Machine-Independent Support for Garbage Collection, Debugging, Exception Handling, and Concurrency (Draft)

Simon Peyton Jones
University of Glasgow

Norman Ramsey
University of Virginia

August 7, 1998

Abstract

For a compiler writer, generating good machine code for a variety of platforms is hard work. One might try to reuse a retargetable code generator from another compiler, but code generators are complex and difficult to use, and they limit one's choice of implementation language. One might try to use C as a portable assembly language, but C limits the compiler writer's flexibility and the performance of the resulting code. The wide use of C, despite these drawbacks, argues for a portable assembly language.

C-- is a new language designed expressly as a portable assembly language. C-- eliminates some of the performance problems associated with C, but in its originally-proposed form it does not provide adequate support for garbage collection, exception handling, and debugging. The problem is that neither the high-level compiler nor the C-- compiler has all of the information needed to support these run-time features. This paper proposes a three-part solution: new language constructs for C--, run-time support for C--, and restrictions on optimization of C-- programs.

The new C-- language constructs enable a high-level compiler to associate initialized data with *spans* of C-- source ranges and to specify "alternate continuations" for calls to procedures that might raise exceptions. The run-time support is an interface (specified in C) that the garbage collector, exception mechanism, and debugger can use to get access to both high-level and low-level information, provided that the C-- program is suspended at a *safe point*. High- and low-level information is coordinated by means of the C-- spans and a common numbering for variables. Finally, the C-- optimizer operates under the constraints that the debugger or garbage collector can change the values of local variables while execution is suspended, and that a procedure call with alternate continuations can return to more than one location.

This three-part solution also provides adequate support for concurrency, so the paper illustrates the problem and the proposed solution with examples from garbage collection, exception handling, debugging, and threads. The paper also includes a model of the dataflow behavior of C-- calls.

A number of open problems remain. The most serious have to do with apparent redundancies among spans and safe points, and with the interaction of debugging support with optimization.

Contents

1 Introduction

Suppose you are writing a compiler for a high-level language. How are you to generate high-quality machine code? You could do it yourself, or you could try to take advantage of the work of others by using an off-the-shelf code generator. Curiously, despite the huge amount of research in this area, only three retargetable, optimizing code generators are freely available: VPO (Benitez and Davidson 1988), ML-RISC (George 1996), and the gcc back end (Stallman 1992). Each of these impressive systems has a rich, complex, and ill-documented interface. Of course, these interfaces are quite different from one another, so once you start to use one, you will be unable to switch easily to another. Furthermore, they are language-specific. To use ML-RISC you must write your front end in ML, to use the gcc back end you must write it in C, and so on.

All of this is rather unsatisfactory. It would be much better to have one portable assembly language that could be generated by a front end and implemented by any of the available code generators. So pressing is this need that it has become common to use C as a portable assembly language (Atkinson *et al.* 1989; Henderson, Conway, and Somogyi 1995; Pettersson 1995; Peyton Jones 1992; Tarditi, Acharya, and Lee 1992; Bartlett 1989b). Unfortunately, C was never intended for this purpose — it is a programming language, not an assembly language. C locks the implementation into a particular calling convention, makes it impossible to compute targets of jumps, and provides no support for garbage collection, exceptions, or debugging, except such debugging support as may be provided by a particular C compiler. An earlier paper discusses C's shortcomings in more detail (Peyton Jones, Oliva, and Nordin 1998).

The obvious way forward is to design a language specifically as a compiler target language. Such a language should serve as the interface between a compiler for a high-level language (the *front end*) and a retargetable code generator (the *back end*). What makes the problem interesting is that we want to retain the high performance one would expect from a code generator crafted specifically for the front end. The design of a portable assembly language should enable the front end to make choices that maximize performance, while enabling the back end to apply the best known code-improvement technology.

Separating the front and back ends complicates run-time support. In general, a front end will be designed in conjunction with a *run-time system*, which helps implement such high-level features as garbage collection, exception handling, debugging, and concurrency. To avoid repetition we refer to such features as *high-level run-time services.* The difficulty is that some of the information required by these run-time services, such as the locations of variables, is known only to the back end. The back end must therefore have its own run-time system, which supports the front-end run-time system.

The primary contribution of this paper is a specification that makes it possible to keep these two run-time systems separate, so that different front ends (and their run-time systems) can be used with different back ends (and their run-time systems), provided both front and back end conform to the specification. The specification has two parts. The front end communicates with the back end by emitting programs in a portable assembly language called C--. The front-end run-time system communicates with the back-end run-time system through a run-time interface (specified in C) called the C-- run-time interface.

This paper builds on the existing design of C-- (Peyton Jones, Oliva, and Nordin 1998). The new material, which enables C-- to support garbage collection, exception handling,

```
/* Ordinary recursion */
export sp1;
sp1( word32 n ) {
  word32 s, p;
  if n == 1 {
      return( 1, 1 );
  } else {
      s, p = sp1( n-1 );
      return( s+n, p*n );
  }
}

/* Tail recursion */
export sp2;
sp2( word32 n ) {
  jump sp2_help( n, 1, 1 );
}

sp2_help( word32 n, word32 s, word32 p ) {
  if n==1 {
      return( s, p );
  } else {
      jump sp2_help( n-1, s+n, p*n )
  }
}
```

```
/* Loops */
export sp3;
sp3( word32 n ) {
    word32 s, p;
    s = 1; p = 1;

  loop:
    if n==1 {
      return( s, p );
    } else {
      s = s+n;
      p = p*n;
      n = n-1;
      goto loop;
    }
}
```

Figure 1: Three functions that compute the sum $\sum_{i=1}^{n} i$ and product $\prod_{i=1}^{n} i$, written in C--.

and debugging, is in three parts: new language constructs for C--, run-time support for C--, and restrictions on optimization of C-- programs. The paper contains many examples of C-- code and C code. These examples have not been compiled, so they are surely riddled with errors and inconsistencies.

CAVEAT: This draft is being circulated for comment while the design is still incomplete. Some paragraphs may bear marginal notes discussing unresolved issues. Particularly sticky issues may be marked as "caveats," as is this paragraph. Caveats may not be intelligible unless you've read the whole paper.

2 The main features of C--

An earlier paper describes the basic design of C-- (Peyton Jones, Oliva, and Nordin 1998). We sketch the design here, with emphasis on those aspects that are relevant for understanding run-time support. Figure 1 gives examples of some C-- procedures that give a flavour of the language. C-- has the following features:

- A C-- program has a well-defined semantics that is independent of any machine. It is not, however, necessarily true that a front end will generate the *same* C-- program for each target architecture, or that a single C-- program will compile for every target architecture. For front ends, our goal is easy retargetability. In some cases, retargeting may be simplified by putting some knowledge of the architecture into the front end, e.g., the sizes of the data types that are most naturally used on that architecture.

- A C-- program is a sequence of procedure, `data`, `const`, `global`, `import`, and `export` declarations. Procedures, data, and constants are private to their compilation units unless named in an `export` directive. Procedures and addresses may be imported from other units using the `import` directive. `global` variables must be declared in every compilation unit.

- C-- supports a bare minimum of data types: a family of `word` types (`word8`, `word16`, `word32`, `word64`), and a family of floating-point types (`float32`, `float64`, `float80`). These types encode only the size (in bits) and the kind of register (general-purpose or floating-point) required for the datum. Not all types are available on all machines.

  The `word` types are used for characters, bit vectors, integers, and addresses (pointers). On each machine, one of the `word` types (typically `word32` or `word64`) is designated the "native word size" of the machine. One (probably the same one) is also designated the "native pointer type." Exported and imported names must have the native pointer type.

- As in any assembler, the operations on the values determine what kind of data is intended; for example, `*` multiplies two signed integers in two's-complement representation, while `*u` multiplies two unsigned integers.

- Memory access (loads and stores) are typed, and denoted with square brackets. Thus the statement:

  ```
  word32[foo] = word32[foo] + 1;
  ```

  loads a `word32` from the location whose address is in `foo`, adds one to it, and stores it at the same location. The mnemonic for this syntax is to think of `word32` as a C-like array representing all of memory, and `word32[foo]` as a particular element of that array. The semantics of the address is not C-like, however; the expression in brackets is the *byte address* of the item.

- Data declarations include labels and initialized data. For example:

  ```
  data {
    foo: word32 0;
         word32 27;
    baz: float64[10] 0.0;
  }
  ```

  The keyword `data` specifies that the data is to be put in the data section. The data block contains two 32-bit words and an array of ten 64-bit floats. `foo` is the address of the first word, and `baz` is the address of the array. `foo` and `baz` are *immutable* addresses; they cannot be assigned to.

- C-- also supports multiple, named data sections. For example:

```
data "debug" {
  ...
}
```

This syntax declares the block of data to belong to the section named "debug". Code is by default placed in the section "text", and a data directive with no explicit section name defaults to the section "data". Procedures can be enclosed in code "mytext" { ... } to place them in a named section "mytext".

C-- assigns no semantics to the names of data sections, except that, when linking object files, the linker is required to concatenate sections with the same name. Particular implementations may, however, assign machine-dependent semantics. For example, a MIPS implementation might assume that data in sections named sdata is addressable from the global pointer and that data in sections named rodata is read-only.

- Like other assemblers, C-- gives programmers the ability to name compile-time constants, e.g., by

  ```
  const GC = 2;
  ```

- C-- variables may be declared global, in which case the C-- compiler attempts to put them in registers. For example, given the declaration

  ```
  global {
    word32 hp;
  }
  ```

  the implementation attempts to put variable hp in a register, but if no register is available, it puts hp in memory. C-- programs use and assign to hp without knowing whether it is in a register or in memory. Unlike a name declared by data, a name declared by global is not an address. In fact, there is no such thing as “the address of a global,”and memory stores to unknown addresses cannot affect the value of a global. This guarantee permits a global to be held in a register, and even if it has to be held in memory, the optimizer does not need to reload it after a store to an unknown memory address. All separately compiled modules must have *identical* global declarations, or horribly strange things will happen.

  global declarations may name specific registers, for example:

  ```
  global {
    word32 hp    "%eax";
    word32 hplim "%ebx";
  }
  ```

  Register names are implementation-dependent.

2.1 Procedures

C-- suports procedures that are both more and less general than C procedures (e.g., multiple results and full tail calls but no varargs). Specifically:

- A C-- procedure, like sp1 in Figure 1, has parameters, like n, and local variables, like s and p. Parameters and variables are expected to be mapped onto machine registers where possible, and only spilled to the stack when necessary. In this absolutely conventional way C-- abstracts away from the number of machine registers actually available.

- C-- supports fully general tail calls, identified as "jumps." Control does not return from jumps, and C-- implementations must deallocate the caller's stack frame before each jump. For example, the procedure sp2_help in Figure 1 uses a jump to implement tail recursion.

- C-- supports procedures with multiple results, just as it supports procedures with multiple arguments. Indeed, a return is somewhat like a jump to a procedure whose address happens to be held in the topmost activation record on the control stack, rather than being specified explicitly. All the procedures in Figure 1 return two results; procedure sp1 contains a call site for such a procedure.

- A C-- procedure call is always a complete statement, which passes expressions as parameters and assigns results to local variables. In particular, a call cannot occur in an expression, as most high-level languages allow. For example, it is illegal to write

  ```
  r = f( g(x) );             /* illegal */
  ```

  because result returned by g(x) cannot be an argument to f. Instead one must write:

  ```
  y = g(x);
  r = f(y);
  ```

  This restriction makes explicit the order of evaluation, the location of each call site, and the names and types of temporaries used to hold the results of calls.

- To handle high-level variables that can't be represented using C--'s primitive types, C-- can be asked to allocate named areas in the procedure's activation record.

  ```
  f (word32 x) {
    word32 y;

    stack { p : word32;
            q : word32[40];
    }
    /* Here, p and q are the addresses of the relevant chunks of
       data. Their type is the native pointer type of the machine. */
  }
  ```

  stack is rather like data; it has the same syntax between the braces, but it allocates on the stack. As with data, the names are bound to the addresses of the relevant locations, and they are immutable. C-- makes no provision for dynamically-sized stack allocation (yet).

- The name of a procedure is a value whose type is the native pointer type. The procedure in a call can be an arbitrary expression, not simply the statically-visible name of a procedure. For example, the following statements are both valid, assuming the procedure `sp1` is defined in this compilation unit:

```
word32[ptr] = sp1;            /* Store procedure address */
...
r,s = (word32[ptr+4])( 4 );   /* Call some stored procedure */
```

- A C-- procedure, like `sp3` in Figure 1, may contain gotos and labels, *but they serve only to allow a textual representation of the control-flow graph.* Unlike procedure names, labels are not values, and they have no representation at run time. Because this restriction makes it impossible for front ends to build jump tables, C-- includes a switch statement, for which the C-- back end generates efficient code.

2.2 Calling conventions

The calling convention for C-- procedures is entirely a matter for the C-- implementation—we call it the *standard C-- calling convention.* In particular, C-- need not use the C calling convention.

The calling convention includes not only a specification of how parameters are passed during calls, but also a specification of how values are returned. The standard calling convention places no restrictions on the number of arguments passed to a function or the number of results returned from a function. The only restrictions are that the number and types of actual parameters must match those in the procedure declaration, and similarly, that the number and types of values returned must match those expected at the call site. These restrictions enable efficient calling sequences with no dynamic checks. (A C-- implementation is not required to check that C-- programs meet these restrictions.)

We note the following additional points:

- If a C-- function does not "escape" — if all sites where it is called can be identified statically — then the C-- back end is free to create and use specialized instances, with specialized calling conventions, for each call site. Escape analysis is necessarily conservative, but a function may be deemed to escape only if it is used other than in a call, or if it is named in an `export` directive.

- Support for unrestricted tail calls requires an unusual calling convention, so that a procedure making a tail call can deallocate its activation record while still leaving room for parameters that do not fit in registers. Section 7.5.1 discusses the issue in detail.

- C-- programs may use multiple calling conventions within a single program. Calling conventions other than the standard one are identified by name, as follows:

Maybe `foreign jump` *would be OK too; but I'm not sure. SLPJ.*

```
foreign convention-name r = f( parameters );
foreign convention-name return( results );
foreign convention-name f( parameters ) { body };
```

foreign calls and returns may be to other C-- procedures or to truly foreign code, provided, of course, that a foreign return returns to a foreign call.

Depending on the linker support available, names exported from C-- compilation units may be visible to foreign code. For example, the following C-- procedure can be called from C:

```
foreign "C" inc( word32 x ) {
              foreign "C" return( x+1 );
}
```

The calling convention used at each call site must match the calling convention given in the procedure declaration. Foreign calling conventions may be more restrictive than C--'s standard convention; for example, the C calling convention only permits one return value. A C-- implementation is not required to implement any calling conventions other than the standard one. Section 7.5 discusses this and related issues to be considered by designers and implementors of C-- calling conventions.

3 The problem of run-time support

When a front end and back end are written together, as part of a single compiler, they can cooperate intimately to support high-level run-time services, such as garbage collection, exception handling, debugging, and concurrency[1]. The primary contribution of this paper is to propose a means by which the front and back end might cooperate at arm's length, and still get good performance. Our guiding principle is this:

> C-- should make it *possible* to implement high-level run-time services, but it should not actually *implement* any of them. Rather, it should provide just enough "hooks" to allow the front-end run-time system to implement them.

This paper identifies the hooks.

Keeping the front end and back end at arm's length requires complex interfaces at both compile time and run time. It might appear more palatable to incorporate garbage collection, exception handling, and debugging into the C-- language, as (say) the Java Virtual Machine does. But doing so would guarantee that C-- would never be used. Different source languages require different support, different object layouts, and different exception semantics—especially when performance matters. No one back end could satisfy all customers.

Why are the interfaces complex? *High-level run-time services need to inspect and modify the state of a suspended program.* A garbage collector must find, and perhaps modify, all live pointers. An exception handler must navigate, and perhaps unwind, the call stack. A debugger must allow the user to inspect, and perhaps modify, the values of variables. All of these tasks require information from both front and back ends. The rest of this section elaborates.

[1] We view debuggers as part of the front-end run-time system, even though on many machines debuggers run in separate address spaces, using operating-system services to manipulate the program being debugged.

Finding roots for garbage collection. If the high-level language requires accurate garbage collection, then the garbage collector must be able to find all the *roots* that point into the heap. If, furthermore, the collector supports compaction, 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.

The difficulty is that neither the front end nor the back end has all the knowledge needed to find roots at run time. Only the front end knows which source-language variables, and therefore which C-- variables, represent pointers into the heap. Only the back end, which maps variables to registers and stack slots, knows where those variables are located at run time. Even the back end can't always identify exact locations; variables mapped to callee-saves registers may be saved arbitrarily far away in the call stack, at locations not identifiable until run time.

Printing values in a debugger. A debugger needs compiler support to print the values of variables. For this task, information is divided in much the same way as for garbage collection. Only the front end knows how source-language variables are mapped onto (collections of) C-- variables. Only the front end knows how to print the value of a variable, e.g., as determined by the variable's high-level-language type. Only the back end knows where to find the values of the C-- variables.

Loci of control A debugger must be able to identify the "locus of control" in each activation, and to associate that locus with a source-code location. This association is used both to plant breakpoints and to report the source-code location when a program faults.

An exception mechanism also needs to identify the locus of control, because in some high-level languages, that locus determines which handler should receive the exception. When it identifies a handler, the exception mechanism unwinds the stack and *changes* the locus of control to refer to the handler.

At run time, loci of control are represented by values of the program counter (e.g., return addresses), but at the source level, loci of control are associated with statements in a high-level language or in C--. Only the front end knows how to associate high-level source locations or exception-handler scopes with C-- statements. Only the back end knows how to associate C-- statements with the program counter.

Liveness. Depending on the semantics of the original source language, the locus of control may determine which variables of the high-level language are visible. Depending on the optimizations performed by the back end, the locus of control may determine which C-- variables are live, and therefore have values. Debuggers should not print dead variables. Garbage collectors should not trace them; tracing dead pointers could cause space leaks. Worse, tracing a register that once held a root but now holds a non-pointer value could violate the collector's invariants. Again, only the front end knows which variables are interesting for debugging or garbage collection, but only the back end knows which are live at a given locus of control.

Exception values. In addition to unwinding the stack and changing the locus of control, the exception mechanism may have to communicate a value to an exception handler. Only the front end knows which variable should receive this value, but only the back end knows where variables are located.

Succinctly stated, each of these operations must combine two kinds of information:

- *Information that only the front end has:*
  - Which C-- parameters and local variables are heap pointers.
  - How to map source-language variables to C-- variables and how to associate source-code locations with C-- statements.
  - Which exception handlers are in scope at which C-- statements, and which variables are visible at which C-- statements.
- *Information that only the back end has:*
  - Whether each C-- local variable and parameter is live, where it is located (if live), and how this information changes as the program counter changes.
  - Which program-counter values correspond to which C-- statements.
  - How to find activations of all active procedures and how to unwind stacks.

In the following section, we propose compile-time mechanisms that the front end can use to record its information, and we propose a run-time interface that gives the front-end run-time system access to both front-end and back-end information.

4 Support for high-level run-time services

We assume that executable programs are divided into three parts, each of which may be found in object files, libraries, or a combination.

- The front end compiler translates the high-level source program into one or more C-- modules, which is separately translated to *object code* by the C-- compiler.
- The front end comes with a (probably large) *front-end run-time system.* This run-time system includes the garbage collector, exception handler, and whatever else the source language needs. It is written in a programming language designed for humans, not in C--; in what follows we assume that the front end run-time system is written in C.
- Every C-- implementation comes with a (hopefully small) *C-- run-time system.* The main goal of this run-time system is to maintain and provide access to information that only the back end can know. It makes this information available to the front end run-time system through a C-language run-time interface, which we describe below. Different front ends may interoperate with the same C-- run-time system.

To make an executable program, we link generated object code with both run-time systems.

Within this setting, our proposal has three main elements:

- Garbage collection, exception handling, and debugging necessarily affect the semantics of procedure calls. In particular, the values of some local variables can be changed unexpectedly, and some procedures can return to more than one location. Section 4.1 describes how these effects are manifested in a C-- program.

- Section 4.2 describes C-- language mechanisms that enable the front end to record private information and to associate it with information available only to the back end. These mechanisms make it possible to solve the information-combining problem described above.

- Section 4.3 discusses how control can be transferred between generated code and a front-end run-time system, and in particular, how a front-end runtime can suspend and inspect running C-- code.

Section 4.4 shows how these elements are realized in a detailed run-time interface. This interface is to be used by the front-end run-time system and to be implemented by the back-end run-time system. It enables the front-end runtime to control interactions between parts of the program, to get access to back-end information, and to recover information deposited by the front end.

In what follows, we use the term "variable" to mean either a parameter of the procedure or a locally-declared variable.

4.1 Semantics of calls, or constraints on the optimizer

High-level run-time services often inspect and modify the state of a running program. This observation necessarily imposes some constraints on the C-- optimizer, because such state changes might violate invariants that it could otherwise maintain. In this section we identify the constraints, and describe C-- language mechanisms that minimise their impact.

4.1.1 Call-site invariants

In the presence of garbage collection and debugging, calls have an unusual property: *live local variables are potentially modified by any call.* For example, a compacting garbage collector might modify pointers saved across a call. Consider

```
f( word32 a ) {
  word32[a+8] = 10;  /* store 10 in 32-bit word pointed to by a+8 */
  g( a );
  word32[a+8] = 0;   /* store 0  in 32-bit word pointed to by a+8 */
  return;
}
```

If g invokes the garbage collector, the collector might modify a during the call to g, so the code generator must recompute a+8 after the call—it would be unsafe to save the common subexpression a+8 across the call. The same constraint supports a debugger that might change the values of local variables. Calls may also modify C-- values that are declared to be allocated on the stack.

A compiler writer might reasonably object to the performance penalty imposed by this constraint; the back end pays for compacting garbage collection whether the front end needs it or not. To eliminate this penalty, the front end can mark C-- parameters and variables as *invariant across calls*, using the keyword invariant, thus:

```
f( invariant word32 a ) {
  invariant word16 b;
  word32 c;
  ...
  g( a, b, c );         /* "a" and "b" are not modified by the call,
                           but "c" might be */
  ...
}
```

The `invariant` keyword places an obligation on the front-end run-time system, not on the caller of `f`. The keyword constitutes a promise to the C-- compiler that the value of an `invariant` variable will not change "unexpectedly" across a call. The run-time system and debugger may not change the values of invariant variables.

In the absence of a debugger, a front end can safely mark non-pointer variables as invariant across calls, and front ends using mostly-copying collectors (Bartlett 1988; Bartlett 1989a) or non-compacting collectors (Boehm and Weiser 1988) can safely mark *all* variables as invariant across calls.

4.1.2 Multiple return continuations

Some high-level run-time services, such as an exception dispatcher or debugger, may need to change the locus of control. They can't simply change the program counter, because two different program points may hold their live variables in different locations, and they may have different ideas about the layout of the activation record and the contents of callee-saves registers. They may even have different ideas about which variables are alive and which are dead. Unconstrained, dynamic changes in locus of control also make life hard for the optimizer: if the program counter can change arbitrarily, there is no such thing as dead code, and a variable live anywhere is live everywhere.

We address these problems in two ways. To support synchronous exception handling, we tell the optimizer exactly how the program counter might be changed dynamically, as described below. To support asynchronous exception handling and debugging, we give the optimizer substantial freedom, requiring only that it record decisions it makes. The debugger can make a reasonable effort at run time, as described in Section 6.3.

When exceptions are synchronous, it is reasonable to assume that raising an exception requires a call.

Typically, handling an exception involves first unwinding the stack to the caller of the current procedure, or its caller, etc., and then directing control to an "exception handler." We support such exceptions by making the following extension to the semantics of a procedure call: *a call might return to more than one location, and the C-- programmer specifies explicitly all the locations to which a call could return.* In effect, the call has many possible continuations instead of just one. An example syntax[2] is:

```
r = f( x )
  also returns v to      { /* alternate continuation 1 */ }
  also returns w1, w2 to { /* alternate continuation 2 */ }
  also aborts;
```

[2] We are still arguing over syntax!

```
  /* code for the normal case */
```

This statement establishes four possible continuations for the call `f(x)`. The “main” continuation simply follows the call, as usual; if this continuation is taken the result is assigned to `r`. The two “alternate” continuations are in the blocks that follow the `returns` … `to` clauses. If alternate continuation 2 were taken, the net effect would be just as if the call above were replaced by:

```
  w1, w2 = f( x ) ;
  /* alternate continuation 2 */
```

That is, the results would be assigned to `w1` and `w2`, and the appropriate code would be executed. To avoid confusion, C-- requires that alternate continuations end with an explicit control transfer (`goto`, `jump`, or `return`).

Unlike `also returns`, `also aborts` does not identify an explicit continuation. Instead, it indicates that the exception dispatcher can terminate this procedure by going to a handler in some calling procedure. In effect, `also aborts` introduces a flow edge from the call site to the procedure exit node, so that the optimizer does not eliminate assignments before the call that might otherwise be considered dead (Hennessy 1981). Any call site in a procedure P that could raise an exception not handled in P should be annotated with `also aborts`.

How are these multiple continuations used? A `return` statement causes execution to transfer to the “main” continuation in the caller. However, an exception dispatcher may abort a running C-- procedure and cause it to return to an alternate continuation, by using the C-- run-time procedures described in Section 4.4. Section 6.2 presents an example exception dispatcher that uses this method.

So far as the optimizer is concerned, then, such a call site is simply a fork in the control graph. The flow edges from the call site to each continuation (or to the exit node) guarantee that the variable locations, liveness, etc., are consistent no matter which continuation gets control after the call. Alternate continuations are assumed to be rarely used, and back ends are encouraged to exploit this assumption, e.g., when allocating registers or deciding on the placement of spills.

Programmers and compiler writers may expect there to be virtually no execution-time penalty for associating alternate continuations with a call site. There may, however, be a significant cost associated with the C-- run-time procedure that finds the alternative continuation, though it is unlikely to be so large as to dominate the cost of exception dispatching. Section 5 discusses implementation techniques.

A future revision of C-- may specify a means by which alternate return addresses could be communicated to `f`, the locations into which `f` could place returned values, and a mechanism by which generated C-- code could arrange a return to an alternate continuation.

CAVEAT: The `also returns` *mechanism limits potential exception implementations. In particular, it presumes that the exception dispatcher restores callee-saves registers, and it requires that exception dispatch be implemented in the run-time system (because there is no C-- value that represents the* `also returns` *continuation). These limitations preclude “ML-style” exceptions, which are very efficient to raise but which have a modest overhead in the normal case. Possible support for such exceptions is presented in Section 7.4.2.*

4.2 Front-end information

Front ends can use a data directive to record arbitrarily complex static information in the form of initialized data. For example, to support garbage collection, a front end could build a data block that says which of the parameters and local variables of a procedure are heap pointers. Only the front-end run-time system depends on the format of such data blocks. We need C-- language support only to identify the interesting data blocks at run time. For example, when tracing a particular procedure activation, the garbage collector needs the block describing the local variables of that procedure. For debugging or exception handling, the front end may record information that applies only to parts of procedures, or to individual C-- statements. We address all these applications by providing a mechanism for *mapping suspended procedure activations to associated* data *blocks.*

As an example, suppose we have a function f(x, y), with no other variables, in which x holds an integer and y holds a pointer into the heap. The front end can encode the heap-pointer information by emitting a data block, or *descriptor*, associating 0 with x and 1 with y:

```
data {
  gc1: word32 2;       /* this procedure has two variables */
       word8  0;       /* x is a non-pointer */
       word8  1;       /* y is a pointer */
}
```

This encoding does not use the names of the variables; instead, each variable is assigned an integer index, based on the textual order in which it appears in the definition of f. Therefore x has index 0 and y has index 1. A more compact encoding would use the indexes more intelligently, consuming only one bit per variable.

To associate this descriptor with f, the front end places the definition of f in a C-- *span*:

```
span GC gc1 {
  f( word32 x, word32 y ) {
    ...code for f...
  }
}
```

A span may apply to a sequence of function definitions, or to a sequence of statements within a function definition. In this case, the span applies to all of f. If execution of f is suspended to perform some run-time service, the back end must be able to map the suspended activation of f, plus the token GC, to the value gc1. More concretely, the C-- run-time system provides the C procedure

```
void *GetDescriptor( activation *a, int token ).
```

A run-time call to GetDescriptor, passing an "activation handle" a and the token GC, returns the address gc1. Activation handles are discussed in Section 4.4.

There are no constraints on the form of the descriptor that gc1 labels; that form is private to the front end and its run-time system. All C-- does is transform span directives into mappings from program counters to values.

Spans cannot overlap, but they can nest; the innermost span bearing a given token takes precedence. One can achieve the effect of overlapping by binding the same data block to multiple spans.

There may be several independent mappings in use simultaneously, e.g., one for garbage collection, one for debugging, and so on. The "token" is an integer used to distinguish these mappings from one another. Again, C-- takes no interest in the tokens; it simply provides a map from a (token, pc) pair to an address.

For exception handling, a mapping may indicate which handler should receive control upon an exceptional return from a procedure call. Handlers may be implemented using alternate continuations. The C-- runtime provides procedures that enable the front-end exception mechanism to transfer control to to alternate continuations, which are identified by number, according to the order in which they are attached to the call.

The C-- back end has considerable freedom to optimize programs; in particular, front ends may not assume that each span maps to a contiguous sequence of machine instructions. All that a front end can count on is that the optimizer preserves the mapping of program counter to span value. In particular,

- The optimizer may split any span into pieces. Pieces map to the same value as the original span.
- The optimizer may freely reorder code (and spans) enclosed within a span.
- The optimizer may remove dead, redundant, or partially redundant code from a span (e.g., it may reuse a common subexpression computed in another span).[3]
- The optimizer may not move code across span boundaries.

CAVEAT: Giving the optimizer this much freedom makes things hard for the debugger. For example, it's not clear what the debugger should do to set a breakpoint. Most of the problems seem to go away if the optimizer is prevented from splitting spans, i.e., if every span maps to a contiguous set of object-code locations.

4.3 Suspension and introspection

All our intended high-level run-time services must be able to suspend a C-- computation, inspect its state, and modify it, before resuming execution.

What does "the state of a suspended C-- computation" look like? In addition to the contents of memory, it includes a *logical stack* of procedure activations. This stack may not correspond exactly to the programmer's intuition about calls and returns, because activations of procedures that end in tail calls may have disappeared. The logical stack is probably, though not necessarily, implemented as a *physical stack* of *activation records*, the layouts of which are determined by the back end.

In many implementations of high-level languages, the run-time system runs on the same physical stack as the program itself. In such implementations, walking the stack or unwinding the stack requires a thorough understanding of system calling conventions, especially if

[3] Provided, of course, that the common subexpression is invariant across any intervening safe points, as discussed below.

an interrupt can cause a transfer of control from generated code to the run-time system. We prefer to imagine that the generated code and the run-time system run on separate stacks, as separate threads:

- The *system thread* runs on the *system stack* supplied by the operating system. Front-end run-time system code runs in the system thread, and it can easily inspect and modify the state of the C-- thread.
- The *C-- thread* runs on a separate *C-- stack*. When execution of the C-- thread is suspended, the state of the C-- thread is saved in the *C-- thread-control block*, or TCB.

Though we call them "threads," "coroutine" may be a more accurate term. The system thread never runs concurrently with the C-- thread, and the two can be implemented by a single operating-system thread.

4.3.1 A coroutine implementation

The C-- run-time system provides a synchronous control mechanism, which a front-end run-time system can use to run as a coroutine with C-- code. The front-end runtime calls the C procedure `Resume(t)` to transfer control to the C-- thread described by thread-control block `t`. Execution continues in the C-- thread until it calls the C-- procedure `yield(r)`, which suspends execution of the C-- thread, saves its state in its control block, and co-routines back to the system thread's `Resume` call. The value `r` that is passed as a parameter to `yield` is returned as the result of `Resume`; it is the thread's *yield code*.

To start a program, execution begins in the front-end run-time system, which uses the C procedure `InitTCB` to initialize the C-- thread, to which it can then transfer control with `Resume`. These C procedures are described in greater detail in Section 4.4.

Though it is not the focus of this paper, we intend that C-- should support many very lightweight threads, in the style of Concurrent ML (Reppy 1991), Concurrent Haskell (Peyton Jones, Gordon, and Finne 1996), and many others. The C procedures `InitTCB` and `Resume`, together with the C-- procedure `yield`, are sufficient for front-end run-time systems to implement a complete non-preemptive threads package, as we discuss in Section 6.4.

4.3.2 Safe points

At a call to `yield`, as at any other call, the code generator guarantees that the live variables are tidily saved away in the activation record or in callee-saves registers. `yield` stores the callee-saves registers, as well as the rest of the thread state, in the thread-control block, so at this point the execution of the C-- thread can safely be suspended, and the front-end runtime has full access to the state of the C-- thread. We call such a point a *safe point*.

More precisely, a program-counter value within a procedure is a safe point if it is safe to suspend execution of the procedure at that point, and to inspect and modify its variables. Accordingly, we require the following precondition for execution in the front-end run-time system:

> A C-- thread can be suspended only at a safe point.

Because any procedure could call `yield`, the code generator must make a safe point at every call site. This safe point is associated with the state in which the call as been made and the procedure is suspended awaiting the return.

It is impractical to try to make every instruction a safe point; recording local-variable liveness and location information for every instruction might represent a truly onerous burden. The front end can therefore insert extra safe points with the C-- statement

```
safepoint;
```

The existence of a safe point constrains the optimizer in the following way:

- the value of any non-invariant variable may change, and hence any use of that variable dominated by the safe point must not be moved before it;
- the safe point uses the values of non-invariant variables, and hence any definitions of such variables that dominate the safe point must not be moved after it;
- the values in memory may change, and hence any memory loads dominated by the safe point must not be moved before it;
- the safe point uses memory, and hence any memory stores that dominate the safe point must not be moved after it.

The optimizer is otherwise free to move code across safe points and to make the static ordering of safepoints in the object code differ from the order in the source code.

CAVEAT: A stronger restriction would prevent the optimizer from moving assignments to C-- *variables as well as memory loads and stores. This restriction might be sufficient to ensure that the observable state of the machine at a safe point would be consistent with the* C-- *source code, where "observable" means "observable by the procedures defined in Section 4.4". (Front ends can achieve such an effect by making all variables non-invariant.) It also might, if enough safepoints are introduced to support debugging, effectively disable the optimizer, which we don't want. We would like to give the optimizer as much freedom as possible without creating a heroic task for the debugger; how to do this is a topic for research.*

A `safepoint` directive may have an optional name, which the back end binds to the program counter that corresponds to the safe point. This name is visible anywhere in the compilation unit containing the function that contains the safe point. A safe-point name is *not* a label and may not be the target of a `goto`. Safe points may have names in order to support debugging; a front end can use the names in `data` declarations to build maps from source locations to program counters.

CAVEAT: We give safe points names only *to support debugging; the debugger has to have a map from source-code locations to object-code locations. Safe-point names seem like an ugly wart; it might be better to supply a map from spans to sets of PCs, but in that case, it's not clear how to ensure that suspension occurs at a safe point.*

4.3.3 Pre-emption

So far we have suggested that a C-- program can only yield control voluntarily, through a `yield` call. What happens if an interrupt or fault occurs, transferring control to the front-end run-time system, and the currently executing C-- procedure is not at a safe point?

This may happen if a user deliberately causes an interrupt, e.g., to request that the stack be unwound or the debugger invoked. It may happen if a hardware exception (e.g., divide by zero) is to be converted to a software exception. It may happen in a concurrent program if timer interrupts are used to pre-empt threads.

How can we achieve the safe-point invariant in the presence of such asynchronous events? There are the following conventional alternatives:

- Allow only asynchronous events that do not inspect or modify the state of the running C-- thread, and always resume it when the event handler completes. An example of such an event would be a timer interrupt. This choice is very restrictive.
- Make sure that wherever execution is suspended, the run-time system safely advances the machine to the next safe point, and that such advancement takes only bounded time. The C-- run-time interface provides the procedure `ExecuteToNextSafePoint`, so the front end need only ensure that time is bounded, which it can do by inserting a safe point into every loop and before every call. (The automatic safe point at a call is not useful; it applies only when the procedure is suspended at the call site, which is probably not what is desired.)
- Suspend execution between safe points, breaking the safe-point invariant. This alternative is most relevant to debugging; if a program is faulty, it may be undesirable, or even impossible, to advance it to the next safe point. In that case, a simple debugger may give approximate or inaccurate results. A more ambitious debugger should find the activation-record descriptor for a nearby safe point, then reason forward or backward by examining the machine instructions.

Asynchronous pre-emption is difficult to implement not only in C--, but in *any* system. Chase (1994b) discusses some implementation techniques.

To implement pre-emption, the C-- runtime must sit between the operating-system interrupt mechanism and the front-end runtime. When it takes an interrupt, it bundles up the C-- thread state into its thread-control block, and it transfers control to the front-end handler. If the C-- thread was suspended at a safe point, all operations in Section 4.4 are valid. If not, only a restricted set of operations are valid. This restricted set must include `ExecuteToNextSafePoint`. It also must include some support for the debugger, in case a fault makes forward execution impossible. The details are a topic for research.

4.4 The C-- run-time interface

This section presents the detailed interface that a front-end run-time system can use to create C-- threads, to transfer control to them, and to inspect and modify their state. Rather than specify representations of thread-control blocks or activation records, we hide them behind simple abstractions. In particular, we specify a set of C procedures that must be provided by the C-- runtime, and that together allow the state of a C-- thread to be inspected and modified.

The state of a C-- thread consists of some saved registers and a logical stack of procedure activations. This logical stack is usually implemented as some sort of physical stack, but the correspondence between the two may not be very direct. In particular, callee-saves registers that logically belong with one activation are not necessarily stored with that activation, or even with the adjacent activation; they may be stored in the physical record of an activation that is arbitrarily far away. This problem is the reason that C's `setjmp` and `longjmp` functions don't necessarily restore callee-saves registers, which is why some C compilers make pessimistic assumptions when compiling procedures containing `setjmp`.

We hide this complexity behind a simple abstraction, the *activation*. The idea of an activation of procedure P is that *it approximates the state the machine will be in when control returns to P*. The approximation is not completely accurate because other procedures may change the global store or P's stack variables before control returns to P. At the machine level, the activation corresponds to the "abstract memory" of Ramsey (1992), Chapter 3, which gives the contents of memory, including P's activation record (stack frame), and of registers. The contents of volatile registers are undefined, and such registers are treated as unavailable.

The activation abstraction hides machine-dependent details and raises the level of abstraction to the C-- source-code level. In particular, the abstraction hides:

- The layout of a thread-control block,
- The layout of an activation record, and the encoding used to record that layout for the benefit of the front end runtime,
- The details of manipulating callee-saves registers, the number of which varies from one architecture to another, and
- The direction in which the stack grows.

All of these matters become private to the back end and the C-- runtime.

In the C-- run-time interface, an activation record is represented by an "activation handle," which is a value of type `activation`. Arbitrary registers and memory addresses are represented by variables, which are referred to by number. Values of the program counter are restricted to safe points.

The procedures in the C-- run-time interface are:

`void InitTCB( tcb *t, int size, program_counter pc )` initialises the thread-control block of a new C-- thread. The control block is pointed to by `tcb` and is `size` bytes long. The initial N bytes, where N is a machine-dependent constant, are used for the actual control block; the remaining bytes are used for the thread's stack. When execution of the thread is begun, it begins at program counter `pc`, which should be the address of a C-- procedure with no parameters. *CAVEAT: Probably,* `InitTCB` *should take two pointers: one to a fixed-size TCB, and one to a stack block. That way, the client is free to allocate stacks and TCBs in different places, which might improve locality.*

`int Resume( tcb *t )` resumes execution of the C-- thread whose control block is pointed to by `t`. When the thread yields control back to the system thread, the value it transfers is returned as the result of `Resume`; this value is the thread's *yield code*.

`void *GetDescriptor( activation *a, int token ).` As noted above, `GetDescriptor` returns the address of the data block associated with the smallest C-- span tagged with `token` and containing the safe point where the activation is suspended.

`void *FindVar( activation *a, int var_index )` interrogates an activation handle for the location of any parameter or local variable in the activation record to which the handle refers. As mentioned earlier, variables are indexed by numbering them in the order in which they are declared, starting with zero. `FindVar` returns the address of the location containing the value of the specified variable. The front end is thereby able to examine or modify the value. `FindVar` returns `NULL` if the variable is dead. It is a checked runtime error to pass a `var_index` that is out of range.

Names bound by `stack` declarations are considered variables for purposes of `FindVar`, even though they are immutable. For such names, `FindVar` returns the value that the name has in C-- source code, i.e., the address of the stack-allocated block of storage. Storing through this address is meaningful; it alters the contents of the activation record `a`. Stack locations are not subject to liveness analysis.

`void *FindDeadVar( activation *a, int var_index ).` If a variable is dead, but its value is still sitting in a register or stack location, it would be unfair to hide the value from a debugger. Accordingly, `FindDeadVar(a, v)` returns a pointer to a location containing the last known value of `v`, if any such location exists, and `NULL` otherwise. It is an unchecked run-time error to store into such a location.

`void FirstActivation( tcb *t, activation *a ).` When execution of a C-- program is suspended, its state is captured in its thread-control block. `FirstActivation` uses the thread-control block to initialise an activation handle that corresponds to the procedure that will execute when the thread's execution is resumed.

`int NextActivation( activation *a )` modifies the activation handle `a` to refer to the activation record of `a`'s caller, or more precisely, to the activation to which control will return when `a` returns. `NextActivation` returns nonzero if there is such an activation record, and zero if there is not. That is, `NextActivation(&a)` returns zero if and only if activation handle `a` refers to the bottom-most record on the C-- stack.

`void SetActivation( tcb *t, activation a )` modifies the thread-control block `t` so that the execution of the thread will resume with activation `a`. It is used to unwind the stack, e.g., when handling exceptions.

`void SetContinuation( tcb *t, int cont_index ).` If thread `t` is suspended at a call site, `SetContinuation` arranges that when the thread is resumed, it will do so at the continuation whose ordinal number is `cont_index`. The normal continuation counts as continuation 0, so `SetContinuation(t,` k`)` may be called only when thread `t` is suspended at a call site with more than k return continuations.

`void *FindReturnLocation( tcb *t, int result_index ).` If thread `t` is suspended at a call site, `FindReturnLocation` returns a pointer to the location in which the thread expects to receive the value(s) returned by the call. `result_index` is the *ordinal number* of the result, starting from 0. `FindReturnLocation(a,` n`)` may be called only when thread `t` is suspended at a call site which will resume with a continuation expecting more than n return values.

InitTCB(t, n, pc)	Initialises a thread-control block t of size n, with initial program counter pc.
Resume(t)	Resumes C-- thread t.
GetDescriptor(&a, token)	Returns pointer to data block associated with the smallest span tagged with token and containing the point where activation a is suspended.
FindVar(a, v)	Returns pointer to mutable location containing value of parameter or local variable v in activation a.
FindDeadVar(a, v)	Returns pointer to location containing last known value of v.
FirstActivation(t, &a)	Sets a to "currently executing" activation of thread t.
NextActivation(&a)	Mutates a to point to the activation to which a will return (normally a's caller).
SetActivation(t, a)	Mutates thread-control block so thread t will resume execution with activation a.
SetContinuation(t, k)	Mutates thread-control block so thread t will resume execution at its k'th continuation.
FindReturnLocation(t, n)	Returns a pointer to the location in which the n'th return value will be returned to thread t.
SetSafePoint(t, sp)	Sets the safe point at which the thread will resume.
ExecuteToNextSafePoint(t)	Runs thread t until it reaches a safe point.

Table 1: The C-- run-time interface

We use result_index, not the number of the variable that will receive the result, because we expect that doing so will significantly reduce the amount of information that must be saved by the back end. For example, under a C-like calling convention, FindReturnLocation(a, 0) might always return the same register.

Notice that, unlike FindVar, FindReturnLocation only applies to the top-most activation — hence we pass a pointer to the thread-control block, rather than an activation handle. For other activations it is nonsense to talk of the return locations, whereas for the topmost activation it makes perfect sense; for example, the first return value might be returned in register 1, which is held in the thread-control block.

void SetSafePoint(tcb *t, safe_point sp) is a more drastic operation, used mainly by debuggers. It forcibly changes the safe point at which the thread will resume to sp, which must itself be a safe point in the suspended procedure. *CAVEAT: This procedure is just flailing at the general problem of altering control flow from within the debugger. It should be ignored until someone has a chance to think out a more complete proposal.*

void ExecuteToNextSafePoint(tcb *t) runs thread t until its topmost activation gets to a safe point. If the topmost activation of thread t is already suspended at a safe point, ExecuteToNextSafePoint is a no-op. This procedure is needed to put the C-- stack into a safe state when its execution is interrupted asynchronously. ExecuteToNextSafePoint may cause a fault without reaching a safe point, and it may run arbitrarily long. If a front-end run-time system uses ExecuteToNextSafePoint, it is up to the front end to ensure that every loop is cut by a safe point.

Table 1 summarises the C-- run-time interface.

5 Implementing the C-- run-time interface

Can spans, multiple return continuations, and the C-- run-time interface can be implemented efficiently? By sketching a possible implementation, we argue that they can. Because the implementation is private to the back end and the back-end run-time system, there is wide latitude for experimentation. Any technique is acceptable provided it implements the semantics above at reasonable cost. We argue below that well-understood techniques do just that.

5.1 Implementing spans

The span mappings of Section 4.2 take their inspiration from table mappings for exception handling, and they can be implemented in similar ways (Chase 1994a). The most common way is to use tables sorted by program counter. If suitable linker support is available, tables for different tokens can go in different sections, and they will automatically be concatenated at link time. Otherwise, tables can be chained together (or consolidated) by an initialisation procedure called at program initialisation time.

5.2 Implementing stack walking

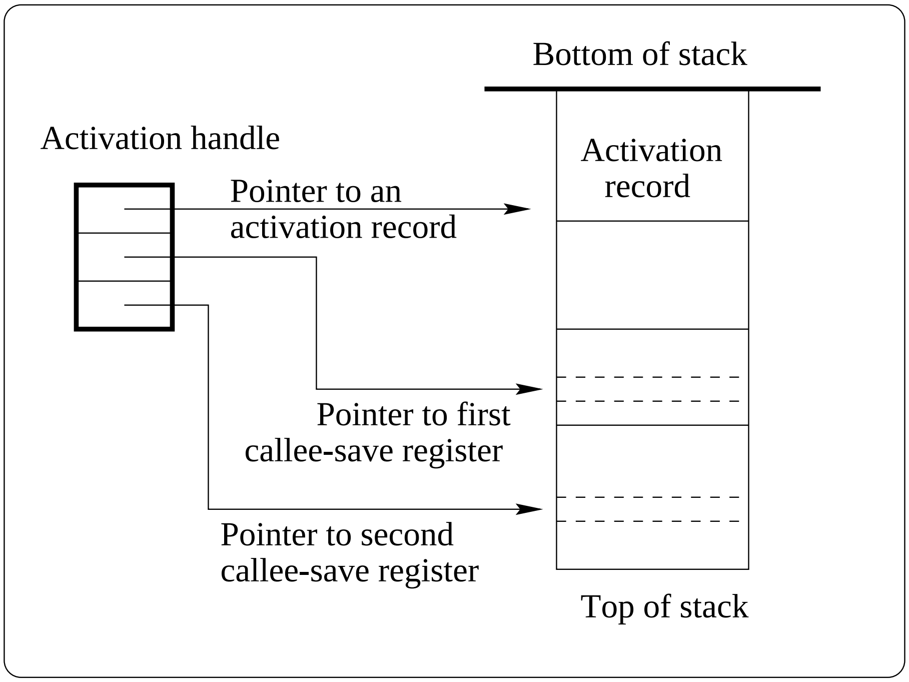
In our sketch implementation, the call stack is a contiguous stack of activation records. An activation handle is a static record consisting of a pointer to an activation record on the stack, together with pointers to the locations containing the values that the non-volatile registers[4] had at the moment when control left the activation record (Figure 2). `FirstActivation` initialises the activation handle to point to the topmost activation record on the stack and to the locations in the thread-control block that hold the values of registers. Depending on the mechanism used to suspend execution, the thread-control block might hold values of all registers or only of non-volatile registers, but this detail is hidden behind the run-time interface. Ramsey (1992) discusses retargetable stack walking in Chapters 3 and 8.

The run-time system executes only when execution of C-- procedures is suspended. We assume that the front-end run-time system uses one of the techniques mentioned in Section 4.3.3 to ensure that C-- execution is suspended only at safe points. For each safe point, the C-- code generator builds a statically-allocated *activation-record descriptor* that gives:

- The size of the activation record; `NextActivation` can use this to move to the next activation record.

- The liveness of each local variable, and the locations of live variables, indexed by variable number. The "location" of a live variable might be an offset within the activation record, or it might be the name of a callee-saves register. `GetVar` uses this "location" to find the address of the true memory location containing the variable's value, either

[4] The non-volatile registers are those registers whose values are unchanged after return from a procedure call. They include not only the classic callee-saves registers, but also registers like the frame pointer, which must be saved and restored but which aren't always thought of as callee-saves registers.


Bottom of stack
Activation handle
Activation record
Pointer to an activation record
Pointer to first callee-save register
Pointer to second callee-save register
Top of stack

The activation handle points to an activation record, which may contain values of some local variables. Other local variables may be stored in callee-saves registers, in which case their values are *not* saved in the current activation record, but in the activation records of one or more called procedures. These activation records can't be determined until run time, so the stack walker incrementally builds a map of the locations of callee-save registers, by noting the saved locations of each procedure.

Figure 2: Walking a stack

by computing an address within the activation record itself, or by returning the address of the location holding the appropriate callee-saves register, as recorded in the activation handle.

- If the safe point is a call site, the locations where the callee is expected to put results returned from the call.[5]

- The locations where the caller's callee-saves registers may be found. Again, these may be locations within the activation record, or they may be *this* activation's callee-saves registers. `NextActivation` uses this information to update the pointers-to-callee-saves-registers in the activation handle.

- If the safe point is a call site, pointers to the code for the alternate continuations.

[5] It would be more efficient to store this information with the callee, because then it could be stored once per function instead of once per call site. Unfortunately, this information is needed to implement `FindReturnLocation` just after the callee has exited, and therefore when the callee may not be known. Therefore, the information has to be stored at each call site. Luckily, this information is determined by the calling convention, so most of it can be shared.

The C-- runtime can map activations to descriptors using the same mechanism it uses to implement the span mappings of Section 4.2. The run-time interface can cache these descriptors in the activation handle, so the lookup need be done only when `NextActivation` is called, i.e., when walking the stack. An alternative that avoids the lookup is to store a pointer to the descriptor in the code space, immediately following the call, and for the call to return to the instruction after the pointer. The SPARC C calling convention uses a similar trick for functions returning structures (SPARC 1992, Appendix D).

The details of descriptors and mapping of activations to descriptors are important for performance. At issue is the space overhead of storing descriptors and maps, and the time overhead of finding descriptors that correspond to PCs. Liskov and Snyder (1979) suggests that sharing descriptors between different call sites has a significant impact on performance. Luckily, these details are private between the back end and the back-end run-time system, so we can experiment with different techniques without changing the approach, the run-time interface, or the front end.

6 Using the C-- run-time interface

This section explains how the C-- run-time interface might be used to help implement a garbage collector, an exception mechanism, a debugger, and a threads package. The first three all use the same technique to walk the stack: allocate an activation handle, initialize it with `FirstActivation`, and call `NextActivation` to move from one activation record to the next.

6.1 Garbage collection

Our primary concern is how the collector finds, and possibly updates, roots. Other tasks, like finding pointers in heap objects and compacting the heap, can be managed entirely by the front-end run-time system (allocator and collector) with no support from the back end. C-- takes no responsibility for heap pointers passed to code written in other languages. It's up to the front end to pin such pointers or to negotiate changing them with the foreign code.

To help the collector find roots in global variables, the front end can arrange to deposit the addresses of such variables in a special data section. To find roots in local variables, the collector must walk the activation stack. For each activation handle `a`, it calls `GetDescriptor(&a, GC)` to get the garbage-collection descriptor deposited by the front end. The descriptor tells it how many variables there are and which contain pointers. For each pointer variable, it gets the address of that variable by calling `FindVar`. If the result is non-`NULL`, the collector marks or moves the object the variable points to, and it may redirect the variable to point to the object's new location. Note that the collector need not know which variables were stored on the stack and which were kept in callee-saves registers; `FindVar` provides the location of the variable no matter where it is. Figure 3 shows a simple collector based on Appel (1989), targeted to the C-- run-time interface and the descriptors shown in Section 4.2.

A more complicated collector might have to do more work to decide which variables represent heap pointers. TIL is the most complicated example we know of (Tarditi *et al.* 1996). In

```
struct gc_descriptor {
  unsigned var_count;
  char heap_ptr[1];
};

void gc(void) {
  activation a;

  FirstActivation(tcb, &a);
  for (;;) {
    struct gc_descriptor *d = GetDescriptor(&a, GC);
    if (d) {
      int i;
      for (i = 0; i < d->var_count; i++)
        if (d->heap_ptr[i]) {
          typedef void *pointer;
          pointer *rootp = FindVar(a, i);
          *rootp = gc_forward(*rootp);  /* copying forward as per Appel */
        }
    }
    if (!NextActivation(&a))
      break;
  }
  gc_copy(); /* from-space to to-space, as per Appel */
}
```

Figure 3: Part of a simple copying garbage collector

TIL, whether a parameter is a pointer or not may depend on the value of another parameter. For example, a C-- procedure generated by TIL might look like this:

```
  f( word32 ty, word32 a, word32 b ) { ... }
```

The first parameter, `ty`, is a pointer to a heap-allocated type record. It is not statically known, however, whether `a` is a heap pointer. At run time, the first field of the type record that `ty` points to describes whether `a` is a pointer. Similarly, the second field of the type record describes whether `b` is a pointer.

To support garbage collection, we attach to `f`'s body a span that points to a statically allocated descriptor, which encodes precisely the information in the preceding paragraph. How this encoding is done is a private matter between the front end and the garbage collector; even this rather complicated situation is easily handled with no further support from C--.

Garbage collection is invoked, via a call to `yield`, when the allocator runs out of space. For example, here is how the code might look if allocation takes place in a single contiguous area bounded by `heap_limit`:

```
  f( word32 a,b,c ) {
    while (hp+12) > heap_limit {
                yield( GC );    /* Need to GC */
    }
    hp = hp+12;
```

```
PROCEDURE TryAMove() =
BEGIN
  TRY
    makeMove(getMove(player));
    next := (next + 1) MOD NUMBER(players);
  EXCEPT
  | IllegalMove(why) => player.illegalmove(why);
  | NoMoreTiles      => player.illegalmove("not enough tiles left");
  END;
  INC(movesTried);
END TryAMove;
```

Figure 4: Example Modula-3 procedure

```
    ...
}
```

6.2 Exceptions

Exceptions have a rich history, and many models of exceptions and exception handling have been proposed and implemented (Goodenough 1975; Liskov and Snyder 1979; Drew and Gough 1994). Exceptions may be associated with handlers *statically*, by source-language constructs, or *dynamically*, by calls to primitives like the POSIX `sigaction`. The *raising* of an exception transfers control to a handler, which may *terminate*, *resume*, or *retry* the code that raised the exception. Finally, there are *synchronous* exceptions, which occur only at well-defined points in the program, and *asynchronous* exceptions, which may occur at arbitrary points.

This paper addresses exceptions that are associated with handlers statically; by definition, dynamically associated handlers need no compile-time support. The C-- run-time interface supports termination and retry models, but not the resumption model. It also supports synchronous exceptions, and possibly asynchronous exceptions via `ExecuteToNextSafePoint` (Section 4.3.3).

C-- does not enforce a particular model of or semantics for exceptions; instead, C-- provides hooks that enable different front-end run-time systems to implement different high-level exception semantics. These hooks do not impose undue overhead on the normal case. (The designers of Modula-3 (Cardelli *et al.* 1992) urge implementors to spend 10,000 instructions in the exceptional case to save 1 instruction in the normal case.) We illustrate the idea of using a single set of hooks to implement various high-level exception semantics by sketching how one might implement exception dispatchers for Modula-3, Eiffel, and Exceptional C.

6.2.1 Modula-3 exceptions

Figure 4 shows a fragment from a game-playing program written Modula-3. Modula-3 uses `TRY-EXCEPT-END` to show handlers and their scopes. The statement sequences to the right of the arrows (`=>`) are handlers for the exceptions `IllegalMove` and `NoMoreTiles`. If either of these exceptions is raised anywhere between `TRY` and `EXCEPT`, control transfers to the appropriate handler. Otherwise, after the assignment to `next`, control skips directly from `EXCEPT` to `END`. After execution of a handler, control also transfers to `END`.

```
void TryAMove() {
  word32 s, t;
  span EXN ex1 {
    t = getMove(player) also returns s to {goto H1;} also returns to {goto H2;};
    makeMove(t)         also returns s to {goto H1;} also returns to {goto H2;};
    t = word32[players];          /* load size of array from its descriptor */
    next = (next + 1) mod t;
  }
  finish:
    movesTried = movesTried + 1;
    return;
  H1:
    t = word32[word32[player]+12]; /* load address of illegalmove method */
    t (s);
    goto finish;
  H2:
    t = word32[word32[player]+12]; /* load address of illegalmove method */
    t (lit1);
    goto finish;
}
data {
  lit1 : word8[22] "not enough tiles left\0";
  ex1 : align  4;
        word32 2;   /* two handlers in scope */
        word32 Exn_IllegalMove;
        word32 0;   /* assign argument to variable 0 (s) */
        word32 Exn_NoMoreTiles;
        word32 -1;  /* no argument  */
}
```

Figure 5: C-- implementation of Modula-3 `TryAMove`.

This code might be translated into the C-- procedure shown in Figure 5. The data section contains both a string literal and an "exception descriptor" that shows which exceptions are handled in the C-- span tagged with `ex1`.

To see how exception dispatch works, let us suppose that `getMove` terminates normally, but `makeMove` discovers that the move cannot be made because there would be no more tiles. `makeMove` would contain the Modula-3 statement

```
  RAISE IllegalMove("Your play goes off the board");
```

which might be translated into a `yield` to awaken the system thread, using the yield code to request exception handling service. The details of the particular exception would be pushed onto a global "exception stack."

```
  push_exn_info(Exn_MoMoreTiles, lit19);
  yield( EXCEPTION );
```

The system thread would invoke the exception dispatcher. The dispatcher would in turn call `FirstActivation(tcb, &a)` to get the activation handle for `makeMove`, and then call `GetDescriptor(&a, EXN)` to find handlers. `GetDescriptor` might return `NULL` if, for example, `makeMove` contained no handlers. The dispatcher would then call `NextActivation(&a)` to get the next frame. This time, `GetDescriptor(&a, EXN)` would return a pointer to `ex1`,

```
struct exn_descriptor {
  int handler_count;
  struct { void *exn_tag; int arg_number; } handlers[1];
}

void dispatcher() {
  activation a;
  void *exn_tag, *argument;

  pop_exn_info(&exn_tag, &argument);
  FirstActivation(tcb, &a);
  for (;;) {
    struct exn_descriptor *d;
    d = GetDescriptor(&a, EXN);
    if (d) {
      int i;
      for (i = 0; i < d->handler_count; i++)
        if (d->handlers[i].exn_tag == exn_tag) {
          SetActivation(tcb, &a);     /* unwind stack */
          SetContinuation(tcb, i+1);  /* choose handler */
          if (d->handlers[i].arg_number >= 0) { /* exn expects value */
            void *result;
            result = FindReturnLocation(tcb, d->handlers[i].arg_number);
            *result = argument; /* Assign result */
          }
          return;        }
    }
    if (!NextActivation(&a))
      abort();   /* unhandled exception: dump core */
  }
}
```

Figure 6: A simplified exception dispatcher for Modula-3

and the dispatcher would identify handler 0 as the handler for the Modula-3 exception `IllegalMove`. It would then use `SetActivation(tcb, &a)` to establish `a` as the activation to resume, and `SetContinuation(tcb, 1)` to cause resumption at the first alternate continuation. Then it would use `FindReturnLocation(tcb, 0)` to find the location to which to assign the value stored on the exception stack as the result to return.

Figure 6 shows a simple dispatcher implemented in C. A real dispatcher for Modula-3 would be more complicated, because it would have to provide for finalization (`TRY-FINALLY-END`), for handlers that receive multiple exceptions, and for better recovery from unhandled exceptions. The dispatcher included with DEC SRC Modula-3 even includes performance optimizations, such as efficient finalization of locks.

6.2.2 Eiffel exceptions

The Eiffel language (Meyer 1992) provides a somewhat different exception model, but one that can be implemented using the same C-- mechanisms. In Eiffel, exception handlers may not be attached to arbitrary sequences of statements, but only to whole procedures (called *routines*). Multiple handlers are not permitted; a handler, if present, receives all possible exceptions. Finally, handlers may not terminate normally, but must either *retry* the execution of their routines or terminate their routines and propagate the exception up the call stack. The high-level semantics are significantly different from Modula-3 semantics, but the implementation in `C--` is almost the same. The new element is the retry, which can be implemented in `C--` by a simple branch (`goto`) to the beginning of the routine. Propagating the exception requires re-invoking the run-time exception dispatcher, which is also possible in Modula-3.

6.2.3 Exceptional C exceptions

Exceptional C (Gehani 1992) offers a limited form of asynchronous exceptions with resumption. In Exceptional C, handlers for asynchronous exceptions do *not* have access to the local variables of the functions in which they appear; such handlers have access only to global variables. Such handlers can be compiled into `C--` procedures, just as they are compiled into C functions in Gehani's implementation. Because the handler does not touch the state of the suspended `C--` procedure, it can resume that procedure even when the procedure is suspended at an unsafe point. Unfortunately, the `C--` run-time interface provides no general way to find the handler or to implement Exceptional C's `retry` or `next`. (`Retry` resumes execution at the beginning of the block protected by the handler, and `next` resumes execution at the end of the block protected by the handler.) The problem is that, if an asynchronous exception occurs at an arbitrary point, it is unsafe to call `NextActivation` to search for the handler—think about interrupting a procedure that is in the middle of saving callee-saves registers. If exceptions are truly asynchronous, all these problems can be solved by guaranteeing that `ExecuteToNextSafePoint` terminates in bounded time, and `C--` can be used for the implementation.

6.3 Debugging

Debugging is not a solved problem. Questions remain in how compilers should support debuggers, how debuggers can be made retargetable, and how debuggers interact with optimizers. This paper does not address these questions. The paper does argue that the `C--` run-time interface provides adequate support for parts of debuggers that are well understood, and that it is compatible with some simple but useful optimizations. Moreover, we illustrate how `C--` can be used to express some tradeoffs between flexibility for the optimizer and power in the debugger.

CAVEAT: This is the weakest section in the paper. The main issues are (1) that named *safe points seem redundant with spans, and (2) it's not at all clear what restrictions on the optimizer are needed even to do something as simple as set a breakpoint. We certainly don't have answers to the more complicated questions, e.g., suppose I don't want to pay the performance penalty of having a safepoint at every source-level statement. In this case, how do I set a breakpoint at a statement, and what can I assume about the machine's state when the breakpoint is taken?*

The tasks of a source-level debugger are

- to display variables and their values,

- to evaluate expressions and assignments,
- to show what procedures are active and where they are suspended,
- to plant and remove breakpoints,
- to change control flow in the target program,
- to call procedures in the target program, and
- to control and interact with different target programs.

For simplicity, this paper ignores the problems of interacting with target programs. It assumes that the debugger runs in the same address space as the target, can call the C-- run-time interface directly, and can change values with simple loads and stores. Some debuggers may run in separate address spaces and use operating-system services to control their targets. Such debuggers might include a C-- run-time system modified to operate on a different address space.

6.3.1 Debugging information

Much of the information the debugger needs is known to the front end. Although many current compilers use specialized "stabs" or other machine-dependent formats to communicate these "debugging symbols" to the debugger, such specialized support is not necessary; ordinary initialized data is sufficient. It is helpful, but not necessary, to have some link-time support so this data can be put in a separate section and not automatically loaded into the target program's memory. The exact format of this data is a private matter between the front end and the debugger; we note in passing that this format need not be machine-dependent (Ramsey and Hanson 1992).

As an example of supporting a debugger using C--, this paper sketches a scheme derived from Ramsey (1992), Chapter 4. Debugging symbols emitted by a high-level front end might include

- For each compilation unit, a table of global symbols, a list of procedures, and a table mapping the names of source files to the procedures generated from those source files.
- For each procedure, a list of all the safe points. For each safe point, a record of its source-code and object-code locations. (The front end has access to the object-code location by means of the safe-point name.) This list is used to set breakpoints in object code when source locations are specified by the user.
- Spans pointing to tables that provide information about the names of the high-level language variables. One table per procedure is not adequate; in languages with nested scopes, the same name may denote an integer at one program point and a floating-point value at another.

  The tables record the location and type of each variable. Locations of global variables are placed directly in the debugging symbols; locations of local variables are given as indices and looked up at debug time using `FindVar`. The type contains the information needed to print the value of the variable and to evaluate expressions involving that variable.

- Spans mapping object-code locations to source-code locations, so the debugger can make an accurate report if a fault occurs when the program is not at a safe point.

These debugging symbols contain the information the debugger needs to display variables and their values and to evaluate expressions and make assignments. From the C-- run-time interface, the debugger would need `GetDescriptor` to find the information associated with the current safe point, and `FindVar` to read and change the values of local variables.

To show active procedures and where they are suspended, the debugger would use procedures `FirstActivation` and `NextActivation`, and also the source-code locations stored in the safe-point spans.

6.3.2 Breakpoints and flow of control

To plant and remove breakpoints, a debugger must be able to map source-code locations to object-code locations. Typical debuggers use a two-level strategy, mapping a source file to a list of procedures, and each procedure to a list of safe points. They then search the safe points for the one nearest the specified source location.

CAVEAT: The problem with setting breakpoints at safepoints is that for them to be of any use at all, one must restrict the optimizer. We haven't yet figured out a good set of restrictions.

A debugger can use `SetSafePoint` to change the flow of control in a suspended procedure. As ever, arbitrary changes in the flow of control are unsafe: for example, what does it mean to transfer control to the middle of a sequence of instructions used to restore callee-saves registers in a procedure epilog? As its name suggests, `SetSafePoint` supports only transfer of control between safe points.

Even if control is transferred only from one safe point to another, the back end may have made different assumptions about the states of C-- variables at different safe points. For example, what if local variable `x` is currently held in a register, but at the new PC, `x` is expected to be on the stack? Worse, what if local variable `y` is dead, but the debugger changes to a safe point at which `y` is live? Some compilers and debuggers handle this problem by making worst-case assumptions at code-generation time, e.g., by putting all variables on the stack at every safe point in the procedure. The C-- run-time interface makes such pessimistic assumptions unnecessary. Before the debugger calls `SetSafePoint`, it must consult `FindVar` to discover, and separately record, the values of all live variables. After calling `SetSafePoint`, must again consult `FindVar` to find the locations of all variables live at the new safe point, and store the values it previously extracted. In the example above, this sequence of calls would copy the value of `x` from its register location to its stack location. Furthermore, if the debugger discovers that the new safe point is associated with a live variable that is currently dead, it must supply a new value for the variable, perhaps by consulting the user.

6.3.3 Debugging and optimization

Even when supporting a debugger, C-- gives the back end some freedom to optimize code, and the front end some ability to trade off debuggability and optimization. Given the specification of `SetSafepoint`, the back end is free to put variables in registers, to split live ranges, and to use state-of-the-art register allocation (Briggs, Cooper, and Torczon 1994; George and Appel 1996). The back end is also free to reorder spans, to break up spans, and to use global-optimization techniques within spans. (The optimizer must not move code across span boundaries.) If the relevant variables are marked as `invariant`, the optimizer can save common subexpressions across calls and other safe points.

The front end has two ways of controlling optimization. Marking variables as invariant across calls gives the optimizer more opportunities for common-subexpression or partial-redundancy elimination, at the cost of preventing a garbage collector or debugger from changing these variables. Defining larger spans and reducing the number of safe points gives the optimizer larger regions in which to operate and more opportunities for optimization, at the cost of providing less precise information and control at debug time.

6.4 Concurrency

Section 4.3, above, describes C--'s support for very lightweight threads. Here we briefly outline how the interface described there can be used to support a simple, conventional threads package. As ever, our goal is to provide the minimal hooks to make a threads package possible; the bulk of the implementation should be in the front end runtime, not the C-- runtime.

When forking a thread, the front-end runtime allocates a thread-control block, which includes space for a stack. It initializes the block by passing to `InitTCB` a pointer that actually points a few words past the start of the allocated area. These initial few words are under the complete control of the front end runtime, to use for any per-thread data it wishes. We call this area the *thread-local data block*. The front-end runtime starts the thread by calling `Resume`. We expect it puts a pointer to the thread-local data block in a known place, and the generated C-- code probably moves that pointer to a register.

Since threads are not interrupted asynchronously, mutual exclusion is not an issue. When a thread wants to synchronise with another thread or block awaiting some event, it simply mutates the state of the appropriate semaphore (or whatever), and `yield`s to the front-end runtime with a suitable yield code. All of this can be done with a call to a support library written in C--, and provided along with the front-end runtime.

This model of concurrency may support a weak form of preemption by allowing "preempted" threads to continue to the next safe point. Front ends can ensure atomicity by keeping safe points out of critical sections. A similar strategy was used with Argus, made cheaper by forcing safe points to coincide with stack-limit checks (Liskov *et al.* 1987). This style of concurrency is very cheap. The operating system is not involved, no protection boundaries are crossed, and shared data structures can be manipulated without OS-level synchronization. The resulting system can have thousands of threads.

6.4.1 High-level threads

Most high-level threads packages will wish to pass arguments to threads, and to have threads return results upon termination. This can be done by putting a closure in thread-local data, and by passing something like the following function to `InitTCB`.

```
extern word32 "register 7" localdata;
extern word32 initial_localdata;

void run_a_closure () {
  word32 f, arg, answer, closure;

  localdata = initial_localdata;
  closure = word32[localdata+closure_slot];
  f   = word32[closure];
  arg = word32[closure+4];
  answer = f(arg);
  word32[localdata+answer_slot] = answer;
  yield(I_HAVE_FINISHED);
}
```

6.4.2 Blocking

Multiplexing lots of C-- threads onto one operating-system thread is all very well, but if a system call blocks, then all C-- threads block. A standard solution for this well-known problem is to use non-blocking I/O exclusively, but not all I/O libraries can be made to use non-blocking I/O.

If the operating system provides multiple threads, an alternative solution is possible. When a C-- thread wants to make a call that may block, it builds a data structure describing the call. Then it yields to the front-end runtime, with a code indicating that it wants a possibly-blocking call to be made. The front-end runtime keeps a pool of OS threads handy, and sends the request to it. Meanwhile, it blocks the C-- thread, and schedules another. When the OS thread completes the call, and the front end runtime is next awakened, it re-schedules the original C-- thread.

6.4.3 Stack overflow

What should happen if the stack of a C-- thread overflows? Since the stack is under the control of C--, it must be C-- that detects stack overflow. This is primarily an implementation issue, but it bears on the run-time interface because the C-- runtime must be able to ask the front-end runtime to allocate additional stack space. It should do so by arranging to `yield` to the front end with a suitable yield code. This way the front end can run the garbage collector if it needs to. (It may be necessary to allocate a small amount of space statically to ensure that there is room enough to save the thread's state in case of stack overflow.) This aspect of C--'s design is not fully thought out.

7 Discussion and related work

7.1 Alternatives to a portable assembler

What are the alternatives to a portable assembly language? One possibility is to use existing retargetable code generators. In the introduction, we mentioned that such code generators are complex, language-specific, and often poorly documented. They are also surprisingly few:

- The simplest and best documented such code generator is the `lcc` code generator (Fraser and Hanson 1991), but it implements no global optimizations.
- Gnu C (Stallman 1992) has been used the most successfully with different front ends, but folklore suggests that the cost is disproportionate. One known problem is that the Gnu C "tree" interface is not documented, but the "RTL" interface assumes that its input was generated from Gnu C "trees."
- SUIF (Hall *et al.* 1996) offers many optimizations, including parallizing optimizations, but it has only one native-code back end; otherwise, it compiles to C.
- Vpo (Benitez and Davidson 1988) has been used with a few different front ends, but it enforces C's calling conventions, does not optimize tail calls, and does not provide hooks for supporting garbage collection.
- ML-RISC (George 1996) is a promising new code generator, but it is still under development, and its interfaces still appear complex.

An alternative to using existing code generators is to use existing virtual machines.

- The Java Virtual Machine (Lindholm and Yellin 1997) is machine-independent and can be expected to be widely implemented, but even the most heavily optimized JVM compiler is unlikely to approach the performance possible with a custom code generator, or with C--. In particular, Java's security model may forbid optimization of tail calls and other important optimizations. Various people have suggested ways around these difficulties, but they involve exactly the same hacks that make C unattractive (e.g., massive switch statements), and because of Java's type security, additionally require many unnecessary run-time type checks. The resulting implementations run like a mangy, badly injured dog (Clausen and Danvy 1998; Wakeling 1998).
- Like Java, the Juice system (Franz 1997) supports machine-independent mobile code, but it is not a virtual machine. Instead, it is based on *slim binaries*, which are specially compressed abstract syntax trees from the Zurich Oberon compiler (Franz and Kistler 1997). These abstract syntax trees would be difficult to target for another front end, but it might be very effective to apply the compression technology to a lower-level language like C--.
- There are a couple of virtual machines whose focus is mostly that of a portable operating system envelope, such as Inferno/Limbo, and Elate. They are proprietry and do not inter-operate natively with (say) Windows or Unix. Using them implies much more than adopting an assembler. *CAVEAT: We are far from confident that these remarks are accurate.*

7.2 Alternatives for high-level run-time services

Given that we *are* designing a portable assembly language, we considered several alternative mechanisms to support high-level run-time services, such as garbage collection, exception handling, and debugging.

- One could build them into the C-- implementation, but such a solution would be nearly useless. The range of possible implementations varies widely with the semantics of the high-level language, and we know of no single implementation strategy that would be suitable for languages as different as, e.g., Haskell and C++. Building a debugger into C-- is even more clearly a bad plan; the right way to build a source-level debugger for a high-level language is *not* on top of a source-level debugger for C--. Because language implementers need freedom to use different implementations of garbage collection, exceptions, and debugging, C-- should provide mechanisms, not solutions.
- Another possible mechanism is to provide a procedural interface between the front end and the back end, and to arrange that the back end perform "upcalls" to the front end to exchange information about the layout of run-time structures. This is the approach taken by ML-RISC (George 1996). At present, however, every such interface is highly specific to its particular back end; ML-RISC provides ML-RISC's interface (and requires that the front end be written in ML), `gcc`'s tree language is very specific to `gcc`, and so on. It is far from clear how to make such an interface independent of the back end. Furthermore, given such an interface, the intermediate program can no longer be represented as an independent object, standing between the front and back end, amenable to independent analysis and processing.

The alternative we have chosen is to combine a C-- *language* with a run-time procedural interface. The language provides the means by which the front end records the information known only to it. The run-time interface provides the means by which the run-time systems gets both the information known only to the back end, and also the information recorded by the front end using the language. The front-end run-time system and debuggers are clients of this interface.

7.3 Alternatives for thread arguments and results

It would be a little easier to build thread packages if initial arguments could be passed to InitTCB, and if `yield` could return a result on thread termination. (There are also other cases in which it is useful for `yield` to pass a value as well as a code, e.g., when generated code uses `yield` to ask the front-end runtime to make a system call on its behalf.) The problem with passing such values in this way is that there are times when the values could be hidden inside the C-- run-time system, inaccessible to the garbage collector. We saw three possible solutions:

1. Extend the C-- run-time interface to make these hidden values accessible. The complexity of this approach seems disproportionate to the benefits.
2. Restrict the use of the interface so the hidden values may not point to the garbage-collected heap. This restriction destroys much of the appeal of the approach, since one will often want to pass and return pointers.
3. Eliminate such arguments and results entirely, and force the generated code to communicate with the front-end runtime using global variables or registers (e.g., a pointer to thread-local data). This is the solution we have chosen.

Under the scheme we have chosen, a threads package would probably use thread-local data to pass arguments to a thread and to get results from it, as shown in Section 6.4.1.

7.4 Alternatives for exception handling

We considered these alternatives for the implementation of exception handlers:

- They could be separate C-- functions. We dismissed this quickly, because a handler in a source-language function f needs access to f's local variables.
- They could be simple labels; we discuss this in Section 7.4.1.
- They could be a quasi-first-class continuation. We discuss this in Section 7.4.2.
- Perhaps they could be named safe points. We haven't considered this alternative carefully.

A more radical alternative would be to introduce exceptions directly into C--. For example, a `try/handle` construct might replace `also returns`. Our current design conflates procedure calls with exception handling; if a front end wishes to "throw an exception" from an activation A to a handler in the same activation, it must use a `goto`. A design with explicit exceptions might keep procedure calls separate from exception handling. Still, any design must provide some means of telling the optimizer where control can flow when an activation is aborted by some exception, so it seems unlikely that procedure calls and exceptions can be separated completely.

7.4.1 Labels are not values

Handlers cannot be referred to simply as labels, because labels are not values, and labels have no representation at run time.

The major reason that labels are not values is that first-class labels cause difficulties for optimization. If a label were a value, it could be stored somewhere, and such values could later be targets of `goto`. That means that the destination of a goto might not be known at compile time. That raises at least two difficulties:

- At the `goto` site, the code generator does not know where to put the local variables, since it does not know the destination. It does not even know how to adjust the stack pointer! (The stack pointer might in principle be at different positions relative to the frame base for different labels in the procedure.)
- The code generator can't do a proper liveness analysis if it doesn't know the destinations of all gotos, so it has to be very pessimistic and assume that everything live at any label is live at any goto. Worse, it has to keep all the live variables in the same locations at all these points.

C-- programmers who wish to use first-class labels within a procedure will be forced to break the procedure up into many small procedures and to use `jump` to transfer control between them. The price of this style is that live "local variables" must be passed as parameters, but this style need not lead to bad code—C-- compilers can be expected to generate very good code for calls to functions that don't escape the local compilation unit.

7.4.2 Continuations are values

Our design makes no provision for allowing generated code to unwind the stack or change control to an alternate continuation—these things must be done in the run-time system. This omission makes it impossible for C-- to implement an "ML-like" model of exception handling. In the ML-like model, a small overhead is added to every `TRY-EXCEPT-END`, but in exchange, exception dispatch is very efficient. The high-level languague maintains a stack of handlers, perhaps pointed to by a register. Every exception goes to the handler on top of the stack, and that handler contains code to identify the exception and pass it on to the next handler if necessary. The rest of this section presents a partly-baked idea that might support this model.

We add two new kinds of statement: one to declare continuations, and one to invoke them. To declare a continuation inside a procedure, write

`continuation` *name* `(` *formal-parameters* `)` `{` *statement-sequence* `}`

The *statement-sequence* must end in an explicit control transfer, and the *formal-parameters* must be variables of the procedure in which the continuation appears. Labels within *statement-sequence* may be targets of `goto`s anywhere in the procedure containing the continuation. Within the procedure containing the continuation, *name* is available as a C-- value. This value encapsulates three things: a stack pointer, a program counter, and a place to put parameters that won't fit in registers. The type of a continuation value is the native pointer type.

To invoke a continuation, write

`invoke` *expression* `(`*args*`)`,

where the *expression* evaluates to a continuation. `invoke` *k* `(`*args*`)` cuts the stack and changes the program counter to the values encapsulated by *k*, passing arguments as explained below.

A continuation *k* encapsulating an activation *A* may be invoked *only* while that activation is suspended at a call site. In keeping with the principle that a C-- programmer specifies explicitly all the locations to which a call could return, this call site must contain one or more `invokes` clauses, one of which must name *k*. The `invokes` clause on a call takes a similar form to "`also returns`":

```
  r = f(x,y) also invokes k;
```

and serves two purposes. First, it warns the back end that the call might return to the label named in the clause. Second, it tells the back end that control flow along that path does not restore callee-saves registers. This means the back end must insure that no variable live on entry to the continuation is in a callee-saves register at the call site.

Because a continuation may be invoked only when its activation is suspended at a call site that names the continuation explicitly, continuations-as-values do not cause the problems for optimization that labels-as-values would. If a front end wishes to invoke a continuation from *within* its own activation, it can use a `goto` with assignment (if it can identify the continuation statically) or it can call a trivial procedure do to the `invoke` on its behalf (if it cannot identify the continuation statically). C-- back ends should inline such procedures with no loss of efficiency.

Once an activation dies, its continuations die, too. Invoking a dead continuation is an *unchecked* run-time error, which it is up to the programmer to avoid. This restriction makes C-- continuations less powerful than Scheme continuations, but it means they can be implemented very efficiently, without stack copying. One possible implementation is for the C-- back end to allocate two words in the current activation record for each continuation in the procedure. It can store a suitable program counter and stack pointer in these words, then use a pointer to the words as the representation of the continuation. (With proper design of the stack-frame layout, the stack pointer also serves to point to a location for overflow parameters.) The store instructions can be placed anywhere they dominate all uses of the continuation.

Using these new constructs, the procedure TryAMove from Figure 4 could be compiled into the C-- code shown in Figure 7. (This example assumes that the native pointer size of the machine is word32.) The code to raise an exception:

```
RAISE exn (val);
```

would be compiled into this C--:

```
k = word32[exn_top];   /* fetch current handler from stack */
exn_top -= sizeof(k);  /* pop stack */
invoke k(exn, val);    /* invoke the handler */
```

It is somewhat unsatisfying to have two different mechanisms (also returns and invokes) for abnormal continuation from a procedure call, especially because the mechanisms are quite similar. Both take a list of variables that are bound at the time of the control transfer—although the syntax of the also returns variables suggests return values and the syntax of the continuation variables suggests parameters, they are semantically identical. Both are associated with statement sequences that end in explicit control transfers. Both have access to all of the procedure's local variables. The important differences have to do with invocation and register-save behavior:

So is this an ugly wart or a thing of beauty?

- The continuation can be invoked only from generated code, and therefore it must be representable as a C-- value. The also returns can be invoked only from the run-time system, and therefore it is identified by index number (as passed to SetContinuation), so it need not have a C-- value.

- Flow to also returns works like an ordinary return in that it restores the values of nonvolatile (callee-saves) registers. This behavior must be implemented by "unwinding the stack," which is expensive. It is suited only to heavyweight exception mechanisms. Flow to a continuation does *not* restore callee-saves registers; instead, it treats all registers as volatile. This behavior may be implemented by "cutting the stack" (i.e., changing the stack pointer), which can be made as efficient as an indirect call. It is suited to lightweight exception mechanisms that are implemented entirely in generated code, without the intervention of the run-time system.

CAVEAT: One particularly troubling restriction is that a continuation cannot be invoke*'d in its own activation. Such invocation is not necessary, because it would be more or less equivalent to a* goto *into the body of the continuation — our current proposal permits such* goto*s, although they make us nervous. Even if the continuation invoked is dynamic, one could always call a procedure to do the invocation. The call site provides a place to*

```
register word32 exn_top;      /* top of exn stack */

void TryAMove() {
  word32 s, t, code, val, k;
  exn_top += sizeof(H);      /* put H on the dynamic exception stack */
  word32[exn_top] = H
  t = getMove(player) also invokes H;
  makeMove(t)         also invokes H;
  t = word32[players];        /* load size of array from its descriptor */
  next = (next + 1) mod t;
  exn_top -= sizeof(H)       /* leave TRY-EXCEPT-END */
 finish:
  movesTried = movesTried + 1;
  return;
  continuation H (code, val) {
    if (code == IllegalMove) {
      t = word32[word32[player]+12]; /* load address of illegalmove method */
      t (val);
      goto finish;
    } else if (code == NoMoreTiles) {
      t = word32[word32[player]+12]; /* load address of illegalmove method */
      t (lit1);
      goto finish;
    } else {
      k = word32[exn_top];
      exn_top -= sizeof(k);
      invoke k(code, val);
    }
}
data {
  lit1 : word8[22] "not enough tiles left\0";
}
```

Figure 7: C-- implementation of Modula-3 `TryAMove` in ML style.

attach suitable `also invokes` *clauses, which we would otherwise need to add to the* `invoke` *construct.*

Partly, it boils down to a point of view: is a continuation a value, is it just a node in the control-flow graph of its parent procedure, or is it both? We are still arguing about this!

7.5 Value-passing conventions

C functions may pass multiple parameters to one another and may return single values. C-- offers many more opportunities for functions to exchange values, as indicated in the following table:

From	*To*	*Frame*	*Overflow*	*Convention*
call	function	keep	callee's frame	"call"
`jump`	function	free	callee's frame	"call"
`return`	call	free	caller's frame	"return"
`invoke`	`continuation`	cut	continuation frame	"invoke"
run-time	`also returns`	unwind	N/A	none needed

The *From* and *To* columns indicate where the control transfer originates and where it ends.

The *Frame* column explains what happens to the originating stack frame (activation record). For example, at a call, the caller keeps its frame, because control will return to it. At a jump, the caller deallocates the frame before transferring control, because control never returns from a jump. "Cut" indicates that `invoke` frees the stack frame as a side effect; the frame disappears without an explicit deallocation. "Unwind" indicates that the run-time system deallocates the frame by calling `SetActivation`.

The last two columns show what happens to values that are passed (parameters or results). The *Convention* specifies rules for placing values in locations (i.e., in registers or memory). Finite automata may be useful for this purpose (Bailey and Davidson 1995). Typical conventions pass some values in registers, but each machine has only finitely many registers. The *Overflow* column shows where values go when they won't fit in registers. For example, a frame containing `continuation` must reserve enough space to hold any overflow parameters used by `invoke`.

Because a single function may be reached by both jumps and calls, jumps and calls must share a value-passing convention. Other conventions, viz., `return` and `invoke`, are independent of the call convention and of each other. Finally, no convention is needed for `also returns`. Control is only transferred to `also returns` from the run-time system, and the run-time system knows to what activation control returns, so it can look up the locations using `FindReturnLocation`. It therefore need not rely on a convention.

"No convention" might be misinterpreted as meaning that every `also returns` can statically see all the places it can be invoked, and vice versa, so they can agree on an *ad hoc* convention. But this is not what we are claiming. The claim is that the back end can choose an *ad hoc* convention at the time it generates code, and that the front-end run-time system automatically accomodates itself to the back end's convention because it has access to the locations only by means of `FindReturnLocation`.

Callee-saves registers are not very useful in programs that use the `continuation/invoke` implementation of exceptions. Callee-saves registers are also perform badly when there are lots of tail calls, because a tail call must restore the callee-saves registers, only (perhaps) for the destination to save them again. Implementors of C-- are therefore encouraged to provide a choice of calling conventions, and in particular, to provide a calling convention that uses no (or few) callee-saves registers.

7.5.1 Implementing unrestricted tail calls

It may seem strange to say that overflow parameters must be passed in the callee's frame. Here is a more exact account of how implementations can support tail calls without restricting the number of parameters.

We assume here that every activation record has a fixed size determined at compile time, that the activation record is pointed to (in a sense made precise below) by a single register, which we call the stack pointer *sp*[6], and that the stack grows downward, toward lower addresses. We further assume that the compiler can compute the following quantities for each procedure:

i	The number of bytes needed to hold incoming parameters that will not fit in registers.
o	The number of bytes needed to hold outgoing parameters, in calls, that will not fit in registers.
o'	The number of bytes needed to hold outgoing parameters, in jumps, that will not fit in registers.
t	The number of bytes needed to hold temporaries, stack-allocated data, spill slots, etc.

Figure 8 shows the layout of an activation record under these assumptions. The figure shows the difficult case $o' > i$, i.e., there are more parameters to a tail call than there are incoming parameters.

On entry, each procedure performs a *limit check*:

`if` sp `- max(`o'`,` $i+t+o$`) <= limit then` *stack overflow.*

It then allocates its frame by sp = $sp - (i+t)$, and after that allocation it finds its incoming parameters either in registers or in the interval $[sp + t, fp + t + i)$. To implement a call for which k bytes of outgoing parameters do not fit in registers, it puts those parameters in the interval $[sp - k, sp)$. Finally, to implement a tail call, it first frees its frame by sp = $sp + (i+t)$, then again puts overflow outgoing parameters in $[sp - k, sp)$. It is perhaps a matter of point of view whether the incoming parameters are deemed to be in the caller's frame and the outgoing in the procedure's own frame, or whether the incoming parameters are deemed to be in the procedure's own frame, and the outgoing in the callee's frame. We have a minor preference for the latter view.

It might seem tidier to have *sp* point to the bottom of the outgoing-parameter area, rather than to the top. Such a scheme has minor disadvantages. If the frame size remains fixed, this scheme wastes stack space, reserving space for the most expensive outgoing call even when it is not needed. If the frame size varies, no stack space need be wasted, but the back end's bookkeeping job becomes more tedious, and it may have to record more information in order to support `FindVar`. If C-- procedures executed on the system stack, however, instead of on a separate C-- stack, such a scheme might be necessary, lest an interrupt destroy the values of outgoing parameters.

[6] Calling it the frame pointer would work equally well.

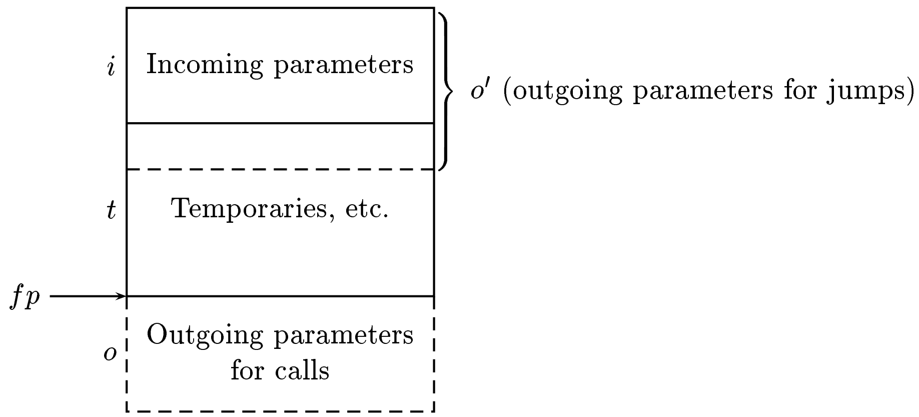

i
Incoming parameters
o' (outgoing parameters for jumps)
t
Temporaries, etc.
fp
o
Outgoing parameters for calls

Figure 8: A C-- activation record

7.6 Formalizing the behavior of calls

With `also returns`, `also invokes`, `also aborts`, and `invariant`, C-- procedure calls are considerably more complicated than calls in C. This section attempts to specify the dataflow behavior of C-- calls. The basic idea is that for each call, the code generator creates one or more "phantom" basic blocks representing the effect of that call. These blocks contain dataflow directives (e.g., **use** and **def**) that the optimizer is not permitted to touch. The optimizer must take these directives into account when optimizing the rest of the procedure.

We narrow our focus to a single call. For any C-- variable or hardware register, there are two potential uses and two potential definitions of interest:

D_r	A definition in the caller, before the call.
D_e	A definition in the callee.
U_r	A use in the caller, after the call.
U_e	A use in the callee.

There are four potential dataflow edges $D_? \rightarrow U_?$, so there are sixteen possible combinations. We dismiss combinations in which $D_r \rightarrow U_r \wedge D_e \rightarrow U_r$, since a use after the call can't be from both the caller and the callee at once. We also suspect that it is irrelevant whether a definition in the callee reaches a use in the callee; $D_e \rightarrow U_e$ is a private matter for the callee. This pruning leaves the 6 combinations shown in Table 2, in which the relevant edges have been given suggestive names.

We can specify the effect of a calling convention by assigning three bits to each register. The value of a bit corresponds to the presence or absence of a check mark in a column of Table 2. The calling convention determines the contents of a phantom basic block as follows:

```
For each hardware register or stack location $r$ do
    If $r$ holds a parameter
        insert **use**$(r)$, creating an edge $D_r \rightarrow U_e$
    If $r$ is not callee-saves (i.e., $r$ is volatile)
        insert **kill**$(r)$, "destroying" an edge $D_r \rightarrow U_r$
```

$D_r \to U_e$	$D_r \to U_r$	$D_e \to U_r$	Description
parameter	*callee-saves*	*result*	
			Scratch register
		$\surd$	Result
	$\surd$		Standard callee-saves
$\surd$			Volatile parameter
$\surd$		$\surd$	Parameter-result
$\surd$	$\surd$		Nonvolatile parameter

Table 2: Register definitions and uses at a call site

.

```
If $r$ holds a result
    insert **def**$(r)$, creating an edge $D_e \to U_r$
```

The word "destroying" is slightly misleading; it means that if an edge $D_r \to U_r$ exists, the code generator has to insert a spill and reload—it can't just delete the edge. This algorithm applies to a call that returns to a normal or an alternate continuation. Note that an alternate continuation in particular might return results to a stack location. If the call `invokes` a continuation, instead of returning, the algorithm is the same, except that along that path there is no such thing as a callee-saves register.[7]

Finally, we can deal with non-invariance as follows. If a non-invariant variable x is live across the call, then by definition there is an edge $D_r \to U_r$ corresponding to x. The call may contain **use**(x), but it may not contain **kill**(x) or **def**(x). For each such variable, we insert **mutate**(x) into the phantom block for the call. **mutate**(x) is equivalent to $x \leftarrow \mu(x)$, where μ is an unspecified function. It has the effect of replacing the edge $D_r \to U_r$ with a pair of edges $D_r \to U_e; D_e \to U_r$.

To translate a call, the back end performs these steps:

1. Create a "prolog block" moving the arguments into the correct locations in registers and on the stack, according to the parameter-passing convention. This block corresponds to code that is executed before the call, and it can be optimized (e.g., by allocating variables to parameter registers).

2. For the call itself, create a non-deterministic fork node in the flow graph, with an outgoing edge for the normal return, plus outgoing edges for each alternate continuation, for each invoked continuation, and for `also aborts` (if present). This node will be translated into the call instruction itself. This node might also include a **def**, e.g., if the call instruction puts the return address in a register.

3. For each outgoing edge from the fork, construct a phantom basic block representing the effect of the call. This phantom block will contain the **def**, **use**, **kill**, and **mutate** directives given above. Each phantom block will have its own "epilog block."

4. The epilog block for the normal return should contain a copy instruction for each result, which should move the result from its conventionally determined location into the

[7] Except for things like the stack pointer or kernel registers that the compiler must not touch.

```
f( word32 a ) {
  word32 b, c;

  b = a;
  c = a;
  b, c = f( c )
    also returns d to {
      return( b + d );
    };
  c = b + c + a;
  return( c );
}
```

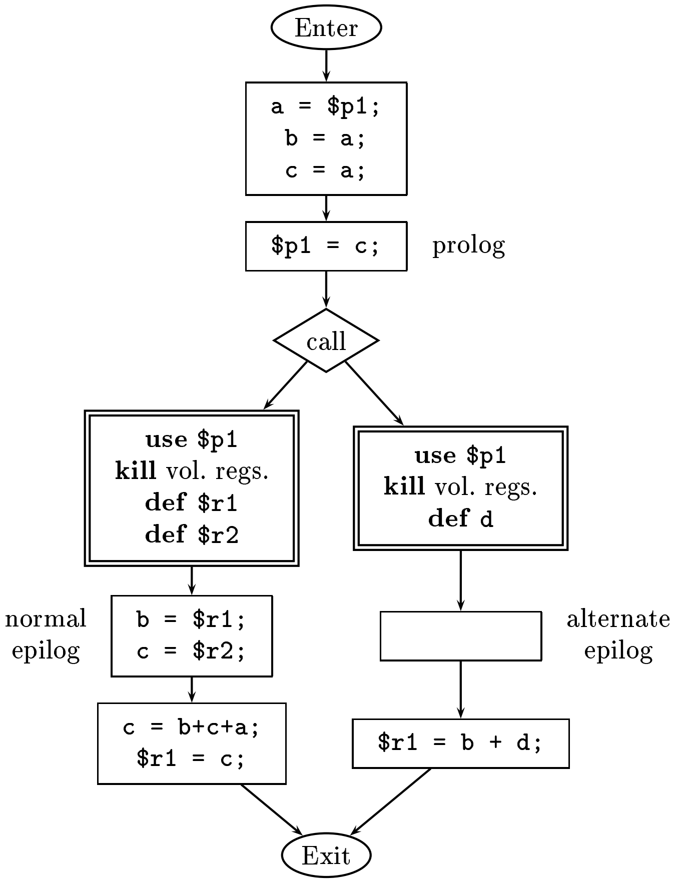


Figure 9: Example control-flow graph for translation of procedure with call

C-- variable named in the C-- call. (For most calling conventions, this block should also be annotated to show that it must follow the call instruction in the generated code.)

5. The epilog blocks for `also returns` continuations can be empty—the **def** operation in the phantom block represents the action of the run-time system in storing the return values in the proper locations.

6. The epilog blocks for `invokes` continuations should contain more copy instructions, again to move the returned values from their conventional locations into the variables named in the `continuation` statement. (Alternatively, one could move these copies to a prolog for the `continuation`.) These blocks then lead to the corresponding `continuation`s. If SSA is used, ϕ functions may have to be inserted.

Assuming parameters are passed in registers `$p1`, `$p2`, ..., and results are returned in registers `$r1`, `$r2`, ..., Figure 9 shows an initial translation from an example C-- function to a flow graph. Figure 10 shows the same flow graph, but with the proper mutations inserted into the phantom blocks. It uses static single-assignment numbering (Alpern, Wegman, and Zadeck 1988; Rosen, Wegman, and Zadeck 1988; Appel 1998) to make it easier to identify flow edges. It also shows the assumptions made about parameters, results, and callee-saves registers (c.s.r.) at entry and exit nodes.

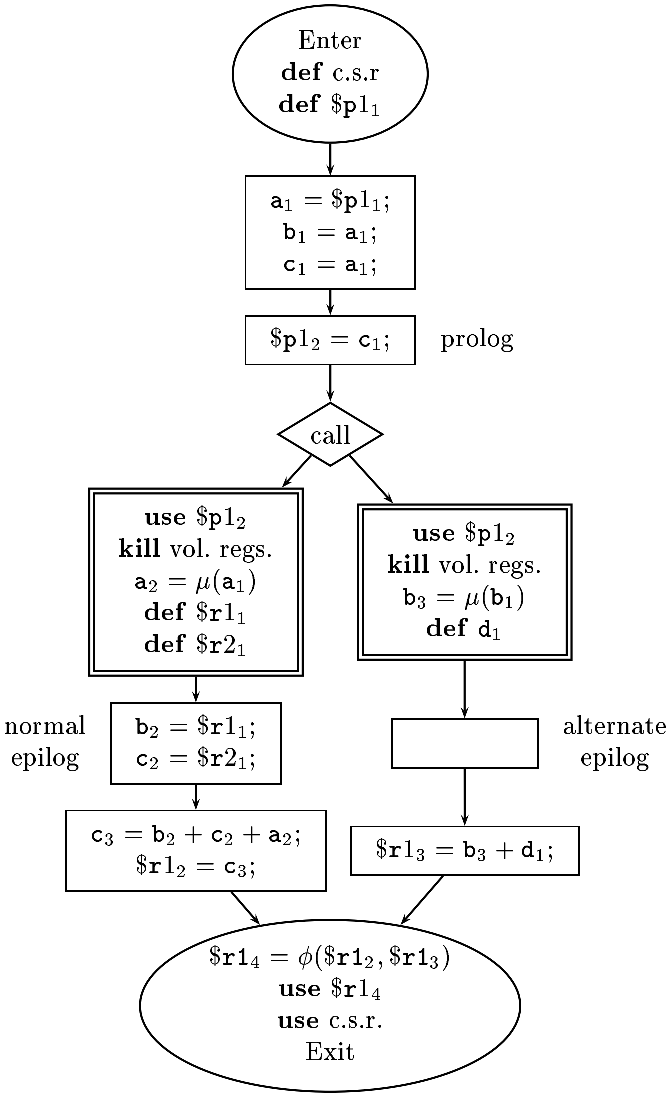


Figure 10: Flow graph with SSA numbering

7.7 Other related work

Chase (1994a) and (1994b) provide helpful, clear explanations of the techniques required to implement both synchronous and asynchronous exceptions. Liskov and Snyder (1979) discuss both the programming methodology in which exceptions should be used and the efficiency of their implementation. Hennessy (1981) discussses the interaction of exceptions and optimizations, especially global and interprocedural optimizations.

Other researchers have used the terms "stopping point" and "GC point" instead of "safe point" (Reppy 1990). Liskov *et al.* (1987) describes an implementation of pre-emption in which the only safe points are in procedure prologs; overhead is reduced by making pre-emption appear as stack overflow.

Linton (1990) describes a widely ported debugger. Hanson and Raghavachari (1996) describes a debugger in which the front end generates code to check for breakpoints dynamically and to maintain "shadow" versions of back-end data structures like the stack. This technique makes the debugger machine-independent at a cost of a factor of 3–4 in both space and time. Ramsey (1992) describes the design of a retargetable debugger, including a discussion of the supporting information recorded by the front end. The approach costs about a factor of 2 in space overhead, but no time overhead. Ramsey (1992) also describes a retargetable stack walker, which could be used to implement `NextActivation` in the C-- run-time interface.

It would be nice to say something about the CAML runtime, which has been used to support several languages, but we don't know of any published work.

Hauser (1996) discusses language support for floating-point exceptions.

Reppy (1990) discusses the use of safe points in the conversion of asynchronous signals to high-level exceptions.

Hieb, Dybvig, and Bruggeman (1991) presents a technique for using non-contiguous chunks of memory to implement a thread stack, and explains how to deal with stack overflow and underflow.

8 Open problems

There are number of aspects of our design that are incomplete.

- There is a significant piece missing from C--, one which is not related to run-time support, but which must be added if we are to meet our goal of competing with compiler-specific code generators. There is no way for the front end to tell the C-- code generator what it knows about aliasing. This knowledge is particularly important for compiling statically typed functional languages, like Standard ML and Haskell. Programs written in these languages do lots of allocation, and therefore do lots of storing into newly allocated records. However, most previously allocated records are immutable, and the front end can easily guarantee, without sophisticated analysis, that loads from these records have no dependencies with stores into newly allocated records. Compiling without this information may cause an unacceptable loss of performance.[8]

- We haven't specified the exact semantics of foreign calls, or how they interact with the scheduling mechanism. For example, if a front-end procedure wants to raise a synchronous exception, it is not clear whether it should make a foreign call into the exception dispatcher or whether it should execute a special kind of `yield`. This omission makes the example in Section 6.2.1 a bit fuzzy.

- They were introduced to support debugging by providing a map from soure-code locations to object-code locations, but named safe points seem to subsume many other mechanisms, especially if one is willing to assume that programs are suspended only at safe points. (This assumption need not hold for faulty code—exactly the case where one wants a debugger.) Given named safe points, is it desirable also to have spans? Is it necessary to have a separate mechanism for alternate continuations?

[8] Private communication from Lal George.

If we were only interested in garbage collection, exception handling, and concurrency, we could eliminate spans and have the front end attach its information to safe points.[9] Because we want to support debugging, however, we need to be able to get information about a faulty procedure no matter where its execution is suspended.

- The C-- run-time interface provides no way for a debugger to call a procedure in the target program. It might suffice to be able to push new activations on a C-- stack. (In single-threaded programs, calls have to share a single stack with the target program and with interrupt frames.) Pushing new activations might also be needed to implement a resumption model of exception handling.

- We need an interface for the C-- run-time system to get memory from the front-end run-time system in case of stack overflow.

- The two-stack model has been used in other systems, including Standard ML of New Jersey (Appel 1990). Although this model typically requires some assembly language to manage the transfer of control, it can attractive because it offers complete freedom to the implementor of the C-- stack.[10] It would still be desirable, however, to be able to run generated C-- code and the run-time system on the same stack, avoiding both assembly language and issues of stack overflow. Regrettably, such an implementation would require changes to the C-- run-time interface. C-- would no longer transfer control by a C-- `yield`, which returns from `Resume`, but would instead call directly into the front-end runtime. Such an interaction may be possible, but it requires further investigation.

- We suspect that compiler writers might want access to alternate continuations from generated C-- code, not just from the run-time system. It might be desirable to emit a direct return to an alternate continuation, for example. On some machines, including the SPARC, this could be done efficiently by representing alternate continuations as branch instructions in the instruction stream itself. The cost might be as low as one extra word per call site. On the other hand, `invoke` is almost as cheap. This is a matter for experimentation.

- It is not clear under what circumstances a "call site" should be considered the location of the call instruction and under what circumstances it should be considered the instruction at the return address. The former is where we want to set breakpoints, but the latter is the locus to which a suspended activation returns.

- The `continuation` and `invokes` constructs could in principle be used to implement coroutines without assistance from the run-time system, but such an attempt might be ill-advised. For example, how would one communicate to the C-- run-time system that one had switched to a different stack, and therefore that a different test should be used for stack overflow? The details remain to be investigated.

- It is not clear how to indicate the type of a literal constant. This information is especially needed when one of the arguments to a C-- procedure is a literal constant.

[9] There may, however, be significant performance advantages (reduction in size of initialized data) to having the back end implement the mappings, enabling it to exploit its knowledge of the ordering of program counters.

[10] To hell with register windows!

- We have not decided whether C-- should support `foreign jump`, that is, a tail call to a procedure with a non-standard calling sequence. NR believes that different front ends may want to use a single C-- implementation with different calling sequences. He conjectures that some people may even want to experiment with multiple calling sequences for calls between C-- procedures, even in a single front end. For example, in a hybrid OO language, he can imagine using one calling sequence for method calls and another for procedure calls, perhaps using different numbers of callee-saves registers. SLPJ is concerned about imposing an unknown implementation burden in exchange for unclear gains.
- We have not precisely defined the idea of statically evaluatable expressions ("constant expressions"). These should include, e.g., integer literals, names declared with `const`, differences of two addresses in the same data section (or stack frame), etc.

Acknowlegements

Thomas Nordin built the first implementation of C-- based on Lal George's ML-Risc code generator. Richard Black, Lal George, Thomas Johnsson, Greg Morrisset, Xavier Leroy, Nilhil, Olin Shivers, and David Watt, have all given valuable feedback.

References

Alpern, Bowen, Mark N. Wegman, and F. Kenneth Zadeck. 1988 (January).
Detecting equalities of variables in programs.
In *Conference Record of the 15th Annual ACM Symposium on Principles of Programming Languages*, pages 1–11, San Diego, California.

Appel, Andrew W. 1989 (February).
Simple generational garbage collection and fast allocation.
Software—Practice & Experience, 19(2):171–183.

———. 1990.
A runtime system.
Lisp and Symbolic Computation, 3:343–380.

———. 1998 (April).
SSA is functional programming.
SIGPLAN Notices, 33(4):17–20.

Atkinson, Russ, Alan Demers, Carl Hauser, Chnristian Jacobi, Peter Kessler, and Mark Weiser. 1989 (July).
Experiences creating a portable Cedar.
Proceedings of the '89 SIGPLAN Conference on Programming Language Design and Implementation, SIGPLAN Notices, 24(7):322–329.

Bailey, Mark W. and Jack W. Davidson. 1995 (January).
A formal model and specification language for procedure calling conventions.
In *Conference Record of the 22nd Annual ACM Symposium on Principles of Programming Languages*, pages 298–310, San Francisco, CA.

Bartlett, Joel F. 1988 (February).
Compacting garbage collection with ambiguous roots.
Technical Report 88/2, DEC WRL, 100 Hamilton Avenue, Palo Alto, California 94301.

———. 1989a (October).
Mostly-copying garbage collection picks up generations and C++.
Technical Report TN-12, DEC WRL, 100 Hamilton Avenue, Palo Alto, California 94301.

———. 1989b.
SCHEME to C: A portable Scheme-to-C compiler.
Technical Report RR 89/1, DEC WRL.

Benitez, Manuel E. and Jack W. Davidson. 1988 (July).
A portable global optimizer and linker.
Proceedings of the ACM SIGPLAN '88 Conference on Programming Language Design and Implementation, in *SIGPLAN Notices*, 23(7):329–338.

Boehm, Hans-Juergen and Mark Weiser. 1988 (September).
Garbage collection in an uncooperative environment.
Software—Practice & Experience, 18(9):807–820.

Briggs, Preston, Keith D. Cooper, and Linda Torczon. 1994 (May).
Improvements to graph coloring register allocation.
ACM Transactions on Programming Languages and Systems, 16(3):428–455.

Cardelli, Luca, James Donahue, Lucille Glassman, Mick Jordan, Bill Kalsow, and Greg Nelson. 1992 (August).
Modula-3 language definition.
SIGPLAN Notices, 27(8):15–42.

Chase, David. 1994a (June).
Implementation of exception handling, Part I.
The Journal of C Language Translation, 5(4):229–240.

———. 1994b (September).
Implementation of exception handling, Part II: Calling conventions, asynchrony, optimizers, and debuggers.
The Journal of C Language Translation, 6(1):20–32.

Clausen, L. R. and Olivier Danvy. 1998 (April).
Compiling proper tail recursion and first-class continuations: Scheme on the Java virtual machine.
Technical report, Department of Computer Science, University of Aarhus, BRICS.

Drew, Steven J. and K. John Gough. 1994 (May).
Exception handling: expecting the unexpected.
Computer Languages, 20(2):69–87.

Franz, Michael and Thomas Kistler. 1997 (December).
Slim binaries.
Communications of the ACM, 40(12):87–94.

Franz, Michael. 1997 (October).
Beyond Java: An infrastructure for high-performance mobile code on the World Wide Web.
In Lobodzinski, S. and I. Tomek, editors, *Proceedings of WebNet'97, World Conference of the WWW, Internet, and Intranet*, pages 33–38. Association for the Advancement of Computing in Education.

Fraser, Christopher W. and David R. Hanson. 1991 (September).
A code generation interface for ANSI C.
Software—Practice & Experience, 21(9):963–988.

Gehani, Narain H. 1992 (October).
Exceptional C or C with exceptions.
Software—Practice & Experience, 22(10):827–848.

George, Lal and Andrew W. Appel. 1996 (May).
Iterated register coalescing.
ACM Transactions on Programming Languages and Systems, 18(3):300–324.

George, Lal. 1996.
MLRISC: Customizable and reusable code generators.
Unpublished report available from `http://www.cs.bell-labs.com/~george/`.

Goodenough, John B. 1975 (December).
Exception handling: Issues and a proposed notation.
Communications of the ACM, 18(12):683–696.

Hall, Mary W., Jennifer M. Anderson, Saman P. Amarasinghe, Brian R. Murphy, Shih-Wei Liao, Edouard Bugnion, and Monica S. Lam. 1996 (December).
Maximizing multiprocessor performance with the SUIF compiler.
Computer, 29(12):84.

Hanson, D. R. and M. Raghavachari. 1996 (November).
A machine-independent debugger.
Software—Practice & Experience, 26(11):1277–1299.

Hauser, John R. 1996 (March).
Handling floating-point exceptions in numeric programs.
ACM Transactions on Programming Languages and Systems, 18(2):139–174.

Henderson, Fergus, Thomas Conway, and Zoltan 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*, pages 1–15, Portland, Or.

Hennessy, John. 1981 (January).
Program optimization and exception handling.
In *Conference Record of the 8th Annual ACM Symposium on Principles of Programming Languages*, pages 200–206, Williamsburg, Virginia.

Hieb, R., R. K. Dybvig, and A. C. Bruggeman. 1991 (June).
Representing control in the presence of first-class continuations.
In *Proceedings of the ACM SIGPLAN '91 Conference on Programming Language Design and Implementation.*

Lindholm, Tim and Frank Yellin. 1997 (January).
The Java Virtual Machine Specification. The Java Series.
Reading, MA, USA: Addison-Wesley.

Linton, Mark A. 1990 (June).
The evolution of Dbx.
In *Proceedings of the Summer USENIX Conference*, pages 211–220, Anaheim, CA.

Liskov, Barbara H. and Alan Snyder. 1979 (November).
Exception handling in CLU.
IEEE Transactions on Software Engineering, SE-5(6):546–558.

Liskov, Barbara, Dorothy Curtis, Paul Johnson, and Robert Scheifler. 1987 (11).
Implementation of Argus.
In *Proceedings of the 11th ACM Symposium on Operating Systems Principles*, pages 111–122.
Published in ACM Operating Systems Review Vol.21, No.5.

Meyer, Bertrand. 1992.
Eiffel: The Language.
London: Prentice Hall International.

Pettersson, M. 1995.
Simulating tail calls in C.
Technical report, Department of Computer Science, Linkoping University.

Peyton Jones, Simon L., A. J. Gordon, and S. O. Finne. 1996.
Concurrent Haskell.
In *Conference Record of the 23rd Annual ACM Symposium on Principles of Programming Languages*, pages 295–308.

Peyton Jones, Simon L., Dino Oliva, and T. Nordin. 1998.
`C--`: A portable assembly language.
In *Proceedings of the 1997 Workshop on Implementing Functional Languages.* Springer Verlag LNCS.

Peyton Jones, Simon L. 1992 (April).
Implementing lazy functional languages on stock hardware: The spineless tagless G-machine.
Journal of Functional Programming, 2(2):127–202.

Ramsey, Norman and David R. Hanson. 1992 (July).
A retargetable debugger.
ACM SIGPLAN '92 Conference on Programming Language Design and Implementation, in *SIGPLAN Notices*, 27(7):22–31.

Ramsey, Norman. 1992 (December).
A Retargetable Debugger.
PhD thesis, Princeton University, Department of Computer Science.
Also Technical Report CS-TR-403-92.

Reppy, John H. 1990 (August).
Asynchronous signals is standard ML.
Technical Report TR90-1144, Cornell University, Computer Science Department.

———. 1991 (June).
CML: a higher-order concurrent language.
In *Proceedings of the ACM SIGPLAN '91 Conference on Programming Language Design and Implementation.* ACM.

Rosen, Barry K., Mark N. Wegman, and F. Kenneth Zadeck. 1988 (January).
Global value numbers and redundant computations.
In *Conference Record of the 15th Annual ACM Symposium on Principles of Programming Languages*, pages 12–27, San Diego, California.

SPARC International. 1992.
The SPARC Architecture Manual, Version 8.
Englewood Cliffs, NJ: Prentice Hall.

Stallman, Richard M. 1992 (February).
Using and Porting GNU CC (Version 2.0).
Free Software Foundation.

Tarditi, David, Anurag Acharya, and Peter Lee. 1992.
No assembly required: compiling Standard ML to C.
ACM Letters on Programming Languages and Systems, 1(2):161–177.

Tarditi, D., G. Morrisett, P. Cheng, C. Stone, R. Harper, and P. Lee. 1996 (May).
TIL: A type-directed optimizing compiler for ML.
Proceedings of the ACM SIGPLAN '96 Conference on Programming Language Design and Implementation, in *SIGPLAN Notices*, 31(5):181–192.

Wakeling, D. 1998 (September).
Mobile Haskell: compiling lazy functional languages for the Java virtual machine.
In *Proceedings of the 10th International Symposium on Programming Languages, Implementations, Logics and Programs (PLILP'98)*, Pisa.
To appear.