the remainder of the dissertation, I use the terms *low-level run-time system* and *C-- run-time system* interchangeably. To avoid ambiguity, instead of *run-time system* I use the more precise term *front-end run-time system* in some places.

2.2 An Overview of C--

An initial proposal for C-- appeared in (Peyton Jones et al. 1997). The version used in this dissertation is essentially the one described in (Peyton Jones et al. 1999). Additional syntax is introduced in later chapters, as needed. For a complete definition of the C-- language, readers are invited to consult the C-- reference manual (Ramsey et al. 2001).

C-- is a portable intermediate language intended to be independent of both source programming language and target architecture. Some of C-- is similar to other compiler intermediate languages, for example the ones described by Muchnick (1997) and Fraser and Hanson (1995).

C-- is not a "write-once, run-anywhere" intermediate language, like JVM code (Lindholm and Yellin 1999). It is neither a language-specific, target-independent code format, like ANDF (Benitez et al. 1991) or slim binaries (Franz and Kistler 1997). These systems are abstractions of high-level languages, while C-- provides *abstractions of hardware*. For instance, C-- variables are an abstraction of machine registers; they hide the actual number and any conventions on their use. Similarly, memory loads and stores in C-- hide the particular addressing modes available in the machine.

C-- hides most, but not all, of the target machine from the front end; front ends must be aware of properties like the size of the native data-pointer and code-pointer types, alignment requirements of the target machine, etc. For example, a C-- program generated for a machine with 64-bit addresses would be different from a C-- program generated for a machine where addresses are 32 bits.

2.3 Types

C-- supports a bare minimum of types: bits types (`bits8`, `bits16`, `bits32`, `bits64`) and floating-point types (`float32`, `float64`). These types encode

only the size and the kind of register (general-purpose or floating-point) required for a C-- value.

By intent, this minimalistic type system does not preserve high-level type information: its sole purpose is to direct the C-- compiler's mapping of C-- variables to registers or memory. A front end uses the `bits` types to represent a number of high-level types, such as booleans, characters, numbers, pointers, bit vectors, etc.

Not all C-- types are available on all machines. For instance, a program that uses `bits64` may be rejected by a C-- compiler that targets a machine that supports only 32-bit values and operations.

2.4 Functions

C-- has parameterised functions. Each function has a fixed number of parameters and a fixed number of results. Each parameter and each result has a fixed type. Note that multiple results are supported.

A function body consists of declarations and statements. Declarations are local variables and local memory allocation. Statements include labels, assignments, memory writes, conditionals, gotos, calls, jumps (tail calls), and returns.

There are no structured loop statements; iteration is implemented using conditionals and gotos or conditionals and (tail) calls.

The number and types of actual parameters and results in a function call must match those in the function definition. The function specified in a call statement can be an arbitrary expression, not simply the statically-visible name of a function.

Control does not return from a `jump` (tail call) and the caller's activation frame is deallocated before the call.

2.4.1 Expressions

C-- expressions yield values of `bits` and `float` types. Values may be assigned to variables, passed as parameters, returned as results, and fetched from and stored in memory.

Expressions can be literals (integer, floating-point and character), names (of variables, functions, code labels, data labels and continuations), memory reads and primitive operators applied to one or more sub-expressions.

The type of function names and code labels is the native code-pointer type. The type of data labels is the native data-pointer type.

The list of primitive operators includes integer arithmetic (`+`, `-`, `*`, `/`, `%`), floating-point arithmetic (`+f`, `-f`, `*f`, `/f`), bit manipulation (`>>`, `<<`, `|`, `&`, `^`), and logical (`||`, `&&`, `==`, `!=`, `<`, `<=`, `>`, `>=`).

There are no separate signed and unsigned `bits` types. Instead, the distinction is in the operators, like in most assembly languages. There are unsigned integer operators for arithmetic (`+u`, `-u`, `*u`, `/u`, `%u`), comparison (`==u`, `!=u`, `<u`, `<=u`, `>u`, `>=u`) and bit shift (`>>u`).

There is no "address-of" operator. However, global and local memory can be allocated explicitly and labelled.

2.4.2 Local Memory

To handle high-level values that cannot be represented using `C--`'s primitive types, like records and arrays, memory can be allocated locally to a function. This declaration

```
stackdata {
  a_record:
  bits8;
  float64[2];
}
```

reserves enough memory to hold a `bits8` and two `float64`s; it also declares a label that names the address of the `bits8`. The label itself is an expression of the native data-pointer type.

An explicit `align` directive provides alignment where that is required. For instance:

```
stackdata {
  a_record:
  bits8;
  align 8;
  float64[2];
}
```

Here, enough padding is allocated after the `bits8` to guarantee that the first `float64` is aligned to a 64 bit boundary (8 bytes).

2.4.3 Memory Access

All memory access is explicit, including the type and size of the value being fetched or stored. For example, this memory write statement

```
bits8[a_record] = 1;
```

stores the literal `1` into the 8-bit memory cell referenced by the label `a_record`. The same syntax is used for load expressions:

```
y = float64[a_record+1];
```

This statement loads a 64-bit float from the address referenced by the expression `a_record+1` and assigns that value to the local variable `y`.

2.4.4 Calling Conventions

The calling convention for `C--` procedures is entirely a matter for the `C--` implementation. In particular, `C--` need not use the native calling conventions to pass parameters and return results.

The programmer can ask for a function to use C's convention, so that the function can be called from a C program. Similarly, external C functions can be called from a `C--` function by specifying the C calling convention at the call site.

2.5 Static Data

Static memory can be allocated and initialised using constructs similar to those found in assembly languages. For example:

```
section ".data" {
  foo: bits32{10}; /* One bits32 initialised to 10 */
  bits32{1,2}; /* Two initialised bits32s */
  align 8;
  float64[2]; /* Uninitialised array of 2 float64s */
  bar: bits8 /* An uninitialised byte */
}
```

Here `foo` is a label that names the address of the first bits32; `bar` labels the address of the last `bits8`. The labels are values of the native data-pointer type; they can be assigned to variables of that type and even used

in other static data initialisers. However, they are immutable values: they cannot be assigned to.

2.6 Continuations

Ramsey and Peyton Jones (2000) have proposed *C-- continuations* as a mechanism to support source-language exception handlers in C--. Below, I explain C-- continuations briefly and refer the reader to (Ramsey and Peyton Jones 2000) for the full details.

In the following code

```
f(bits64 x, float64 y) {
  bits64 w;
  ...
  return;
  continuation k(w):
  ... statements mentioning x, y, w
}
```

k represents a control-flow target "with arguments". k can be invoked from within f, but continuations are most useful for non-local transfers of control, such as exception handling.

The above code declares the name k as an *r*-value of the native data-pointer type. (For instance, bits64 in a 64-bit architecture.) As an *r*-value, k cannot be assigned to, but it can be passed to a procedure, used in the right hand side of an assignment, or stored in memory.

The x in continuation k(x) is not a binding instance; the "formal parameters" of a continuation must be variables of the enclosing function, and therefore do not need type declarations.

There are different C-- constructs to invoke a continuation. They all have the same denotational semantics, but different run-time costs. Here I describe one of them. The statement cut to k(*arguments*); transfers *arguments* to conventional locations,[1] truncates the stack to k's activation, and sets the program counter to k's program counter.

[1] These locations are determined by the C-- implementation.

2.7 An Example

Finally, I show the code of a C-- function that uses most of the constructs explained above. The function does not compute anything interesting; it is merely a representative collection of C-- declarations and statements.

```
/* A function of two parameters. The types of a function's
   results are not explicitly stated */
f(bits8 x, float32 y, bits64 z){
  /* A local variable declaration */
  bits8 a;
  /* The type system does not distinguish character and
     integer literals. Both can be assigned to a bits8 */
  a = 1;
  a = 'a';
  bits32 b, c, d;
  /* A call to a function that returns two results */
  b,c = g(a * 10);
  /* A conditional */
  if(b < c) {
    /* A tail call */
    jump g(a * 10);
    /* The next statement is never executed. Control does not
       return from a jump */
    a = 1;
  }
  /* A code label */
  L:
  if(b < 0) {
    b = b + 1;
    /* A call to a C function */
    foreign "C" putchar('a');
    /* A goto */
    goto L;
  }
  /* A local memory declaration (an array) */
```

```
  stackdata {
    tbl: bits32[10]; /* tbl is a data label */
  }
  /* Labels and function names are values. They can
     be used in assignments, ... */
  b = L;
  b = tbl;
  b = f;
  /* and stored in memory */
  bits32[tbl] = f;
  bits32[tbl+4] = L;
  bits32[tbl+8] = tbl;
  /* Two indirect calls; they both call f */
  c,d = b(x, y);
  c,d = (bits32[tbl])(x, y);
  if(c == d) {
    /* Invoke the continuation stored in z */
    cut to z(d);
  }
  /* f returns two results */
  return(0, 1);
}
```

Chapter 3

High-level Annotations for Intermediate Languages

Building a state-of-the-art code generator that targets several architectures entails a high implementation effort. At the very least, the code generator performs instruction selection, register allocation and emission of assembly language or machine code. In practice we want the code generator also to perform low-level optimisations (independent of the source language), such as instruction scheduling, common subexpression elimination, loop-invariant code hoisting, unreachable code elimination, copy propagation, peephole optimisations, etc (Muchnick 1997; Bacon et al. 1994).

Because of the costs involved in implementing an optimising code generator, using an off-the-shelf, source-language-independent back end is very appealing to compiler writers. MLRISC (George and Leung 2000), VPO (Benitez and Davidson 1988) and the GCC back end (Stallman 2001) are all examples of freely-available code generators.

Portable compiler target languages are also of interest to compiler writers. C has been used as a portable assembly language in compilers for Prolog, Mercury, Scheme, SML, Haskell, APL, and many more. C ensures wide portability and reasonable performance, since good C compilers exist for virtually every target. More recently, JVM code has been gaining popularity as a portable assembler for source languages other than Java. `C--` is a portable compiler target language that lacks the shortcomings of C and JVM code as assembly languages (Peyton Jones et al. 1999).

To obtain high-quality object code, it is important that the compiler

front end shares with the back end any high-level information that can be used to perform aggressive low-level optimisations. Program information that may be readily discovered by the front end can be expensive or even impossible to rediscover in the back end. In monolithic compilers, the different phases can share information about the source program via in-memory data structures like symbol tables or program dependence graphs (Ferrante et al. 1987), or via auxiliary files for cross-module optimisations.

When using an off-the-shelf back end, the only way to communicate program properties to the code generator is via constructs in the intermediate language. The portable back ends and intermediate languages mentioned above provide little or no support to encode high-level semantic information. For instance, with VPO the only information that can be passed to the code generator is whatever can be expressed as register transfer lists.

In this chapter I identify high-level program properties that can be used to perform aggressive back-end optimisations and/or reduce compilation time. The focus is on static program knowledge that is readily available at the front end, or that is the result of source-language-dependent analysis.

Here, we are interested in communicating static program properties to *language-independent back ends*, regardless of the source language (which may be procedural, object-oriented, or declarative).

Some compilers compute such high-level information and exploit it in the back end, but generic intermediate languages and code generators described in the literature do not provide ways of expressing most of the static properties presented here.

My contribution is to identify an extensive set of such high-level properties and to propose suitable annotations for intermediate languages to convey this information to the code generator.

The annotations are generic. They can be retrofitted to any existing intermediate language, not just `C--`. And they express language-independent program properties. As such, they can be used in language implementations of any paradigm (procedural, object-oriented, or declarative). I intend the annotations as suggestions to designers of compiler intermediate languages.

3.1 Annotations for Low-Level Optimisations

Compiler front ends for modern programming languages have abundant information about programs: types, side effects, control flow due to exceptions, etc. I shall assume the following principle:

> *Do not throw away high-level information that can be exploited for low-level optimisations and that the back end cannot (easily) recover.*

The annotations can be inferred by the compiler, but may also be provided by the programmer, in source languages that support them. In the latter case, the compiler translates source-level annotations into annotations of the intermediate language. Since the annotations are not specific to any particular back end or intermediate language, I do not propose concrete syntax.

3.1.1 Control Flow of Function Calls

Some functions do not return normally to their caller, but terminate the program abruptly. The standard libraries of many programming languages include non-returning functions: `exit`, `abort` and `longjmp` in C; `System.exit` in Java; `error` and `System.exitWith` in Haskell; etc. In GNU C, programmers can annotate functions as `noreturn` (Stallman 2001).

I propose that calls in the intermediate language can be annotated as non-returning. The back end can use this information to remove unreachable code and to perform better register allocation.

If a call to function F does not return, control will never reach the instruction immediately following the call. In the back end, the unreachable code elimination phase can delete all code that is dominated by the call. (A statement S_1 *dominates* statement S_2 if every control path that reaches S_2 passes through S_1.)

Sometimes, calls to non-returning functions are not present in the source program, but are generated by the front end during the translation to low-level code. For instance, in a language with run-time error checking, the front end emits an array-bounds check for every array access (except where it can statically determine that the index is within bounds). An error check typically contains a call to a non-returning error function. The low-level

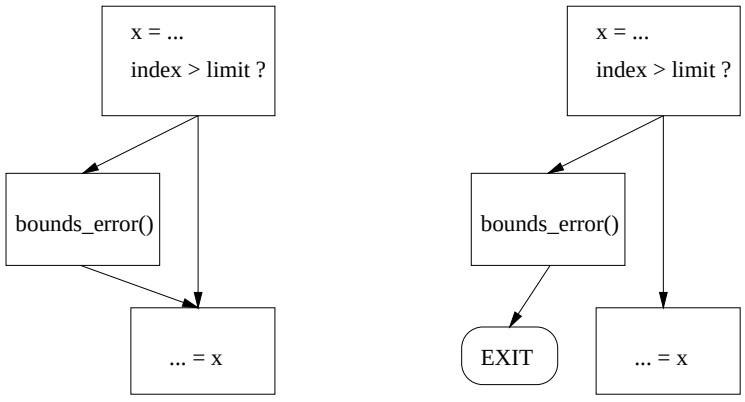
code may end up with many such calls that do not return. If they are identified, the back end can perform better register allocation. For instance, the following code fragment:

```
x = ...;
arr[index] = x;
```

may be expanded to this before code generation:

```
x = ...;
if (index > limit) {
   array_bounds_error();
}
arr[index] = x;
```

Here, `x` cannot be allocated to a scratch (caller-saves) register, since this register might be overwritten by the function `array_bounds_error`. The register allocator can either spill `x` across the call, or allocate it to a callee-saves register. However, if the call is annotated as non-returning, `x` can be allocated to a scratch register, which is cheaper than spilling or using a callee-saves register. (Using a callee-saves register in a function imposes an indirect cost of saving it before its first definition and restoring it after its last use.) Essentially, the annotation lets the back end transform the control-flow graph (CFG) as shown in Figure 3.1. In the CFG 3.1(b), `x` is not live across the call to `array_bounds_error`, and thus may be allocated to a scratch register.



(a) CFG when the call does return.

(b) CFG when the call does not return.

Figure 3.1: Effect of a "does not return" annotation on the CFG.

3.1.2 Control Flow due to Exceptions

Exceptions are the preferred method for signalling error conditions in programming languages that provide them. For instance, Java throws `IndexOutOfBoundsException` for illegal array accesses, rather than terminating the program.

In a control-flow graph representation of the program, a call that may raise an exception terminates a basic block and creates a control-flow edge to the handler as well as an edge to the next statement after the call. For example, the Caml code in Figure 3.2 has the CFG of 3.2(b). Either `f` or `g` could throw `Exn1` or some other exception, but only `Exn1` is caught in this example. If any other exception were thrown by `f` or `g`, control would flow to the `EXIT` pseudo-node of the CFG.

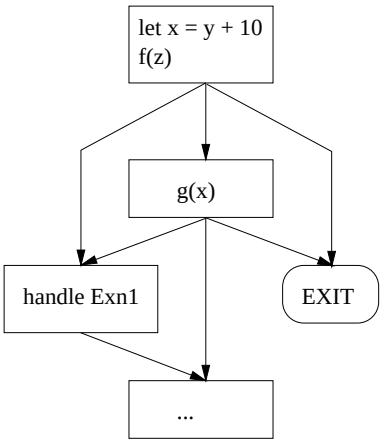
A portable intermediate language must have a way of conveying to the back end the additional control-flow edges due to exceptions. Without them, the code generator cannot build an accurate CFG. In `C--`, extra edges for exceptions are expressed via the `also` annotation (Ramsey and Peyton Jones 2000). MLRISC has a similar construct. For the example above, in `C--` the calls `f(z)` and `g(x)` require an `also` annotation to the handler code.

Exception analysis (Leroy and Pessaux 2000; Yi and Ryu 2001) is useful to diagnose programming errors due to possibly uncaught exceptions. Interestingly, knowing that an exception is always thrown is useful information for back-end optimisations. I propose an `always` annotation for these situations. The annotation carries the control-flow successor of a call. In `C--`, it would carry a continuation, in MLRISC it would carry a label, and similarly in other intermediate languages. With `always` annotations, the back end can simplify the CFG by removing edges that will never be taken at run-time.

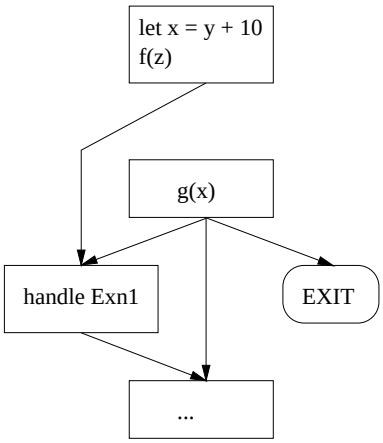
In the example, if the static analysis can prove that `f(z)` always throws exception `Exn1`, the call can be annotated to give the CFG of 3.2(c). The annotation makes it clear that the only possible successor to the call `f(z)` is the exception handler. The code `g(x)` is now unreachable and the CFG can be simplified to the one in 3.2(d). Moreover, if there are no other uses of `x`, the definition `let x = y + 10` is dead code and can be deleted. This, in turn, might enable further dead code elimination if, for example, `y` is used only in the definition of `x`.

On the other hand, if the static analysis can prove that `f(z)` always

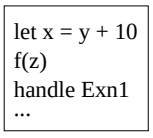
```
let x = y + 10 in
(try
  f(z); g(x)
with Exn1 ->
  handle_exn());
...
```



(b) Additional control-flow edges due to exceptions.



(c) CFG when `Exn1` is guaranteed to be thrown.



(d) Optimised CFG when `Exn1` is guaranteed to be thrown.

Figure 3.2: Effect of an "always raises" annotation on the CFG.

throws an exception different from `Exn1`, then the call can be annotated as non-returning (as described in Section 3.1.1) to indicate that the only successor is the `EXIT` pseudo-node. In this case, everything dominated by the call in the original CFG is now unreachable.

All the front end has to do is annotate the statement `f(z)` during the translation to intermediate code. Standard back-end optimisations like unreachable and dead code elimination take care of the rest.

3.1.3 Branch Prediction

Accurate branch prediction is extremely useful for several low-level optimisations. Code motion (Hailperin 1998), control and data speculation (Dulong et al. 1999), global instruction scheduling (Young and Smith 1998), branch alignment (Young et al. 1997) and good spill-cost estimates in register allocation all benefit from accurate predictions of the outcome of conditional branches.

The back end can make good guesses for some branches: backward loop branches are likely to be taken, equality tests of floating point numbers are likely to yield false, etc. (Ball and Larus (1993) apply simple local heuristics to MIPS machine instructions and report average hit rates of 80%.) However, using knowledge of high-level properties of the program, the compiler front end can make better predictions than the back end of the outcome of some conditional branches. Below I list a few examples.

- The conditional tests of run-time error checks inserted by the front end (arrays bounds checking, invalid argument type for an operation, etc) are likely to select the no-error case.
- Pattern matching on lists is more likely to succeed on non-empty lists.
- If a program contains a non-exhaustive pattern match, the compiler typically inserts a catch-all pattern with code that reports a pattern match failure (by raising an exception, for instance). The conditional branch to this code can be annotated by the front end as very likely not to be taken.
- Dynamic allocation of objects in garbage-collected languages is translated in many implementations into low-level code similar to this:

```
ptr = allocate_memory(size);
if (ptr > limit) { garbage_collect(); }
... use ptr ...
```

The comparison `ptr > limit` will fail in most cases.

- In GNU C, there is a primitive mechanism for the programmer to provide branch prediction information (`__builtin_expect`).

These are a few of many possible examples. In all cases, the front end has the advantage of high-level information to make better predictions than the back end.

I propose that all constructs of the intermediate language that have multiple control-flow successors accept (optional) probability annotations. These constructs include conditional branches, `if then else`, `switch`, and indirect jumps where the possible targets are listed. The extra control-flow edges for exceptions introduced by `also` annotations (Section 3.1.2) can also be annotated with branch probabilities.

3.1.4 Data Prefetching

Data prefetching aims to reduce the cost of cache misses by transferring data from main memory to the processor before it is actually used in a computation. Many current processors have some form of prefetch instruction. Judicious placement of prefetch instructions can substantially reduce the execution time of memory-bound programs. The compiler back end can insert prefetch instructions during the instruction scheduling phase. Occasionally, however, it is the front end and not the back end that knows best where it can be profitable to insert a prefetch instruction. In these cases, a portable intermediate language should be equipped with prefetch annotations.

In garbage-collected languages, writes to newly-allocated objects have cache miss rates close to 100%. (On machines with write-validate caches, write misses do not cause the processor to stall. But on machines with fetch-on-write or write-around policies write misses are costly.) If we use sequential allocation from a contiguous free space, prefetch annotations can be inserted at allocation points (Appel 1998). Sometimes, high-level language constructs can serve as prefetch hints: in C99 (ISO/IEC 1999), the use of `static` in this function prototype

```
float dot_product(float x[static 6], float y[static 6]);
```

is a promise to the compiler that `x` and `y` point to arrays containing at least six floats. A C99 front end can emit a prefetch annotation before every call to `dot_product`. Finally, programmers can provide prefetch hints at the source level that are to be translated to prefetch annotations in the low-level intermediate language.

3.1.5 Pointer Maps for Accurate Garbage Collection

To support accurate garbage collection, compilers emit *pointer maps* that describe to the collector the location of local variables and temporaries that contain pointers to heap-allocated objects. Pointer variables local to a function can be either in registers or in the activation frame of that function. The compiler back end emits a pointer map for every program point where a function can be suspended for garbage collection—*gc-points*. Allocation points are gc-points, since an attempt to allocate when there is not enough free memory can start a garbage collection cycle. Function calls are gc-points as well, since the callee, or anything it calls, may try to allocate memory.

The total size of the pointer maps of a program varies from system to system. The Java compiler described by Agesen et al. (1998) builds maps that consume an average of 57% of the JVM code. Tarditi (2000) uses a compact representation to achieve an average of 3.6% of code size for optimised Java programs compiled to the x86. Compact representations and compression schemes trade space consumption for increased decoding time during garbage collection.

Not enough attention has been paid in the garbage collection literature to the issue of *unnecessary* pointer maps. (To my knowledge, only Diwan et al. (1992) mention it, but very briefly.) Many call sites are not gc-points, because the callee (and anything it calls transitively) do not allocate memory during its execution and therefore cannot trigger a garbage collection. The pointer maps for all these call sites will never be consulted by the garbage collector, and therefore waste space in the object code. This is highly undesirable for systems where memory is scarce, like PDAs or hand-held computers.

The standard libraries of many garbage-collected languages contain a wealth of functions that cannot start a garbage collection (mathematical functions, character manipulation, traversal of data structures, I/O, inter-

face to the operating system, etc). Calls to these functions can be annotated by the front end, so that the back end does not emit pointer maps for them.

In a multithreaded environment, however, all calls are potentially gc-points, even if the called function itself does not allocate: if an active thread starts a collection, the collector has to scan the stack and registers of suspended threads as well. Thus, in the presence of multithreading, pointer maps are needed for all suspension points, and no call should be annotated as not being a gc-point. However, even in languages that support threads, many programs are single-threaded. If the compiler can determine this for a program, then it is safe for the front end to annotate calls that are not gc-points. (In some implementations of Java user-defined finalizers are executed by system threads. In this case, to assert that no allocation takes place during a method invocation—i.e. that the call site is not a gc-point—, the compiler needs to check that none of the finalizers allocate memory in addition to checking that the program does not start multiple threads.)

3.1.6 Memory Disambiguation

Disambiguation of memory accesses is vital for several back-end optimisations including redundant load/store elimination, scheduling for hiding memory latency, load/store hoisting, and common subexpression elimination of load/stores. For instance, in the C code shown below:

```
*p = ...;
if(...) {
  ... use p ...
} else {
  x = *q;
}
...
```

if there are no other uses of `p`, the statement `*p = ...;` can be hoisted into the true branch, *provided that* `p` *and* `q` *cannot point to the same memory location.*

I propose annotating loads, stores and calls with sets of tags that represent abstract memory regions. Two memory access operations may refer to the same run-time location if their tag sets intersect. Calls take two sets of tags, `ref` and `mod`, for locations that are read and written, respectively. If a

function accesses only private memory, calls to it are annotated with empty `mod` and `ref` sets. In the front end a number of analyses can be used to generate `mod/ref` sets, including type-based disambiguation (Diwan et al. 1998), points-to analysis (Cheng and Hwu 2000), and algorithmic analysis (Novack et al. 1995).

3.1.7 Side Effects of Functions

Some functions do not perform I/O operations, nor access the global variables of the program, nor raise exceptions. A *pure* function returns the same result when called with the same arguments, independent of the calling context. In Haskell, all functions that do not have monadic types are pure. In ML, type and effect systems (Benton and Kennedy 2000) can assign non-computational types to pure functions. The standard libraries of many programming languages contain many pure functions (mathematical, string manipulation, etc.). In GNU C, programmers can annotate function declarations as `pure` or `const`. (The return value of a `pure` function may depend on the values of its parameters and/or the values of global variables. `const` is more strict: the result may *not* depend on the contents of global variables.)

In the compiler back end, calls to pure functions are subject to dead code elimination and code motion optimisations, like loop invariant hoisting, common subexpression elimination, and partial redundancy elimination (Briggs and Cooper 1994). For example, in the following code:

```
y = cos(z);
...
for (...) {
  vec[i] = cos(x) + ...
}
```

if the call to the cosine function has no side effects and there are no uses of `y`, then the assignment `y = cos();` is dead code. Also, if the back end can determine that `x` is loop-invariant, then the second call to `cos` can be hoisted outside the loop.

I propose a `no_effects` annotation for function calls in the intermediate language. As in GCC, it is useful to distinguish between accessing global memory and performing other side effects like I/O or raising exceptions.

Thus, `no_effects` is equivalent to GCC's `pure` annotation. `no_effects` does not say anything about memory reads and stores. This is communicated to the back end with `mod/ref` lists instead. In the previous example, the statement `y = cos(z);` cannot be deleted if it can modify a global location that can be read later in the program. Calls to `cos` would be annotated with both `no_effects` and empty `mod/ref` lists. (This is equivalent to a `const` function using GCC's annotations.) In this code:

```
for (...) {
  ... f(x) ...
  vec[i] = g(x);
}
```

if `x` is loop-invariant, the call `g(x)` can be hoisted outside the loop if it is annotated `no_effects` *and* it has no read/write conflicts of global memory with `f`.

Notice that if a function may diverge for some of its inputs, deleting a call that is dead code, or moving it across basic blocks, may change the program result when compiled with optimisations turned on. This may not be allowed by the source language definition. If this is the case, non-termination should be considered a side effect and only calls that can be proven to terminate should be annotated with `no_effects`.

Side-effects annotations and `mod/ref` annotations are also useful when the target is a stack-based language. Consider the following code:[1]

```
x = f(a);
y = g(b);
z = h(y,5,x);
```

where `f` has no side effects, `g` might have side effects, and `f` and `g` don't access the same global variables. In the absence of any annotations, the back end emits the code shown in Figure 3.3(a). With appropriate `no_effects` and `mod/ref` annotations, the calls to `f` and `g` can be permuted to save the two `swap` instructions, resulting in the code of Figure 3.3(b).

[1] This example was suggested to me by Andrew Kennedy.

```
push a
call f
push b
call g
swap
push 5
swap
call h
```

(a) Unoptimised code.

```
push b
call g
push 5
push a
call f
call h
```

(b) Optimised code.

Figure 3.3: Code for a stack-based target.

3.2 Related Work

Annotations have been used successfully to assist back-end optimisations in several compilers. Cho et al. (1998) communicate data dependence information of C programs to the GCC back end. The data is stored in separate files and is retrieved by the back end via a set of provided query functions. They report average speedups of 11% on a MIPS R10000. Several recent papers describe the use of Java class file attributes to store program information. Krintz and Calder (2001) describe a framework to annotate JVM code with information collected off-line. They convey counts of local variable usage and control-flow information. The annotations are then used by dynamic compilers to guide optimisation. They report reduced optimised compilation overhead by 78% and speedups of 7% on average for a set of Java programs. Pominville et al. (2001) encode optimisation information in class file attributes for elimination of array bounds checks. They show speedups of up to 10% in the Kaffe virtual machine with a just-in-time compiler.

Some of these related projects exploit static program information for language-specific optimisations, like array bounds checking elimination in Java. In contrast, the annotations proposed here can be used to convey information that can be used for language-independent optimisations in the back end.

In a typed intermediate language, it is possible to use a very expressive type system which can encode complex propositions and proofs about a program (Shao et al. 2002), possibly including all the properties captured by the annotations proposed here.

The factored control-flow graph (FCFG) (Choi et al. 1999) is a program representation that is more compact than traditional CFG representations

for programs that have many operations that may raise exceptions. The main advantage of a FCFG is that it results in smaller program graphs. The annotations of Section 3.1.2 are useful for code generators that use either CFGs or FCFGs.

It remains to evaluate the actual performance gains that an optimising code generator can achieve using the annotations proposed here. Cheng and Hwu (2000) and Ghiya et al. (2001) describe thorough studies of the importance of different memory dissambiguation techniques for optimizations. Young and Smith (1999) present a performance analysis of branch prediction. Mowry et al. (1992) evaluate different prefetching techniques applied to scientific codes. I am not aware of similar evaluations of the impact of GCC's `pure` and `const` annotaions, or of effects type systems proposed for optimization of functional languages (Tolmach 1998; Benton and Kennedy 2000).

Chapter 4

C-- Support for Run-time Services

C--'s design is unique in that it includes not only an intermediate language, for use by the front end, but also a low-level run-time system, for use by the front-end run-time system (Peyton Jones et al. 1999; Ramsey and Peyton Jones 2000). Using C--'s run-time system, a garbage collector can locate pointers in the stack unambiguously and an exception dispatcher can unwind the stack.

This chapter extends the low-level run-time system for C--, making the following contributions:

- I extend the interface to support the implementation of generational stack collection (Cheng et al. 1998) (Section 4.4).
- I generalise and simplify the interface so that an accurate garbage collector can locate pointers in the stack efficiently (Section 4.5).
- I augment the interface to support stack walking in source languages that provide a foreign function interface (Section 4.7).

Though demonstrated here in the specific context of C--, the concept of a low-level run-time system applies equally well to compiler-target languages other than C-- (including C!) and also to generic code generators, such as MLRISC (George and Leung 2000) and the GCC back end (Stallman 2001).

4.1 Introduction

The run-time system for a programming language provides such high-level services as garbage collection (GC) and exception handling.

A garbage collector must find, and perhaps modify, all pointers to dynamically-allocated objects—the *GC roots*, or *root set* (Jones 1996). An *accurate* garbage collector needs to identify unambiguously the location and liveness of all such pointers. If, furthermore, the collector might relocate objects, the locations of heap objects may change during garbage collection, and the collector must be able to redirect each root to point to the new location of the corresponding heap object. Figure 4.1 sketches how a simple, accurate garbage collector scans the stack of activation frames for GC roots.

```
void gc_scan_stack() {
  activation_type act;

  act = top activation;
  for (;;) {
    for each root r in activation act {
      process r;
    }
    if (act==bottom activation) {
      /* Reached bottom of stack: we are done */
      break;
    } else {
      act = caller's activation;
    }
  }
}
```

Figure 4.1: Algorithm for scanning the stack in an accurate garbage collector.

In some implementations of exception handling, when an exception is raised, the exception mechanism unwinds the call stack until it finds an activation where a handler for that exception is in scope;[1] then it invokes that handler. Depending on the semantics of the source language, additional actions may have to be executed as activations are unwound. For instance, in C++, as the call stack is unwound, destructors must be invoked for those

[1] We say that *a handler is in scope*, or that control is *inside the scope of a handler* when the program counter lies within the section of the program protected by the handler.

objects whose dynamic scope terminates. Figure 4.2 shows the pseudo-code of a simple exception dispatcher.

```
void dispatch(exception_type exn) {
  activation_type act;

  act = top activation;
  for (;;) {
    if (a handler for exn is in scope) {
      restore values of non-volatile registers;
      SP = SP of activation act;
      invoke handler, passing exn as a parameter;
    } else if (act==bottom activation) {
      /* Reached bottom of stack */
      report_unhandled_exception(exn);
    } else {
      call destructors in open scopes of act;
      act = caller's activation;
    }
  }
}
```

Figure 4.2: Algorithm of a simple exception dispatcher.

4.2 Compiler Support for Run-Time Services

In order to provide some of its services, the run-time system of a programming language requires information available only to the compiler.

4.2.1 Accurate Garbage Collection

Accurate garbage collectors must be able to find all pointers in the stack and in the registers at any point in the program at which collection may occur. To support the garbage collector, the compiler can emit *pointer maps*, static tables that describe the location of live local variables and temporaries that contain pointers to heap-allocated objects (Diwan et al. 1992). (In contrast, conservative collectors (Boehm and Weiser 1988) need little or no support from the compiler to locate heap pointers in the call stack: the collector scans every location in the stack.)

The collector needs a pointer map for every gc-point. The gc-points are allocation points, function calls and, in multithreaded languages with pre-emptive scheduling, all other program points where a thread can be suspended. Furthermore, when a collection is triggered by one of the threads, all other threads must be advanced up to a gc-point. If a thread is executing a loop, advancing might take an arbitrary amount of time. To make this amount bounded, gc-points are inserted in every loop.[2]

4.2.2 Exception Handling

To support exception handling by stack unwinding[3] the compiler emits static tables that describe which exception handlers are in scope at any point in the program at which an exception can be raised (Koenig and Stroustrup 1990).

The term *safe point* is commonly used to refer to all program points where the program can be interrupted by the run-time system, and for which the compiler emits static descriptors. I will use it in the remaining of the chapter.

4.2.3 Static Information to Support the Run-Time System

Garbage collection and exception handling require two kinds of information from the compiler. The first kind of information is available to the front end:

- which C-- function parameters and local variables are heap pointers;
- which exception handlers are in scope at which program points;
- which destructors are in scope at which program points.

The second kind of information is only available to the back end:

- whether each local variable and parameter is live, where it is located (if live), and how this information changes as the program counter changes;

[2] Alternatives that avoid the need to advance threads are to make every machine instruction a gc-point (Stichnoth et al. 1999) or to scan the topmost activation of every thread conservatively (Barabash et al. 2001).

[3] Exception handling can also be implemented in the generated code, without support from a run-time system. Ramsey and Peyton Jones (2000) briefly survey different exception handling mechanisms used in several language implementations.

- how to pass parameters to an exception handler or to a destructor;
- the locations where the non-volatile (callee-saves) registers are saved, if any;[4]
- the size of each activation.

If C is used as a portable assembly language and garbage collection is required, a conservative collector (Boehm and Weiser 1988) must be used. If, on the other hand, the compiler targets JVM code, our programs must use the garbage collector provided by a JVM. The problem is that JVM implementations are optimised for the allocation patterns of Java, which are often different from those of other languages. For instance, functional languages allocate memory at higher rates than Java (Wakeling 1999).

C--'s run-time system provides access to the information that the back end generates, through a well-defined interface. Using C--'s run-time system, a garbage collector can locate pointers in the stack unambiguously and an exception dispatcher can unwind it. The only intimate cooperation required is between the C-- compiler and its run-time system; the front end works with C-- at arm's length, through a well-defined language and a well-defined run-time interface.

4.3 The C-- Run-Time Interface

This section presents the original C-- run-time interface, as proposed by Peyton Jones et al. (1999). Using this interface, a front-end run-time system can walk the call stack of the program and inspect the contents of the individual activation frames.

The state of a suspended C-- computation consists of a logical stack of function activations[5] and some saved registers. In the C-- run-time interface, an activation frame on the stack is represented by an *activation handle*, which is a value of type `cmm_activation_T`.

Callee-saves registers that logically belong to one activation are not necessarily stored with that activation; they may be stored in the physical frame of an activation that is arbitrarily far away.

[4] A compiler might choose not to use any of the machine registers in a callee-saves fashion.

[5] This logical stack can be implemented using the system stack, but also by other means, such as allocating activations in the heap.

C-- can support high-level run-time services, such as garbage collection, as follows. When garbage collection is required, control is transferred to the front-end run-time system. The garbage collector can navigate the stack using a function to start at the topmost activation (`cmm_top_activation`) and a function to move one activation down (`cmm_callers_activation`); the latter also signals when the bottom-most activation is reached. In each activation on the stack, the garbage collector finds the location of the local variables using `cmm_find_var`. An exception dispatcher can use these same functions to unwind the stack.[6]

The activation handle abstraction hides the following machine-dependent details:

- the layout of an activation frame,
- the direction in which the stack grows within the target machine's address space, and
- the details of manipulating callee-saves registers.

C-- hides the complexity of walking the stack behind the following two functions:[7]

> `void cmm_top_activation(cmm_activation_T *act)`. Before control is transferred to the front-end run-time system, the state of a C-- program is captured by making a call to the C-- run-time system. `cmm_top_activation` uses that state to initialise activation handle `act` to refer to the topmost C-- activation on the call stack.
>
> `int cmm_callers_activation(cmm_activation_T *act)` modifies activation handle `act` to refer to the activation to which control will return when `act` returns. `cmm_callers_activation` returns zero if there is no such activation frame, which means that `act` already refers to the bottom-most activation on the C-- stack.

There is also a function that allows inspection and modification of the variables of a C-- function:

[6] Additional functions are needed to transfer control to an exception handler. These are described by Ramsey and Peyton Jones (2000) and will not be discussed here.

[7] Without loss of generality we assume that the front-end run-time system is implemented in C. Thus, I describe here a C interface. C-- implementations can provide interfaces with languages other than C.

`void *cmm_find_var(cmm_activation_T *act, int var_index)`
asks an activation handle for the location of a particular parameter or local variable in the activation. `cmm_find_var` returns the address of the specified variable. The front end is thereby able to examine or modify the value. `cmm_find_var` returns `NULL` if the variable is dead.

Notice that `cmm_find_var` always returns a pointer to a memory location, even though the specified variable might be held in a register at the moment at which garbage collection is required. But by the time the garbage collector is walking the stack, the `C--` implementation must have stored all the registers away, and it is up to the `C--` run-time system to figure out where the variable is, and to return the address of the location holding it.

`C--` supports the collector only in locating pointers in the `C--` call stack. Other tasks, such as finding pointers in heap objects, can be managed entirely by the front-end run-time system (allocator and collector) with no support from `C--`. To find roots stored in global variables, the collector and the front end can establish a private protocol and no further support is required from `C--`. For example, the front end can arrange to deposit global roots in a special data section.

Figure 4.3 shows a refinement of the algorithm of Figure 4.1 that uses the functions of the `C--` run-time interface.

It remains that we describe how the garbage collector knows which variables are roots.

4.3.1 Determining Which `C--` Variables are Roots

Suppose the garbage collector is examining a particular activation frame. It can use `cmm_find_var` to locate a particular variable. But how can it know whether that variable is a pointer? (See the fragment of pseudo-code in Figure 4.3 that reads "`if (`i^{th} *variable is a pointer into the heap*`)`".) This information is known only to the front end, which must make it available to the garbage collector. The following mechanism is proposed by Peyton Jones et al. (1999):

- The front end builds a static initialised data block—a *front-end gc-descriptor*—that says which `C--` parameters and local variables correspond to heap pointers in the source program. The *format* of the

```
typedef void *pointer;

void gc_scan_stack() {
  cmm_activation_T act;

  cmm_top_activation(&act);
  for (;;) {
    int i;
    for (i = 0; i < number of variables in act; i++) {
      if (i^th variable is a pointer into the heap) {
        pointer *rootp = cmm_find_var(&act, i);
        /* Trace GC root, if live */
        if (rootp != NULL) gc_forward(rootp);
      }
    }
    cmm_callers_activation(&act) || break;
  }
}
```

Figure 4.3: A simple stack scanning function, targeted to C--.

gc-descriptor is private to the front end and its run-time system; C-- neither knows nor cares.

- The front end uses special C-- syntax to associate a range of program counters to a gc-descriptor.
- The C-- run-time system provides a function to map a gc-point to the corresponding gc-descriptor (`cmm_get_descriptor`).

For illustration purposes, let us assume that front-end gc-descriptors have the following type:

```
struct front_end_gc_descriptor {
  unsigned var_count;
  unsigned char is_ptr[1];
};
```

This represents a list of booleans, one for each parameter and local variable. The list is encoded as an array of variable length with `is_ptr[i]` being the value of the i^{th} boolean and `var_count` the length of the array.[8] (A

[8] An array of variable length can be implemented in C by defining an array type of size one element and assigning to a value of this type a dynamically-allocated array of the desired size.

production compiler would of course use a more compact encoding than a character for every boolean, for instance a bit map.)

When the garbage collector scans the stack, it calls the following C-- run-time function:

`void *cmm_get_descriptor(cmm_activation_T *act)`,

which returns the address of the gc-descriptor of the gc-point where `act` is suspended. The garbage collector finds the live pointers in an activation as follows. For each local variable, it determines whether it is a pointer by consulting the front-end gc-descriptor. For those variables that are known to be pointers, it gets their address by calling `cmm_find_var`.

The code of a sample function to scan the stack is shown in Figure 4.4. (This implements the algorithm of Figure 4.1.)

```
struct front_end_gc_descriptor {
  unsigned var_count;
  unsigned char is_ptr[1];
};
typedef void *pointer;

void gc_scan_stack() {
  cmm_activation_T act;

  cmm_top_activation(&act);
  for (;;) {
    struct front_end_gc_descriptor *d;
    int i;

    d = cmm_get_descriptor(&act);
    for (i = 0; i < d->var_count; i++) {
      if (d->is_ptr[i]) { /* if i^th local is a root */
        pointer *rootp = cmm_find_var(&act, i);
        /* trace root, if live */
        if (rootp != NULL) gc_forward(rootp);
      }
    }
    cmm_callers_activation(&act) || break;
  }
}
```

Figure 4.4: Using front-end gc-descriptors to determine which C-- variables are roots.

In essence, the construction of pointer maps for garbage collection is split between front end and back end, and the two sources of information are combined *at run-time, during garbage collection.* The back end emits static descriptors that map live C-- variables to their run-time locations. The front end emits static descriptors that map all C-- variables to booleans. Tight coupling is required between front end and back end, since they *must use the same numbering scheme* for C-- variables in their respective descriptors. A conceptual view of this architecture is shown in Figure 4.5. In the figure, dashed arrows depict interactions that occur at run-time.

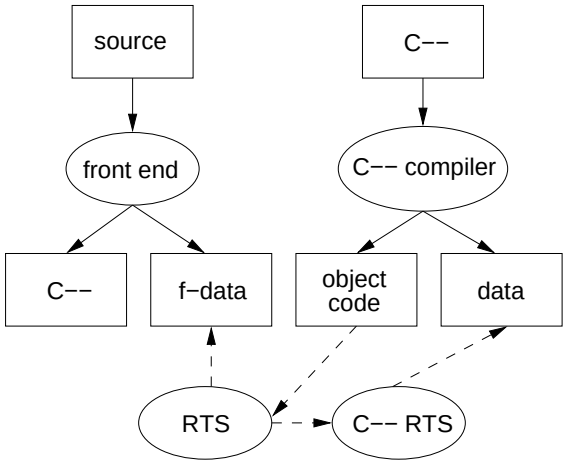


Figure 4.5: Architecture of compiler and RTS using the C-- interface of Peyton Jones et al. (1999).

4.3.2 Implementing the C-- run-time interface

An activation handle (of type `cmm_activation_T`) is an abstract type whose contents can only be accessed through the functions in the C-- run-time interface. It can be implemented as a record consisting of a pointer to an activation frame on the stack, together with pointers to the locations containing the values that the callee-saves registers had at the moment when control left the activation. The shape of an activation handle varies between machines, since different architectures have different number of registers. By making the activation handle an abstract type, this detail is hidden from the front end run-time system.

The function `cmm_top_activation` initialises an activation handle to point to the topmost activation frame on the C-- stack and to the private

locations in the C-- run-time system that hold the values of callee-saves registers.

The C-- code generator builds a statically-allocated table of *activation descriptors*, indexed by safe point. Each activation descriptor contains:

- The size of the activation frame. (`cmm_callers_activation` needs this to move to the next activation record in the call stack.)
- The locations of live local variables. The location of a live variable might be an offset within the activation frame, or it might be the name of a callee-saves register. `cmm_find_var` uses this location to find the address of the true memory location containing the variable's value—either an address within the activation frame itself, or the address of the location holding the appropriate callee-saves register, as recorded in the activation handle.
- The locations where any modified callee-saves registers have been saved. `cmm_callers_activation` uses this information to update the pointers to callee-saves-registers in the activation handle.
- The location of the return address. `cmm_callers_activation` uses it to index into the table of activation descriptors, giving the descriptor of the next activation in the stack.

Figure 4.6 shows the whole machinery in action during a traversal of the call stack. The activation handle combines the static information in an activation descriptor with the dynamic value of the stack pointer and the locations of saved registers. Through the activation handle, the front-end run-time system can access the local variables of an activation. In the figure, the handle refers to an activation that, when it was suspended, had two live locals. `local-1` was stored in a slot in the activation frame and `local-2` was assigned to callee-saves register number 2. The contents of this register was later saved by another function in its own activation frame, in a location tracked by the activation handle. Similarly, the current function uses callee-saves register 2 to hold `local-2` and saves its previous contents in its own activation. Nothing is assigned to callee-saves register 1 by this activation, so there was no need to save it. However, the location where callee-saves register 1 was most recently saved is still tracked by the activation handle;

an activation suspended further down in the stack might have assigned a local to this register.

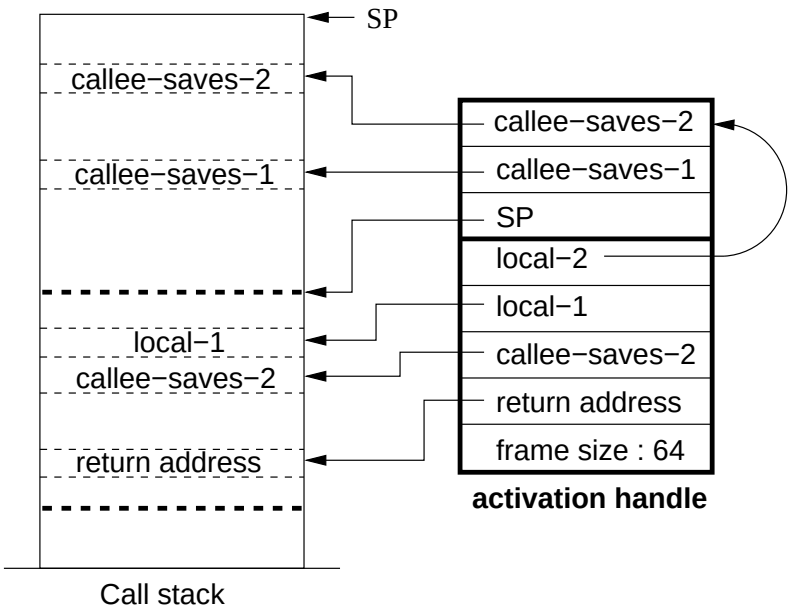


Figure 4.6: During traversal of the stack, the local variables of a suspended activation are accessible via the activation handle.

The C-- run-time interface described in this section has some limitations that hinder the implementation of certain services of the high-level run-time system. In the following sections I describe these limitations and propose an improved C-- run-time interface that fixes the problems. More specifically, the issues that are addressed are: support for generational stack collection (Cheng et al. 1998) (Section 4.4), a more efficient method of locating GC roots for accurate garbage collection (Section 4.5), and support for stack walking in the presence of foreign calls and callbacks (Section 4.7).

4.4 C-- Run-Time Support for Generational Stack Collection

In many programs, most objects live a very short time, while a small percentage of them live much longer (Wilson 1994). Objects with a long life survive many collections and are copied at every collection, over and over. This is a major source of inefficiency in garbage collectors. *Generational collection* (Lieberman and Hewitt 1983; Ungar 1984) avoids much of this

copying by segregating objects into multiple areas by age, and collecting areas containing older objects less often (Wilson 1994).

Cheng et al. (1998) have observed that in programs with deep stacks, the cost of scanning all the activations can become the dominant cost of garbage collection. Since most deep stacks are not frequently unwound, most of the old stack frames are unchanged across successive collections. If the collector can determine which stack frames are unchanged, then the cost of root scanning can be reduced by reusing the information from the previous collection. Cheng et al.'s *generational stack collector* identifies unchanged frames as follows:

> Every time the stack is scanned, the collector changes the return address of every n^{th} stack frame to a special stub function, while recording the original return address in a table. (In their experiments, they use a value of 25 for n.) When the collector encounters a marked frame, it knows that its root information has not changed since the previous collection. It is now safe to stop scanning the rest of the stack. The stub function simply returns to the address stored in the table.

A generational stack collector that uses C--'s run-time interface needs to inspect and modify the return address stored in an activation frame. The original C-- interface provides functions to inspect and modify only the locals of a suspended activation. An additional function is necessary that provides the same level of access to the return address of an activation. The following function serves this purpose:

```
void **cmm_saved_return_address(cmm_activation_T *act)
```

`cmm_saved_return_address` asks activation handle `act` for the location of the return address in the activation frame to which `act` refers. Figure 4.7 shows a simple function that implements generational stack scanning, targeted to C--'s run-time interface.

4.5 Compile-Time Pointer Maps

I have discovered that the two sources of information required to construct pointer maps can be combined at compile-time, rather than at run-time,

```
void gc_generational_scan_stack() {
  cmm_activation_T act;

  cmm_top_activation(&act);
  for (;;) {
    void **loc_ret_addr;

    loc_ret_addr = cmm_saved_return_address(&act);
    /* Stop scanning now if this activation has been
       previously marked */
    if (*loc_ret_addr == addr_of_stub_function) break;
    /* Record return address and mark this activation */
    record_in_table(*loc_ret_addr);
    *loc_ret_addr = addr_of_stub_function;
    /* Scan this activation */
    <...>
    /* Move to activation of caller, if any */
    cmm_callers_activation(&act) || break;
  }
}
```

Figure 4.7: Generational stack collection targeted to C--.

resulting in improved performance, smaller object code size and a simpler interface between the front-end run-time system and C--.

To generate consolidated pointer maps in the same way as monolithic compilers do (Diwan et al. 1992), we can proceed as follows:

> Instead of constructing the pointer map as two separate structures, one emitted by the front end and the other emitted by the C-- compiler, the front end can express, in C-- syntax, which C-- variables need to be included in the pointer maps. Then, the C-- compiler emits a unified map that merges the two sources of information.

This scheme has some important advantages over the method proposed by Peyton Jones et al. (1999):

- A source of artificial coupling between the front-end run-time system and C-- disappears, namely the need to use a specific numbering scheme for C-- variables.
- The implementation of the front end is simpler. Front-end gc-descriptors are not required for garbage collection. The front-end implementor does not need to devise a space-efficient encoding for descriptors, nor write the code that emits them.
- Garbage collection is more efficient. Scanning each activation involves less work.
- The size of the program's object code is smaller. Only one kind of static descriptor is required and it includes less information (only live pointers vs. all variables).

The new code to scan the stack using the simplified interface is shown in Figure 4.8. This code uses a function that has not been introduced before:

> `int *cmm_var_count(cmm_activation_T *act)` returns the number of live pointers referred to by an activation handle.

Compared to the original code of Figure 4.4, there are several improvements:

```
void gc_scan_stack() {
  cmm_activation_T act;
  cmm_top_activation(&act);
  for (;;) {
    int i;
    for (i = 0; i < cmm_var_count(act); i++) {
      pointer *rootp = cmm_find_var(&act, i);
      /* only live roots reported by cmm_find_var */
      gc_forward(rootp);
    }
    cmm_callers_activation(&act) || break;
  }
}
```

Figure 4.8: Garbage collector targeted to C-- using unified pointer maps.

- The loop count is smaller. The value returned by `cmm_var_count` is the number of locals that are pointers and are live at the gc-point, rather than the total number of locals of the C-- function. Accordingly, `cmm_find_var(&act,i)` returns the address of the i^{th} variable tracked by activation descriptor `act`. Notice that this differs from the behaviour of `cmm_find_var` in the original interface (Section 4.3), where `cmm_find_var` returns the address of the i^{th} variable of the procedure.

- Two conditional tests are eliminated: the check for "pointer-ness", and the check for liveness.

- The call to `cmm_get_descriptor` is eliminated.

Without these optimisations, there could be noticeable garbage collection overheads in language implementations that use C--. In some cases, there may be many active activations in the call stack when a collection is necessary. (Cheng et al. (1998) have observed that, in programs with deep recursion, there can be more than 4000 activations in the call stack.)

Furthermore, since the pointer maps are now smaller, the size of the program's object code is reduced.

It remains that I describe the syntax that the front end uses to indicate what C-- formal parameters and local variables that correspond to heap pointers in the source language. It suffices to add a qualifier to each decla-

ration, as follows:

```
f(bits64 arg1, bits64 gc_root arg2) {
  bits64 gc_root local1;
  bits64 local2;
  ...
}
```

In this example `arg2` and `local1` have been marked using the keyword `gc_root`. The C-- compiler will record information only about these two locals in any pointer maps that it emits for function `f`.

An alternative to the `gc_root` qualifier would be to add reference types to C--. Below I discuss why this does not work in all cases.

Ambiguous Types at Compile-Time

Sometimes it is not known at compile-time whether a variable is a pointer. For instance, in the TIL compiler (Tarditi et al. 1996), whether a function parameter is a pointer may depend on the *run-time* value of another parameter. That is, in order to support garbage collection in C--, it is not enough to add reference types to C--: in the TIL compiler variables other than pointers need to be inspected by the garbage collector. When types are not known at compile-time the front end must annotate as `gc_root` every C-- local that *might* need to be inspected or modified at run-time. C-- provides access to the run-time location of all marked variables; it is up the front-end run-time system to determine what is a pointer and what is not.

4.5.1 Support for Debugging

In some situations, the front-end run-time system may want to inspect or modify C-- variables other than live pointers. This happens, for instance, in a debugger targeted to the C-- run-time system (Peyton Jones et al. 1999). In this case, the C-- compiler needs to emit debugging descriptors that give the location of *all* parameters and local variables of a function, not only heap pointers. But this can be requested when it is needed, with an argument to the C-- compiler.

The insight behind the interface proposed in this chapter (in contrast to the original C-- interface) is the following:

Back-end descriptors for garbage collection are independent from back-end descriptors for debugging.

In the new setting, the costs of debugging are paid only when this service is required; the performance of garbage collection is not affected in any significant way.

4.6 The Problem of Language Interoperability

An increasing number of high-level programming languages provide a foreign-function interface (FFI) that enables interoperability with code written in other languages—typically C. A FFI gives access to the wealth of libraries that have been developed in C. In fact, it can be argued that providing a FFI is becoming essential for the success of a modern high-level programming language (though not a sufficient condition). A few language implementations that provide a FFI are Java (Liang 1999), Caml (Leroy et al. 2002), SML (Blume 2001), Haskell (Finne et al. 1998), Scheme (Serrano 1994) and Mercury (Henderson et al. 2002), but the list goes on.

FFIs support, at least, calls from the source language to the foreign language, but many of the recent systems also support *callbacks* from the foreign code to the source language. For instance, the Objective Caml implementation provides means to interface with C code, including callbacks to Caml. The API includes functions to create Caml objects and to throw Caml exceptions from C code. The Glasgow Haskell Compiler (GHC) (GHC Team 2001) supports calls to and from C, and so do the Mercury Compiler, the Bigloo Scheme compiler (Serrano 1994) and the GNU Java compiler (FSF 2001).

4.6.1 Foreign Function Interfaces and Stacks

A critical implementation decision for a foreign function interface is how the call stacks of the high-level language and the foreign language are laid out. This has a direct impact on the implementation of run-time services that need to walk the call stack, like garbage collection and exception handling. The two alternatives are:

- The high-level language and the foreign code use the same call stack.

- The foreign code uses a separate call stack.

Currently, most implementations use one stack, since it makes the rest of the run-time system simpler. In what follows, I will call this the *single-stack model*.

4.6.2 Foreign Calls and Garbage Collection

FFI implementations that use a single stack model use one of the following methods to implement garbage collection:

1. **Conservative scanning of the whole stack** (both source and foreign activations). Mercury, Bigloo and the GNU Java compiler do this. This is also the method of choice for garbage-collected languages that use C as a portable assembly language.

2. **Accurate scanning of the source activations only.** The Caml garbage collector scans Caml activations accurately but skips foreign frames. (The foreign code may still hold pointers into the Caml heap, but it must "register" their location by calling a function in the Caml run-time system.)

3. **Hybrid approaches**. The Java run-time system described by Barabash et al. (2001) uses *mostly accurate stack scanning*: all foreign activations and the topmost activation of every thread are scanned conservatively; the remaining activations are scanned accurately. The main advantage of this scheme is that, when a collection is required, threads can be stopped anywhere, and there is no need to advance their execution until a safe point.

To scan the whole stack conservatively, the front-end run-time system does not require support from C--. However, to implement accurate or mostly accurate scanning, the front end run-time system requires the services of C--'s run-time system.

4.6.3 Foreign Calls and Exception Handling

Ramsey and Peyton Jones (2000) survey popular alternatives for implementing exception handling. Stack cutting, native-code stack unwinding and continuation-passing style do not require support from the run-time

system. Run-time stack unwinding, on the other hand, is implemented with a stack-walking mechanism.

As for garbage collection, when the stack contains foreign activations, the policy of the front end may be to skip them, or to traverse them by calling a foreign unwinder. For instance, if the foreign language were C++, its semantics requires that destructors be invoked when locals go out of scope.

4.7 C-- Run-Time Support for Foreign Calls

The C-- stack-walking interface proposed by Peyton Jones et al. (1999) does not support walking a stack that contains foreign activation frames. Here, I extend the interface to make this possible.

Scanning a stack with foreign activations can be supported by C-- as follows:

- At every foreign call, the C-- compiler emits code that saves (somewhere) the state of the C-- computation. (A precise definition of what is included in this state appears in Section 4.7.2.)
- When `cmm_callers_activation` cannot find the caller of the current C-- activation, it determines whether the caller is a foreign activation (a callback from foreign code to C-- code), or whether we have reached the bottom of the stack. If the former, `cmm_callers_activation` queries the saved state to continue the traversal in the C-- activation that made the foreign call. Otherwise, there are no more activations and the stack traversal is complete.

However, this implements only one of the alternatives of Section 4.6.2, namely "accurate scanning of source activations only". It is not sufficient if a garbage collector wishes to do mostly accurate scanning (Barabash et al. 2001), for instance. Furthermore, it imposes the cost of saving the state *every time* the program makes a call to foreign code. Some language implementations may allow foreign calls, but no callbacks; or the front end may know that certain foreign calls make no callbacks, even if the source language allows it. In either of these cases, it is wasteful that every foreign call pay the cost of saving a state that will never be consulted. What C--

should provide is a *mechanism* to save the state at selected foreign calls, according to *policies of the front end*.

Below I extend the C-- run-time interface so that a stack walker is aware of the boundaries between sequences of C-- and foreign activations. With knowledge of these boundaries, the stack walker can *either skip the foreign activations or walk them using mechanisms external to C--.*

4.7.1 Walking a Mixed Stack

Before giving the full details of how the C-- state is saved when there is a call to foreign code, or what exactly needs to be saved, it is useful to see the code that walks a stack containing foreign activations. I show code for both jumping over and traversing the sequences of foreign activations. Since the purpose of the examples is to show how stack navigation is done, it is left unspecified what the front-end run-time system does to each C-- activation that it visits.

How to Skip Foreign Activations

```
cmm_activation_T act;

cmm_top_activation(&act);
for (;;) {
  examine activation act
  if (cmm_callers_activation(&act) == 0) {
    /* Caller is not a C-- function. Reached bottom of a
       sequence of C-- activations. Move to previous C--
       sequence, if any */
    cmm_saved_activation(&act) || break;
  }
}
```

In this example, the run-time system skips foreign activations. When the caller of a C-- function is foreign (a callback from foreign code to C--), the traversal is continued at the previous known C-- sequence (if any), by calling `cmm_saved_activation`. (Notice that the loop always terminates. `cmm_callers_activation` either updates the reference `act` and returns 1, or leaves the reference unchanged and returns 0.)

How to Traverse Foreign Activations

If, instead of skipping the foreign activations, the run-time system must traverse them (perhaps with a conservative collector, as in mostly accurate scanning (Barabash et al. 2001)), we need to know their limits in the stack. This is depicted in Figure 4.9.

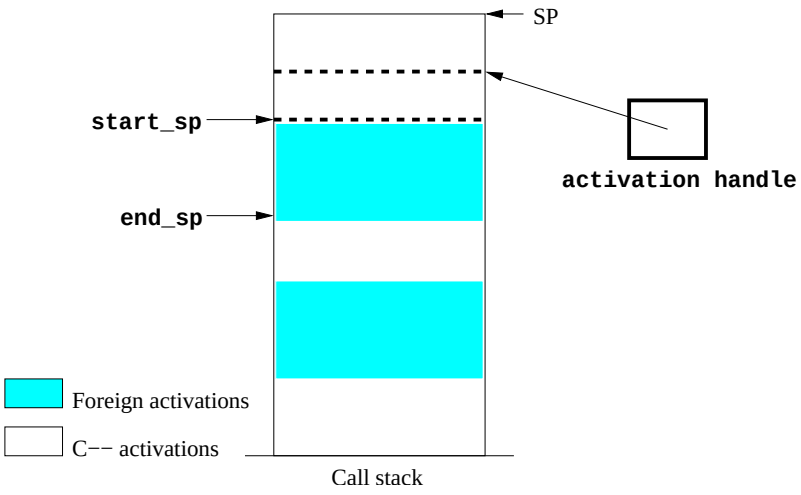


Figure 4.9: Stack limits of a sequence of foreign activations.

Two new functions in the C-- run-time system return the values of the stack pointer that delimit the foreign activations (`start_sp` and `end_sp` in the figure). `cmm_callers_sp(&act)` returns the stack pointer of the activation that `act` will return to. `cmm_saved_sp()` returns the stack pointer of the C-- activation that made the foreign call. (`cmm_saved_sp` returns `NULL` if there no such activation. This signals that the stack traversal is complete.)

Figure 4.10 shows example code that uses these limits to scan a sequence of foreign activations.

Implementation

In order to continue the traversal, `cmm_saved_activation` consults some state—a C-- *context*—that was saved just before the foreign call was made. (The means to save a C-- context will be described later.)

Because there may be more than one sequence of foreign activations in the call stack that the run-time system is traversing, the C-- run-time system maintains *a stack of saved contexts.* Figure 4.11 depicts the contexts

```
cmm_activation_T act;

cmm_top_activation(&act);
for (;;) {
  examine activation act
  if (cmm_callers_activation(&act) == 0) {
    /* Caller is not a C-- function. Reached bottom of
       a sequence of C-- activations. Walk foreign
       activations, if any */
    void *start_sp = cmm_callers_sp(&act);
    void *end_sp = cmm_saved_sp();
    foreign_walk(start_sp, end_sp);
    /* Continue visiting C-- frames, if any */
    cmm_saved_activation(&act) || break;
  }
}
```

Figure 4.10: Code to walk foreign activations.

stack at an execution point where there have been two foreign calls, both of which made a callback to C-- code.

During the traversal, one of the saved contexts is designated as *current*. Initially, `cmm_top_activation` makes the top saved context the current one.

The call `cmm_saved_activation(&act)` updates activation handle `act` to point to the activation captured by the current C-- context, and moves `current_context` down one position. Figure 4.12 depicts the situation during a stack traversal, before and after `cmm_saved_activation` is called.

`cmm_saved_activation` returns zero when the saved contexts stack is empty. This signals that the stack does not contain any more C-- activations under the one that `act` points to. The traversal is now complete.

Storage for the Stack of C-- Contexts

The memory to hold the stack of C-- contexts is managed by the C-- run-time system without further support from the high-level run-time system. C-- can use its own heap or it can use the call stack as the allocation area for C-- contexts. In the latter scheme, a context is saved in the activation frame of the C-- function that made the foreign call. (A pointer to the previous context is also saved in that activation, in order to implement a stack by means of a linked list.)

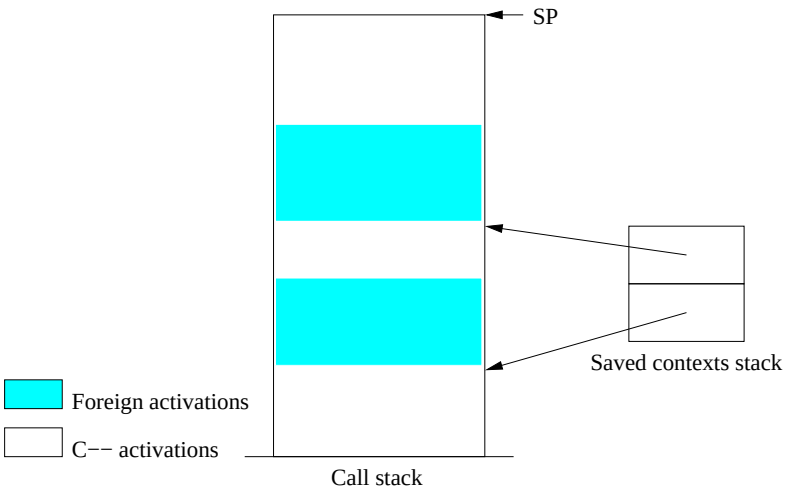


Figure 4.11: A stack containing foreign activations and its associated saved contexts stack.

4.7.2 Saving the C-- State

Finally I describe exactly what state is contained in a C-- context and what interface the C-- run-time system provides to save this state in its private *saved contexts stack*.

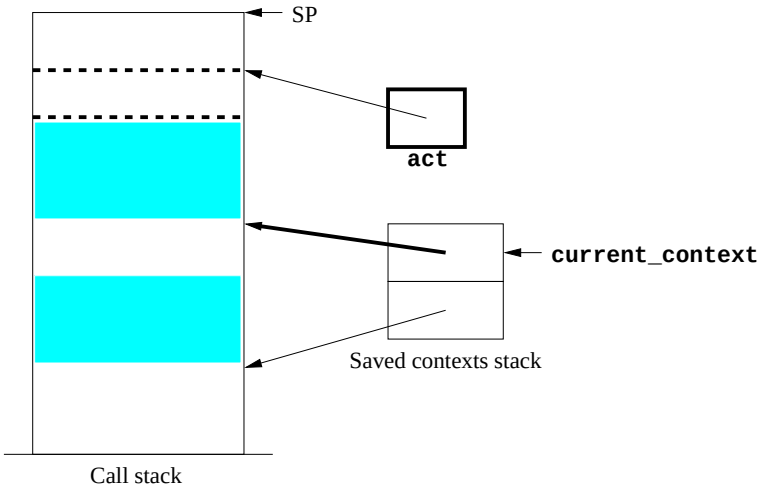
The State in a C-- Context

A C-- context must capture the state of a C-- activation that calls foreign code. More precisely, it must capture the state at the program counter where control leaves that C-- activation. The C-- state that needs to be saved is everything that `cmm_saved_activation` will need to initialise an activation handle, namely the following:

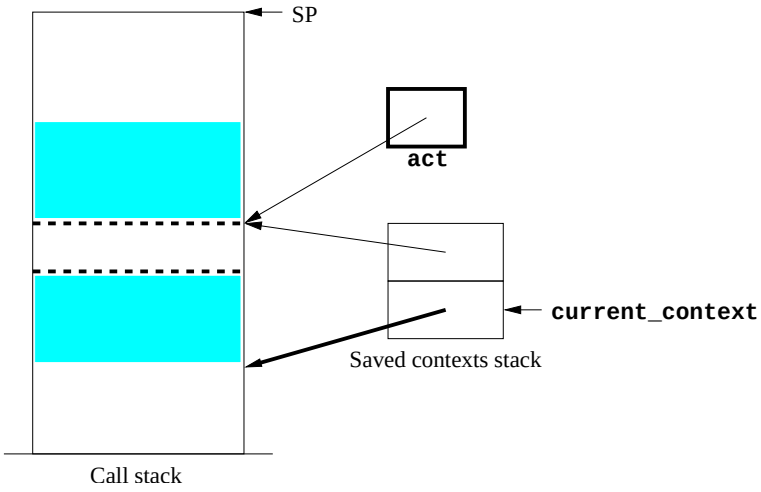
- The (static) activation descriptor of the C-- activation.
- The (dynamic) value of the stack pointer of the C-- activation.
- The (dynamic) locations where the callee-saves registers have been saved, if any.

Hence, the saved context must contain the following:

- The return address of the foreign call. This return address is a safe point and there exists an activation descriptor for this address, emitted



(a) Before calling `cmm_saved_activation`.



(b) After calling `cmm_saved_activation`. `act` points to the activation captured by `current_context`. `ccurrent_context` is updated.

Figure 4.12: Traversing a stack that contains foreign activations.

by the C-- compiler. `cmm_saved_activation` will map the return address to that descriptor.

- The value of the stack pointer.
- The values of the callee-saves registers.

Notice that the values of the caller-saves registers are not needed, since they cannot contain anything that will be live after the foreign call returns.

Saving the C-- Context

Before each foreign call that might be suspended while the front-end run-time system runs, a context must be pushed. When the call returns, the context must be popped. The C-- run-time system provides two functions to push and pop the C-- state, respectively. A foreign call that might be suspended while the run-time system runs must be wrapped between calls to these two functions, as shown below.[9]

```
cmm_push_state();
r = foreign "C" f(expr_1,...,expr_n);
cmm_pop_state();
```

Notice that we need to push a context that reflects the C-- state when control leaves the C-- activation at the foreign call, and this sequence pushes a context that reflects the state when control leaves the activation at the call to `cmm_push_state` itself! Perhaps the values of the callee-saves registers and the stack pointer are not the same at these two program points. Fortunately, this is not the case. Neither `cmm_push_state` nor the code to pass parameters to `f` overwrite the callee-saves registers. The stack pointer remains unchanged as well; even if the foreign call requires parameters to be passed on the stack, the C-- compiler can arrange it so that this does not require modifying the stack pointer during call setup.

Alas, the context must include also the return address of the foreign call. Hence, this address must be a parameter of `cmm_push_state`. The problem is that, in general, it is not possible to write a label in source C-- that refers exactly to the return address of a call. For instance, given this code:

[9] Recall that the qualifier `foreign "C"` tells the compiler that it should use the C calling convention for the parameters and result of the call.

```
r = foreign "C" f(expr_1,...,expr_n);
L:
```

the C-- compiler generates low-level code similar to this:

```
formal_arg_1 = expr_1
...
formal_arg_n = expr_n
call f
r = formal_ret
L:
```

The label names the address of the instruction following `r = formal_ret`, and not the return address of the call.

What we need is C-- syntax to label the return instruction of a function call. An annotation to call statements can serve this purpose:

```
cmm_push_state(R);
r = foreign "C" f(expr_1,...,expr_n) "return_label" R;
cmm_pop_state();
```

In summary, these two functions and the `return_label` annotation provide all the required functionality. Moreover, for the front end, the annotation is very lightweight in terms of additional notation and, for the C-- compiler, it is straightforward to translate to assembly or machine code.

4.8 Support for Multi-Threaded Languages

Modern programming languages increasingly support multiple threads of execution, either as built-in language constructs or as a library that provides an interface to threads services provided by the operating system.

Typically, the front-end run-time system maintains a *thread control block* (TCB) for every thread that is created by the program. This data structure includes a thread identifier, a pointer to the thread's stack, etc. In a front-end run-time system targeted to C--, the TCB must contain an extra field for use by the C-- run-time system. This imposes negligible space overhead per thread. In addition, the C-- run-time system provides a multi-threading version of the functions that save and consult C-- contexts, namely `cmm_top_activation`, `cmm_push_state`, `cmm_pop_state` and

`cmm_saved_activation`. The multi-threading variants take the field in the TCB reserved for `C--` use as an extra parameter.

4.9 Related Work

The method of traversing a run-time call stack with the aid of descriptors emitted by the compiler has been described many times in the specific contexts of exception handling, garbage collection and debugging. For instance, in his PhD dissertation, Boquist (1999) describes a compiler for the functional language Haskell where the back end exports a function `gc_find_roots()` that the garbage collector uses to locate roots in the stack. The novelty in `C--` is that it provides an interface that is language-independent and can be used to support a variety of high-level run-time services.

Hudson et al. (1991) have implemented a language-independent garbage collector toolkit. The toolkit provides a generational garbage collector. Language implementations that use the toolkit must provide functions that implement language-specific functionality. These include a function to copy an object in memory and a function to find pointers inside an object. In addition, the language-specific part is responsible for locating all root pointers in the stack and in global memory.

Wilson and Johnstone (1993) have developed a real-time garbage collector that is suitable for any garbage collected language implementation. According to (Wilson 1994), it has been adapted for use with C++, Eiffel, Scheme, and Dylan.

Microsoft has recently developed a common intermediate language and a common language run-time system (Meijer and Gough 2001), with similar goals of those of the Java virtual machine. In this system, though, there is more emphasis on supporting multiple programming languages. All languages share the same run-time system and execute in the same virtual machine. Nevertheless, the intermediate language is a substrate to implement object-oriented languages. As such, its run-time system is optimised for that case.

Chapter 5

Compiling C--

This chapter describes the implementation of cmmc, a compiler for C--. It divides into two sections, describing translation of C-- into assembly code and optimisations, respectively.

5.1 Compiling C--

cmmc uses standard compiler technology, a lexer generator (ML-Lex (Appel et al. 1994)) and a parser generator (ML-yacc (Tarditi and Appel 2000)), to build an abstract syntax tree. Next, cmmc performs simple type-checking to detect erroneous C-- constructs, like using operators with values of the wrong type. Finally, it uses the generic code generator MLRISC (George and Leung 2000) to perform instruction selection, to perform register allocation and to generate assembly language.

cmmc implements only some of the low-level optimisations listed in Section 1.1. Nevertheless, the performance of the resulting code is satisfactory. (Chapter 7 presents a performance evaluation.)

Much of the work performed by cmmc consists of standard code generation technology. This chapter describes only the following C--specific issues:

- The calling convention that C-- must use, given the requirement to support tail-call optimisation.
- The differences between C-- continuations and exception handlers in high-level languages. The compilation of C-- continuations.

- A novel way of implementing reserved machine registers.

5.1.1 The C-- Calling Convention and Tail Call Optimisation

A calling convention specifies what machine resources (registers or memory locations) are used to pass parameters to functions and to return results. A calling convention also specifies what registers must be preserved by the called function (*callee-saves* registers) and which ones may be overwritten (*caller-saves*), but this will not be discussed further here.

Compiler writers should use the most efficient calling convention available in the target architecture. In this section I demonstrate that the combination of two features of C-- (tail-call optimisation and separate compilation) disallow the use of the most efficient convention in the general case. Section 5.2.3, Optimising Overflow Parameters of Known Functions, describes special cases where C-- functions can use the optimal convention.

Description of Calling Conventions

In the x86 architecture the convention is to pass all parameters on the stack. Newer architectures tend to have more registers than the x86 (32 or more, while the x86 has only 8) and since accessing memory is more costly than accessing registers, manufacturers specify calling conventions that use some of the registers to pass parameters (and return results). The remaining parameters are passed in memory, starting at a known offset from the stack pointer. Parameters that are passed in memory, according to the calling convention, will be referred to as *overflow* parameters.

Below I describe different standard conventions for passing overflow parameters. We will see that some calling conventions are more efficient than others.

To show what the different conventions specify, I will use as an example a call to a function `f` with two overflow parameters. The actual values of the parameters are `x` and `y`. It is assumed that 4 bytes are required to hold each value, that the first overflow parameter must be placed at the top of the stack, and that the stack grows from high addresses to low addresses. The code to pass the parameters that go in registers is not shown.

- **Convention 1:** The caller pushes overflow parameters before the call and pops them after the call.

```
sp <- sp - 8
[sp]<- x
[sp+4] <- y
call f
sp <- sp + 8
```

This is the convention of the x86 on Unix. The cost is two instructions at each call point to adjust the stack pointer. However, the x86 also provides a `push` instruction, which decrements the stack pointer automatically. If a `push` instruction is used, the cost in the x86 is one instruction per call. In a variant of this convention, GCC lets the arguments of all the calls in the same basic block pile up on the stack and deallocates them with a single increment of the stack pointer after the last call.

- **Convention 2:** The caller pushes overflow parameters before the call, and the callee pops them before it returns.

```
sp <- sp - 8
[sp] <- x
[sp+4] <- y
call f
```

This is the x86 convention on the Windows operating system. The callee can deallocate its arguments together with the rest of its own activation frame, in one single increment. Deallocating in the callee leads to better code density, since the deallocation is not repeated at every call site.

- **Convention 3:** Space for overflow parameters is allocated as part of the caller's frame. The stack pointer does not have to be decremented or incremented at every call site.

```
[sp] <- x
[sp+4] <- y
call f
```

Space is reserved in the caller's frame to pass the overflow parameters to all the calls that the caller makes. This memory area is large enough to hold the overflow parameters of the call with the largest number of

parameters. This space is allocated (resp. deallocated) by the caller at entry (resp. at exit), and not once for every call site. This is the cheapest convention in terms of instruction count, and it is used by many modern processors, including Alpha, Sparc, and PPC.

Tail Call Optimisation

The operational semantics of C-- requires that every function deallocate its activation frame when it makes a `jump`. Clinger (1998) has expressed the space behaviour of tail calls as follows:

> "A tail call does not cause an immediate return, but passes the responsibility for returning from the procedure that performs the tail call to the function it is calling. In other words, the activation of a procedure extends from the time it is called to the time that it performs either a return or a tail call."

With this rule, the space consumed by a sequence of tail calls is constant, instead of linear on the size of the sequence. This allows us to express iterative computations using recursion, and still use constant space (Clinger 1998).

Tail call optimisation is an important optimisation for programs where tail calls are very frequent, such as those written in functional or logic languages. (In fact, it is a *required* optimisation in the Scheme programming language (Abelson et al. 1998).)

Tail-call optimisation is increasingly important in contexts other than functional languages. Recent experimental and theoretical research has shown that some matrix computations that have been traditionally implemented by looping can run faster when expressed with (tail-)recursive algorithms (Yi et al. 2000). Such algorithms naturally exhibit memory locality properties that can be exploited by the multi-level memory hierarchies common in modern microprocessors.

Overflow Parameters in C--

The requirement that the activation of a function be deallocated when the function makes a `jump` constrains the choice of calling convention for overflow parameters of `jump`s. But because a C-- function can be reached both from

normal calls and from tail calls, the same calling convention must be used for both.

While implementing a calling convention for C--, I realised (together with Simon Peyton Jones and Lal George) that, in order to support unrestricted tail call optimisation, C-- cannot use Convention 3. The following example shows why:

```
g() {
  ...
  jump f(1,2,...,20);
}
```

In Convention 3, overflow parameters are passed in the caller's frame (g in the example). On the other hand, the jump requires that g's frame be deallocated before transferring control to f. These two conditions together imply that g can only pass f's overflow parameters in the frame of g's caller (to which f will return). The only possible way of doing this is if *every* caller of g reserves enough space for f's overflow parameters. And indeed for the overflow parameters of anything that f jumps to! In the presence of separate compilation, the compiler cannot know the whole call graph. Therefore, if C-- has to guarantee constant space consumption of computations that use tail-recursion, Convention 3 cannot be used.

In fact, Convention 1 cannot be used either, since there is work to do after the call returns, namely pop the overflow parameters. Since tail calls do not return to their caller (but to their caller's caller) the opportunity to pop the parameters is lost, which means that tail calls grow the stack. This breaks the constant space consumption property.

In conclusion, *in order to support unrestricted tail call optimisation, C-- must use Convention 2 to pass overflow parameters.*

Section 5.2.3 shows how this restriction can be lifted for functions that satisfy certain conditions.

5.1.2 C-- Continuations

Ramsey and Peyton Jones (2000) have proposed *C-- continuations* as a mechanism to support source-language exception handlers in C--.

At run-time, a C-- continuation encapsulates both a control-flow target and a specific C-- activation on the stack. When a continuation is invoked,

all activations above that of the continuation are removed from the call stack before transferring control to the continuation's program counter. Once an activation dies, its continuations die too. Invoking a dead continuation results in a run-time error. In the following code `k` is passed to a function, which can later invoke it.

```
f(bits64 x, float64 y) {
  bits64 w;
  ...
  g(x, k) also cuts to k ;
  /* k may be invoked by g, or by something g calls */
  ...
  return;
  continuation k(w):
  ... code for k, mentioning x, y, w ...
}
```

The annotation `also cuts to k` at `g`'s call site indicates that control might flow from the call directly to `k`. This allows the code generator to build an accurate control-flow graph (Hennessy 1981; Choi et al. 1999).

There are two different ways of invoking a `C--` continuation: *stack cutting* and *stack unwinding* (Ramsey and Peyton Jones 2000). In turn, each one can be implemented in code generated by the `C--` compiler or in code in `C--`'s run-time system. This gives a total of four combinations, each with a different cost model. `cmmc` supports one of them, stack cutting via generated code. The statement `cut to k(`*arguments*`);` transfers *arguments* to conventional locations,[1] truncates the stack to `k`'s activation, and sets the program counter to `k`'s program counter. This is done in constant time, without walking the call stack.

Notice that if the source language is C++, exception handling cannot be implemented with `C--`'s `cut to`. The semantic of C++ requires that object destructors be invoked as soon as the dynamic scope of the object terminates (Stroustrup 1997). When the stack is unwound for exception handling, activations are removed from the stack, terminating the scope of their variables. Destructors may have to be invoked at that time. (In order to support C++, `cmmc` would have to implement stack unwinding.)

[1] These locations are determined by the `C--` implementation.

5.1.3 Run-time Representation of C-- Continuations

At run-time, both a C-- continuation and a source-language exception handler encapsulate a control flow target and a specific activation on the stack. In addition, each has the property that its life span terminates when its activation returns. This suggests that perhaps a C-- compiler can use for C-- continuations the same run-time representation that other compilers use for exception handlers. However, I will show that this is not possible.

The most space-efficient representation for exception handlers is the following. When the scope of an exception handler is entered, its program counter is pushed onto the call stack. Then the value of stack pointer itself serves as the run-time value of the handler. In this representation, a single address captures both the stack pointer and the program counter of an exception handler. This is shown in Figure 5.1.

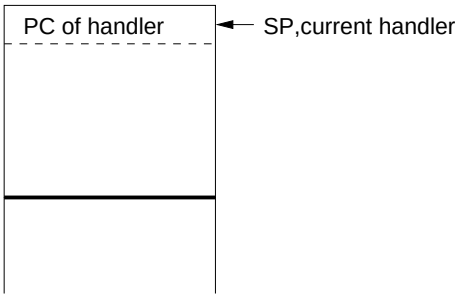


Figure 5.1: An efficient run-time representation of exception handlers.

When an exception is raised, the code to resume execution at the current exception handler is simple and efficient:

```
SP <- current_handler
JMP (MEM[current_handler])
```

This representation hinges on the fact that, at any one time, there is only one *current* exception handler. More precisely, each expression in the scope of an exception handler has two *continuations*:[2] the *normal continuation* (the next instruction after the expression) and the *exceptional continuation* (the code of the exception handler). Exception handlers can be nested, but then the innermost one becomes the current exceptional continuation. When evaluation leaves the scope of the innermost one, the enclosing one

[2]Here, the term "continuation" means "the rest of the computation" (Reynolds 1998), rather than a C-- continuation.

becomes current. In other words, *every expression has only one exceptional continuation.*

In contrast, in C--, more than one continuation may be current or "active" (in the sense that they can be invoked) at one point in the program. Consider this code:

```
f() {
  ...
  g(k1, k2) also cuts to k1, k2;
  ...
  continuation k1():
  ...
  continuation k2():
  ...
}
```

k1 and k2 are both passed to g, which can invoke either. Therefore, at the call site *both* continuations are active. The consequence is that a C-- continuation cannot be represented as compactly as an exception handler.

For each active continuation we must allocate a (program counter, stack pointer) tuple. The continuation value (of the native data-pointer type) is the address of the tuple. Since the life of a continuation is at most that of its activation, the tuple can be allocated in the activation frame of its enclosing function. Figure 5.2 shows the activation frame of f, containing the representations of continuations k1 and k2.

Non-nested exception handlers can use a single memory location to store the value of the current exception handler. Similarly, in C--, two continuations that are not "live" simultaneously can also share the same memory area. To determine if two continuations are live at the same time the C-- compiler may perform a custom liveness analysis on continuations. However, cmmc does not implement such analysis and allocates separate memory in the activation frame for each C-- continuation of a function.

Invoking a Continuation

To cut to a continuation, the following actions must be taken:

1. Pass the continuation arguments.

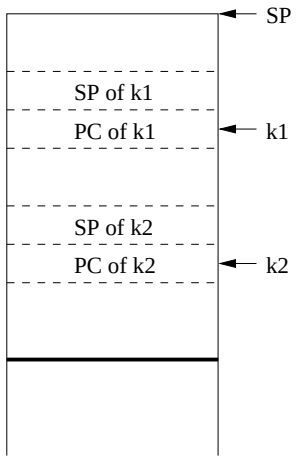


Figure 5.2: Run-time representation of `C--` continuations in an activation of function `f`.

2. Update the stack pointer.

3. Jump to the continuation code.

`cmmc` passes continuation arguments in the same registers and stack locations that it uses to return function results. The translation of `cut to` to machine code follows directly from the run-time representation of continuations. Given:

`cut to (`*expr*`)(`*args*`);`

expr must point to a (program counter, stack pointer) tuple. (`C--` specifies that it is a run-time error to `cut to` an expression that does not evaluate to a `C--` continuation. A `C--` compiler may emit code that tries to check this at run-time, but `cmmc` does not.) To update the stack pointer and jump to the continuation code, `cmmc` emits the following low-level code:[3]

```
/* evaluate expr into a fresh temporary t */
t  <- expr
SP <- MEM[t - 8]
JMP (MEM[t])
```

Compared to a normal function call, the additional cost is two loads and an indirect branch instead of a direct one. As a special case, if the target of a `cut to` is a continuation of the same function activation, as in this code:

[3]For an architecture where addresses occupy 8 bytes.

```
f() {
  ...
  cut to k(1);
  ...
  continuation k(arg):
  ...
}
```

`cmmc` emits the optimal code, namely pass the actual parameters and branch to a label local to `f`.

```
arg <- 1
goto f_k
```

Here, it is assumed that the control flow target of the continuation `k` is represented internally by the `C--` compiler by the label `f_k`.

Initialising Continuation Values

When a `C--` function is invoked, for each of its local continuations, a tuple must be allocated in memory and its contents must be initialised *before the continuation may be invoked*. (Local invocations do not require initialisation, though, since they translate to local branches and the stack pointer remains the same.) There are two reasonable alternatives to initialise a continuation:

- Early, at function entry.
- Lazily, before the continuation can be invoked.

If we initialise the continuation lazily, the initialisation cost (two stores) is paid only along control paths that actually use the continuation. However, this can incur redundant initialisations if the same continuation is used more than once and we initialise before every use. For instance, given the code in Listing 5.3, `k` cannot be invoked in the true branch, and there is no need to initialise in that control-flow path. On the other hand, initialisation must complete before the call to `g`. If initialisation is done immediately before the call, the continuation tuple would be written every time round the loop. Standard back-end optimisations, like code motion and dead store elimination, may be able to eliminate some or all of the redundant stores. If

the C-- compiler does not implement such optimisations, an alternative is to initialise each continuation *in the dominator of all its uses.* cmmc initialises immediately before each use.

```
f(bits64 x) {
  ...
  if(...) {
    h(10);
    return;
  } else {
    loop:
    g(k) also cuts to k;
    ...
    goto loop;
  }
  ...
  continuation k(arg):
  ...
}
```

Figure 5.3: Continuation k cannot be invoked in the true branch.

5.1.4 Reserved Machine Registers

C-- provides support for assigning global variables to specific machine registers. C-- global register declarations provide the same functionality as global register variables in GNU C (Stallman 2001). Common uses of reserved registers by front ends include:

- *Allocation pointer* and *allocation limit* for fast memory allocation in system where the allocation space is a contiguous region of free memory (Appel 1998).
- *Exception pointer*, to hold the current exception handler (Appel 1992).
- Dedicated registers in parts of the run-time system written in assembly language.

The C-- declaration

```
register bits64 global_var "r14";
```

has two effects: it allocates variable `global_var` to register `r14`, and it reserves register `r14` globally in the `C--` module where the declaration appears. That is, even in a `C--` function that does not mention `global_var`, register `r14` will not be used to colour local variables.

All separately compiled `C--` modules must have identical `register` declarations. Otherwise, it is impossible to guarantee that all functions use the same location for a certain global.

`register` declarations name specific machine registers and are, therefore, target-dependent. `C--` programs should avoid reserving registers that are used by the native calling conventions. Typically this includes the stack pointer, the return address register, and the function result register. But some native calling conventions reserve more, like a global pointer to implement position-independent code (Evans and Eckhouse 1999). `C--` compilers should reject programs that try to reserve such registers.

Implementing Reserved Machine Registers

A `C--` compiler can implement reserved registers as follows.

1. References to global variables are directly translated to their physical register, before register allocation.

2. During register allocation, reserved registers are not available to colour any temporary.

3. Local variables are initially assigned to temporaries. Later, they will be assigned to available physical registers or memory by the register allocator.

For instance, this `C--` statement:

```
global_var = global_var + x;
```

translates to this instruction in low-level intermediate code:

```
r14 <- r14 + t245
```

assuming that `x` gets assigned to temporary `t245`. Later, the register allocator will assign `t245` to a physical register or memory location. This is the implementation used by the GNU C compiler.

5.1.5 A Better Implementation

`cmmc` uses a smarter implementation of reserved registers, which allows reserved registers to be used for other variables in parts of the program where the reserved register is dead.

Consider the following code:

```
f(bits64 x) {
 bits64 y;

 y = x+1;
 ... uses y
 global_var = x + 16;
 return;
}
```

Assume that `... uses y` is a section of code that has high register pressure (perhaps an unrolled loop), and that `global_var` is not mentioned in that section of code. Because there are more live variables than registers, something has to be spilled, for instance `y`. However, it would be harmless to assign `y` to register `r14`! What makes this safe is the fact that `global_var` is dead throughout that section of code. Intuitively, if the first thing we do with a reserved register in a function is assign to it, then that means that we do not care about its previous value; therefore it is safe to use the register for other purposes temporarily.

Notice, however, that the same cannot be done for this other function:

```
f'(bits64 x) {
 bits64 y;

 y = x+1;
 ... uses y
 g();
 global_var = x + 16;
 return;
}
```

because `g` may consult the current value of `global_var`. In this case `global_var` is live at the same time as `y`. (The call statement acts as a use of `global_var`.)

`cmmc` implements reserved registers as follows.

1. References to global variables are directly translated to their physical register, before register allocation.
2. Reserved registers are available during register allocation to colour temporaries.
3. When doing liveness analysis of a function, reserved registers are considered live inside functions called by the current function and after statements that exit the current function (`return`, `jump` and `cut to`).
4. Local variables are initially assigned to temporaries. Later, they will be assigned to available physical registers or memory by the register allocator.

To my knowledge, this method of making reserved registers available when they are dead is novel.

When a front end reserves a few machine registers, the number of spills may increase in user programs, since the register allocator has fewer registers to colour program variables. With the technique described above this problem is alleviated.

5.2 Optimising C--

The effort of optimising an intermediate language pays over and over every time a compiler uses the intermediate language to translate a program. In the case of a language-independent intermediate language like C--, that might be used by several client compilers, potentially many more source programs benefit from the optimisations.

This section describes the following optimisations that `cmmc` implements:

- Efficient calling conventions for known functions.
- Tail-recursion optimisation.
- The use of conditional move instructions.

5.2.1 Optimising Parameter Passing

C-- functions are not required to use the native calling convention of the target architecture. C-- compilers are free to use different registers, or a larger number of them, for faster function calls (Lueh and Gross 1997).

Where possible, cmmc uses a better calling convention than the native one. For instance, it uses more machine registers to pass arguments. (Of course, the native convention must be used for `foreign` calls and `foreign` function definitions.)

5.2.2 Optimising Register Parameters of Known Functions

We say that a function is *known* when all its call sites can be identified statically. Otherwise, we say that it is *escaping*. Some compilers specialise the calling convention of known functions (Kranz et al. 1986; Appel 1992). cmmc uses similar techniques. I will first show the code that cmmc emits for functions that are not known. Then, to illustrate the specialisation of the calling convention, I will show the code that is emitted for known functions.

Consider the following code, where neither `f` nor `g` are known functions:

```
f(bits64 x) {
  ...
  g(0, x);
  ...
}
```

Assume that the calling convention for escaping functions is to pass the first few parameters in registers starting with register 1. This is the low-level code that cmmc would produce:

```
f:
  /* copy formal parameter to temporary */
  x <- R1
  ...
  /* pass actual parameters in R1, R2 */
  R1 <- 0
  R2 <- x
  call g
  ...
```

The register allocator will try to coalesce (assign the same register to) a temporary and a machine register if they are connected by copy instructions (George and Appel 1996). When this strategy succeeds, the copy instruction can be deleted. Here, `x` is connected to both `R1` and `R2`. The allocator can assign `x` to either of them, and remove one of the copies. Let us assume that it selects `R2`. This results in the following code:

```
f:
  /* copy formal parameter to temporary */
  /* x assigned to R2 */
  R2 <- R1
  ...
  /* pass actual parameters in R1, R2 */
  R1 <- 0
  R2 <- R2
  call g
  ...
```

Now, the copy `R2 <- R2` can be deleted.

It is also possible to coalesce two temporaries if they are connected by copies. If a function is known, `cmmc` uses temporaries rather than machine registers for its formal parameters. This exposes even more possibilities for coalescing than before. For instance, assume that the parameter to `f` is passed in temporary `tf1`, and the parameters to `g` are passed in `tg1` and `tg2`:

```
f:
  /* copy formal parameter to temporary */
  x <- tf1
  ...
  /* pass actual parameters in tg1, tg2 */
  tg1 <- 0
  tg2 <- x
  call g
  ...
```

Now the allocator could coalesce `x, tf1` and `tg2`, and remove two copies instead of one, to produce this optimal code:

```
f:
  ...
  tg1 <- 0
  call g
  ...
```

Of course, not only `f` but all callers of `g` must use the same temporaries (`tg1` and `tg2`) as the formal parameters of `g`.

`cmmc` uses a simple escape analysis to determine whether a function escapes its compilation unit. The analysis is purely syntactic and determines that a function may escape when:

- Its name is listed in an `export` declaration, or
- Its name is used in a position other than the name of the callee in a call or `jump` statement. This includes its use as a parameter, in a `return`, in the right hand side of an assignment, etc.

This analysis is necessarily conservative, but I believe that it provides a good compromise of cost vs. precision.

5.2.3 Optimising Overflow Parameters of Known Functions

If f is a known function, the following optimisations are possible:

- Convention 3 (Section 5.1.1) can be used if there are only calls and no jumps to f.
- Convention 3 can be used if every jump to f is from functions f'_i that use convention 3 themselves and have no fewer overflow parameters than f. (A caller of any one of the functions f'_i reserves in its frame enough space to hold f'_i's overflow parameters. This space is also large enough to hold f's overflow parameters, since they are no more than those of f'_i.)

5.2.4 Tail-Recursion Optimisation

Many tail-calls are also *tail-recursive* (i.e. the callee is the same function as the caller). For instance, in functional languages iterative computations are often expressed by recursion.

Tail-recursive calls can be compiled more efficiently than ordinary tail calls. The steps for an ordinary tail call are:

1. Move actual parameters to formal parameters.
2. Restore callee-saves registers.
3. Pop the stack frame of the calling function.
4. Jump to the callee.

A compiler that does not optimise tail-recursive calls would translate this function

```
f(bits64 x) {
  bits64 y;
  y = ...;
  ...
  jump f(x-y);
}
```

to this low-level code:[4]

```
1  f:
2    SP <- SP - 32
3    save callee-saves
4    /* copy formal in R1 to temporary x */
5    x <- R1
6    y <- ...
7    ...
8    /* pass actual parameter in R1 */
9    R1 <- x - y
10   restore callee-saves
11   SP <- SP + 32
12   JMP f
```

There are some obvious inefficiencies here. First, the stack pointer is incremented before the branch, only to be decremented by the same amount

[4]Assume that 32 bytes of space are required in the activation frame to hold local variables or spilled registers.

at the branch target. Similarly, the callee-saves registers are restored and saved before and after the branch.

Steele was the first to point out that a tail-recursive call can be compiled to a "goto with arguments" (Steele 1977). If the compiler performs tail-recursion optimisation, the inefficiencies with the stack pointer and the callee-saves registers can be eliminated, resulting in this code:

```
1  f:
2    SP <- SP - 32
3    save callee-saves
4    RECURSE:
5    /* copy formal in R1 to temporary x */
6    x <- R1
7    y <- ...
8    ...
9    /* pass actual parameter in R1 */
10   R1 <- x - y
11   goto RECURSE
```

But, in fact, the compiler can emit an even better sequence:

```
1  f:
2    SP <- SP - 32
3    save callee-saves
4    /* copy formal in R1 to temporary x */
5    x <- R1
6    RECURSE:
7    y <- ...
8    ...
9    /* pass actual parameter in temporary */
10   x <- x - y
11   goto RECURSE
```

This is essentially the same code that would be generated if the iteration were expressed with a loop, instead of with recursion.

Notice that if the copy `x <- R1` can be deleted by the coalescing phase, then the final machine code would be identical in both cases. But if `x` and `R1` cannot be coalesced, the second sequence executes one less instruction in every iteration.

cmmc implements tail-recursion elimination; in particular, it generates the optimal sequence.

5.2.5 Conditional Moves

Several RISC architectures have special instructions for conditional moves, which are more efficient that conditional branches. A typical conditional move instruction has the following format:

```
R1 <- R2 if cond R3
```

The semantics is to move the contents of register R2 into R1 if the condition cond holds for R3. cond is usually limited to simple comparisons with the value zero (equal to, less than, etc).

For instance, a straightforward translation of this conditional assignment:

```
if (c==1) {x = y;} else {x = z;}
```

is the following:

```
R <- c == 1
if !R goto ELSE
x <- y
goto ENDIF
ELSE:
x <- z
ENDIF:
```

In target architectures that have conditional move instructions, the following sequence executes faster than the above sequence:

```
R <- c == 1
x <- y
x <- z if !R
```

This improved translation is possible only if the alternative assignments have simple right hand sides and identical left hand sides. The following conditional assignment can also be translated to an efficient sequence that uses a conditional move:

```
if (cond) {x = y;}
```

And so can patterns like the following, where the conditional assignments are implicit:

```
if (cond) {f(x);} else {f(y);}
if (cond) {return(x);} else {return(y);}
```

`cmmc` emits conditional move instructions for patterns where conditional assignments are explicit.

Chapter 6

Targeting C--

This chapter describes the use of the C-- infrastructure in two different compilers. These are the first compiler implementations to use C-- to generate code. I show that C-- can be used to compile very dissimilar source languages, C and Caml.

The systems that have been used in the evaluation are lcc (Fraser and Hanson 1995) and the Caml compiler from INRIA (Leroy et al. 2002). The two compilers translate very dissimilar intermediate languages into C--. lcc's intermediate language is imperative, low-level, and fairly close to C--. Caml's intermediate language is functional and has some higher-level constructs, like exception handlers. Each translation stresses different aspects of C--, providing broad coverage overall.

The discussion in this chapter is centered on C and Caml, but the lessons that have been learnt are general and apply to other languages as well.

The chapter ends with a discussion about the need to generate target-dependent C-- and how this may increase the complexity of the front end.

6.1 Translating C to C--

The lcc C compiler has proved to be an ideal system for a first evaluation of C-- as a portable compiler intermediate language. lcc is lean, its source code is freely available, and its design and implementation are fully documented in a book (Fraser and Hanson 1995).

Moreover, lcc's intermediate language (IL) has a very small semantic gap with C--. The IL representation of a C program makes explicit all things that are implicit in the source code, but that are specified in the ANSI

C standard; for example, type conversions when passing parameters and returning results. By and large, the IL has a very similar structure to C--. C expressions that have no direct counterparts in C--, such as function calls and assignments in an expression context, are translated to C-- statements; structured control-flow statements are lowered to conditional or indirect branches; and so on.

This structure has allowed me to fit a small generic emitter that translates lcc's IL to C--. The emitter consists of 610 lines of C (excluding comments), plus a machine-specific file that describes the sizes and the memory alignment requirements of the primitive data types.[1] This file is around 70 lines of C.

In the rest of this section I describe a few interesting issues that arise when translating the C language to C--. (Appendix B shows the translation of additional high-level constructs to C--. It also compares C-- to another generic intermediate language.)

6.1.1 Indirect gotos and Labels as Values

The first C-- specification included only direct gotos, where the destination was restricted to be a label in the same function as the goto.

The lcc C-- emitter has exposed that this limitation hinders the emission of efficient machine code for some common constructs of source languages. The case selection code for a C switch statement can be implemented with nested conditional branches, or with branch tables. When there are more than three alternatives, the fastest selection code is a branch table (Fraser and Hanson 1995). Threaded code interpreters (Bell 1973) also use branch tables for efficiency.

To build branch tables in C--, we need to be able to store labels in memory. To branch to a label stored in a table, an indirect goto is necessary. (In an indirect goto, the branch target is an arbitrary expression of the native code-pointer type). These are exactly the same as the "labels as values" feature and the "computed goto" statement of GCC (Stallman 2001).

Labels-as-values and indirect gotos have been subsequently added to the C-- specification so that branch tables can be implemented.

Unlike a computed goto in GCC, an indirect goto in C-- includes a list of the possible targets, so that an optimiser does not have to make pessimistic

[1] See Section 6.3 for a more detailed description of the contents of this file.

assumptions about the control flow of an indirect `goto`. Without this list, the optimiser might be forced to assume that control may be transferred to any statement of the function. The assembly language of the IA64 has the same feature (Tal et al. 1999).

Figure 6.1 shows a trivial C program and its `C--` translation by `lcc`.[2] Notice the indirect `goto` (line 15) that lists all potential targets (`L1, L2, L3, L4`). Notice also the statically-allocated branch table (line 29). In `C--`, code labels have global scope, so that their addresses can be stored into global branch tables.

6.1.2 In-Memory Locals

C has an "address-of" operator (`&`) that returns the address where a value is stored. In C, local variables that have their address taken[3] cannot be allocated to machine registers, since registers have no memory address. When generating `C--`, `lcc` allocates such variables not to `C--` variables, but to memory locations in the function's frame, using `stackdata`. Any use of (assignment to) the locals in C is translated to an explicit load from (store to) the `stackdata` location in `C--`.

C variables that are declared `volatile` are also allocated to `stackdata`. (The semantics of C requires that an optimiser does not affect the number and order of references to a `volatile` variable (Harbison and Steele 1995, p. 83).) By allocating them to `stackdata` instead of `C--` variables, we make sure that the `C--` compiler will not place them in registers. This guarantees that the register allocator will not eliminate references to them (for example by coalescing (George and Appel 1996)).

Finally, formal parameters that have their address taken must be copied into local `stackdata`, and all their uses and definitions must be translated to explicit loads and stores. In the current `C--` emitter every parameter is copied unconditionally, at function entry, even if the parameter is never mentioned in the function body. Two better alternatives are the following:

- Insert the copy in the dominator of all uses and definitions of the

[2] This and subsequent code listings that show the `C--` output produced by the front end have been edited for readability, including added comments, indentation, the names of local variables and labels, etc. None of the edits alter the number and relative order of `C--` statements.

[3] In `lcc`, the information about what variables have their address taken is readily available in the symbol table built by the front end.

```
1  void f(int i) {
2    int x;
3    switch (i) {
4      case 1: x = 1; break;
5      case 2: x = 2; break;
6      case 3: x = 3; break;
7      case 4: x = 4; break;
8    }
9  }
10
11 foreign "C" f(bits32 i) {
12   bits32 x;
13   if (i < 1) { goto END_SWITCH; }
14   if (i > 4) { goto END_SWITCH; }
15   goto (bits64[TBL + (%zx64(i-1) << 3)]) (L1, L2, L3, L4);
16   /* %zx64 stands for zero-extension to 64 bits */
17   L1: x = 1;
18   goto END_SWITCH;
19   L2: x = 2;
20   goto END_SWITCH;
21   L3: x = 3;
22   goto END_SWITCH;
23   L4: x = 4;
24   END_SWITCH:
25   foreign "C" return();
26 }
27 section ".rdata" {
28   align 8;
29   TBL: bits64{L1,L2,L3,L4};
30 }
```

Figure 6.1: A C switch statement translated to C-- using indirect gotos.

parameter. In this way the copy is executed only in program paths that reference the parameters. If the parameter is not mentioned in the body, then no copy is inserted.

- Insert the copy naively at function entry but use a phase of dead-store elimination to clean up useless stores. In practice this requires good aliasing information. (See Section 3.1.6.)

Figure 6.2 shows a trivial C function that includes a `volatile` variable, and that takes the addresses of a formal parameter and of a local variable. The `C--` code emitted by `lcc` follows the C code.

Notice that the C local variables `z` and `vol` are explicitly allocated to memory in the `C--` translation. They are allocated to `stackdata` locations with labels `z2` and `vol2`, respectively.

A memory location is also reserved to copy the parameter `x`. `x` is copied to this memory location at function entry (line 22). In the rest of the function, `x2` is the label of the address where the value of parameter `x` is stored. For instance, the assignment of `x`'s address to `ptr1` appears in line 23. Similarly, `z`'s address (`z2`) is assigned to `ptr2` in line 24.

All uses of `x`, `z`, and `vol` translate to loads from their addresses.

6.1.3 Tail Call Optimisation in C

Tail calls can be implemented more efficiently than ordinary calls ((Appel 1998, chapter 15), Chapter 5 of this dissertation). Tail calls occur very frequently in functional languages, and they are not uncommon in C. Clinger (1998) measured the number of static tail calls and tail-recursive calls in several Scheme and C programs. His measurements for C are shown in Table 6.1.

Program	Calls	Tail calls	Tail-Recursive calls
`lcc`	4492	6.5%	4.2%
`jpeg`	2122	6.2%	5.4%
`grep`	656	4.6%	3.7%

Table 6.1: Static frequency of tail calls in C (Clinger 1998).

C calls that appear in tail position can be translated to `C-- jump` statements, so that the `C--` compiler performs tail call optimisation. However, a `C-- jump` deallocates the caller's frame, and in C the contents of the caller's

```
1
2  void f(int x){
3    int* ptr1, ptr2;
4    int z, i;
5    volatile int vol;
6
7    ptr1 = &x;
8    ptr2 = &z;
9    i = x;
10   i = z;
11   i = vol;
12   return;
13 }
14
15 foreign "C" f(bits32 x) {
16   stackdata { x2: bits32; }
17   bits64 ptr1, ptr2;
18   stackdata { z2: bits32; }
19   bits32 i;
20   stackdata { vol2: bits32; }
21
22   bits32[x2] = x;
23   ptr1 = x2;
24   ptr2 = z2;
25   i = bits32[x2];
26   i = bits32[z2];
27   i = bits32[vol2];
28   foreign "C" return();
29 }
```

Figure 6.2: Translation of `volatile` variables into C--.

frame can be accessed even after a call in tail position. The caller's frame is potentially needed in any of the following cases:

- A pointer to a local variable is passed as an argument to the tail call. The pointer may be dereferenced by the callee (or by anything it calls).
- A pointer to a local variable has been stored in a global variable before the tail call, or passed as an argument in a previous call. (I use "before" and "previous" in the sense of *predecessor in the control flow graph.*)
- Control may return to the caller after the tail call. In C this can occur when there has been a call to `setjmp` previous to the tail call and the callee (or anything it calls) invokes `longjmp`.

These situations are considered by C compilers that implement tail-call optimisation, but unfortunately it is difficult to find published descriptions of all the possible instances. For the above cases, it is unsafe to translate a C tail call into a `C-- jump`. An optimising C compiler targeted to `C--` can emit `jump`s for C calls in tail position if it is able to determine that the caller's frame will not be accessed after the tail call. `lcc`'s `C--` backend is quite simple and does not perform any such analysis. Therefore, it does not emit `jump`s.

6.1.4 Unsigned Integer Loads in `C--`

Several high-level languages, including C, provide signed and unsigned integers as primitive types. Similarly, the instruction sets of modern architectures include instructions for signed and unsigned arithmetic and memory loads. For example, the Alpha instruction set provides the instructions `ldb` (load byte) and `ldbu` (load byte unsigned).

An 8-bit unsigned load instruction in an architecture with 64-bit registers sets bits 8 to 63 of the register to zero. An 8-bit signed load instruction extends bit 7 (the sign bit) of the loaded value through bits 8 to 63 of the register. For example, for the C code in Figure 6.3, `lcc` emits, for the Alpha architecture, load-byte (`ldb`) to read `c` and load-byte-unsigned (`ldbu`) to read `uc`.

While `C--` has distinct operators for signed and unsigned arithmetic operations, it only provides one kind of load operator. To make sure that `lcc` targeting `C--` emits exactly the same machine instructions, the difference

```
char c;
unsigned char uc;

void f() {
  char a;
  unsigned char b;

  a = c;
  b = uc;
  return;
}
```

Figure 6.3: C example that reads signed and unsigned variables.

must be conveyed using other C-- constructs. The most natural way is as follows:

```
foreign "C" f() {
  bits8 a;
  bits8 b;

  a = %sx8(bits8[c]); /* sign extend */
  b = %zx8(bits8[uc]); /* zero extend */
  foreign "C" return();
}
```

Here, `%sx8` and `%zx8` are unary operators that perform sign and zero extension respectively.

A naive C-- compiler would emit for this sequence a load instruction followed by a sign or zero extend instruction. `cmmc` recognises the patterns `%sxN(bitsN[addr])` and `%zxN(bitsN[addr])`, and, where available, emits the appropriate signed or unsigned load instructions.

In my opinion, however, it would be better if C-- distinguished between signed and unsigned load operations. This distinction is present in several compiler intermediate languages, e.g. the ones described in (Muchnick 1997) and (Fraser and Hanson 1995). A possible syntax for C-- is to qualify memory references, for example like this: `bits8(unsigned)[uc]`.

6.1.5 Unsupported Features of C

Unlike C, C-- does not support functions with variable argument lists. When lcc's C-- emitter encounters the definition of a C function whose prototype has a variable argument list, it emits an error message.

However, a *call* to a C function with a variable argument list may be translated to a normal C-- call in some targets. For instance, in the calling convention of the Alpha, the machine code required to pass parameters is the same for all calls, irrespective of the prototype of the callee.

In other conventions (for instance the IA64), the parameter passing code is different for functions with variable argument lists. In these cases, there might be no C-- sequence that translates to the desired machine code. For such calls to functions with variable argument lists, lcc's C-- emitter generates an error message.

6.2 Translating Caml to C--

The Caml compiler is based on the idea of compilation by transformation (Kelsey and Hudak 1989; Appel 1992; Peyton Jones 1996). The front end builds an abstract syntax tree, which is translated to assembly language via a series of intermediate languages, each one closer to the machine than the previous one.

Two of these intermediate languages are at a level of abstraction close to C--: LLFL[4] and Mach. LLFL is a low-level, weakly-typed functional language. During compilation, LLFL is lowered down to Mach, an abstract machine language similar to three-address code (Aho et al. 1986). Mach still contains a few fairly high-level constructs, like memory allocation, a structured if, and a raise operator.

Mach is similar to a low-level C--, but it is machine-specific. It contains a large core that is machine-independent, plus a few machine-specific instructions, like push for the x86, or memory barrier for the IA64. This makes Mach unsuitable as the intermediate language to translate to C--. Thus, LLFL is the natural intermediate language to translate to C--.

[4]Low-level Functional Language. In the compiler sources, this language is actually named C--. I will call it LLFL to avoid confusion.

6.2.1 Translating LLFL into C--

There is no formal definition of the LLFL language, just a datatype in the sources of the Caml compiler. Some salient characteristics of LLFL are the following:

- The only syntactic category is `expression`; there are no statements. The syntax is based on prefix application. (It is like the syntax of LISP's S-expressions.)
- LLFL expressions are like those found in untyped functional intermediate languages of other compilers: `let` bindings, function applications, operator applications, `if` expressions, literals, memory loads and stores, etc.
- The only explicit types are those of the formal parameters of functions and the results of function applications.
- Applications of functions and operators can be nested. For instance, `f(g(x))` and `(x+y)*z` are allowed.
- Operators include arithmetic, bit manipulation, and integer/floating-point conversions.
- Higher-level expressions include `try, raise` (to handle and raise exceptions, respectively), `alloc` (to allocate memory), `checkbound` (to check array bounds), and `loop` (to iterate).
- Function definitions cannot be nested.

In summary, the semantic gap with C-- is moderate, making the translation non-trivial, but still manageable. I will describe in some detail those aspects of the translation of LLFL to C-- that require careful design. They are:

- Translating LLFL tail calls into C-- `jump`s.
- Translating LLFL expressions to C-- statements.
- Inferring C-- types for LLFL variables and for temporaries created during the translation.

- Identifying heap pointers for garbage collection.
- The translation of exception raising and handling.

I address the first four points in the remainder of this section. The translation of exceptions is described in Section 6.2.3.

Translating LLFL Tail Calls

Function applications in tail position are translated to `C-- jump` statements. For instance, this function definition in source Caml:

```
let f x y = (if x > y then x else g y)
```

is translated to the following `C--`[5]:

```
f(bits64 x, bits64 y) {
  if (x > y) {
    return(x);
  } else {
    jump g(y);
  }
}
```

Unlike in C, *all* LLFL function applications in tail position can be safely translated to `C-- jump`s. It cannot be the case that the caller's frame contains a memory cell that may be accessed after the call. (Recall that Section 6.1.3 discussed the limitations for C.)

The translation is syntax-directed. When a function application appears in tail position, a `jump` is emitted. For any other expression in tail position, a `return` is emitted.

From LLFL Expressions to `C--` Statements

In general, when translating a language with rich expressions (like a functional language) to a language of simple expressions and statements (like `C--`), complex subexpressions must be evaluated by statements in the target language. The results of the evaluation of the subexpressions are temporarily stored in variables. Most importantly, *the order of evaluation of expressions*

[5] A 64-bit architecture is assumed in this example.

must be preserved in the translation to C--. For instance, consider this Caml function definition:

```
let f x y z = max (g x) (if x > y then y else z)
```

Its representation in LLFL is as follows:

```
function f (x: addr, y: addr, z: addr) =
  (max
     ((g x): addr)
     (if (> x y) y z)
  ): addr
```

To translate this function definition to C--, the following must be done:

1. The function application `g x` must be emitted as a statement, and its result stored in a C-- variable.
2. Similarly for the expression `if x > y then y else z`.

Here is the result of the translation into C--:

```
f(bits64 x, bits64 y, bits64 z) {
  bits64 t1;
  if (x > y) {
    t1 = y;
  } else {
    t1 = z;
  }
  bits64 t2;
  t2 = g(x);
  jump max(t2, t1);
}
```

The order of evaluation may look incorrect (the `if` expression is evaluated before the function call), but the Caml language specification does not impose a left-to-right order. In order to support efficient partial applications of functions, the Caml compiler emits code that evaluates subexpressions right-to-left (Leroy 1990), and I have chosen to follow the same convention in the translation to C--.

The translation from the functional language to C-- is a variation of the A-normalisation algorithm (Flanagan et al. 1993). A-normalisation is

a source-to-source transformation for functional languages that names all intermediate results and makes control flow explicit. The resulting program can be easily translated to a low-level language like three-address code, or C--. For instance, this function definition:

```
let f x y = x * g(x) + y
```

looks like this after A-normalisation:

```
let f x y =
  let t1 = g(x) in
  let t2 = x * t1 in
  t2 + y
```

I have implemented a variant of the algorithm of Flanagan et al. (1993). In this variant, the results of expressions with no side effects are not given names, and thus the example above is A-normalised to this code:

```
let f x y =
  let t1 = g(x) in
  (x * t1) + y
```

This variant of A-normalisation produces expression trees, rather than three-address code. This facilitates certain back-end optimisations that work better on expression trees than on three-address code. For instance, instruction selection based on tree pattern matching can identify complex patterns that match CISC instructions (Appel 1998). Indeed, several architectures have multiply-and-add instructions, which match nicely with the code in the example shown above. Also, there exist optimal algorithms for instruction selection (Pelegrí-Llopart and Graham 1988; Ertl 1999) and register allocation (Sethi and Ullman 1970) for expression trees. (They take advantage of the fact that subexpressions can be computed in any order.)

The C-- emitter in fact A-normalises the LLFL code on-the fly, as it emits C--. Perhaps the implementation might be more clear if done as a separate pass of A-normalisation, followed by translation to C--. The resulting C-- code would be the same in either case, though.

Type Inference

In LLFL only formal parameters and the results of function application have explicit types. `let`-bound variables and temporaries generated during the translation to C-- need an explicit declaration in C--, including a type.

A local type inference pass is required during the translation of LLFL to C--. The inference pass is syntax-driven. The type of a function application is given explicitely in LLFL; the type of an `if` expression is the type of the true and false subexpressions; the type of a binary operator application is the type of its operands; etc. For instance, this Caml code:

```
let f x y = g (if x > y then x else h x)
```

is translated to the following C-- code:

```
f(bits64 x, bits64 y) {
  bits64 t;
  if (x > y) {
    t = x;
  } else {
    t = h(x);
  }
  jump g(t);
}
```

The temporary `t` was generated in the translation to C--, and a type was inferred for it.

Identifying Heap Pointers for Garbage Collection

The C-- emitter marks in the program text those arguments and local variables that correspond to heap pointers in the Caml program. Marking is done using the keyword `gc_root`, as described in Section 4.5. For instance, for this Caml fragment:

```
let f xs =
  let tail = List.tl xs in ...
```

the following C-- code is generated:

```
f(bits64 gc_root xs){
  bits64 gc_root tail;
  tail = List_tl(xs);
  ...
}
```

Here, both `xs` and `tail` are pointers to cons-cells, which are heap-allocated.

6.2.2 Exceptions and Exception Handling in Caml

Exceptions in Caml are first-class values of type `exn`. They are declared with the keyword `exception` followed by a constructor declaration. For instance,

```
exception Division_by_zero,
```

or

```
exception Error of string.
```

To throw an exception, Caml provides `raise`. To handle an exception, the `try` expression is used:

`try` *expr* `with` *handler*

Where *handler* has this form:

$pattern_1$ `->` $expr_1$ `|` `...` `|` $pattern_n$ `->` $expr_n$

We say that *expr* is the expression *protected by the handler*. Evaluation of the `try` expression first evaluates *expr* and returns this value if the evaluation of *expr* does not raise any exceptions. If the evaluation of *expr* raises an exception, the exception value is matched against the patterns $pattern_1$, ..., $pattern_n$, in order. If the matching against $pattern_i$ succeeds, the associated expression $expr_i$ is evaluated, and its value becomes the value of the whole `try` expression. If none of the patterns matches the value of *expr*, the exception value is raised again (Leroy et al. 2002). During the evaluation of *expr* we say that control is *inside the scope of the handler*.

Caml Implementation of Exception Handling

The Caml compiler implements exception handling as follows. An exception handler encapsulates a program counter (that of the handler's code) and a stack pointer (the activation where the scope of the handler is entered). Nested exception handlers are implemented using a stack of handlers. When control enters the scope of a handler, this handler is pushed onto the handler stack. We say that this is the *current handler.* When control leaves the scope of a handler, the handler is popped from the stack. (The stackability of exception handlers is folklore knowledge among language implementors, and it has been proven formally recently (Polakow and Yi 2001).)

Caml implements the stack of exception handlers as a linked list, with its elements stored in the activation frames of the functions that install handlers. For instance, given this code fragment:

```
let f x = ... try g(x) with Division_by_zero -> 0 ...
```

Figure 6.4 depicts the call stack during an invocation of `f`, at the point where the handler has been pushed, but before `g` has been called. Notice in particular that the stack pointer and the top of the handler stack (`curr_handler`) coincide. That is, `curr_handler` captures both an activation (a stack pointer) and a program counter.

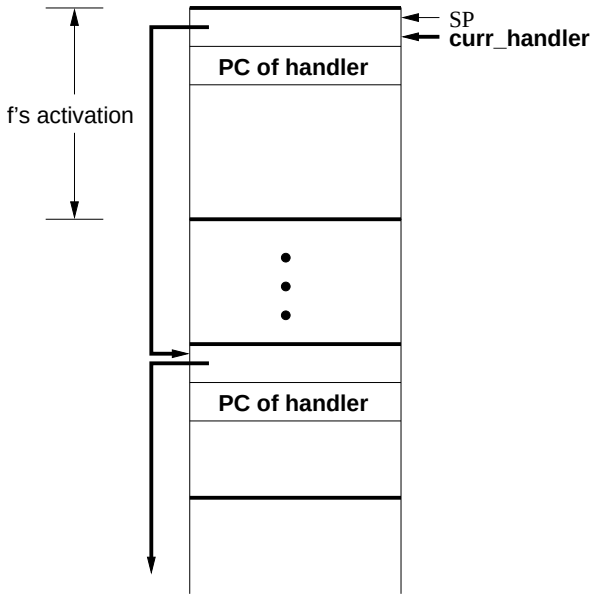


Figure 6.4: Stack of Caml handlers implemented as a linked list in the call stack.

To raise an exception, the stack is cut to the activation of the current handler (the topmost one on the handler stack) and control is transferred to the program counter of that handler. Additionally, the topmost handler is popped from the handler stack.

Cost Model of the Caml Implementation

The cost of this implementation is as follows. Raising an exception takes constant time (adjust the stack pointer and transfer control). Entering or leaving the scope of a handler has a small (constant) cost: that of pushing and popping the handler onto the handler stack. Additionally, stack cutting imposes an indirect cost. In a `try` expression such as this:

```
try f() with handler
```

if an exception is raised during the evaluation of `f()`, the values of any callee-saves registers that have been modified in `f()` (or anything it calls) are not restored when control reaches the handler. Stack cutting reduces the utility of callee-saves registers. More specifically:

- A function that contains a `try` expression must save *all* the callee-saves and restore them when it returns (or makes a tail call). This can be expensive.
- If the expression protected by the handler includes function calls (the most common case), all local variables that are live into or across the handler cannot be assigned to callee-saves registers (Ramsey and Peyton Jones 2000). They cannot be assigned to caller-saves registers either, because these may be overwritten by the call. Therefore these variables must be spilled to memory.

In part because of this, the Caml compiler uses all registers as caller-saves. (Other reasons are: using callee-saves registers makes stack scanning more complex (Peyton Jones et al. 1999); frequent tail calls also reduce the utility of callee-saves registers (Peyton Jones and Ramsey 1998).)

Ramsey and Peyton Jones (2000) describe the cost models of two other exception handling mechanisms (run-time stack unwinding and native-code stack unwinding).

6.2.3 Caml Exceptions in C--

I have implemented Caml exceptions in C-- in the same way that the native Caml compiler does. More specifically, a stack of handlers is maintained, also implemented as a linked list; a machine register is reserved to point to the topmost handler. Each handler is represented by a C-- continuation.

Figure 6.5 depicts a situation like the one in Figure 6.4, where the activations in the call stack contain the linked list of C-- continuations. As Figure 6.5 shows, compared to the native Caml compiler implementation of handlers, using C-- continuations to implement handlers is slightly more expensive. Continuations take up one more word of storage and incur an extra indirection. A precise comparison of the costs of exceptions in the two implementations is given in Chapter 7.

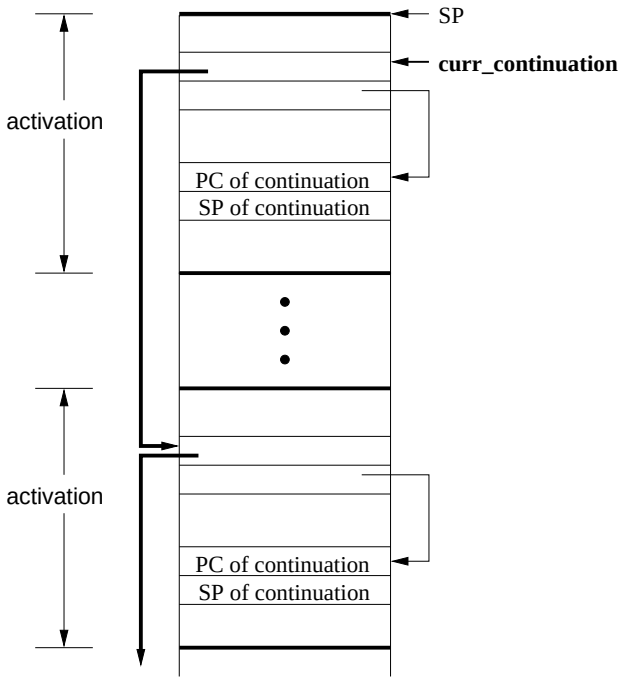


Figure 6.5: Stack of C-- continuations implemented as a linked list in the call stack.

Translation of `raise`

To raise an exception, we pop the topmost C-- continuation from the continuations stack, and cut to it, passing the exception as the argument. The translation of the expression `raise e` is C-- code with the following structure:

```
k <- pop continuations_stack
cut to (k)(e);
```

For instance, consider this simple Caml function:

```
let f ex = raise ex
```

All this function does is `raise` its argument (which must be of type `exn`). This gets translated to the C-- code of Figure 6.6. There, `curr_continuation` is a global variable held in a reserved register.

Translation of Exception Handlers

Exception handlers in Caml are lowered into a more primitive construct in LLFL:

```
f (bits64 ex) {
  bits64 k;

  /* k = topmost continuation */
  k = bits64[curr_continuation + 8];
  /* pop continuation */
  curr_continuation = bits64[curr_continuation];
  cut_to (k)(ex);
}
```

Figure 6.6: C-- code for `raise`.

`TRY(`*expr*`,` *exn*`,` *handler*`)`

Here *expr* is the expression protected by the exception handler, *exn* is a fresh identifier that appears free inside *handler*, and *handler* is an expression equivalent to this:

`if` *exn* `matches` $pattern_1$ `then` $expr_1$
`...`
`else if` *exn* `matches` $pattern_n$ `then` $expr_n$
`else raise` *exn*

The translation need not be nested conditions; sometimes, a `switch` expression with the same semantics is emitted by the Caml front end.

The above `TRY` expression is translated to C-- code that has the following structure:

```
push continuation k
evaluate expr
pop continuation

bits64 exn;
continuation k(exn):
statements for handler
```

To "push" a continuation, the C-- emitter allocates a tuple (link, continuation) in the activation (with `stackdata`). This tuple holds the continuation and a link to the previous topmost continuation.

```
stackdata {
  previous: bits64;
  cont: bits64;
}
```

This space can be reused for all unnested `TRY` expressions, but if there is a nested `TRY` inside *expr*, the front end must allocate a separate `stackdata` area.

A Larger Example

I will now show a more elaborate `try` expression in Caml and its full translation to `C--`.

```
let f x y = try (g(); h x) with Division_by_zero -> x | _ -> y
```

The function `f` returns the value returned by `h x` if no exceptions are raised during the evaluation of `g(); h x`. Otherwise, it returns `x` if there was a division by zero, or `y` if any other exception was raised.

If an exception is raised, it is first matched against `Division_by_zero`; if it matches, `x` is returned. Otherwise, the exception matches the wildcard `_`, and `y` is returned.

This is translated to the `C--` code shown in Figure 6.7. (For conciseness, the code that matches `exn` with the representation of `Division_by_zero` is omitted.)

Control-Flow Annotations

In the translation to `C--`, every function call in the expression protected by a handler is annotated with `also cuts to`, even if some of them may not raise an exception. (In Figure 6.7, the calls `g()` and `h(x)` are annotated.)

The results of a static exception analysis (Leroy and Pessaux 2000) can be used to determine more precisely what functions may raise what exceptions, in order to eliminate unnecessary `also cuts` annotations.

6.3 Generating Target-dependent `C--`

`C--` is not a "write-once, run anywhere" language. Things like the size of a data pointer vary from target to target and front ends must emit `C--` code

```
1  /* reserved machine register to point to topmost
2     continuation */
3  register bits64 curr_continuation 15;
4
5  f (bits64 x, bits64 y) {
6    stackdata {
7      previous: bits64;
8      cont: bits64;
9    }
10
11   /* push continuation k */
12   bits64[previous] = curr_continuation;
13   bits64[cont] = k;
14   curr_continuation = previous;
15
16   g() also cuts to k;
17   bits64 t;
18   t = h(x) also cuts to k;
19
20   /* pop continuation k */
21   curr_continuation = bits64[curr_continuation];
22
23   return(t);
24
25   bits64 exn;
26   continuation k(exn):
27   if (... /* exn matches Division_by_zero */) {
28     return(x);
29   } else {
30     return(y);
31   }
32 }
```

Figure 6.7: Exception handling code in C--.

with target-dependent sizes and alignments. For instance, consider this C function definition:

```
void f(long index) {
  long vector[7];
  vector[index + 2] = 0;
  return;
}
```

It is translated by `lcc`'s `C--` back end to the following Alpha-specific `C--` code:

```
foreign "C" f(bits64 index) {
  stackdata {
    align 8;
    vector: bits64[7];
  }
  bits64[vector + (index * 8) + 16] = 0;
  foreign "C" return();
}
```

The Alpha is a 64-bit architecture, and the Alpha convention specifies that `long`s are 64-bit wide. For best performance, a vector of `long`s is aligned on an 8-byte boundary.

The same C code is translated by `lcc`'s back end to the following x86-specific `C--` code:

```
foreign "C" f(bits32 index) {
  stackdata {
    align 4;
    vector: bits32[7];
  }
  bits32[vector + (index * 4) + 8] = 0;
  foreign "C" return();
}
```

Notice that sizes, alignments and array indexing expressions are different. The x86 is a 32-bit architecture and `lcc` chooses to represents `long`s by 32 bits here.

Running a C-- program emitted for a 64-bit architecture on a 32-bit architecture may produce wrong results. In fact, the program may even be rejected by the C-- compiler altogether.

Some people have expressed concern that this might be a disadvantage of C-- compared to C as a target language. However, my experience in targeting lcc and the Caml compiler to C-- has shown that sizes and alignments of primitive types can be easily dealt with by having a very simple target machine description.

It is common to use such descriptions in compilers that emit code for several target architectures. For instance, lcc uses *type metrics* to describe the size and alignment of primitive C types in each architecture (Fraser and Hanson 1995, chapter 5). All lcc's back ends (including the C-- back end) use the type metrics to emit target-specific code. For instance, here is the description of C types that lcc uses for the Alpha processor:

```
Interface alphaIR = {
  1, 1, 0, /* char */
  2, 2, 0, /* short */
  4, 4, 0, /* int */
  8, 8, 0, /* long */
  8, 8, 0, /* long long */
  4, 4, 1, /* float */
  8, 8, 1, /* double */
  8, 8, 1, /* long double */
  8, 8, 0, /* T * */
  0, 1, 0, /* struct */
  ...
  }
```

The first column is the size of the data in bytes. The second column gives the minimum alignment in memory. For instance, ints occupy 4 bytes and must be aligned to an address multiple of 4. The third column says whether literals of the corresponding type are valid as instruction operands. (For instance, in the Alpha, floating-point literals are invalid.) This column is ignored by the C-- emitter, since it only makes sense for back ends that emit assembly language.

The main inconvenience is that a machine description must be created for every architecture that is supported by the compiler. However, this is a

one-time task, and it can be automated with the aid of configuration tools (Vaughan et al. 2000).

6.4 Targeting the Caml RTS to C--

I have retrofitted the Caml garbage collector to use the C-- run-time interface. Only the function that scans the stack for roots had to be changed. All the rest remained exactly the same. This includes object allocation, management of free space, pointer chasing, etc. The modified garbage collector is similar to what is described in Chapter 4, with a change that I explain here.

Caml has a foreign function interface, which allows Caml functions to call C functions, and callbacks from C to Caml (Leroy et al. 2002). Caml code and C code execute in the same stack. Recall from Chapter 4 the code of a garbage collector targeted to C--, in the case of a call stack that may contain foreign activations:

```
1  cmm_activation_T act;
2
3  cmm_top_activation(&act);
4  for (;;) {
5    <scan activation act>
6    if (cmm_callers_activation(&act) == 0) {
7      /* Caller is not a C-- function. Reached bottom of
8         a sequence of C-- activations. Move to previous
9         C-- sequence, if any */
10     cmm_saved_activation(&act) || break;
11   }
12 }
```

Recall also that `cmm_saved_activation` initialises activation handle `act` from information stored in a stack of saved C-- contexts.

In Caml, foreign calls and callbacks take place via stubs in the Caml run-time system, which already maintains its own stack of saved Caml contexts. The Caml RTS targeted to C-- uses that stack, and there is no need to build the C-- one (with `cmm_push_state`, as described in Chapter 4). To find pointers in the call stack, the following code is used:

```
1  cmm_activation_T act;
2
3  cmm_top_activation(&act);
4  for (;;) {
5    <scan activation act>
6    if (cmm_callers_activation(&act) == 0) {
7      /* Caller is not a C-- function. Reached bottom of
8         a sequence of C-- activations. Move to previous
9         C-- sequence, if any */
10     void *sp, ret_addr;
11     sp = cmm_callers_sp(&act);
12     ret_addr = CamlSavedRetAddr(sp);
13     sp = CamlSavedSp(sp);
14     cmm_set_activation(&act, ret_addr, sp) || break;
15   }
16 }
```

This code uses the following function from the C-- run-time interface that was not described in Chapter 4:

```
int cmm_set_activation(cmm_activation_T *act, void *gc_point,
                       void *sp)
```

`cmm_set_activation` initialises an activation handle from saved values of a stack pointer and a return address. This is similar to what `cmm_saved_activation` does, except that the values of the saved callee-saves registers are not provided. Therefore, it is only safe to use `cmm_set_activation` if the code generated by the C-- compiler uses no callee-saves registers.

Callee-saves registers are not too effective for code that contains many tail calls and sets exception handlers often, which is the case in Caml programs. In addition, callee-saves registers incur some cost during garbage collection (the collector must keep track of them as it moves from one activation to the next) and when foreign functions are called (they must be saved away before a foreign call). For these reasons, the Caml compiler emits machine code that does not use any callee-saves registers.

`cmmc` provides a command-line switch to request that code be emitted that uses no callee-saves registers. The Caml front end uses this facility.

Chapter 7

Evaluation

This chapter presents an evaluation of the C and Caml compilers that have been re-engineered to use `C--` in the back end, as described in Chapter 6.

7.1 Compiler Construction Using `C--`

The main benefit of using the `C--` infrastructure to build a compiler is the reduced implementation cost. It is much easier to write and debug a `C--` emitter than a complete code generator. In addition, the `C--` code generation infrastructure can be used in multiple compilers, which allows to leverage improvements to the `C--` compiler and reduces maintenance costs.

`lcc`'s back end for the alpha consists of 9 files, totalling 4740 lines of code (LOC). This includes a code-generator generator but excludes the instruction selection module that is automatically generated by the code-generator generator. `lcc`'s `C--` emitter consists of 2 files, totalling 680 LOC. Caml's back end for the alpha consists of 30 files and 5010 LOC. This includes instruction selection, register allocation and assembly code generation. Caml's `C--` emitter consists of 2 files, totalling 1850 LOC. The semantic gap between the intermediate language and `C--` is larger in the Caml compiler and this accounts for the larger size of the emitter. Table 7.1 summarises these figures.

The actual savings are, in fact, substantially larger than what these numbers suggest. Line count is a very crude measurement of implementation cost and, when comparing compiler back ends, it does not do justice to the real effort behind the implementation. Register allocators are notoriously difficult to debug. Implementing an instruction selector and an assembly

	lcc	lcc/C--	Caml	Caml/C--
files	9	2	30	2
LOC	4740	680	5010	1850

Table 7.1: Size of the back end.

code emitter requires much more effort and attention to low-level detail than writing a C-- emitter.

7.2 Performance Evaluation

A number of C and Caml programs were compiled with `lcc` (version 4.1) and the Caml compiler (version 3.01) respectively, and then with the same compilers targeting C--. The resulting C-- code was translated to assembly language by `cmmc`, and the system assembler and linker were used to generate executable programs. Only the code of the benchmarks was compiled via C--; the standard libraries were not recompiled. For the Caml/C-- compiler, the garbage collector was modified as described in Section 6.4.

All programs where run with inputs large enough so that the resulting running times were significant. Times were measured with the UNIX `time` command and rounded to the nearest hundredth of a second. Each program was run three times; the highest time was discarded and the reported time is the arithmetic average of the other two measurements.

7.2.1 C Benchmarks

Table 7.2 contains brief descriptions of the C programs that have been evaluated. `tsp` is from a set of freely-available benchmarks; `qsort` is part of `lcc`'s test suite; `compress` and `go` are from the SPEC95 suite (All the other C programs in SPEC95 contain functions with variable numbers of arguments, which are not supported by C--.); all the rest are from the SPEC2000 suite.

The C programs were run on a lightly-loaded Alpha running Digital UNIX V3.2C. To have an additional data point, the C programs were also compiled with `gcc` (version 2.7.2.1), using optimisation level `-O3`.

Table 7.3 shows the execution times, and Figure 7.1 displays them normalised with respect to the `lcc` results. The size of the code generated by

tsp	Solve the travelling salesman problem.
qsort	Initialise and then sort an array of integers.
compress	Compress and decompress files in memory.
go	Play the game of "Go".
gzip	Compress and decompress files in memory.
vpr	Placement and routing of integrated circuits.
mcf	Combinatorial optimisation of vehicle scheduling.
crafty	Solve chess board layouts.
parser	Syntactic parser of English.
gap	Solve group theory problems.
vortex	Object-oriented database transactions.
bzip2	Compress and decompress files in memory.
twolf	Placement and routing of integrated circuits.

Table 7.2: Description of C benchmarks.

`lcc/C--` is 5% smaller on average than what `lcc` generates.

Analysis

The results of the C benchmarks are as expected. `lcc` has a simple back end that does not include many optimisations (Fraser and Hanson 1995). `cmmc`, on the other hand, uses MLRISC as its code generator, which implements more optimisations than `lcc`'s back end. In particular, MLRISC comes with a sophisticated register allocator (George and Appel 1996). As a result, programs compiled with `cmmc` spill substantially less and contain fewer copy instructions than when compiled with `lcc`. Also, `cmmc` does better instruction selection than `lcc` and performs simple peephole optimisations after register allocation.

`gcc` with optimisation level `-O3` performs a number of optimisations that are not implemented by `lcc` nor `cmmc`, including global common subexpression elimination, loop strength reduction, loop unrolling, a second pass of instruction scheduling after register allocation, and many more. The combined effect of all these optimisations accounts for the differences in execution times that have been measured.

7.2.2 Caml Benchmarks

Table 7.4 contains brief descriptions of the Caml benchmarks.

	lcc	lcc/C--	Ratio	gcc -O3	Ratio
tsp	2:21.17	2:17.09	.97	1:56.85	.83
qsort	38.96	30.99	.80	28.10	.72
compress	20:36.18	18:17.96	.89	13:02.37	.63
go	7:25.97	5:56.11	.80	4:01.44	.54
gzip	2:09:30.04	1:47:09.50	.83	1:28:06.19	.68
vpr	1:41:57.92	1:28:41.72	.87	1:11:22.34	.70
mcf	2:29:36.39	2:01:11.23	.81	1:35:45.56	.64
crafty	1:13:34.08	1:06:12.67	.90	52:14.42	.71
parser	3:01:31.18	2:34:17.50	.85	2:03:26.22	.68
gap	1:43:14.74	1:23:37.73	.81	1:18:28.65	.76
vortex	2:04:02.42	1:37:59.52	.79	1:21:52.56	.66
bzip2	1:43:49.17	1:25:07.92	.82	1:14:45.38	.72
twolf	3:52:07.87	3:17:18.75	.85	2:21:36.48	.61

Table 7.3: Execution times of C benchmarks (hours:minutes:seconds).

bdd	Solve a binary decision diagram problem.
boyer	Verify a proof with the Boyer-Moore theorem prover.
fft	Solve a fast Fourier transform problem.
quicksort	Initialise and then sort an array of integers.
xml	Parse an XML document and make multiple traversals.
exceptions	Raise and catch exceptions.

Table 7.4: Description of Caml benchmarks.

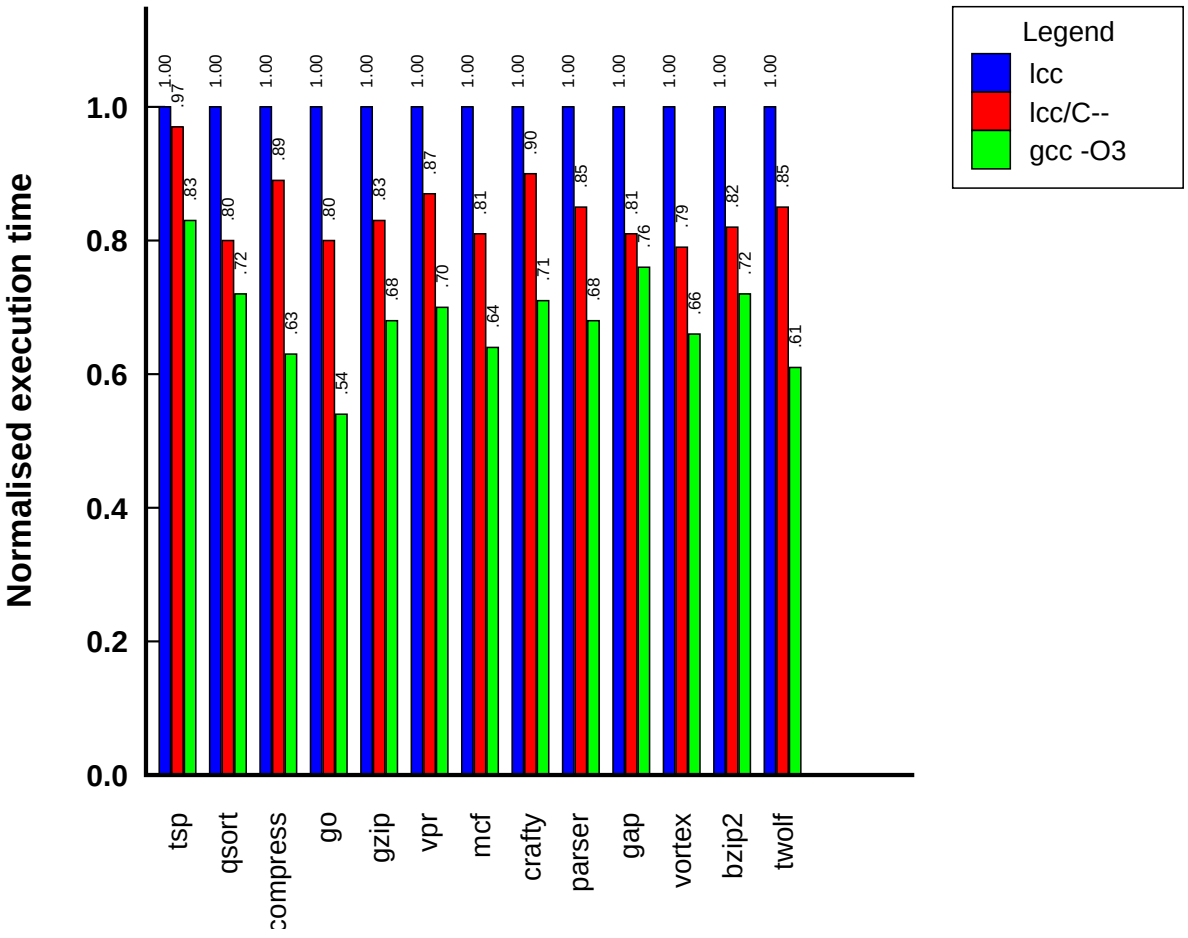


Figure 7.1: Normalised execution times of C benchmarks.

The first four benchmarks are part of the Caml test suite. `exceptions` is a synthetic benchmark that raises and catches exceptions. The intention of the `exceptions` benchmark is to try to measure the overhead of implementing Caml exception handlers using `C--` continuations. The complete code of `exceptions` is shown in Figure 7.2.

```
exception E of int

let g x = raise (E x)

let rec f x =
  try
    if x > 0 then g x else (Printf.printf "done\n"; 0)
  with E n ->
    f (x-1)

let _ = f 40000000
```

Figure 7.2: Code of `exceptions` benchmark.

The Caml programs were run on a lightly-loaded Alpha running RedHat Linux 6.1 (kernel version number 2.2.13-0.9).

Table 7.5 shows the execution times, and Figure 7.3 displays them normalised with respect to the times of the native Caml compiler for the Alpha. The size of the code generated by Caml/`C--` is less than 1% larger on average than what the Caml compiler generates.

	Caml	Caml/`C--`	Ratio
bdd	16.25	18.68	1.15
boyer	25.73	26.28	1.02
exceptions	15.05	18.01	1.20
fft	22.20	24.03	1.08
quicksort	37.56	54.55	1.45
xml	47.04	51.27	1.09

Table 7.5: Execution times of Caml benchmarks (seconds).

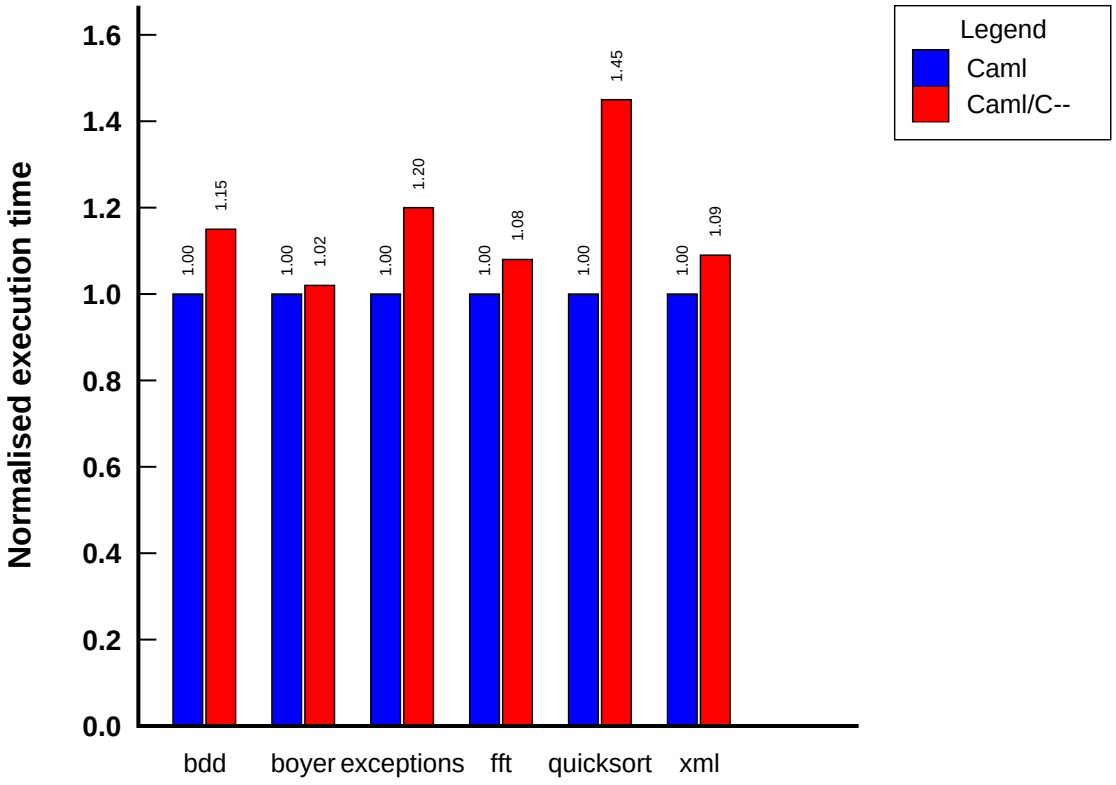


Figure 7.3: Normalised execution times of Caml benchmarks.

Analysis

A number of factors account for the slower execution times of the Caml compiler targeting `C--` vs. the Caml compiler using its own back end. I list below the ones for which I have evidence, but there may be more. The order in the list does not reflect the relative weight of each item because it is difficult to charge a precise percentage of the total overhead to any single item.

Allocation Merging

Caml allocates memory for dynamic objects from a contiguous region of free memory. An *allocation limit* points to the end of the free region, and an *allocation pointer* points to the last allocated memory cell. A function in the run-time system implements allocation like this:

```
void alloc_mem(int size) {
  retry_allocation:
  alloc_ptr = alloc_ptr + size;
  if (alloc_ptr > alloc_limit) {
    gc();
    goto retry_allocation;
  }
}
```

The cost of one allocation when enough free memory is available is the following: a function call, an add instruction, a comparison, a conditional branch (which is expensive in current architectures), and a function return.[1]

Functional languages allocate memory at very fast rates (Diwan et al. 1995), and it is important to minimise the costs of allocation. When more that one allocation is requested in the same basic block, it is profitable to combine them into a single allocation. The Caml compiler has an optimisation phase that does precisely that.

For instance, this Caml expression:

```
let l = x::y::zs in ...
```

[1]The compiler inlines calls to `alloc_mem` when the option "optimise for space" is selected.

is translated by the Caml compiler to this LLFL code:

```
let l = alloc(x, alloc(y,zs)) in ...
```

This code allocates and initialises two new cons cells.[2] This is subsequently translated to this code in the Mach intermediate language:

```
t = alloc_mem(8 + 8) (* sizeof(y) + sizeof(zs) *)
[t-...] = y
[t-...] = zs
l = alloc_mem(8 + 8) (* sizeof(x) + sizeof(t) *)
[l-...] = x
[l-...] = t
...
```

In the Mach language allocation is already separated into two more primitive actions: request free memory and store values into that memory. After the phase that combines allocations, the following code results:

```
t = alloc_mem(8 + 8 + 8 + 8)
[t-...] = y
[t-...] = zs
l = t - (8 + 8)
[l-...] = x
[l-...] = t
...
```

Here, the first allocation requests memory for two cons cells and the second call to `alloc_mem` has been substituted by a simple subtraction.

The `C--` back end for Caml performs a direct translation of LLFL code to `C--`, and merging of allocations has not been implemented. Unfortunately, the LLFL code is not in the "appropriate" form to perform this optimisation, since a notion of basic block is required—allocations can be merged only if they occur in the same basic block. Mach code is the ideal intermediate language for allocation merging, but this is too late in the pipeline. (Recall from Section 6.2 that translation to `C--` must be from a higher-level intermediate language than Mach, since Mach code is machine-dependent and

[2]Cons cells (and all other heap objects) contain an extra integer field, a garbage collection header, which I have omitted for presentation purposes. This header describes to the garbage collector the location of pointers in the allocated object. In this example the code that is actually emitted is `alloc(2048,x,alloc(2048,y,zs))`.

some of its constructs cannot be expressed in `C--`.) A-normalised (Flanagan et al. 1993) LLFL code could serve as the input of an allocation merging pass before translation to `C--`.

Accuracy of garbage collection

The gc-descriptors emitted by `cmmc` are type-accurate, but not liveness-accurate (Agesen et al. 1998). That is, only actual pointers are included in gc-descriptors, but a pointer that is dead at a gc-point may end up in the gc-descriptor for that gc-point.[3] This inaccuracy may prevent some garbage from being reclaimed early. This can increase the frequency of garbage collection. It can also make each garbage collection take longer, because more objects have to be copied.

In contrast, the gc-descriptors emitted by the native Caml back end are both type- and liveness-accurate.

Instruction scheduling, memory alignment of branch targets and register allocation

The amount of slowdown of `quicksort` (45.23%) was unexpected. A visual inspection of the generated `C--` revealed no obvious inefficiencies except some unreachable code (a branch immediately followed by another branch). Fixing the `C--` emitter to avoid this resulted in no measurable improvement, though. Inspection of the generated assembly language shows that the main differences between `cmmc` and the native Caml compiler are better instruction scheduling, inclusion of alignment directives for branch targets, and fewer spills and reloads.

Instruction scheduling. The Caml code generator performs a simple pass of instruction scheduling within basic blocks (Muchnick 1997). `cmmc` does not currently implement scheduling. In some programs, scheduling can reduce execution time by a few percent points. In `quicksort`, this figure is 3.6%.[4]

Alignment of branch targets. In modern microprocessors, branches generally execute faster if the branch target falls on the start of a cache line.

[3] This is a limitation of the current implementation of `cmmc`. It is not caused by the use of `C--` as an intermediate language.

[4] This was measured by disabling instruction scheduling in the Caml compiler and comparing execution times.

Compilers emit memory alignment directives for branch targets to ensure that they end up on the "right" addresses. In `quicksort`, branch alignment reduces execution time by 1.9%.

Register allocation. The register allocator in the Caml compiler performs live range splitting of variables that cannot be coloured (Cooper and Simpson 1998). For some Caml programs, this strategy is very successful and results in fewer spills and reloads than in the code generated by `cmmc`. In the case of `quicksort`, the additional memory accesses occur inside the inner loop, which exacerbates the differences in execution times.

Register Usage of Special Functions

Caml intermediate code contains frequent calls to the function `alloc_mem`. This function is a special function in Caml's run-time system written in assembly language that preserves the values of all machine registers. The register allocator of the Caml compiler takes advantage of this knowledge, but this information cannot be communicated to the `C--` register allocator, since `C--` provides no syntax for that. Since calls to `alloc_mem` are frequent, this loss of information could result in measurable increases in execution time. In Section 8.2 I propose to extend `C--` with annotations to express register usage of functions.

Exceptions

Raising and handling exceptions using `C--` continuations is slightly more expensive than the implementation used by the native Caml compiler. `C--` continuations are strictly more expressive than exception handlers (Section 6.2.3), and this has a small run-time cost.

Table 7.6 shows the number and kind of instructions required to enter the scope of a handler, leave the scope of a handler, and raise an exception, all of them in the Alpha back end. Caml targeted to `C--` requires less ALU instructions, but more loads and stores, which are more expensive. (The instruction count of raising an exception does not include the instructions for passing the exception argument to the Caml handler nor to the `C--` continuation. However, this is exactly the same in the native Caml compiler and in the version targeted to `C--`.)

I expect that in real Caml programs the overhead due to the implementation of exceptions should be unnoticeable.

		load	store	ALU
enter scope of handler	Caml	0	2	2
	Caml/C--	0	4	1
leave scope of handler	Caml	1	0	1
	Caml/C--	1	0	0
raise exception	Caml	2	0	2
	Caml/C--	4	0	0

Table 7.6: Instruction counts of exceptions in an Alpha.

7.3 Summary

I have shown that targeting C-- in two real compilers for C and Caml involves substantially less effort than implementing a whole code generator from scratch.

With the lcc/C-- compiler we win both ways, improved performance and reduced implementation effort. The code produced by the Caml/C-- compiler provides a very reasonable trade-off against a reduced implementation effort.

Some optimisations are currently missing from cmmc; two examples are instruction scheduling and live range splitting. If the C-- compiler is further improved and merging of allocations is added to the Caml C-- emitter, it is reasonable to expect that Caml/C-- can produce code of the same quality as that produced by the original Caml compiler. Furthermore, all compilers targeting C-- will benefit in the future from improved optimisations in cmmc.

Chapter 8

Proposed Extensions to C--

In the design of a generic intermediate language there are two important goals:

- All the constructs of a variety of source languages should be expressible in the intermediate language without loss of essential semantic information.
- The intermediate language should translate to efficient machine code for a wide range of target architectures.

The intermediate language designer has to resolve the tension between language simplicity and trying to support every source language and every target architecture. As Muchnick says, "Intermediate language design is largely an art, not a science" (Muchnick 1997).

I have evaluated C-- as a target language for C and Caml, and I have demonstrated that C-- can effectively support a wide variety of high-level language constructs. In Section 8.1 I illustrate the tension mentioned above by identifying a high-level construct present in C, C++ and Java that cannot be expressed in C--. This loss of semantic information can result in machine code that violates the semantics of the source program. I propose an extension to C-- to capture this source-level semantics.

In Section 8.2 I enumerate a number of constructs that can be added to C-- so that high-level information can be used to generate more efficient code. Like the annotations proposed in Chapter 3, these constructs are useful for any generic intermediate language, not just C--.

8.1 The `volatile` Type Qualifier

Consider the following C function definition:

```
1  void f(int *iptr, volatile int *viptr) {
2    *iptr = 0;
3    *iptr = 1;
4    *viptr = 0;
5    *viptr = 1;
6  }
```

The statement in line 2 is a *dead store* and can be eliminated (Muchnick 1997). The similar-looking one in line 4 cannot be removed, though. In C, a store through a `volatile` pointer cannot be optimized away, even if it appears to be dead:

> An lvalue expression of a `volatile`-qualified type should not participate in optimisations that would increase, decrease, or delay any references to, or modifications of, the object (Harbison and Steele 1995, p. 83).

According to this restriction, several common back-end optimisations are illegal on references to `volatile` values, including dead store elimination, partial redundancy elimination (Morel and Renvoise 1979) and register promotion (Cooper and Lu 1997).

The `volatile` type qualifier is present not only in C, but also in C++ and Java. (See (Harbison and Steele 1995) for the motivation and examples of use of the `volatile` type qualifier.)

`C--` does not provide the equivalent of `volatile`. Thus, `lcc` translates the function above to the following `C--` code:

```
1  foreign "C" f(bits64 iptr, bits64 viptr) {
2    bits32[iptr] = 0;
3    bits32[iptr] = 1;
4    bits32[viptr] = 0;
5    bits32[viptr] = 1;
6    foreign "C" return();
7  }
```

Given this code, an optimizing C-- compiler could delete the two stores in lines 2 and 4. But the `volatile` qualifier in the source C program explicitely forbids that stores through the pointer `viptr` be altered!

This is an example of semantic information that is essential to the code generator but cannot be expressed in C--. The loss of this information can result in incorrect code being generated. Given the simplicity of C--'s type system, perhaps the best way to deal with `volatile` values in C-- is not by qualifying types in value declarations, but by annotating individual loads and stores. For instance, the syntax of memory references in C-- could be extended from *type*`[`*expr* [*aligned*]`]` to *type*`[`*expr* [*aligned*] [`"volatile"`]`].

8.2 C-- Extensions for Performance

In this section I enumerate two constructs that could be added to C-- so that common high-level constructs can be compiled to more efficient machine code via C--.

Fine Control of Register Usage

The following code sequence occurs frequently in Caml's Mach intermediate language:

```
x = ...
ptr = alloc_mem(sz)
[ptr] = x
```

`alloc_mem` is a function in Caml's run-time system written in assembly language that preserves the values of all machine registers. The register allocator of the Caml compiler takes advantage of this knowldege and is free to assign `x` to a caller-saves register. (This is cheaper than saving `x` in memory across the call to `alloc_mem` and also cheaper than using a callee-saves register, which incurs the cost of saving and restoring the register at function entry and exit respectively.)

To the C-- compiler, though, `alloc_mem` is like any other function; the compiler has to assume that it may overwrite the caller-saves registers. Therefore, it has to spill `x`, and indeed any other variables that are live across the call to `alloc_mem`. Since such calls are very frequent in Caml programs, this can have a noticeable effect on the performance of the generated code.

Some form of annotation for calls in C-- would be very useful to express information about the particular register usage of frequently-called special functions.

Testing for Overflow

Some front ends would benefit if C-- provided a language construct to test for overflow of arithmetic operations. Some programming languages, for instance SML (Milner et al. 1990), require that an exception be thrown when overflow occurs in an arithmetic operation. This requires a check for overflow after every potentially-overflowing arithmetic operation. The SML to C compiler checks for overflow by bit-testing of the operands and results of every arithmetic operation that may overflow (Tarditi et al. 1992). This is substantially more expensive than what can be achieved using assembly language directly, since overflow flags are often visible. C-- should provide a construct to test for overflow that is convenient for the front end to emit and that can be translated to efficient machine code. For instance, C-- could provide a built-in unary operator `%overflow`. When this operator appears in the condition of an `if` statement, the C-- compiler would emit a "branch-on-overflow" instruction, or code to check the overflow bit and branch, depending on what the target architecture provides.

Chapter 9

Conclusions

In this dissertation I have described the development of a generic code generation infrastructure, consisting of a compiler and a low-level run-time system for `C--`, a portable compiler intermediate language. In addition I have proposed improvements and enhancements to `C--` and to its run-time system.

The starting point for my work was an initial proposal for an intermediate language and a low-level run-time system. Neither of them had been implemented completely nor evaluated in a real compiler. I have refined and generalised their design and built an implementation. The complete system provides a *practical code generation infrastructure.*

I have shown that this code generation infrastructure (1) can be used to compile very diverse source programming languages; (2) substantially reduces the overall effort required to implement a programming language; (3) can capture and retain high-level information useful for low-level optimisations; (4) can capture and retain high-level information necessary to support run-time services; (5) does not increase the complexity of the compiler's front end, the run-time system, or the interface between the two; and (5) produces code competitive to that generated by monolithic compilers.

A detailed summary of the contributions follows.

9.1 Summary of Contributions

I have implemented the first compiler and run-time system for `C--`. I have also retrofitted two existing compilers to use `C--` as a target language, including one for a language that supports both garbage collection and exception handling. (These are the first compilers to use `C--`.) I have shown that `C--`

can be used to compile very dissimilar source languages.

As a direct result of the implementation, I have proposed improvements to both C-- and its run-time interface. I have proposed that C-- be extended with syntax to express all of the following:

- `volatile` values (Section 8.1);
- indirect branches and first-class labels (Section 6.1.1);
- special register usage of functions (Section 8.2);
- necessary pointer maps for accurate garbage collection (Section 3.1.5).

The first one is essential to convey high-level semantics of several source languages, including C and Java; without this extension it is not possible to translate these languages to C-- in a way that preserves the original semantics of programs.[1] The other extensions are important for efficiency. In addition, I have extended the C-- run-time interface with functionality to support the following high-level run-time services:

- generational stack collection (Section 4.4);
- stack walking in languages that provide a foreign function interface (Section 4.7).

Finally, I have augmented C--'s run-time interface so that accurate garbage collection can be implemented more easily and efficiently than is possible with the original interface (Section 4.5).

The following are my main contributions to the compilation of C--:

- The translation of continuations (Section 5.1.3). C-- continuations are more expressive than exception handlers. The consequence is that continuations require a run-time representation different from the one that is often used for exception handlers. It turns out that this representation has a slight run-time cost. I have identified the costs of using C-- continuations to implement exception handling.

[1] The only feature of C that C-- does not support directly is functions with a variable number of arguments.

- A novel compilation method for globally reserved machine registers (Section 5.1.5). The result is that the register allocator can use a reserved register for other variables in parts of the program where the reserved register is dead. This can decrease spills.

- The selection of the parameter passing convention (Section 5.1.1). To support tail-call optimisation, `C--` may use only one of several possible calling conventions. I also identify situations in which it is possible to use different, more efficient, calling conventions.

To communicate high-level program properties from the front end of the compiler to the code generator, I have proposed that the intermediate language be extended with syntax to express all of the following:

- control flow of non-returning calls (Section 3.1.1);

- control flow due to exceptions (Section 3.1.2);

- branch prediction (Section 3.1.3);

- data prefetching (Section 3.1.4);

- memory disambiguation (Section 3.1.6);

- side effects of functions (Section 3.1.7);

These constructs are general: they can be retrofitted to other existing or future intermediate languages, not just `C--`.

9.2 Directions for Future Research

9.2.1 Debugging

Debuggers need compiler support to examine and modify the values of variables, to insert breakpoints, etc. (Hanson 1999). The back-end information required for debugging is much the same as what is needed to inspect the stack for garbage collection or exception handling. Front-end information is necessary too, such as the types of variables, their scopes, etc. These two sources of information are emitted by the compiler and consumed during a debugging session.

It would be possible to write a debugger for a language that uses C-- in the back end. Front-end and back-end information would be emitted separately and combined by the debugger, using the C-- run-time interface (Peyton Jones et al. 1999). Instead of implementing a debugger from scratch, the compiler could generate debugging information in a standard debugging format, so that an off-the-shelf debugger could be used. In order to do this, the front-end information would be combined with the debugging information emitted by the C-- compiler, at compile time. The mechanisms that C-- should provide to interpret this information need to be investigated.

9.2.2 Heap-allocated Activations

One option to implement first-class continuations is to allocate activation frames in the heap instead of in the stack (Appel 1992). This method has the advantage that other language features, such as concurrency, are simpler to implement (Reppy 1999).[2] There are intermediate languages, lower level than C--, in which the front end manages the allocation of activations itself (George and Leung 2000). This provides great flexibility, but also requires that the front end maintain the mapping from spilled variables to spill locations. This is clearly a back-end issue, since it involves machine-dependent details. If the back end could take care of it, it would simplify the front end considerably. An open question is how to support heap-allocated activations in C-- in such a way that spilling is implemented by the C-- compiler, without complicating the front end unduly.

9.2.3 Type Systems to Express Program Properties

An expressive type system provides an alternative to annotations to communicate static program properties to the code generator. Typed intermediate languages have been developed in which it is possible to encode complex propositions and proofs about a program (Shao et al. 2002). An open research question is how to overlay a more expressive type system onto C--.

[2] E.g., the run-time system does not have to implement a mechanism to resize the stack of a thread, since no "stack overflow" errors occur at run-time.

9.2.4 Annotations for Mobile Intermediate Representations

This dissertation has not addressed the issue of annotation safety when the intermediate representation is mobile code. In this scenario, malicious third parties can alter the annotations while the code is in transit. Tampering with branch prediction and prefetching annotations can result in slower execution time, but it cannot alter the original semantics of the program. Others, like annotations that express control-flow or register usage, are sensitive and would need a safety mechanism if they are transmitted across insecure channels. The standard tools of cryptography can be used for these situations, but language-based mechanisms, like proof-carrying code (Necula 1997), constitute a promising research direction.

Appendix A

Syntax of C--

compilation unit ⇒ {*toplevel*}

toplevel ⇒ `section` *string* `{` {*section*} `}`
| *decl*
| *function*

section ⇒ *decl*
| *function*
| *datum*

decl ⇒ `import` *name* {`,` *name*} `;`
| `export` *name* {`,` *name*} `;`
| `register` *type name int* `;`

datum ⇒ *name* `:`
| `align` *int* `;`
| *type* [*size*] [*init*] `;`

init ⇒ `{` *expr* {`,` *expr*} `}`
| *string*

size ⇒ `[` [*expr*] `]`

body ⇒ {*decl* | *stackdecl* | *stmt*}

function ⇒ [*conv*] *name* `(` [*formals*] `)` `{` *body* `}`

formal ⇒ *type name*

actual ⇒ *expr*

formals ⇒ *formal* {`,` *formal*}

actuals ⇒ *actual* {`,` *actual*}

stackdecl ⇒ `stackdata` `{` {*datum*} `}`

stmt ⇒ `;`
| `if` *expr* `{` *body* `}` [`else` `{` *body* `}`]
| `switch` *expr* `{` {*arm*} `}`
| *lvalue* `=` *expr* `;`
| [*name* {`,` *name*} `=`] [*conv*] *expr* `(` [*actuals*] `)` {*flow*} `;`
| [*conv*] `jump` *expr* `(` [*actuals*] `)` `;`
| [*conv*] `return` [`(` [*actuals*] `)`] `;`
| *name* `:`
| `continuation` *name* `(` *name* {`,` *name*} `)` `:`
| `goto` *expr* [`(` *name* {`,` *name*} `)`] `;`
| `cut to` *expr* `(` [*actuals*] `)` `;`

arm ⇒ `case` *range* {`,` *range*} `:` `{` *body* `}`

range ⇒ *expr* [`..` *expr*]

lvalue ⇒ *name*
| *type* `[` *expr* [*aligned*] `]`

flow ⇒ `also cuts to` *name* {`,` *name*}
| `also aborts`

expr ⇒ *int*
| *float*
| `'` *char* `'`
| *name*
| *type* `[` *expr* [*aligned*] `]`
| `(` *expr* `)`
| *expr op expr*
| `~` *expr*
| `%` *name* `(` [*actuals*] `)`

type ⇒ *bitsn*
| *floatn*

conv ⇒ `foreign` *string*

aligned ⇒ `align` *int*

op ⇒ `+` | `-` | `*` | `/` | `%` | `&` | `|` | `^` | `@<<` | `>>` | `==` | `!=` | `>` | `<` | `>=` | `<=`

Appendix B

Comparing C-- to MLRISC

In this appendix I compare, by means of an example, C-- and MLRISC, two generic intermediate languages (ILs) of different semantic level. In general, the semantic level of an IL determines how much translation work has to be done by the front end and how much remains to be done by the back end. The example has been chosen to highlight the differences between the two ILs, rather than to provide an exhaustive comparison.

C-- and MLRISC have been designed from the outset with language independence in mind. This is in contrast to many other ILs like ANDF, GCC's Tree language (FSF 2002) or JVM code, which are biased towards a specific language or a family of related languages. Most ILs support a number of primitive types, simple expressions (literals, variables, arithmetic, logical, bit manipulation, type conversion, etc.), and simple statements (assignments, basic control flow, etc.). High-level ILs provide abstractions of programming languages. For instance, GCC's Tree language supports all the primitive and aggregate types of C and C++, plus class and method types. Low-level ILs such as C-- provide abstractions of hardware. They support hardware types (word and floating point), hide restrictions of the architecture and the instruction set (limited number of registers, limited addressing modes, etc.), and hide the conventions of the operating system (parameter passing, system stack, etc.). C-- and MLRISC do not support aggregate types, such as records, arrays, or variants. As a consequence, a front end that targets C-- or MLRISC has to translate aggregate declarations and references to explicit memory allocation and memory references. C-- provides direct support for function declarations, function calls and parameter passing, while MLRISC does not. Neither C-- nor MLRISC support

directly class and method definitions, nor method invocation. MLRISC is lower-level than `C--`, which means that a front end has to do more work to target MLRISC than to target `C--`. In exchange, the front end that targets MLRISC has complete control over parameter passing, allocation of activation frames, and even what code to generate when a virtual register must be spilled.

Chapter 6 shows how to translate a number of higher-level constructs into `C--`(conditional expressions, function calls nested inside expressions, exception handling, etc.). This appendix demonstrates the translation of additional constructs that are not supported directly by `C--` (nested scopes, access to aggregate values and access to sub-word values.). It also shows constructs that translate directly to `C--` but need to be expanded to primitive constructs in MLRISC (activation frame set-up, accessing function parameters and results, and management of callee-saves registers.). The differences arise due to the lack of direct support for function definitions and parameter passing in MLRISC.

The source code of the example, in C, appears in Figure B.1. The `C--` translation is as emitted by `lcc`, edited only for readability. The only assumption is that C `int`s are represented in 32 bits; it is shown in Figure B.2. The MLRISC code is emitted by `cmmc`, targeting the Alpha architecture; it appears in Figure B.3.

In the MLRISC code, integer registers are named `$0` to `$31`, floating point registers are named `$f0` to `$f31` and temporaries are named as virtual registers with number different from any physical register. (In this example numbers above 500 are used.) The MLRISC code is much more target-specific. For instance, all the details of the calling convention are explicit. This includes allocation/deallocation of activation frames, parameter passing in certain registers, and saving/restoring the callee-saves registers. All this code would be similar for other architectures, except that the specific registers would vary. The exception is two instructions commented as Alpha-specific that update register `$29` on function entry and return. In this example, all the copies between callee-saves registers and temporaries can be eliminated by register coalescing. The same is true of the copies between parameter registers and temporaries. A clever compiler would generate no machine code for all those copies.

```
extern int f(int, int, int, int, int, int, int);
extern int g(void);

int example(int parameter1, int parameter2, int parameter3,
            int parameter4, int parameter5, int parameter6,
            int parameter7) {

  int local_var, incoming_result;

  /* Aggregate value */
  struct { float i; float j;} local_struct;

  /* Sub-word values */
  struct { int a:5, b:5, c:3;} bitfields;

  /* Nested scope */
  { int local_var = 0; }

  /* Access aggregate value */
  local_struct.j = 0.0;

  /* Access sub-word value */
  bitfields.a = local_var;

  /* Access parameters, results, and pass parameters */
  local_var = parameter1 + parameter7;

  incoming_result = f(1,2,3,4,5,6,7);

  return(0);
}
```

Figure B.1: C code.

```
import f, g;
export example;

foreign "C" example(bits32 parameter1, bits32 parameter2,
                    bits32 parameter3, bits32 parameter4,
                    bits32 parameter5, bits32 parameter6,
                    bits32 parameter7 ) {

  bits32 local_var, incoming_result;

  /* struct { float i; float j;} local_struct;
     struct { int a:5, b:5, c:3;} bitfields;
     Structs are allocated in memory */
  stackdata { align 4; local_struct: bits32[2];}
  stackdata { align 4; bitfields: bits32;}

  /* { int local_var = 0; }
     Nested scopes are implemented by renaming identifiers
     that already exist in outer scopes */
  bits32 local_var1;
  local_var1 = 0;

  /* local_struct.j = 0.0;
     Access to struct is an explicit memory store */
  float32[local_struct +u 4] = 0.0;

  /* bitfields.a = local_var;
     Access to sub-word value via bit manipulation */
  bits32[bitfields] = (bits32[bitfields] & -32) |
                      (((local_var << 27) >> 27) & 31);

  /* Access to parameters, results, and parameter passing are
     supported directly in C-- */
  local_var = parameter1 + parameter7;

  incoming_result = foreign "C" f(1, 2, 3, 4, 5, 6, 7);

  foreign "C" return (0);
}
```

Figure B.2: C-- code for the C program in Figure B.1.

```
example:
/* Alpha-specific code */
$29 := $27

/* struct { float i; float j;} local_struct;
   struct { int a:5, b:5, c:3;} bitfields;
   Allocate activation frame */
$sp := $sp - (16)

/* $f2-$f9 are FP callee-saves registers. Save them
   to temporaries */
$f527,$f528,$f529,$f530,$f531,$f532,$f533,$f534 := $f2,$f3,$f4,
                                                   $f5,$f6,$f7,$f8,$f9

/* $9-$15, $26 are integer callee-saves registers.
   Save them to temporaries */
$519,$520,$521,$522,$523,$524,$525,$526 := $9,$10,$11,$12,$13,
                                            $14,$15,$26

/* Incoming parameters in registers $16-$21 and stack location.
   Copy them to temporaries */
$518 := mem.32[$sp + (16)]
$512,$513,$514,$515,$516,$517 := $16,$17,$18,$19,$20,$21

/* { int local_var = 0; } */
$541 := 0

/* local_struct.j = 0.0;
   Access to struct is an explicit memory store.
   Register $f31 contains the value 0.0 */
mem.s[$sp + (8)] := $f31

/* bitfields.a = local_var;
   Access to sub-word value via bit manipulation */
mem.32[$sp + (4)] := ((mem.32[$sp + (4)]) & -32) |
                     ((($539 << 27) ~>> 27) & 31)
```

```
/* local_var = parameter1 + parameter7; */
$539 := $512 + $518

/* incoming_result = f(1,2,3,4,5,6,7);
   Pass parameters in registers $16-$21 and stack location */
$16 := 1
$17 := 2
$18 := 3
$19 := 4
$20 := 5
$21 := 6
mem.32[$sp] := 7
call $f
/* Alpha-specific code */
$29 := $26

/* Incoming result in register $0 */
$540 := $0

/* return (0);
   Return result in register $0 */
$0 := 0

/* Restore values of FP and integer callee-saves registers */
$f2,$f3,$f4,$f5,$f6,$f7,$f8,$f9 := $f527,$f528,$f529,$f530,
                                   $f531,$f532,$f533,$f534
$9,$10,$11,$12,$13,$14,$15,$26 := $519,$520,$521,$522,$523,
                                  $524,$525,$526

/* Deallocate activation frame. Return */
$sp := $sp + (16)
return
```

Figure B.3: MLRISC code for the C program in Figure B.1.

Bibliography

Abelson, H., R. K. Dybvig, C. T. Haynes, G. J. Rozas, N. I. A. IV, D. P. Friedman, E. Kohlbecker, G. L. Steele Jr., D. H. Bartley, R. Halstead, D. Oxley, G. J. Sussman, G. Brooks, C. Hanson, K. M. Pitman, and M. Wand (1998, August). Revised report on the algorithmic language Scheme. *Higher-Order and Symbolic Computation 11*(1), 7–105. 64

Agesen, O., D. Detlefs, and J. E. B. Moss (1998, 17–19 June). Garbage collection and local variable type-precision and liveness in Java virtual machines. See PLDI (1998), pp. 269–279. *SIGPLAN Notices* 33(5), May 1998. 7, 27, 116

Aho, A. V., R. Sethi, and J. D. Ullman (1986). *Compilers: Principles, Techniques and Tools*. Addison-Wesley. 90

Appel, A. W. (1992). *Compiling with Continuations*. Cambridge University Press. 71, 75, 90, 126

Appel, A. W. (1998). *Modern Compiler Implementation in ML*. Cambridge University Press. 26, 71, 86, 94

Appel, A. W., J. S. Mattson, and D. R. Tarditi (1994). *A lexical Analyzer Generator for Standard ML. Version 1.6.0*. Available from `http://cm.bell-labs.com/cm/cs/what/smlnj/doc/ML-Lex/`. 61

Bacon, D. F., S. L. Graham, and O. J. Sharp (1994, December). Compiler transformations for high-performance computing. *ACM Computing Surveys 26*(4), 345–420. 2, 19

Ball, T. and J. R. Larus (1993, 23–25 June). Branch prediction for free. See PLDI (1993), pp. 300–313. *SIGPLAN Notices* 28(6), June 1993. 25

Barabash, K., N. Buchbinder, T. Domani, E. K. Kolodner, Y. Ossia, S. S. Pinter, J. Shepherd, R. Sivan, and V. Umansky (2001, April 23–24). Mostly accurate stack scanning. In *Proceedings of the Java^{TM} Virtual Machine Research and Technology Symposium (JVM-01)*, Monterey, California, USA, pp. 153–170. USENIX Association. 36, 51, 52, 54

Bartlett, J. (1989). SCHEME to C: a portable Scheme-to-C compiler. Technical Report RR 89/1, DEC WRL. 5

Bell, J. R. (1973, June). Threaded code. *Communications of the ACM 16*(6), 370–372. 83

Benitez, M., P. Chan, J. Davidson, A. Holler, S. Meloy, and V. Santhanam (1991, March). ANDF: Finally an UNCOL after 30 years. Technical Report TR-91-05, University of Virginia, Department of Computer Science, Charlottesville, VA. 7, 12

Benitez, M. E. and J. W. Davidson (1988, 22–24 June). A portable global optimizer and linker. In *Proceedings of the ACM SIGPLAN'88 Conference on Programming Language Design and Implementation (PLDI)*, Atlanta, Georgia, pp. 329–338. *SIGPLAN Notices* 23(7), July 1988. 4, 19

Benton, N. and A. Kennedy (2000). Monads, effects and transformations. In A. Gordon and A. Pitts (Eds.), *Electronic Notes in Theoretical Computer Science*, Volume 26. Elsevier Science Publishers. 29, 32

Benton, N., A. Kennedy, and G. Russell (1998, June). Compiling Standard ML to Java bytecodes. See ICFP (1998), pp. 129–140. 5

Blume, M. (2001). No-longer-foreign: Teaching an ML compiler to speak C "natively". In *Electronic Notes in Theoretical Computer Science*, Volume 59(1). Elsevier Science. 50

Boehm, H.-J. (2002, 16-18 January). Bounding space usage of conservative garbage collectors. In *Conference Record of POPL'02: The 29th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages"*. 7

Boehm, H.-J. and M. Weiser (1988). Garbage collection in an uncooperative environment. *Software Practice and Experience 18*(9), 807–820. 7, 35, 37

Boquist, U. (1999, April). *Code Optimisation Techniques for Lazy Functional Languages*. Ph. D. thesis, Chalmers University of Technology, Gothenburg, Sweden. 60

Bothner, P. (1998). Kawa: Compiling Scheme to Java. In *Lisp Users Conference*, Berkeley, CA. 5

Briggs, P. and K. D. Cooper (1994, 20–24 June). Effective partial redundancy elimination. In *Proceedings of the ACM SIGPLAN'94 Conference on Programming Language Design and Implementation (PLDI)*, Orlando, Florida, pp. 159–170. *SIGPLAN Notices* 29(6), June 1994. 29

Chase, D. (1994, June). Implementation of exception handling, Part I. *The Journal of C Language Translation 5*(4), 229–240. 3

Cheng, B.-C. and W.-m. W. Hwu (2000, 18–21 June). Modular interprocedural pointer analysis using access paths: Design, implementation, and evaluation. See PLDI (2000), pp. 57–69. *SIGPLAN Notices* 35(5), May 2000. 29, 32

Cheng, P., R. Harper, and P. Lee (1998, 17–19 June). Generational stack collection and profile-driven pretenuring. See PLDI (1998), pp. 162–173. *SIGPLAN Notices* 33(5), May 1998. 8, 33, 44, 45, 48

Cho, S., J. Tsai, Y. Song, B. Zheng, S. Schwinn, X. Wang, Q. Zhao, Z. Li, D. Lilja, and P. Yew (1998, August). High-level information : An approach for integrating front-end and back-end compilers. In *Proceedings of the 1998 International Conference on Parallel Processing (ICPP '98)*, Washington - Brussels - Tokyo, pp. 346–355. IEEE USA. 31

Choi, J.-D., D. Grove, M. Hind, and V. Sarkar (1999, September 6). Efficient and precise modeling of exceptions for the analysis of Java programs. In *Proceedings of the ACM SIGPLAN-SIGSOFT Workshop on Program Analysis for Software Tools and Engineering*, Volume 24.5 of *Software Engeneering Notes (SEN)*, N. Y., pp. 21–31. ACM Press. 31, 66

Clinger, W. D. (1998, 17–19 June). Proper tail recursion and space efficiency. See PLDI (1998), pp. 174–185. *SIGPLAN Notices* 33(5), May 1998. 64, 86

Codognet, P. and D. Diaz (1995, June 13–18). WAMCC: Compiling Prolog to C. In L. Sterling (Ed.), *Proceedings of the 12th International Conference on Logic Programming*, Cambridge, pp. 317–332. MIT Press. 5

Conway, M. (1958, October). Proposal for an UNCOL. *Communications of the ACM 1*(10), 5–8. 7

Cooper, K. D. and J. Lu (1997, 15–18 June). Register promotion in C programs. See PLDI (1997), pp. 308–319. *SIGPLAN Notices* 32(5), May 1997. 120

Cooper, K. D. and L. T. Simpson (1998). Live range splitting in a graph coloring register allocator. In *Proceedings of the 7th International Conference on Compiler Construction (CC'98)*, Volume 1383 of *LNCS*, Lisbon (Portugal), pp. 174–188. Springer–Verlag. 117

Diwan, A., K. S. McKinley, and J. E. B. Moss (1998, 17–19 June). Type-based alias analysis. See PLDI (1998), pp. 106–117. *SIGPLAN Notices* 33(5), May 1998. 29

Diwan, A., J. E. B. Moss, and R. L. Hudson (1992, 17–19 June). Compiler support for garbage collection in a statically typed language. In *Proceedings*

of the ACM SIGPLAN'92 Conference on Programming Language Design and Implementation (PLDI), San Francisco, California, pp. 273–282. *SIGPLAN Notices* 27(7), July 1992. 3, 7, 27, 35, 47

Diwan, A., D. Tarditi, and J. E. B. Moss (1995, August). Memory subsystem performance of programs with intensive heap allocation. *ACM Transactions on Computer Systems 13*(4), 244–273. 7, 114

Dulong, C., R. Krishnaiyer, D. Kulkarni, D. Lavery, W. Li, J. Ng, and D. Sehr (1999). An overview of the Intel IA-64 compiler. *Intel Technology Journal* (Q4). 25

Ertl, M. A. (1999, January 20–22,). Optimal code selection in DAGs. In *Conference Record of POPL'99: The 26th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, San Antonio, Texas, pp. 242–249. 94

Evans, J. S. and R. H. Eckhouse (1999). *Alpha RISC architecture for programmers*. Prentice-Hall PTR. 72

Ferrante, J., K. J. Ottenstein, and J. D. Warren (1987, July). The program dependence graph and its use in optimization. *ACM Transactions on Programming Languages and Systems 9*(3), 319–349. 20

Finne, S., D. Leijen, E. Meijer, and S. Peyton Jones (1998, June). H/Direct: A binary foreign function interface for Haskell. See ICFP (1998), pp. 153–162. 50

Flanagan, C., A. Sabry, B. F. Duba, and M. Felleisen (1993, 23–25 June). The essence of compiling with continuations. See PLDI (1993), pp. 237–247. *SIGPLAN Notices* 28(6), June 1993. 93, 94, 116

Franz, M. and T. Kistler (1997, December). Slim binaries. *Communications of the ACM 40*(12), 87–94. 12

Fraser, C. W. and D. R. Hanson (1995). *A Retargetable C Compiler: Design and Implementation*. Addison-Wesley. 12, 82, 83, 89, 104, 109

FSF (2001). *The GNU Compiler for the Java Programming Language*. FSF. Available from `http://http://gcc.gnu.org/java/`. 50

FSF (2002). *GNU Compiler Collection (GCC) Internals*. "Free Software Foundation". Available from `http://gcc.gnu.org/onlinedocs/gccint/`. 130

George, L. and A. W. Appel (1996, May). Iterated register coalescing. *ACM Transactions on Programming Languages and Systems 18*(3), 300–324. 76, 84, 109

George, L. and A. Leung (2000, November). MLRISC: A framework for retargetable and optimizing compiler back ends. Unpublished report available from `http://www.cs.nyu.edu/leunga/www/MLRISC/Doc/html/index.html`. 4, 19, 33, 61, 126

GHC Team (2001). *The Glasgow Haskell Compiler User's Guide*. Available from `http://www.haskell.org/ghc/`. 50

Ghiya, R., D. Lavery, and D. Sehr (2001, June 20–22,). On the importance of points-to analysis and other memory disambiguation methods for C programs. In *Proceedings of the ACM SIGPLAN '01 Conference on Programming Language Design and Implementation*, Snowbird, Utah, pp. 47–58. *SIGPLAN Notices,* 36(5), May 2001. 32

Hailperin, M. (1998, November). Cost-optimal code motion. *ACM Transactions on Programming Languages and Systems 20*(6), 1297–1322. 25

Hanson, D. R. (1999, August). A machine-independent debugger — revisited. *Software—Practice and Experience 29*(10), 849–862. 125

Harbison, S. P. and G. L. Steele (1995). *C—A Reference Manual* (Fourth ed.). Upper Saddle River, NJ 07458, USA: Prentice-Hall. 84, 120

Hausman, B. (1994). Turbo erlang: Approaching the speed of C. In E. Tick and G. Succi (Eds.), *Implementations of Logic Programming Systems*, pp. 119–135. Kluwer Academic Publishers. 5

Henderson, F. (2002, June). Accurate garbage collection in an uncooperative environment. In D. Detlefs (Ed.), *ISMM'02 Proceedings of the Third International Symposium on Memory Management*, ACM SIGPLAN Notices, Berlin, pp. 150–156. ACM Press. 7

Henderson, F., T. Conway, and Z. Somogyi (1995). Compiling logic programs to C using GNU C as a portable assembler. In *ILPS'95 Postconference Workshop on Sequential Implementation Technologies for Logic Programming*, Portland, Or, pp. 1–15. 5

Henderson, F., T. Conway, Z. Somogyi, D. Jeffery, P. Schachte, S. Taylor, C. Speirs, and T. Dowd (2002). *The Mercury Language Reference Manual*. Available from `http://www.cs.mu.oz.au/research/mercury/`. 50

Hennessy, J. (1981, January 26–28,). Program optimization and exception handling. In *Conference Record of the Eigth Annual ACM Symposium on Principles of Programming Languages*, Williamsburg, Virginia, pp. 200–206. ACM SIGACT-SIGPLAN. 66

Hudson, R. L., J. E. B. Moss, A. Diwan, and C. F. Weight (1991, September). A language-independent garbage collector toolkit. Technical Report COINS 91-47, University of Massachusetts at Amherst, Department of Computer and Information Science. 60

ICFP (1998, June). *Proceedings of the ACM SIGPLAN International Conference on Functional Programming (ICFP'98)*. 137, 139

ISO/IEC (1999, December). *ISO/IEC 9899:1999 Standard for the C programming language (C99)*. Available from `http://www.iso.ch`. 26

Jones, R. E. (1996, July). *Garbage Collection: Algorithms for Automatic Dynamic Memory Management*. Wiley. With a chapter on Distributed Garbage Collection by R. Lins. 6, 7, 34

Kahrs, J. (2001, 6 October). JVM as UNCOL. Message posted to the compilers mailing list. Available from `http://compilers.iecc.com/comparch/article/01-10-016`. 5

Kelsey, R. and P. Hudak (1989, January 11–13,). Realistic compilation by program transformation – detailed summary. In *Conference Record of the Sixteenth Annual ACM Symposium on Principles of Programming Languages*, Austin, Texas, pp. 281–292. ACM SIGACT-SIGPLAN: ACM Press. 90

Koenig, A. and B. Stroustrup (1990, July/August). Exception handling for C++. *Journal of Object Oriented Programming 3*(2), 16–33. 3, 36

Kranz, D. A., R. Kelsey, J. Rees, P. H. an d James Philbin, and N. Adams (1986, June). ORBIT: An optimizing compiler for Scheme. In *Proceedings of the ACM SIGPLAN '86 Symposium on Compiler Construction*, Volume 21(7) of *ACM SIGPLAN Notices*, Palo Alto, CA, pp. 219–233. ACM Press. 75

Krintz, C. and B. Calder (2001, 20–22 June). Using annotations to reduce dynamic optimization time. In *Proceedings of the ACM SIGPLAN'01 Conference on Programming Language Design and Implementation (PLDI)*. 31

Leroy, X. (1990). The ZINC experiment, an economical implementation of the ML language. Technical Report 117, INRIA. 93

Leroy, X., D. Doligez, J. Garrigue, D. Rémy, and J. Vouillon (2002). The Objective Caml system: Documentation and user's manual. Available from `http://caml.inria.fr`. 1, 50, 82, 96, 105

Leroy, X. and F. Pessaux (2000, March). Type-based analysis of uncaught exceptions. *ACM Transactions on Programming Languages and Systems 22*(2), 340–377. 23, 101

Liang, S. (1999). *Java Native Interface: Programmer's Guide and Specification*. Reading, MA, USA: Addison-Wesley. 50

Lieberman, H. and C. E. Hewitt (1983). A real-time garbage collector based on the lifetimes of objects. *Communications of the ACM 26(6)*, 419–429. Also report TM–184, Laboratory for Computer Science, MIT, Cambridge, MA, July 1980 and AI Lab Memo 569, 1981. 44

Lindholm, T. and F. Yellin (1999, April). *The Java Virtual Machine Specification* (Second ed.). The Java Series. Addison Wesley Longman, Inc. 5, 12

Lueh, G.-Y. and T. Gross (1997, 15–18 June). Call-cost directed register allocation. See PLDI (1997), pp. 296–307. *SIGPLAN Notices* 32(5), May 1997. 75

Meijer, E. and J. Gough (2001). Technical overview of the common language runtime. Available from `http://research.microsoft.com/~emeijer/`. 60

Milner, R., M. Tofte, and R. Harper (1990). *The Definition of Standard ML*. MIT Press. 122

Morel, É. and C. Renvoise (1979, February). Global optimization by suppression of partial redundancies. *Communications of the ACM 22*(2), 96–103. 120

Mowry, T. C., M. S. Lam, and A. Gupta (1992, September). Design and evaluation of a compiler algorithm for prefetching. In *Fifth International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS V)*, Boston, pp. 62–73. 32

Muchnick, S. S. (1997). *Advanced compiler design and implementation*. San Mateo, California, USA: Morgan Kaufmann Publishers. 2, 12, 19, 89, 116, 119, 120

Necula, G. C. (1997, January 15–17,). Proof-carrying code. In *Conference Record of POPL'97: The 24th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, Paris, France, pp. 106–119. 127

Novack, S., J. Hummel, and A. Nicolau (1995). A simple mechanism for improving the accuracy and efficiency of instruction-level disambiguation. In *Languages and Compilers for Parallel Computing*, Volume 1033 of *LNCS*. 29

Otto, T. P. (2001). Apl2c. the APL compiler. Available from `http://www.apl2c.com`. 5

Pelegrí-Llopart, E. and S. L. Graham (1988, January 13–15,). Optimal code generation for expression trees: An application of BURS theory. In *Conference Record of the Fifteenth Annual ACM Symposium on Principles of Programming Languages*, San Diego, California, pp. 294–308. ACM SIGACT-SIGPLAN: ACM Press. 94

Peyton Jones, S., N. Ramsey, and F. Reig (1999, September). C--: a portable assembly language that supports garbage collection. In *International Conference on Principles and Practice of Declarative Programming*, pp. 1–28. LNCS 1702. 5, 8, 12, 19, 33, 37, 39, 42, 47, 49, 52, 98, 126

Peyton Jones, S. L. (1996, 22–24 April). Compiling Haskell by Program Transformation: A Report from the Trenches. In H. Nielson (Ed.), *ESOP'96 — European Symposium on Programming*, Volume 1058 of *Lecture Notes in Computer Science*, Linköping, Sweden, pp. 18–44. Springer-Verlag. ISBN 3-540-61055-3. 90

Peyton Jones, S. L., T. Nordin, and D. Oliva (1997). `C--`: A portable assembly language. In *Proceedings of the 1997 Workshop on Implementing Functional Languages*. Springer Verlag LNCS. 12

Peyton Jones, S. L. and N. Ramsey (1998, August 8). Machine-independent support for garbage collection, debugging, exception handling, and concurrency (draft). Technical Report CS-98-19, Department of Computer Science, University of Virginia. 98

PLDI (1993, 23–25 June). *Proceedings of the ACM SIGPLAN'93 Conference on Programming Language Design and Implementation (PLDI)*, Albuquerque, New Mexico. *SIGPLAN Notices* 28(6), June 1993. 136, 139

PLDI (1997, 15–18 June). *Proceedings of the ACM SIGPLAN'97 Conference on Programming Language Design and Implementation (PLDI)*, Las Vegas, Nevada. *SIGPLAN Notices* 32(5), May 1997. 138, 142, 146

PLDI (1998, 17–19 June). *Proceedings of the ACM SIGPLAN'98 Conference on Programming Language Design and Implementation (PLDI)*, Montreal, Canada. *SIGPLAN Notices* 33(5), May 1998. 136, 138

PLDI (2000, 18–21 June). *Proceedings of the ACM SIGPLAN'00 Conference on Programming Language Design and Implementation (PLDI)*, Vancouver, British Columbia. *SIGPLAN Notices* 35(5), May 2000. 138, 144, 146

Polakow, J. and K. Yi (2001, March). Proving syntactic properties of exceptions in an ordered logical framework. In *The Fifth International Symposium on Functional and Logic Programming (FLOPS'01)*, Volume 2024 of *Lecture Notes in Computer Science*, Tokyo, pp. 61–77. 96

Pominville, P., F. Qian, R. Vallee-Rai, L. Hendren, and C. Verbrugge (2001). A framework for optimizing Java using attributes. In *CC 2001*, Volume 2027 of *LNCS*, pp. 334–354. 31

Ramsey, N. and S. Peyton Jones (2000, 18–21 June). A single intermediate language that supports multiple implementations of exceptions. See PLDI (2000), pp. 285–298. *SIGPLAN Notices* 35(5), May 2000. 16, 23, 33, 36, 38, 51, 65, 66, 98

Ramsey, N., S. Peyton Jones, C. Lindig, T. Nordin, D. Oliva, and P. Nogueira Iglesias (2001, November). *The C-- Reference Manual*. Available at `http://www.cminusminus.org`. 12

Reppy, J. H. (1999). *Concurrent Programming in ML*. Cambridge University Press. 126

Reynolds, J. C. (1998). *Theories of Programming Languages*. Cambridge, England: Cambridge University Press. 67

Schinz, M. and M. Odersky (2001). Tail call elimination on the Java Virtual Machine. In N. Benton and A. Kennedy (Eds.), *Electronic Notes in Theoretical Computer Science*, Volume 59. Elsevier Science Publishers. 5

Serrano, M. (1994, December). Bigloo user's manual. Technical report 0169, INRIA-Rocquencourt, France. 50

Sethi, R. and J. D. Ullman (1970, October). The generation of optimal code for arithmetic expressions. *Journal of the ACM 17*(4), 715–728. 94

Shao, Z., B. Saha, V. Trifonov, and N. Papaspyrou (2002, January). A type system for certified binaries. In *Conference Record of POPL'02: The 29th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, pp. 217–232. 31, 126

Stallman, R. M. (2001). *Using and Porting the GNU Compiler Collection (GCC)*. Free Software Foundation. Available from `http://gcc.gnu.org/onlinedocs/gcc-3.0/gcc.html`. 4, 19, 21, 33, 71, 83

Steel, T. B. (1961). A first version of UNCOL. In *Proceedings of the Eastern Joint Computer Conference*, pp. 371–377. Association of Computing Machinery. 7

Steele, G. L. (1977). Debunking the "expensive procedure call" myth, or procedure call implementations considered harmful, or LAMBDA, the ultimate GOTO. In *ACM Conference Proceedings*, pp. 153–162. Association for Computing Machinery. 79

Stichnoth, J. M., G.-Y. Lueh, and M. Cierniak (1999, 1–4 May). Support for garbage collection at every instruction in a Java compiler. In *Proceedings of the ACM SIGPLAN'99 Conference on Programming Language Design and Implementation (PLDI)*, Atlanta, Georgia, pp. 118–127. *SIGPLAN Notices* 34(5), May 1999. 36

Strong, J., J. Wegstein, A. Tritter, J. Olsztyn, O. Mock, and T. B. Steel (1958). The problem of programming communication with changing machines: A proposed solution: Report of the share ad-hoc committe on universal languages. *Communications of the ACM 1*(8), 12–18. 7

Stroustrup, B. (1997). *The C++ Programming Language (3rd Edition)*. Reading, Mass.: Addison-Wesley. 66

Tal, A., V. Bassin, S. Gal-On, and E. Demikhovsky (1999, November). Assembly language programming tools for the IA-64 architecture. *Intel Technology Journal* (Q4), 10. 84

Tarditi, D. (2000, October). Compact garbage collection tables. In T. Hosking (Ed.), *Proceedings of the Second International Symposium on Memory Management*, Minneapolis, MN. ACM Press. ISMM is the successor to the IWMM series of workshops. 27

Tarditi, D., P. Lee, and A. Acharya (1992, June). No assembly required: compiling standard ML to C. *ACM Letters on Programming Languages and Systems 1*(2), 161–177. 5, 122

Tarditi, D., G. Morrisett, P. Cheng, C. Stone, R. Harper, and P. Lee (1996, 21–24 May). TIL: A type-directed optimizing compiler for ML. In *Proceedings of the ACM SIGPLAN '96 Conference on Programming Language Design and Implementation*, Philadelphia, Pennsylvania, pp. 181–192. *SIGPLAN Notices* 31(5), May 1996. 49

Tarditi, D. R. and A. W. Appel (2000). *ML-Yacc User's Manual Version 2.4*. Available from `http://cm.bell-labs.com/cm/cs/what/smlnj/doc/ML-Yacc/`. 61

Tolmach, A. (1998). Optimizing ML using a hierarchy of monadic types. *Lecture Notes in Computer Science 1473*, 97–?? 32

Ungar, D. M. (1984, April). Generation scavenging: A non-disruptive high performance storage reclamation algorithm. *ACM SIGPLAN Notices 19*(5), 157–167. Also published as ACM Software Engineering Notes 9, 3 (May 1984) — Proceedings of the ACM/SIGSOFT/SIGPLAN Software Engineering Symposium on Practical Software Development Environments, 157–167, April 1984. 44

Vaughan, G. V., B. Elliston, T. Tromey, and I. L. Taylor (2000). *GNU Autoconf, Automake and Libtool*. Carmel, IN, USA: New Riders Publishing. 105

Wakeling, D. (1999, November). Compiling lazy functional programs for the Java virtual machine. *Journal of Functional Programming 9*(6), 579–603. 5, 7, 37

Wentworth, E. P. (1990). Pitfalls of conservative garbage collection. *Software Practice and Experience 20*(7), 719–727. 7

Wilson, P. R. (1994, January). Uniprocessor garbage collection techniques. Technical report, University of Texas. Expanded version of the IWMM92 paper. 44, 45, 60

Wilson, P. R. and M. S. Johnstone (1993, October). Truly real-time non-copying garbage collection. In E. Moss, P. R. Wilson, and B. Zorn (Eds.), *OOPSLA/ECOOP '93 Workshop on Garbage Collection in Object-Oriented Systems*. 60

Yi, K. and S. Ryu (2001). A cost-effective estimation of uncaught exceptions in Standard ML programs. *Theoretical Computer Science 277*(1–2), 185–217. 23

Yi, Q., V. Adve, and K. Kennedy (2000, 18–21 June). Transforming loops to recursion for multi-level memory hierarchies. See PLDI (2000), pp. 169–181. *SIGPLAN Notices* 35(5), May 2000. 64

Young, C., D. S. Johnson, D. R. Karger, and M. D. Smith (1997, 15–18 June). Near-optimal intraprocedural branch alignment. See PLDI (1997), pp. 183–193. *SIGPLAN Notices* 32(5), May 1997. 25

Young, C. and M. D. Smith (1998, November 30–December 2,). Better global scheduling using path profiles. In *Proceedings of the 31st Annual International Symposium on Microarchitecture*, Dallas, Texas, pp. 115–123. IEEE Computer Society TC-MICRO and ACM SIGMICRO. 25

Young, C. and M. D. Smith (1999, September). Static correlated branch prediction. *ACM Transactions on Programming Languages and Systems 21*(5), 1028–1075. 32