C--: a portable assembly language that supports garbage collection

Simon Peyton Jones[1], Norman Ramsey[2], and Fermin Reig[3]

[1] simonpj@microsoft.com, Microsoft Research Ltd
[2] nr@cs.virginia.edu, University of Virginia
[3] reig@dcs.gla.ac.uk, University of Glasgow

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, 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 for this purpose. The use of a portable assembly language introduces new problems in the support of such *high-level run-time services* as garbage collection, exception handling, concurrency, profiling, and debugging. We address these problems by combining the `C--` language with a `C--` *run-time interface.* The combination is designed to allow the compiler writer a choice of source-language semantics and implementation techniques, while still providing good performance.

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 appear to be 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 most 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;

Bartlett 1989b; Peyton Jones 1992; Tarditi, Acharya, and Lee 1992; Henderson, Conway, and Somogyi 1995; Pettersson 1995; Serrano and Weis 1995). 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, provides no support for garbage collection, and provides very little support for exceptions or debugging (Section 2).

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*). The language would not only make the compiler writer's life much easier, but would also give the author of a new code generator a ready-made customer base. In an earlier paper we propose a design for just such a language, `C--` (Peyton Jones, Oliva, and Nordin 1998), but the story does not end there. Separating the front and back ends greatly complicates run-time support. In general, the front end, back end, and run-time system for a programming language are designed together. They cooperate intimately to support such high-level features as garbage collection, exception handling, debugging, profiling, and concurrency — *high-level run-time services.* If the back end is a portable assembler like `C--`, we want the cooperation without the intimacy; an implementation of `C--` should be independent of the front ends with which it will be used.

One alternative is to make all these high-level services part of the abstraction offered by the portable assembler. For example, the Java Virtual Machine, which provides garbage collection and exception handling, has been used as a target for languages other than Java, including Ada (Taft 1996), ML (Benton, Kennedy, and Russell 1998), Scheme (Clausen and Danvy 1998), and Haskell (Wakeling 1998). But a sophisticated platform like a virtual machine embodies too many design decisions. For a start, the semantics of the virtual machine may not match the semantics of the language being compiled (e.g., the exception semantics). Even if the semantics happen to match, the engineering tradeoffs may differ dramatically. For example, functional languages like Haskell or Scheme allocate like crazy (Diwan, Tarditi, and Moss 1993), and JVM implementations are typically not optimised for this case. Finally, a virtual machine typically comes complete with a very large infrastructure — class loaders, verifiers and the like — that may well be inappropriate. Our intended level of abstraction is much, much lower.

Our problem is to enable a *client* to implement high-level services, while still using `C--` as a code generator. As we discuss in Section 4, supporting high-level services requires knowledge from *both* the front and back ends. The insight behind our solution is that `C--` should include not only a low-level assembly language, for use by the compiler, but also a low-level run-time system, for use by the front end's run-time system. The only intimate cooperation required is between the `C--` back end and its run-time system; the front end works with `C--` at arm's length, through a well-defined language and a well-defined *run-*

time interface (Section 5). This interface adds something fundamentally new: the ability to inspect and modify the state of a suspended computation.

It is not obvious that this approach is workable. Can just a few assembly-language capabilities support many high-level run-time services? Can the front-end run-time system easily implement high-level services using these capabilities? How much is overall efficiency compromised by the arms-length relationship between the front-end runtime and the `C--` runtime? We cannot yet answer these questions definitively. Instead, the primary contributions of this paper are to identify needs that are common to various high-level services, and to propose specific mechanisms to meet these needs. We demonstrate only how to use `C--` to implement the easiest of our intended services, namely garbage collection. Refining our design to accommodate exceptions, concurrency, profiling, and debugging has emerged as an interesting research challenge.

2 It's impossible — or it's C

The dream of a portable assembler has been around at least since UNCOL (Conway 1958). Is it an impossible dream, then? Clearly not: C's popularity as an assembler is clear evidence that a need exists, and that something useful can be done.

If C is so popular, then perhaps C is perfectly adequate? Not so. There are many difficulties, of which the most fundamental are these:

- The C route rewards those who can map their high-level language rather directly onto C. A high-level language procedure becomes a C procedure, and so on. But this mapping is often awkward, and sometimes impossible.

  For example, some source languages fundamentally require *tail-call optimisation*; a procedure call whose result is returned to the caller of the current procedure must be executed in the stack frame of the current procedure. This optimisation allows iteration to be implemented efficiently using recursion. More generally, it allows one to think of a procedure as a labelled extended basic block that can be *jumped to*, rather than as sub-program that can only be *called*. Such procedures give a front end the freedom to design its own control flow.

  It is very difficult to implement the tail-call optimisation in C, and no C compiler known to us does so across separately compiled modules. Those using C have been very ingenious in finding ways around this deficiency (Steele 1978; Tarditi, Acharya, and Lee 1992; Peyton Jones 1992; Henderson, Conway, and Somogyi 1995), but the results are complex, fragile, and heavily tuned for one particular implementation of C (usually `gcc`).
- A C compiler may lay out its stack frames as it pleases. This makes it difficult for a garbage collector to find the live pointers. Implementors either arrange not to keep pointers on the C stack, or they use a conservative garbage collector. These restrictions are Draconian.

- The unknown stack-frame layout also complicates support for exception handling, debugging, profiling, and concurrency. For example, an exception-handling mechanism needs to walk the stack, perhaps removing stack frames as it goes. Again, C makes it essentially impossible to implement such mechanisms, unless they can be closely mapped onto what C provides (i.e., `setjmp` and `longjmp`).
- A C compiler has to be very conservative about the possibility of memory aliasing. This seriously limits the ability of the instruction scheduler to permute memory operations or hoist them out of a loop. The front-end compiler often knows that aliasing cannot occur, but there is no way to convey this information to the C compiler.

So much for fundamental issues. C also lacks the ability to control a number of important low-level features, including returning multiple values in registers from a procedure, mis-aligned memory accesses, arithmetic, data layout, and omitting range checks on multi-way jumps.

In short, C is awkward to use as a portable assembler, and many of these difficulties translate into performance hits. A portable assembly language should be able to offer better performance, as well as greater ease of use.

3 An overview of `C--`

In this section we give an overview of the design of `C--`. Fuller descriptions can be found in Peyton Jones, Oliva, and Nordin (1998) and in Reig and Peyton Jones (1998). Figure 1 gives examples of some `C--` procedures that give a flavour of the language. Despite its name `C--` is by no means a subset of C, as will become apparent; C was simply our jumping-off point.

3.1 What is a portable assember?

`C--` is an *assembly language* — an abstraction of hardware — not a high-level programming language. Hardware provides computation, control flow, memory, and registers; `C--` provides corresponding abstractions.

- `C--` expressions and assignments are abstractions of computation. `C--` provides a rich set of computational operators, but these operators work only on machine-level data types: bytes, words, etc. The expression abstraction hides the particular combination of machine instructions needed to compute values, and it hides the machine registers that may be needed to hold intermediate results.
- `C--`'s `goto` and `if` statements are abstractions of control flow. (For convenience, `C--` also provides structured control-flow constructs.) The `if` abstraction hides the machine's "condition codes;" branch conditions are arbitrary Boolean expressions.

```
/* Ordinary recursion */
export sp1;
sp1( bits32 n ) {
  bits32 s, p;
  if n == 1 {
      return( 1, 1 );
  } else {
      s, p = sp1( n-1 );
      return( s+n, p*n );
  }
}

/* Tail recursion */
export sp2;
sp2( bits32 n ) {
  jump sp2_help( n, 1, 1 );
}

sp2_help( bits32 n, bits32 s, bits32 p ) {
  if n==1 {
      return( s, p );
  } else {
      jump sp2_help( n-1, s+n, p*n )
  }
}
```

```
/* Loops */
export sp3;
sp3( bits32 n ) {
    bits32 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;
    }
}
```

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

- C-- treats memory much as the machine does, except that addresses used in C-- programs may be arbitrary expressions. This abstraction hides the limitations of the machine's addressing modes.

- C-- variables are an abstraction of registers. A C-- back end puts as many variables as possible in registers; others go in memory. This abstraction hides the number and conventional uses of the machine's registers.

- In addition, C-- provides a procedure abstraction, the feature that looks least like an abstraction of a hardware primitive. However, many processor architectures provide direct support for procedures, although the nature of that support varies widely (procedure call or multiple register save instructions, register windows, link registers, branch prediction for return instructions, and so on). Because of this variety, calling conventions and activation-stack management are notoriously architecture dependent and hard to specify. C-- therefore offers prodedures as a primitive abstraction, albeit in a slightly unusual form (Section 3.4).

Our goal is to make it easy to retarget front ends, not to make every `C--` program runnable everywhere. Although every `C--` program has a well-defined semantics that is independent of any machine, a front end translating a single source program might need to generate two different `C--` programs for two different target architectures. For example, a `C--` program generated for a machine without floating-point instructions would be different from a `C--` program generated for a machine without floating-point instructions.

Our goal contrasts sharply with "write once; run anywhere," the goal of such distribution formats as Java class files, Juice files (Franz 1997), and ANDF or TenDRA (Macrakis 1993). These formats are abstractions of high-level languages, not of underlying machines. Their purpose is binary portability, and they retain enough high-level language semantics to permit effective compilation at the remote installation site.

Even though `C--` exposes a few architecture-specific details, like word size, the whole point is to hide those details, so that the front end job can largely independent of the target architecture. A good `C--` implementation therefore must do substantial architecture-dependent work. For example:

- Register allocation.
- Instruction selection, exploiting complex instructions and addressing modes.
- Instruction scheduling.
- Stack-frame layout.
- Classic back-end optimisations such as common-subexpression elimination and copy propagation.
- If-conversion for predicated architectures.

Given these requirements, `C--` resembles a typical compiler's intermediate language more than a typical machine's assembly language.

3.2 Types

`C--` supports a bare minimum of data types: a family of `bits` types (`bits8`, `bits16`, `bits32`, `bits64`), 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; for example, a `C--` program emitted by a front-end compiler for a 64-bit machine might be rejected if fed to a `C--` implementation for a 32-bit machine. It is easy to tell a front end how big to make its data types, and doing so makes the front end's job easier in some ways; for example, it can compute offsets statically.

The `bits` types are used for characters, bit vectors, integers, and addresses (pointers). On each architecture, a `bits` type is designated the "*native word* *type*" of the machine. A "*native code-pointer type*" and "*native data-pointer type*" are also designated; exported and imported names must have one of these pointer types. On many machines, all three types are the same, e.g, `bits32`.

3.3 Static allocation

`C--` offers detailed control of static memory layout, much as ordinary assemblers do. A data block consists of a sequence of labels, initialised data values, uninitialised arrays, and alignment directives. For example:

```
data {
  foo:  bits32{10};       /* One bits32 initialised to 10 */
        bits32{1,2,3,4}; /* Four initialised bits32's */
        bits32[8];        /* Uninitialised array of 8 bits32's */
  baz1:
  baz2: bits8             /* An uninitialised byte */
  end:
}
```

Here `foo` is the address of the first `bits32`, `baz1` and `baz2` are both the address of the `bits8`, and `end` is the address of the byte after the `bits8`. The labels `foo`, `baz1`, etc, should be thought of as addresses, not as memory locations. They are all immutable constants of the native data-pointer type; they cannot be assigned to.

How, then, can one access the memory at location `foo`? Memory accesses (loads and stores) are typed, and denoted with square brackets. Thus the statement:

```
bits32[foo] = bits32[foo] + 1;
```

loads a `bits32` 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 `bits32` as a C-like array representing all of memory, and `bits32[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. Further, `foo`'s type is always the native data-pointer type; the type of value stored at `foo` is specified by the load or store operation itself. So this is perfectly legal:

```
bits8[foo+2] = bits8[foo+2] - 1;
```

This statement modifies only the byte at address `foo+2`.

Unlike C, `C--` has no implicit alignment or padding. Therefore, the address relationships between the data items within a single `data` block are machine-independent; for example, $\texttt{baz1} = \texttt{foo} + 52$. An explicit `align` directive provides alignment where that is required.

`C--` 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-- expects that, when linking object files, the linker concatenates sections with the same name. (For backwards compatibility with some existing linkers, front ends may wish to emit an alignment directive at the beginning of each C-- section.) C-- assigns no other semantics to the names of data sections, but particular implementations may assign machine-dependent semantics. For example, a MIPS implementation might assume that data in sections named "rodata" is read-only.

3.4 Procedures

C-- supports procedures that are both more and less general than C procedures — for example, C-- procedures offer multiple results and full tail calls, but they have a fixed number of arguments. Specifically:

- A C-- procedure, such as sp1 in Figure 1, has *parameters*, such as n, and *local variables*, such as s and p. Parameters and variables are 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. As with registers, C-- provides no way to take the address of a parameter or local variable.
- 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. Although high-level languages allow a call to occur in an expression, C-- forbids it. For example, it is illegal to write

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

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

```
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. (For similar reasons, assignments in `C--` are statements, not expressions, and `C--` operators have no side effects. In particular, `C--` provides no analog of C's "`p++`.")

– 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 (bits32 x) {
  bits32 y;

  stack { p : bits32;
          q : bits32[40];
  }
  /* Here, p and q are the addresses of the relevant chunks
     of data. Their type is the native data-pointer type. */
}
```

`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 `C--` expression of native code-pointer type. The procedure specified in a call statement 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, or imported from another one:

```
bits32[ptr] = sp1;          /* Store procedure address */
...
r,s = (bits32[ptr])( 4 );   /* Call stored procedure */
```

– A `C--` procedure, like `sp3` in Figure 1, may contain `goto`s 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 from labels, `C--` includes a switch statement, for which the `C--` back end generates efficient code. The most efficient mix of conditional branches and indexed branches may depend on the architecture (Bernstein 1985).

Jump tables of procedure addresses (rather than labels) can be built, of course, and a `C--` procedure can use the `jump` statement to make a tail call to a computed address.

3.5 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 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 need not 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 specialised instances, with specialised calling conventions, for each call site. Escape analysis is necessarily conservative, but a function may be deemed to escape only if its name 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.
- `C--` allows the programmer to specify a particular calling convention (chosen from a small set of standard conventions) for an individual procedure, so that `C--` code can interoperate with foreign code. For example, even though `C--`'s standard calling convention may differ from C's, one can ask for a particular procedure to use C's convention, so that the procedure can be called from an external C program. Similarly, external C procedures can be called from a `C--` procedure by specifying the calling convention at the call site.

  Some `C--` implementations may provide *two* versions of C's calling convention. The lightweight version would be like an ordinary C call, but it would be useful only when the C procedures terminate quickly; if control were transferred to the run-time system while a C procedure was active, the run-time system might not be able to find values that were in callee-saves registers at the time of the call. The heavyweight version would keep all its state on the stack, not in callee-saves registers, so the run-time system could handle a stack containing a mix of C and `C--` activations.

3.6 Miscellaneous

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 {
  bits32 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 storage allocated by `data`, there is no such thing as "the address of a global", so memory stores to unknown addresses cannot affect the value of a global. This permits a global to be held in a register and, even if it has to be held in memory, the optimiser does not need to worry about re-loading 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 (implementation-dependent) registers, for example:

```
global {
  bits32 hp    "%ebx";
  bits32 hplim "%esi";
}
```

We remarked in Section 2 that the front end may know a great deal about (lack of) aliasing between memory access operations. We do not yet have a way to express such knowledge in `C--`, but an adaptation of Novack, Hummel, and Nicolau (1995) looks promising.

4 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, profiling, concurrency, and debugging. In the `C--` framework, the front and back ends work at arm's length. As mentioned earlier, 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.

Separating *policy* from *mechanism* in this way is easier said than done. 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 is the separation between front and back end hard to achieve? *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 profiler must correlate object-code locations with source-code locations, and possibly navigate 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.

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.

A profiler must map loci of control into entities that are profiled: procedures, statements, source-code regions, etc.

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.

5 Support for high-level run-time services

The main challenge, then, is arranging for the back end and front end to share information, without having to implement them as a single integrated unit. In this section we describe a framework that allows this to be done. We focus on garbage collection as our illustrative example. Other high-level run-time services can fit in the same framework, but each requires service-specific extensions; we sketch some ideas in Section 7.

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

5.1 The framework

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 are separately translated to *generated 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 in Section 5.2. 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.

In outline, `C--` can support high-level run-time services, such as garbage collection, as follows. When garbage collection is required, control is transferred to the front-end run-time system (Section 6.1). The garbage collector then walks the `C--` stack, by calling access routines provided by the `C--` run-time system (Section 5.2). In each activation record on the `C--` stack, the garbage collector finds the location of each live variable, using further procedures provided by the `C--` runtime. However, the `C--` runtime cannot know which of these variables holds a pointer. To answer this question, the front-end compiler builds a statically-allocated data block that identifies pointer variables, and it uses a `span` directive (Section 5.3) to associate this data block with the corresponding procedure's range of program counter values. The garbage collector combines these two sources of information to decide whether to treat the procedure's variable as a root. Section 5.4 describes one possible garbage collector in more detail.

5.2 The C-- run-time interface

This section presents the core run-time interface provided by the C-- run-time system. Using this interface, a front-end run-time system can inspect and modify the state of a suspended C-- computation. Rather than specify representations of a suspended computation or its activation records, we hide them behind simple abstractions. These abstractions are presented to the front-end run-time system through a set of C procedures.

The state of a C-- computation 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. Notably, 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` (Harbison and Steele 1995, §19.4).

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 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 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 (whether to use callee saves registers is entirely up to the C-- implementation), 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.

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

`void *FindVar( activation *a, int var_index )` asks an activation handle for the location of any parameter or local variable in the activation record to which the handle refers. The variables of a procedure are indexed by numbering them in the order in which they are declared in that procedure, 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.

`void FirstActivation( tcb *t, activation *a ).` When execution of a `C--` program is suspended, its state is captured by the `C--` run-time system. `FirstActivation` uses that state to initialise an activation handle that corresponds to the procedure that will execute when the program'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.

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

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.

5.3 Front-end information

Suppose the garbage collector is examining a particular activation record. It can use `FindVar` to locate variable number 1, but how can it know whether that variable is a pointer? The front end compiler cooperates with `C--` to answer this question, as follows:

- The front end builds a static initialised data block (Section 3.3), or *descriptor*, that says which of the parameters and local variables of a procedure are heap pointers. The format of this data block is known only to the front-end compiler and run-time system; the `C--` run-time system does not care.
- The front end tells `C--` to associate a particular range of program counters with this descriptor, using a `span` directive.

– The C-- run-time system provides a call, GetDescriptor, that maps an activation handle to the descriptor associated with the program counter at which the activation is suspended.

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

```
data {
  gc1: bits32 2;       /* this procedure has two variables */
       bits8  1;       /* x is a pointer */
       bits8  0;       /* y is a non-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.

Many other encodings are possible. The front end might emit a table that uses one bit per variable, instead of one byte. It might emit a list of the indices of variables that contain pointers. It might arrange for pointer variables to have continuous indices and emit only the first and last such index.[1] The key property of our design is that *the encoding matters only to the front end and its runtime system.* C-- does not know or care about the encoding.

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

```
span GC gc1 {
  f( bits32 x, bits32 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. There may be several independent span mappings in use simultaneously, e.g., one for garbage collection, one for exceptions, one for debugging, and so on. C-- uses integer *tokens* to distinguish these mappings from one another; GC is the token in the example above. C-- takes no interest in the tokens; it simply provides a map from a (token, PC) pair to an address. Token values are usually defined using a const declaration (Section 3.6).

When the garbage collector (say) walks the stack, using an activation handle a, it can call the following C-- run-time procedure:

[1] This scenario presumes the front end has the privilege of reordering parameters; otherwise, it would have to use some other scheme for parameters.

`void *GetDescriptor( activation *a, int token )` returns the address of the descriptor associated with the smallest `C--` span tagged with `token` and containing the program point where the activation `a` is suspended.

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.

The front end may emit descriptors and spans to support other services, not just garbage collection. For example, to support exception handling or debugging, the front end may record the scopes of exception handlers or the names and types of variables. `C--` supports multiple spans, but they must not overlap. Spans can nest, however; 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.

5.4 Garbage collection

This section explains in more detail how the `C--` run-time interface might be used to help implement a garbage collector. Our primary concern is how the collector finds, and possibly updates, roots. Other tasks, such as 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 is up to the front end to pin such pointers or to negotiate changing them with the foreign code. We defer until Section 6.1 the question of how control is transferred from running `C--` code to the garbage collector.

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 `NULL`, the variable is dead, and need not be traced. Otherwise 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 2 shows a simple copying collector based on Appel (1989), targeted to the `C--` run-time interface and the descriptors shown in Section 5.3.

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 TIL, whether a parameter is a pointer may depend on

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

Fig. 2. Part of a simple copying garbage collector

the value of another parameter. For example, a `C--` procedure generated by TIL might look like this:

```
f( bits32 ty, bits32 a, bits32 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--`.

5.5 Implementing the `C--` run-time interface

Can spans and the `C--` run-time interface 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.

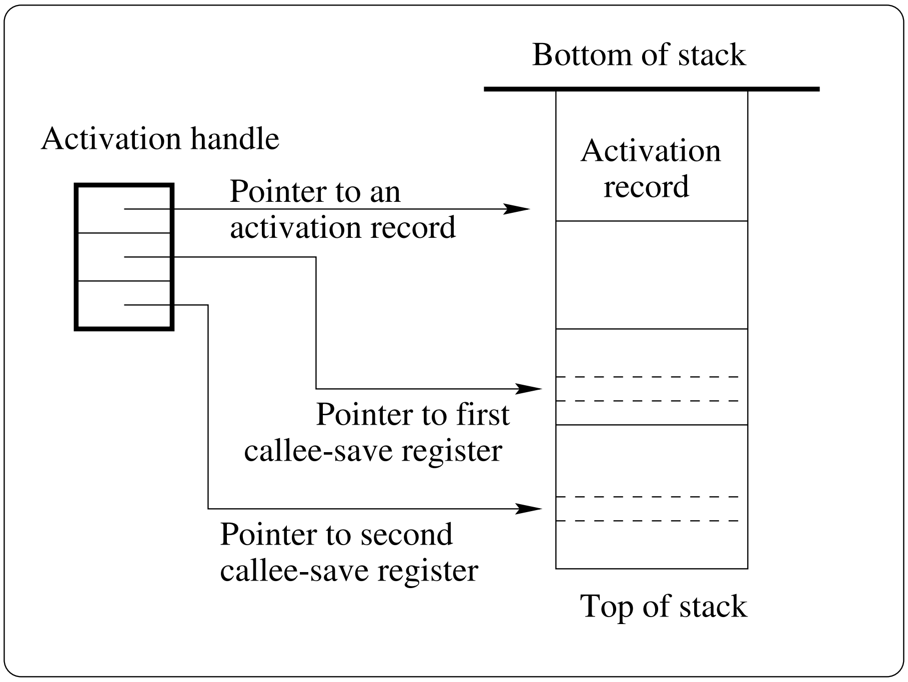
Implementing spans The span mappings of Section 5.3 take their inspiration from table mappings for exception handling, and the key procedure, `GetDescriptor`, can be implemented in similar ways (Chase 1994a). The main challenge is to build a mapping from object-code locations (possible values of the program counter) to source-code location ranges (spans). 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 when the program starts.

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[2] had at the moment when control left the activation record (Figure 3). `FirstActivation` initialises the activation handle to point to the topmost activation record on the stack and to the private locations in the `C--` runtime that hold the values of registers. Depending on the mechanism used to suspend execution, the runtime might have 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 `C--` execution is suspended only at a "safe point." Broadly speaking, a safe point is a point at which the `C--` run-time system is guaranteed to work; we discuss the details in Section 6.2. 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

[2] 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.



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.

Fig. 3. Walking a stack

containing the variable's value, either 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 (Figure 3).

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

The `C--` runtime can map activations to descriptors using the same mechanism it uses to implement the span mappings of Section 5.3. 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. Because these details are private between the back end and the back-end run-time system, we can experiment with different techniques without changing the approach, the run-time interface, or the front end.

6 Refining the design

The basic idea of providing a run-time interface that allows the state of a suspended `C--` computation to be inspected and modified seems quite flexible and robust. But working out the detailed application of this idea to a variety of run-time services, and specifying precisely what the semantics of the resulting language is, remains challenging. In this section we elaborate some of the details that were not covered in the preceding section, and discuss mechanisms that support run-time services other than garbage collection. Our design is not finalised, so this section is somewhat speculative.

6.1 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.

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 an interrupt can cause a transfer of control from generated code to the run-time system. We prefer not to expose this implementation technique through the `C--` run-time interface, but to take a more abstract view. The `C--` runtime therefore operates as if 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. The front-end run-time system 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.

We have to say how a `C--` thread is created, and how control is transferred between the system thread and a `C--` thread.

- The system thread calls `InitTCB` to create a new thread. In addition to passing the program counter for a `C--` procedure without parameters, the system thread must provide space for a stack on which the thread can execute, as well as space for a thread-control block.
- The system thread calls `Resume` to transfer control to a suspended `C--` thread.
- Execution of a `C--` thread continues until that thread calls the `C--` procedure `yield`, which suspends execution of the `C--` thread and causes a return from the system thread's `Resume` call. The `C--` thread passes a *yield code*, which is returned as the result of `Resume`.

For example, garbage collection can be invoked via a call to `yield`, when the allocator runs out of space. Here is how the code might look if allocation takes place in a single contiguous area, pointed to by a heap pointer `hp`, and bounded by `heap_limit`:

```
f( bits32 a,b,c ) {
  while (hp+12 > heap_limit) {
              yield( GC );    /* Need to GC */
  }
  hp = hp+12;
  ...
}
```

It may seem unusual, even undesirable, to speak of two "threads" in a completely sequential setting. In a more tightly-integrated system it would be more usual simply to *call* the garbage collector. But simply making a foreign call to the garbage collector will not work here. How is the garbage collector to find the top of the `C--` portion of the stack that it must traverse? What if live variables (such as `a`, `b`, `c`) are stored in C's callee-saves registers across the call to the garbage collector? Such complications affect not only the garbage collector, but any high-level run-time service that needs to walk the stack. Our two-thread conceptual model abstracts away from these complications by allowing the system thread to inspect and modify a tidily frozen `C--` thread.

Using "threads" does not imply a high implementation cost. Though we call them threads, "coroutines" 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.

Another merit of this two-thread view is that it extends smoothly to accommodate multiple `C--` threads. Indeed, 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.

6.2 Safe points

When can the system thread safely take control? We say 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. 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.

A call to `yield` must be a safe point, and because any procedure could call `yield`, the code generator must ensure that every call site is a safe point. This safe point is associated with the state in which the call has been made and the procedure is suspended awaiting the return. `C--` does not guarantee that every instruction is a safe point; recording local-variable liveness and location information for every instruction might increase the size of the program by a significant fraction (Stichnoth, Lueh, and Cierniak 1999).

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. The answer to the question remains a topic for research; asynchronous pre-emption is difficult to implement, not only in `C--` but in *any* system. Chase (1994b) and Shivers, Clark, and McGrath (1999) discuss some of the problems. One common technique is to ensure that every loop is cut by a safe point, and to permit an interrupted program to execute until it reaches a safe point. `C--` therefore enables the front end to insert safe points, by inserting the `C--` statement

```
safepoint;
```

6.3 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 this function, in which `a+8` is a common subexpression:

```
f( bits32 a ) {
  bits32[a+8] = 10;  /* put 10 in 32-bit word at address a+8 */
  g( a );
  bits32[a+8] = 0;   /* put 0  in 32-bit word at address 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 `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 bits32 a ) {
  invariant bits16 b;
  bits32 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.

If variables will not be changed by 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.

7 Exceptions and other services

In Section 4 we argued that many high-level run-time services share at least some requirements in common. In general, they all need to suspend a running `C--` thread, and to inspect and modify its state. The spans of Section 5.3 and the run-time interface of Section 5.2 provide this basic service, but each high-level run-time service requires extra, special-purpose support. Garbage collection is enhanced by the `invariant` annotation of Section 6.3. Exception handling requires rich mechanisms for changing the flow of control. Interrupt-based profiling requires the ability to inspect (albeit in a very modest way) the state of a thread interrupted *asynchronously*. Debugging requires all of the above, and more besides. We believe that our design can be extended to deal with these situations, and that many of `C--`'s capabilities will be used by more than one high-level service. Here we give an indicative discussion of just one other service, exception handling.

Making a single back end support a variety of different exception-handling mechanisms is significantly harder than supporting a variety of garbage collectors, in part because exceptions alter the control flow of the program. If raising an exception could change the program counter arbitrarily, chaos would ensue; 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. In other words, unconstrained, dynamic changes in locus of control make life hard for the register allocator and the optimiser; if the program counter can change arbitrarily, there is no such thing as dead code, and a variable live anywhere is live everywhere.

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. Many of the mechanisms used for garbage collection are also useful for exception handling; for example, stack walking and spans can be used to find exactly which handler should catch a particular exception. But the mechanisms we have described so far don't allow for changes in control flow. `C--` controls such changes by requiring annotations on procedure calls.

The key idea is that, in the presence of exceptions, *a call might return to more than one location, and every* `C--` *program specifies explicitly all the locations to which a call could return.* In effect, a call has many possible continuations instead of just one. When an activation is suspended at a call site, three outcomes are possible.

- The call returns normally, and execution continues at the statement following the call.
- The call raises an exception that is handled in the activation, so the call terminates by transferring control to a different location in that activation.
- The call raises an exception that is not handled in the current activation, so the activation is aborted, and the run-time system transfers control to a handler in some calling procedure.

`C--`'s call-site annotations specify these outcomes in detail.

We are currently refining a design that supports suitable annotations, plus a variety of mechanisms for transfer of control (Ramsey and Peyton Jones 1999). Exception dispatch might unwind the stack one frame at a time, looking for a handler, or it might use an auxiliary data structure to find the handler, then "cut the stack" directly to that handler in constant time. Our design also permits exception dispatch to be implemented either in the front-end run-time system or in generated code. The `C--` run-time system provides supporting procedures that can unwind the stack, change the address to which a call returns, and pass values to exception handlers.

8 Status and conclusions

The core design of `C--` is stable, and an implementation based on ML-RISC is freely available from the authors. This implementation supports the features described in Section 3, but it does not yet include the `span` directive or a `C--` run-time system.

Many open questions remain. What small set of mechanisms might support the entire gamut of high-level language exception semantics? How about the range of known implementation techniques? Support for debugging is even harder than dealing with exceptions. Exactly what is the meaning of a breakpoint? How should breakpoints interact with optimization? What are the primitive "hooks" required for concurrency support? How should `C--` cope with pre-emption?

These questions are not easily answered, but the prize is considerable. Reuse of code generators is a critically important problem for language implementors. Code generators embedded in C compilers have been widely reused, but the nature of C makes it impossible to use the best known implementations of high-level run-time services like garbage collection, exception handling, debugging, and concurrency — C imposes a ceiling on reuse.

We hope to break through this ceiling by taking a new approach: design a low-level, reusable compiler-target language *in tandem with* a low-level, reusable run-time system. Together, `C--` and its run-time system should succeed in *hiding* machine-dependent details of calling conventions and stack-frame layout. They should eliminate the distinction between variables living in registers and variables living on the stack. By doing so, they should

- Permit sophisticated register allocation, even in the presence of a garbage collector or debugger.
- Make the results of liveness analyses available at run time, e.g., to a garbage collector.
- Support the best known garbage-collection techniques, and possibly enable experimentation with new techniques.

Although the details are beyond the scope of this paper, we have some reason to believe `C--` can also support the best known techniques for exception handling, as well as supporting profiling, concurrency, and debugging.

Acknowledgements

We thank Xavier Leroy, Simon Marlow, Gopalan Nadathur, Mike O'Donnell, and Julian Seward for their helpful feedback on earlier drafts of this paper. We also thank Richard Black, Lal George, Thomas Johnsson, Greg Morrisett, Nikhil, Olin Shivers, and David Watt for feedback on the design of `C--`.

References

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

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.

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. In *ACM Conference on Programming Languages Design and Implementation (PLDI'88)*, pages 329–338. ACM.

Benton, Nick, Andrew Kennedy, and George Russell. 1998 (September). Compiling Standard ML to Java bytecodes. In *ACM Sigplan International Conference on Functional Programming (ICFP'98)*, pages 129–140, Balitmore.

Bernstein, Robert L. 1985 (October). Producing good code for the case statement. *Software Practice and Experience*, 15(10):1021–1024.

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

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, LR and O 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.

Conway, ME. 1958 (October). Proposal for an UNCOL. *Communications of the ACM*, 1(10):5–8.

Diwan, A, D Tarditi, and E Moss. 1993 (January). Memory subsystem performance of programs using copying garbage collection. In *21st ACM Symposium on Principles of Programming Languages (POPL'94)*, pages 1–14. Charleston: ACM.

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.

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

Harbison, Samuel P. and Guy L. Steele, Jr. 1995. *C: A Reference Manual*. fourth edition. Englewood Cliffs, NJ: Prentice Hall.

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.

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

Macrakis, Stavros. 1993 (January). *The Structure of ANDF: Principles and Examples*. Open Systems Foundation.

Novack, Steven, Joseph Hummel, and Alexandru Nicolau. 1995. *A Simple Mechanism for Improving the Accuracy and Efficiency of Instruction-level Disambiguation*, Chapter 19, pages 289–303. Number 1033 in Lecture Notes in Computer Science. Springer-Verlag.

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 (January). Concurrent Haskell. In *23rd ACM Symposium on Principles of Programming Languages (POPL'96)*, pages 295–308, St Petersburg Beach, Florida.

Peyton Jones, SL, D Oliva, and T Nordin. 1998. `C--`: A portable assembly language. In *Proceedings of the 1997 Workshop on Implementing Functional Languages (IFL'97)*, Lecture Notes in Computer Science, pages 1–19. Springer Verlag.

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 Simon L. Peyton Jones. 1999. Exceptions need not be exceptional. Draft available from `http://www.cs.virginia.edu/nr`.

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

Reig, F and SL Peyton Jones. 1998. The `C--` manual. Technical report, Department of Computing Science, University of Glasgow.

Reppy, JH. 1991 (June). CML: a higher-order concurrent language. In *ACM Conference on Programming Languages Design and Implementation (PLDI'91)*. ACM.

Serrano, Manuel and Pierre Weis. 1995 (September). Bigloo: a portable and optimizing compiler for strict functional languages. In *2nd Static Analysis Symposium*, Lecture Notes in Computer Science, pages 366–381, Glasgow, Scotland.

Shivers, Olin, James W. Clark, and Roland McGrath. 1999 (September). Atomic heap transactions and fine-grain interrupts. In *ACM Sigplan International Conference on Functional Programming (ICFP'99)*, Paris.

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.

Steele, GL. 1978. Rabbit: a compiler for Scheme. Technical Report AI-TR-474, MIT Lab for Computer Science.

Stichnoth, JM, G-Y Lueh, and M Cierniak. 1999 (May). Suppport for garbage collection at every instruction in a Java compiler. In *ACM Conference on Programming Languages Design and Implementation (PLDI'99)*, pages 118–127, Atlanta.

Taft, Tucker. 1996. Programming the Internet in Ada 95. In Strohmeier, Alfred, editor, *1996 Ada-Europe International Conference on Reliable Software Technologies*, Vol. 1088 of *Lecture Notes in Computer Science*, pages 1–16, Berlin. Available through `www.appletmagic.com`.

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. In *ACM Conference on Programming Languages Design and Implementation (PLDI'96)*, pages 181–192. Philadelphia: ACM.

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.