Reprinted from *Proceedings of the ACM SIGPLAN 2000 Conference on Programming Language Design and Implementation* (PLDI 2000)

A Single Intermediate Language That Supports Multiple Implementations of Exceptions

Norman Ramsey
Harvard University
nr@eecs.harvard.edu

Simon Peyton Jones
Microsoft Research Ltd
simonpj@microsoft.com

ABSTRACT

We present mechanisms that enable our compiler-target language, C--, to express four of the best known techniques for implementing exceptions, all within a single, uniform framework. We define the mechanisms precisely, using a formal operational semantics. We also show that exceptions need not require special treatment in the optimizer; by introducing extra dataflow edges, we make standard optimization techniques work even on programs that use exceptions. Our approach clarifies the design space of exception-handling techniques, and it allows a single optimizer to handle a variety of implementation techniques. Our ultimate goal is to allow a source-language compiler the freedom to choose its exception-handling policy, while encapsulating the architecture-dependent mechanisms and their optimization in an implementation of C-- that can be used by compilers for many source languages.

1. INTRODUCTION

C-- is a compiler-target language intended to be independent of both source programming language and target architecture (Peyton Jones, Oliva, and Nordin 1997; Peyton Jones, Ramsey, and Reig 1999). Its design accommodates a variety of source languages and leaves room for back-end optimization, all without upcalls from the back end to the front end.

C-- is *not* a universal intermediate language (Conway 1958) or a "write-once, run-anywhere" intermediate language (Lindholm and Yellin 1997). Rather, C-- encapsulates compilation techniques that are well understood, but difficult to implement. Such techniques include instruction selection, register allocation, instruction scheduling, and scalar optimizations of imperative code with loops. Beyond this, C-- also encapsulates the architecture-specific run-time support required for high-level run-time services such as garbage collection, concurrency, debugging, and exception dispatch. It is inappropriate for a back end like C-- to *implement* such services, so the challenge is to identify low-level, primitive mechanisms that a back end should provide, on top of which a C-- client can implement high-level services.

This paper explains how C-- encapsulates the techniques compilers use to support exception dispatch. It makes the following contributions:

- We present the two mechanisms that C-- uses to specify interprocedural control flow: weak continuations that do not outlive their procedure activations, and call-site annotations (Section 4). These mechanisms support, *in a single framework*, three well-known ways of implementing exceptions. C-- also supports continuation-passing style, a fourth implementation technique. Every native-code compiler of which we are aware uses one of these four techniques to implement exceptions.
- To define these mechanisms precisely, we present the intermediate language *Abstract C--* and its formal operational semantics (Section 5). Abstract C-- is easily derived from C-- source.
- It is not immediately obvious how standard analyses and optimizations should be implemented in the presence of exceptions. We therefore present an algorithm for converting abstract C-- into a dataflow graph (Section 6). This may be a novel way of documenting an intermediate representation, and it should be directly useful to implementors, who can then use standard dataflow analyses to produce accurate and safe code even in the presence of exceptions and exception handlers; exceptions need not be treated as special cases. Therefore, a single optimizer should suffice for all C-- programs, regardless of the original source language.
- Our approach is simple and can be applied to other intermediate forms, and it illuminates the design space of exception-dispatch mechanisms.

Not everything here is new. For example, any compiler writer who thinks hard may decide to use extra flow edges to express the optimization constraints of exceptions (Hsieh, Gyllenhaal, and Hwu 1996). But the literature on optimizing in the presence of exceptions is sparse; Hennessy (1981) and Chase (1994b) are rare exceptions. We do not know of other work that presents *language-independent* techniques for such optimization.

PLDI 2000, Vancouver, British Columbia, Canada.

2. WHAT'S THE PROBLEM?

Since we intend to provide exception mechanisms that suffice to support a variety of programming languages, we begin by surveying the mechanisms that are widely used. We are aware of four techniques that are used to transfer control from the point where a source-language exception is raised to the point at which that exception is handled.

- *Stack cutting* sets the stack pointer and the program counter to point directly to the handler. In a native-code compiler, this technique can be very fast, but it does not restore the values of callee-saves registers; in general, these values may be distributed throughout the stack. This technique may be best suited to implementations that use no callee-saves registers. Objective CAML uses this technique, as do many Common Lisps and pre-Scheme Lisps.

  In C code, `setjmp` and `longjmp` cut the stack, but they typically save and restore lots of state: the size of a `jmp_buf` is 6 pointers on Pentium/Linux, 19 on Sparc/Solaris, and 84 on Alpha/Digital-Unix. Although `setjmp` and `longjmp` do not always use the entire buffer, they are significantly more expensive than a native-code stack cutter, which saves 2 pointers. On the SPARC, `longjmp` pays the additional penalty of flushing register windows. Because of the performance penalty, `setjmp` and `longjmp` are used only by implementations that compile to C, such as SRC Modula-3.

- *Run-time stack unwinding* uses the run-time system to unwind the stack one frame at a time until the handler is reached. The run-time system restores the values of callee-saves registers as it unwinds the stack, typically by interpreting tables deposited by the back end. Operating systems may provide support for run-time stack unwinding; for example, the MIPS ABI provides a "run-time procedure table," and Digital Unix includes a well-specified ABI for call stacks. The native-code back ends of Polytechnic Modula-3 use this technique, as do the the Java Virtual Machine and many C++ compilers.

- *Native-code stack unwinding* uses specialized code in each procedure to unwind the stack when a "nonlocal return" or "exceptional termination" is called for. Because the compiler generates native code to do the unwinding, no interpretive overhead is involved. The Self compiler uses this technique.

- In *continuation-passing style*, the potential exception handlers are represented by an exception continuation. Generated code raises an exception by making a tail call to this continuation. The continuation decides which handler applies. Again, the compiler generates specialized code for each handler. Standard ML of New Jersey uses this technique.

C-- supports continuation-passing style through fully general tail calls (Peyton Jones, Oliva, and Nordin 1997), which require no further explanation. We therefore discuss only the first three mechanisms. Although these mechanisms are well understood, supporting any of them requires intimate cooperation among the optimizer, the code generator, and the run-time system.

```
/* 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 );
  }
}
```

```
/* 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;
}   }
```

```
/* 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 );
  }
}
```

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

- Correct exception dispatch depends both on the semantics of exceptions in the source language and on the representation of the call stack on the target machine; the interactions may be subtle.

- Optimization is fundamentally affected by exceptions. Optimizing compilers may have rather *ad hoc* modifications that make the optimizers "do the right thing" for the exception semantics of one particular language.

The easy way out is for the code generator to know the language-specific details of exception semantics, and for the run-time system to know the code generator's stack layouts and register-saving protocols. But such intimate cooperation is not available to a *reusable* code generator, which must support multiple source languages on multiple target architectures.

A main contribution of this paper is to show how a reusable code generator can cooperate with front ends at arm's length, and yet still support a variety of exception semantics in an architecture-independent way. Our key observation is that *a language alone cannot provide a sufficiently flexible interface.* C-- includes not only a language, which cooperates with the the source-language compiler, but also a run-time system, which cooperates with the source-langage run-time system.

3. C--: A PORTABLE ASSEMBLER

Before discussing how C-- deals with exceptions, we sketch the language and its run-time system; more details can be found in Peyton Jones, Oliva, and Nordin (1997) and Peyton Jones, Ramsey, and Reig (1999). To give a feel for C--, Figure 1 presents three C-- procedures, each of which computes the sum and product of the integers $1..n$.

3.1 The C-- language

Much of C-- is unremarkable. C-- has parameterized procedures with declared local variables. A procedure body consists of a sequence of statements, which include assignments, conditionals, `goto`s, calls, and `jump`s (tail calls).

Figure 1 illustrates two features that are common in assemblers, but less common in programming languages. First, a procedure may return multiple results. For example, all the procedures in Figure 1 return two results, and `sp1` contains a call to a multi-result procedure (namely `sp1` itself). Second, a C-- procedure may explicitly tail-call another procedure. For example, `sp2` tail-calls `sp2_help` (using "`jump`"), and the latter tail-calls itself. A tail call has the same semantics as a regular procedure call followed by a return, but it is guaranteed to deallocate the caller's resources (notably its activation record) before the call.

C-- has an extremely modest type system: the only types are words and floating-point values of various sizes, e.g., `bits8`, `bits16`, `bits32`, `bits64`, `float32`, and `float64`. For each target architecture, each implementation of C-- designates one of the `bits`n types as the "native data-pointer type" and one as the "native code-pointer type." For example, the name of a procedure, like `sp1` in Figure 1, denotes an immutable value of the native code-pointer type.

By intent, the C-- type system does not protect the programmer—its sole purpose is to direct the C-- compiler's use of machine resources; in particular, its mapping of variables to registers. For example, C-- does not check the number or types of arguments passed to a procedure.

Both local and global variables model machine registers, not memory locations. Therefore, variables have no addresses—whenever possible, the C-- back end maps them to registers. C-- also permits the programmer to declare names that refer to blocks of memory allocated either globally or in a procedure's activation record, but these names do *not* stand for variables; they stand for addresses of memory blocks, and as such they denote immutable values of the native data-pointer type.

All memory access is explicit. For example, the statement

```
bits32[x] = bits32[y] + 1;
```

loads a 32-bit word from the memory location whose address is in the variable `y`, increments it, and stores it in the memory location whose address is in the variable `x`.

3.2 Programs as graphs

We regard a C-- program as *the textual description of a control-flow graph*, or rather, of a set of named control-flow graphs, one for each procedure. C-- syntax is designed to *make all intraprocedural control-flow edges explicit.*

For example, a *label*, like `loop` in `sp3`, names a *node* in the graph, and a `goto` creates an edge to the specified label. The target of a `goto` must be a label in the same procedure. A label is a value, so the target of a `goto` can be computed at run time, but such a `goto` must statically identify all possible targets, so the C-- compiler can include those edges in its control-flow graph.

3.3 The run-time systems

We assume that an executable program is built by linking together three parts, each of which may be found in object files, libraries, or a combination.

- The front end translates the high-level source program into one or more C-- modules, which the C-- compiler translates to *generated object code.*
- The front end comes with a (probably large) *front-end run-time system.* This run-time system implements all *policy*, as well as any mechanisms that depend on the source language. These may include a garbage collector, exception dispatcher, thread scheduler, etc. The front-end run-time system is written in a programming language designed for humans, not in C--; here we assume it is written in C.
- Every C-- implementation comes with a (hopefully small) *C-- run-time system.* This run-time system encapsulates architecture-specific *mechanisms*, and it provides services to the front end run-time system through a C-language run-time interface (Peyton Jones and Ramsey 1998). Different front ends may interoperate with the same C-- run-time system.

The main service provided by the C-- run-time interface is to present the state of a suspended C-- computation ("thread") as a stack of abstract *activations.* Operations are provided to walk down the stack; to get information from an activation; to make a particular activation become the topmost one; and to change the resumption point of the topmost activation. These operations are summarized in Table 1; the latter three operations are discussed in Section 4. Given knowledge of stack layout, implementing these operations is straightforward; the representation of an activation is likely to include copies of callee-saves registers and a pointer to an activation record on the real call stack.

C-- syntax enables a front end to associate with each call site one or more arbitrary static data blocks, or *descriptors*, each of which is allocated and initialized by the front end. The syntax is not important in this paper. At run-time, the C-- run-time interface provides `GetDescriptor`, which returns the `n`'th descriptor associated with a particular activation.

How is control transferred between a running C-- computation and a run-time system? A front-end runtime may create many C-- threads, each of which makes a coroutine call to the front-end runtime to request services like thread switching or exception dispatch. Or a runtime may create a single C-- thread, which runs on the system stack and requests services by making ordinary calls to the front-end runtime. In either case, the C-- thread initiates the interaction by calling the special C-- procedure `yield`.

`Resume(t)`	Resumes C-- thread `t`.
`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)`	Arranges for thread `t` to resume execution with activation `a`.
`SetUnwindCont(t, n)`	Arranges for thread `t` to resume execution by unwinding to the `n`'th continuation of the activation with which it is set to resume.
`SetCutToCont(t, k)`	Arranges for thread `t` to resume execution by cutting the stack to continuation `k`.
`FindContParam(t, n)`	Returns a pointer to the location in which the `n`'th parameter of the currently-set continuation will be returned to thread `t`.
`GetDescriptor(a, n)`	Returns a pointer to the `n`'th descriptor associated with activation `a`.

Table 1: The C-- run-time interface.

4. IMPLEMENTING HIGH-LEVEL EXCEPTIONS IN C--

We now turn our attention to exceptions. Whatever the details, exceptions change the flow of control. The destination of an exceptional control transfer is usually called a *handler*; finding a handler and transferring to it is called *exception dispatch*. C-- models handlers as *continuations*, and it provides several control-transfer mechanisms. C-- uses annotations on call sites to tell the optimizer what exceptional control transfers can take place. These annotations also help implement the control-transfer mechanisms. This section describes the mechanisms; Appendix A shows examples of their use.

4.1 C-- Continuations

We model an exception handler as a C-- continuation, which is a bit like a label with parameters.

```
f( bits32 x, bits32 y ) {
  float64 w;
   ...
  g( x, k ) also cuts to k ;
              /* k may be "cut to" by g, or
   ...             by something g calls */
  return;
  continuation k( x ):
    ... code for k, mentioning x, y, w ...
}
```

Here, `k` is a continuation, which is passed to `g`. A continuation can be declared only inside a procedure. The `x` in `continuation k(x)` is *not* a binding instance; the "formal parameters" of a continuation must be variables of the enclosing procedure, and therefore they need no type declarations. Like every C-- continuation, `k` denotes a value, which can be used to transfer control to it, as shown in Section 4.2. The value `k` encapsulates a stack pointer and a program counter.

A continuation value may be passed to procedures or stored in data structures; its type is the native data-pointer type. Once an activation dies, however, its continuations die too. Invoking a dead continuation is an *unchecked* run-time error, which it is up to the high-level front end to avoid. C-- continuations are therefore less powerful than (say) Scheme continuations, but they can be implemented very efficiently, without stack copying. To avoid unchecked errors, a front end might protect invocations with a run-time check, or it might impose invariants that guarantee no dead continuation can ever be invoked.

Execute in	Stack walk required? No	Yes
Generated code	`cut to`	`return` <*m*/*n*>
Run-time system	`SetCutToCont`	`SetActivation` and `SetUnwindCont`

Figure 2: Alternatives for control transfer.

The annotation "`also cuts to k`" at `g`'s call site indicates that control might flow from the call directly to `k`. We discuss call-site annotations in Section 4.4.

4.2 Transferring control to a C-- continuation

A C-- procedure can transfer control to a continuation in any of four ways, each of which has a different cost model. The four mechanisms that C-- provides offer compiler writers alternatives for each of two trade-offs.

- *Does raising an exception involve walking the stack?* Walking the stack makes raising an exception expensive, but can make it cheaper to enter the scope of a handler.
- Should the bulk of the work be done *in generated code or in the run-time system?* Using generated code results in larger executables, but they may be faster.

Figure 2 shows the four mechanisms that support these alternatives. In the first row are C-- primitives, which result in generated code. In the second row are entry points in the C-- run-time interface (Table 1). We describe of the four design choices in Figure 2 in turn, starting with the top left corner and working counterclockwise.

Stack cutting (first column of Figure 2)

Given a continuation value, the C-- primitive "`cut to`" invokes the continuation, transfering control directly to the continuation without walking the stack.

```
cut to k( arguments );
```

Here, `k` is a continuation value, such as that passed to `g` in the example of Section 4.1. The `cut to` primitive transfers arguments to conventional locations[1], truncates the stack to `k`'s activation, and sets the program counter to `k`'s program

[1]These locations, which are typically registers, are determined by a calling convention that is private to C--.

counter. All of this takes constant time; there is no stack walk.

Instead of using `cut to`, the C-- program may `yield` to the front-end run-time system. Once the latter finds a continuation value, it can duplicate the effect of `cut to` by

1. calling `SetCutToCont` (Table 1), which cuts the stack,
2. calling `FindContParam`, which identifies the locations in which the continuation expects parameters,
3. storing the actual parameters in those locations, and
4. calling `Resume` to pass control back to generated code.

Stack cutting takes constant time, but it imposes the cost of recording the continuation value to be `cut to`. Because C-- places few constraints on the identification of target continuations, different front ends can use different implementation techniques. Here are two common choices:

- Have the program keep track of a single "exception continuation," perhaps in a register. Commonly, when control is transferred to this continuation, it updates the register to point to a new exception continuation.
- Keep a global stack of continuations; choose the topmost one. Alternatively, keep information about which exceptions were handled by the continuation, and choose the first one that applies to the given exception.

Both choices impose a small cost whenever execution enters or leaves the scope of an exception handler, regardless of whether the exception is raised.

The stack-cutting technique also reduces the utility of callee-saves registers. Normally, we could keep `y` and `w` in callee-saves registers across the call to `g`. But the stack-cutting technique cannot restore the values of `y` and `w` before entering `k`. Why not? Because the values of `y` and `w` might have been spilled into *any* activation in the stack between `k`'s own activation and the place where the exception is raised. Stack cutting therefore imposes a small performance penalty on any call that can `cut to` a continuation; the callee-saves registers must be considered killed by flow edges from the call to any `cut to` continuations. This penalty, too, is paid regardless of whether the continuation is used.

Stack unwinding (second column of Figure 2)

A well-known alternative to stack cutting is to walk the stack, one activation at a time, to discover the topmost activation that can handle a given exception. The idea is that it should cost nothing to enter or leave the scope of a handler; in exchange, we are willing to pay more to raise an exception. C-- supports this approach as well.

To use the unwinding technique, the C-- program may `yield` to the front-end run-time system, indicating (in some manner that C-- neither knows nor cares about) which exception to raise. The front-end runtime then initialises an activation handle with `FirstActivation`, and walks the stack using `NextActivation` (Table 1). For each activation it calls `GetDescriptor` to find the static descriptor deposited for the activation by the front-end compiler. If this descriptor indicates that this activation can handle the exception, the front-end runtime uses `SetActivation` to arrange that execution will resume at the activation thus identified.

The suspended call site in this activation should look something like this:

```
g( x ) also unwinds to k0, k1;
```

Here, `k0` and `k1` are continuations, defined in the same procedure as the call to `g`. The `also unwinds to` annotations indicate control flow, just like the `also cuts to` annotation discussed in Section 4.1. In addition, `also unwinds to` supports the `SetUnwindCont` run-time interface call (Table 1). The front-end runtime uses `SetUnwindCont(t, n)` to arrange that when execution is resumed, it will resume at the `n`'th continuation in the "`unwinds to`" list of the call site. Finally, it can use `FindContParam` as before to find where to store parameters to the continuation, and `Resume` to resume execution. Figure 9 in Appendix A shows the details.

The unwinding technique described so far is somewhat interpretive: the front-end runtime walks the stack, looking at descriptor information until it finds a handler. C-- also makes it possible to compile this stack walk, by allowing a procedure to *return abnormally to its caller*, thus:

```
return <0/2> ( return values );
```

This tells C-- that the caller has two abnormal return continuations (in addition to the normal return point), and causes a return to the first (index `0`) of these two. A normal return to such a call site would be written

```
return <2/2> ( return values );
```

The call site to which such a call returns must specify precisely the correct number of `returns to` continuations as are specified in the `return` statement:

```
g( x ) also returns to k0, k1;
```

where `k0` and `k1` are, as before, continuations declared in the same procedure as the call site. The statement `return <0/2> (p,q)` would return to continuation `k0`, passing `p` and `q` from the return site to the parameters of `k0`. The normal return continuation is always the last, so a normal return in this case would be `return <2/2>`. An unannotated `return` is equivalent to `return <0/0>`.

Because `return` transfers control only from a procedure to its caller, all procedures must cooperate to get the effect of compiled unwinding. In a language that specifies statically what exceptions a procedure may raise, one might compile each call site with an abnormal-return continuation for each possible exception. Alternatively, one might use a single abnormal-return continuation to dispatch all exceptions. C-- supports both styles efficiently, leaving the choice to the implementor of the front end.

How are these abnormal returns implemented? It would be possible simply to return an additional value from each procedure, which the caller could test to see whether the callee had requested stack unwinding or a normal return. Such a test, however, would add an overhead at every call. The overhead can be eliminated by means of a clever code-generation trick. At the call site, the call instruction is followed not by the code to be executed after a normal re-

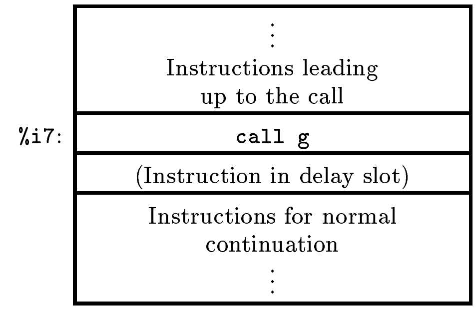

Figure 3: Standard SPARC instruction sequence at call site.

Normal return is `jmp %i7+8`.

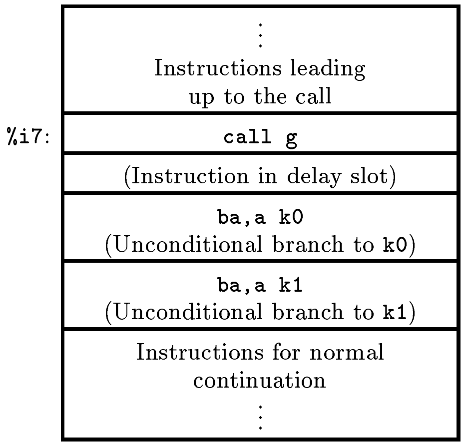

Figure 4: SPARC instruction sequence using the branch-table method.

Normal return is `jmp %i7+16`; return to continuation `k0` is `jmp %i7+8`.

turn, but by a table of branches to continuations (Atkinson, Liskov, and Scheifler 1978). Figures 3 and 4 show an example, using the SPARC instruction set. Figure 3 shows the instruction sequence at an ordinary call site. The `call` instruction leaves its address in register `%i7`, and the instruction in the delay slot is executed immediately after the call, before control is transferred to `g`. The standard return instruction is `jmp %i7+8`, which skips past the instruction in the delay slot and resumes executing the normal continuation.

Figure 4 shows the instruction sequence at the annotated call site `g( x ) also returns to k0, k1`. The callee uses `jmp %i7+16` for a normal return; it uses `jmp %i7+8` or `jmp %i7+12` to return to continuation `k0` or `k1`. This technique has no dynamic overhead in the normal case.[2] Even in the abnormal case, the only dynamic overhead is a branch to a branch, which is much cheaper than branch followed by test and conditional branch. Atkinson, Liskov, and Scheifler (1978) calls this technique the "branch-table method," noting that because it adds words to every call site, the space overhead may be "considerable." On other processor architectures, the run-time overhead may also be considerable; the unusual return address used in the normal case may require extra instructions, and it may also confuse branch-prediction hardware. Still, the branch-table method is used in the Self compiler to implement nonlocal returns.[3] It can also support an efficient implementation of the "vectored returns" used in the Glasgow Haskell Compiler (Peyton Jones 1992, §9.4).

Whether a stack walk is implemented interpretively, in the run-time system, or in native code, using nonlocal return, it can easily restore the values of the callee-saves registers. Indeed, `NextActivation` does so automatically[4]. So the unwinding technique allows callee-saves registers to be used at *every* call site, even if those values might be used in a continuation.

[2] Even the branch-prediction hardware may work without overhead, because under the standard SPARC calling convention, some C procedures return to `%i7+8` and some to `%i7+12`.

[3] Private communication from Craig Chambers, January 1999.

4.3 Primitive operations that can fail

C-- expressions represent pure computations on values; they are evaluated without side effects, which occur only as the result of assignments or calls. What, then, are we to do when such computations fail because of a fault detected by the hardware or the operating system? For example, what about a divide instruction that traps when the divisor is zero? For each such operation we provide two variants:

- The fast-but-dangerous variant (`%divu`, say) generates the shortest possible code sequence (usually one instruction), but its behavior is unspecified if it fails. `%divu(x,0)` might cause an interrupt, kill the process, or silently give the wrong answer. The exact behavior will vary between architectures. (An alternative would be to guarantee process abortion, but for certain processors this alternative would impose run-time costs that might be annoying in cases where the divisor was provably non-zero.)
- The slow-but-solid variant (`%%divu`) maps failure into a `yield`. For example, `%%divu` is indistinguishable from a procedure defined as follows:

```
%%divu( bits32 p, bits32 q ) {
  if q == 0 then { yield( DIVZERO ) }
  return( %divu( p, q ) )
}
```

[4] Since `x` and `y` may be in different callee-saves registers at different call sites, the same continuation may need different prologues at different call sites. These prologues roughly correspond to ϕ-nodes in SSA form. The dispatcher must choose the prologue that is appropriate for the call site at which `f` is suspended. Luckily, this choice can be hidden behind the C-- run-time interface.

Note that if the run-time system fails to unwind or cut the stack, the behavior of the subsequent call to `%divu` is unspecified.

A use of `%%divu` takes the form of a procedure call, together with its `also` annotations. As well as making the control flow explicit, writing faulting operations as calls also ensures that the operations are evaluated in a well-defined order. The C-- implementation can choose whether to perform the test for zero explicitly (slow, but easy), or instead catch the interrupt and map it into a `yield` (fast, but tricky).

4.4 Informing the optimizer

Exception dispatch changes the flow of control. If the optimizer knows only that control might be transferred unexpectedly to a continuation, it has to make pessimistic assumptions; for example, some Ada compilers require that if *any* exception handler uses a variable, that variable must *always* be kept in memory. As another example, C compilers make pessimistic assumptions about local variables in the presence of `setjmp` and `longjmp`. The only portable way to guarantee that local variables will have the right values after a `longjmp` is to declare them `volatile`, which many C compilers interpret to mean "always keep the values of these variables on the stack."

C-- supports aggressive optimization by requiring that the front end tell the optimizer explicitly how an exception dispatch might change the flow of control. This information is conveyed through the annotations attached to call sites, some of which we have mentioned already. The following example shows the complete set:

```
r = g( x ) also cuts    to k1
           also unwinds to k2, k3
           also returns to k4
           also aborts;
```

In the normal case, the call to `g( x )` returns a value, which is placed in `r`. However, if the call to `g` raises an exception, the exception dispatcher may *cut the stack* by invoking continuation `k1` (with a loss of callee-saves registers), *unwind the stack* to continuations `k2` or `k3`, *return to alternate continuation* `k4`, or *abort the execution* of the procedure activation containing the call (e.g., by unwinding or cutting the stack past that activation). The names appearing in these annotations, like `k1`, `k2`, `k3`, and `k4`, are always names of continuations declared in the same procedure as the call site; the annotations may *not* name variables or expressions.

The "`also`" annotations add extra flow edges, from the call site to the specified continuations or to the exit node of the procedure (in the case of `also aborts`). These edges express precisely the constraints that exception handling imposes, but no more. The annotations cannot reasonably be inferred by the C-- compiler on its own; only the front-end compiler knows which calls can flow to which continuations and which calls can abort. If the control flow cannot be determined accurately, the front end, not C--, decides what approximation is useful.

The `also cuts to` annotation may also be attached to a `cut to` statement. An unannotated `cut to` is considered simply to exit the current procedure, but if the `cut to` could transfer control to a continuation in the same procedure, it must have an `also cuts to` annotation naming that continuation.

5. OPERATIONAL SEMANTICS OF C--

So far, our treatment of C-- has been informal, as is common in in descriptions of exception handling.[5] But without a precise specification it is impossible to say for sure whether a particular optimization changes the behavior of the program. There is also the risk that a front-end compiler and a C-- compiler might disagree about what happens in some obscure circumstance. Accordingly, in this section we describe C-- formally and precisely.

We define *Abstract C--*, a language that resembles the flow-graph representations used in optimizing compilers, and we give it a formal operational semantics. The operational semantics uses *transition rules* to specify the permissible behaviors of both the generated code and the run-time system. We also sketch the translation of C-- to Abstract C-- and the implementation of Abstract C-- on hardware.

A *program* in Abstract C-- is a partial map χ from names to *procedures*. A procedure is a *control-flow graph* formed using the *nodes* defined in Table 2. The range of χ includes only nodes of the form $Entry\ \vec{k}\ p$ or $Yield$.

The mutable state of the C-- abstract machine has 7 components:

1. the *control* p, which represents the current node,
2. the *local environment* ρ, which maps names to *values*,
3. a set cs containing the variables of ρ that are stored in callee-saves registers,
4. a *unique integer uid*, which is used to enforce the restriction against using dead continuations,
5. a *memory* M, mapping addresses to values,[6]
6. an *argument-passing area* A, which is a list of values, and
7. a *stack* s, which is either empty or is a tuple consisting of a *continuation bundle*, a local environment, a callee-saves variable set, a unique id, and a stack. A continuation bundle encodes the possible outcomes of a procedure call.

We write a state as follows:

$$p \quad \rho \quad cs \quad uid \quad M \quad A \quad s$$

5.1 Environments, values, and expressions

An *environment* is a partial function from names to values. We write the empty environment as $\perp$. To define an environment that is like ρ except that it maps v to e, we write $\rho[v \mapsto e]$. We generalize this notation to $\rho[\vec{v} \mapsto \vec{e}]$ when we have a list of variables $\vec{v}$ and a list of expressions $\vec{e}$, both the same length. To define an environment that is like ρ except that it is undefined on the variables in set s, we write $\rho \setminus s$.

[5]The ML community, which has a long-standing tradition of formal definitions and analyses, is an honorable exception.
[6]It is straightforward to generalize to a machine with separate address spaces for instructions and data.

Entry $\vec{k}$ p	The unique entry node of a procedure with continuations $\vec{k}$ and first node p.
Exit j n	Normal exit from a procedure, representing a return to continuation j (the call site must have exactly n alternate return continuations tagged with `also returns to`).
CopyIn $\vec{v}$ p	Put results from a call, or parameters to a procedure or continuation, into variables $\vec{v}$, and continue with p.
CopyOut $\vec{e}$ p	Make the values of expressions $\vec{e}$ results of a call, or the parameters to a procedure or continuation, and continue with p.
CalleeSaves cs p	Make cs the set of variables in callee-saves registers (by spilling or reloading), and continue with p.
Assign l e p	Assign e to l, and continue with p.
Branch c p_t p_f	Branch to p_t or p_f when c is true or false.
Call e_f κ	Call procedure e_f, returning to one of the nodes in the *continuation bundle* κ. A continuation bundle is a quadruple $(\vec{p}_r, \vec{p}_u, \vec{p}_c, \mathit{abort})$, where $\vec{p}_r$ are the nodes for continuations listed in `also returns to`, plus the node for normal returns, $\vec{p}_u$ are the nodes for continuations listed in `also unwinds to`, $\vec{p}_c$ are the nodes for continuations listed in `also cuts to`, and *abort* is either *True* (when a call site is annotated with `also aborts`) or *False* (otherwise).
Jump e_f	Tail call procedure e_f. Exits the current procedure.
CutTo e_c	Cut the stack to continuation e_c. Exits the current procedure.
Yield	Execute a procedure in the run-time system.

Table 2: Kinds of nodes in control-flow graph.

To enable variables to denote procedures and continuations as well as basic C-- values, we define a *value* as one of the following forms:

$\mathit{Bits}_n\ k$	The n-bit value k
$\mathit{Code}\ p$	(A pointer to) the node p
$\mathit{Cont}(p, u)$	A continuation to the node p in the stack frame with unique id u.

Because C-- expressions have no side effects, we can give the semantics of an expression e simply by giving an evaluation function $\mathcal{E}[\![e]\!]\rho M$. The exact definition of $\mathcal{E}$ is not relevant to this paper, so we present an abbreviated definition that gives the idea. For simplicity, we assume that the names of local variables are different from the names of procedures, so for any ρ, dom $\rho \cap$ dom $\chi = \emptyset$. Under this assumption, $\mathcal{E}[\![e]\!]\rho M$ might be defined as follows:

$$\begin{array}{ll}
\mathcal{E}[\![v]\!]\rho M = \rho(v) & \text{if } v \in \text{dom } \rho \\
\mathcal{E}[\![v]\!]\rho M = \mathit{Code}\ (\chi(v)) & \text{if } v \in \text{dom } \chi \\
\mathcal{E}[\![\mathit{type}\,[e]\,]\!]\rho M = \mathit{load}_{type}(M, \mathcal{E}[\![e]\!]\rho M) & \\
\mathcal{E}[\![e_1 + e_2]\!]\rho M = \mathit{Bits}_m\ (n_1 + n_2) & \text{if } \mathcal{E}[\![e_i]\!]\rho M = \mathit{Bits}_m\ n_i
\end{array}$$

Rules like the + rule would apply to the other built-in operators. The load_{type} and store_{type} operations use the native byte order of the target machine.

5.2 Transitions of the abstract machine

The C-- abstract machine executes a program by entering the initial state

$$\chi(\texttt{main}) \quad \perp \quad \emptyset \quad 0 \quad \perp \quad \mathit{nil} \quad \mathit{empty}$$

The machine makes transitions until it reaches a state in which no transitions are possible. If, in that state, the control is *Exit* 0 0 and the stack is empty, we say the program has *terminated normally*; otherwise it has *gone wrong.*

A set of transition rules describes the allowable transitions. Each transition rule has the form:

$$\boxed{\begin{array}{l}\langle \mathit{State\ 1} \rangle \\ \Longrightarrow \\ \langle \mathit{State\ 2} \rangle\end{array}}$$

If the machine is in a state matching $\langle \mathit{State\ 1} \rangle$ then it can move in one step to $\langle \mathit{State\ 2} \rangle$ (suitably instantiated). The rest of this section gives the transition rules.

A procedure's entry node binds the procedure's continuations into an empty environment. The incoming environment, ρ, is discarded. (The values of parameters are bound later by a *CopyIn* node.)

$$\begin{array}{llllll}
\mathit{Entry}\ \vec{k}\ p & \rho & cs & uid & M & A \quad s \\
\Longrightarrow & & & & & \\
p & \mathit{addConts}(\perp, \vec{k}, uid) & \emptyset & uid & M & A \quad s
\end{array}$$

where each k in $\vec{k}$ is a pair (v, p) and *addConts* creates bindings as follows:

$$\begin{array}{l}
\mathit{addConts}(\rho, \vec{k}, uid) \\
\quad = \rho, \qquad \text{where } \vec{k} \text{ is empty} \\
\quad = \mathit{addConts}(\rho[v \mapsto \mathit{Cont}(p, uid)], \vec{k}', uid), \\
\qquad\quad \text{where } \vec{k} \text{ is } (v, p) \text{ followed by } \vec{k}'.
\end{array}$$

The sequence $\vec{k}$ lists the continuations declared in the procedure body. Each $k \in \vec{k}$ is a (v, p) pair, where v is the name of the continuation and p is the graph node representing it.

Exit pops an activation, returning to the return continuation named in a `return`. The value-passing area A may hold return values, placed there by a preceding *CopyOut* node.

$$\begin{array}{lllllll}
\mathit{Exit}\ j\ n & \rho & cs & uid & M & A & ((\kappa, \rho', cs', uid', s') \\
\Longrightarrow & & & & & & \\
\vec{p}_r{}'[j] & \rho' & cs' & uid' & M & A & s'
\end{array}$$

where $\kappa = (\vec{p}_r{}', \vec{p}_u{}', \vec{p}_c{}', \mathit{abort})$, and $|\vec{p}_r{}'| = n + 1$.

The value-passing area A holds arguments and results that are passed among procedures and continuations. The *CopyIn* and *CopyOut* nodes transfer values from and to that area. To enable an implementation to reuse registers in A, we specify that *CopyIn* replace A by the empty list, *nil*. *CopyOut* may overwrite A whatever its state.

$$\begin{array}{llllllll} \mathit{CopyIn}\ \vec{v}\ p & \rho & cs & uid & M & A & s \\ \Longrightarrow \\ p & \rho[\vec{v} \mapsto A] & cs & uid & M & nil & s \end{array}$$

$$\begin{array}{lllllll} \mathit{CopyOut}\ \vec{e}\ p & \rho & cs & uid & M & A & s \\ \Longrightarrow \\ p & \rho & cs & uid & M & \mathcal{E}[\![\vec{e}]\!]\rho M & s \end{array}$$

The optimizer may move values into and out of callee-saves registers.

$$\begin{array}{lllllll} \mathit{CalleeSaves}\ cs'\ p & \rho & cs & uid & M & A & s \\ \Longrightarrow \\ p & \rho & cs' & uid & M & A & s \end{array}$$

CalleeSaves nodes are introduced only by optimizers; they are not part of the direct translation of any C-- program into Abstract C--. A *CalleeSaves* node corresponds to a mix of spills and reloads.

Assignment to a variable changes the local environment; assignment to memory changes memory.

$$\begin{array}{lllllll} \mathit{Assign}\ v\ e\ p & \rho & cs & uid & M & A & s \\ \Longrightarrow \\ p & \rho[v \mapsto \mathcal{E}[\![e]\!]\rho M] & cs & uid & M & A & s \end{array}$$

$$\begin{array}{lllllll} \mathit{Assign}\ \mathit{type}\,[a]\ e\ p & \rho & cs & uid & M & A & s \\ \Longrightarrow \\ p & \rho & cs & uid & M' & A & s \end{array}$$
where $M' = store_{type}(M, \mathcal{E}[\![a]\!]\rho M, \mathcal{E}[\![e]\!]\rho M)$.

Conditional branch is straightforward:

$$\begin{array}{lllllll} \mathit{Branch}\ c\ p_t\ p_f & \rho & cs & uid & M & A & s \\ \Longrightarrow \\ \textbf{if } \mathcal{E}[\![c]\!]\rho M \textbf{ then } p_t \textbf{ else } p_f & \rho & cs & uid & M & A & s \end{array}$$

A call pushes a new activation. (Parameters will have already been placed in A by a *CopyOut* node.) The continuation bundle is saved on the stack, because the callee, not the caller, determines what is executed after the call. Each activation record must have a unique *uid*.

$$\begin{array}{lllllll} \mathit{Call}\ e_f\ \kappa & \rho & cs & uid & M & A & s \\ \Longrightarrow \\ p_f & \perp & \emptyset & u & M & A & (\kappa, \rho, cs, uid, s) \end{array}$$
where $\mathcal{E}[\![e_f]\!]\rho M = \mathit{Code}\ p_f$, and u is a fresh, unique identifier.

Tail calls are simpler; the appropriate continuation bundle is already on the stack.

$$\begin{array}{lllllll} \mathit{Jump}\ e_f & \rho & cs & uid & M & A & s \\ \Longrightarrow \\ p_f & \perp & \emptyset & u & M & A & s \end{array}$$
where $\mathcal{E}[\![e_f]\!]\rho M = \mathit{Code}\ p_f$, and u is a fresh, unique identifier.

When the machine tries to `cut to` a continuation belonging to an activation different from the current one ($uid'' \neq uid'$), it removes a frame from the stack and tries again. A real implementation cuts the stack in constant time, but the abstract machine removes the activations one at a time, so it can go wrong if a suspended call does not have an `also aborts` annotation.

$$\begin{array}{lllllll} \mathit{CutTo}\ e_c & \rho & cs & uid & M & A & (\kappa, \rho', cs', uid', s') \\ \Longrightarrow \\ \mathit{CutTo}\ e_c & \rho & cs & uid & M & A & s' \end{array}$$
where $\mathcal{E}[\![e_c]\!]\rho M = \mathit{Cont}(p'', uid'')$
$uid'' \neq uid'$
$\kappa = (\vec{p}_r{}', \vec{p}_u{}', \vec{p}_c{}', \mathit{True})$

When the machine finds the right activation ($uid'' = uid'$), it checks that the call site has an `also cuts to` annotation, and it transfers control to the appropriate continuation. The *uid* check ensures that the machine never invokes a dead continuation; a program that tries to do so goes wrong. As discussed in Section 4.2, `cut to` does not restore values stored in callee-saves registers; we model this behavior by removing them from the saved environment ρ'.

$$\begin{array}{lllllll} \mathit{CutTo}\ e_c & \rho & cs & uid & M & A & (\kappa, \rho', cs', uid', s') \\ \Longrightarrow \\ p'' & \rho' \setminus cs' & \emptyset & uid' & M & A & s' \end{array}$$
where $\mathcal{E}[\![e_c]\!]\rho M = \mathit{Cont}(p'', uid'')$
$uid'' = uid'$
$\kappa = (\vec{p}_r{}', \vec{p}_u{}', \vec{p}_c{}', \mathit{abort})$
$p'' \in \vec{p}_c{}'$

The *Yield* node models execution in the run-time system. Unlike the other rules, the rules for *Yield* do not fully specify which transitions take place; that is, there are states in which more than one transition is permitted. This underspecification allows the run-time system to implement a variety of different high-level exception semantics while still respecting the single C-- semantics.

The run-time system may unwind the stack if the suspended procedure has an `also aborts` annotation:

$$\begin{array}{llllll} \mathit{Yield} & \rho & cs & uid & M & A & (\kappa, \rho', cs', uid', s') \\ \Longrightarrow \\ \mathit{Yield} & \rho & cs & uid & M & A & s' \end{array}$$
where $\kappa = (\vec{p}_r{}', \vec{p}_u{}', \vec{p}_c{}', \mathit{True})$.

The run-time system may change memory, then transfer control to the normal return continuation, or to a continuation listed in an `also returns to` or `also unwinds to` annotation. This transition restores callee-saves registers.

$$\begin{array}{lllllll} \mathit{Yield} & \rho & cs & uid & M & A & (\kappa, \rho', cs', uid', s') \\ \Longrightarrow \\ p' & \rho' & cs' & uid' & M' & A' & s' \end{array}$$
where $\kappa = (\vec{p}_r{}', \vec{p}_u{}', \vec{p}_c{}', \mathit{abort})$, $p' \in \vec{p}_r{}' \cup \vec{p}_u{}'$, and A' is the right length.

The run-time system passes parameters A' to the continuation p'; the values of these parameters are unspecified, but there must be exactly as many parameters as p' expects.

Entry $\vec{k}$ *p*	**def** $\widehat{M}$. For each $v \in \vec{k}$, **def** v. **def** $A[i], 1 \le i \le N$, where N is the number of parameters of the procedure.
Exit j n	**use** $\widehat{M}$. **use** $A[i], 1 \le i \le N$, where N is the number of results of the procedure.
CopyIn $\vec{v}$ *p*	For each $i \in 1..\lvert\vec{v}\rvert$, $\vec{v}\,[i] := A[i]$.
CopyOut $\vec{e}$ *p*	For each $i \in 1..\lvert\vec{e}\rvert$, $A[i] := \vec{e}\,[i]$.
CalleeSaves cs p	No effect on dataflow.
Assign v e p	For each $v' \in fv(e)$, **use** v'. Then **def** v.
Assign type`[`*a*`]` *e p*	For each $v' \in fv(a) \cup fv(e)$, **use** v'. Then **def** $\widehat{M}$.
Branch c p_t p_f	For each $v \in fv(c)$, **use** v.
Call e_f $(\vec{p_r}, \vec{p_u}, \vec{p_c}, abort)$	For each $v \in fv(e_f)$, **use** v. **use** $\widehat{M}$. **def** $\widehat{M}$. **use** $A[i], 1 \le i \le N$, where N is the number of parameters to the call. Along the edge to $\vec{p_r}[j]$ **def** $A[i], 1 \le i \le N$, where N is the number of parameters to the continuation $\vec{p_r}[j]$. $\vec{p_u}[j]$ **def** $A[i], 1 \le i \le N$, where N is the number of parameters to the continuation $\vec{p_u}[j]$. $\vec{p_c}[j]$ **def** $A[i], 1 \le i \le N$, where N is the number of parameters to the continuation $\vec{p_c}[j]$. For each v that could be in cs when the code is executed, **kill** v. If *abort* is *True*, place **use** $A[i], 1 \le i \le N$, where N is the number of results of the procedure, along the edge to the exit node.
Jump e_f	For each $v \in e_f$, **use** v. **use** $\widehat{M}$. **use** $A[i], 1 \le i \le N$, where N is the number of parameters to the jump.
CutTo e_c	For each $v \in e_c$, **use** v. **use** $\widehat{M}$. **use** $A[i], 1 \le i \le N$, where N is the number of parameters to the cut to.
Yield	Not in any optimized procedure.

$fv(e)$ is the free variables of e, possibly including the variable $\widehat{M}$, which represents memory.

Table 3: Dataflow rules for Abstract C--.

The run-time system may also transfer control to a continuation listed in an `also cuts to` annotation, without restoring callee-saves registers.

$$\begin{array}{llllll} \textit{Yield} & \rho & cs & uid & M & A & (\kappa, \rho', cs', uid', s') \\ \Longrightarrow \\ p'' & \rho' \setminus cs' & \emptyset & uid' & M' & A' & s' \end{array}$$

where $\kappa = (\vec{p_r}', \vec{p_u}', \vec{p_c}', abort)$, $p'' \in \vec{p_c}'$, and A' is the right length.

Finally, the run-time system, in the form of the garbage collector, may read and write not only the values in memory, but also the *live* values stored in any ρ anywhere on the stack of the abstract machine. This possibility is not expressed by our formal semantics; to do so would require an even more complicated abstract machine, which would record the set of live variables at each call site.

5.3 Translating C-- to Abstract C--

Translation of everything except continuations, calls, jumps, and cuts should be obvious. To translate a continuation, create a *CopyIn* node naming the parameters of the continuation, and whose successor is the statement following the continuation. Associate this node with the continuation. To translate a call, create a *CopyOut* node that puts the values of the parameters in the value-passing area, and the successor of which is a *Call* node containing an expression designating the procedure to be called. The *Call* node's continuation bundle is computed from the `also` annotations as described above. If the call returns values, the normal continuation should be a *CopyIn* node binding the return values to the variables on the left-hand side of a call. Jumps and cuts are translated similarly, with the simplification that they never return.

5.4 Implementing the abstract machine

The C-- abstract machine is designed to be very like a real machine, hiding only the details of the registers, the calling convention, and the instruction set. The control p corresponds to the program counter. A local environment ρ corresponds to parameters and local variables that are stored in registers or in the activation record; the set cs identifies the variables that are stored in callee-saves registers. The *CalleeSaves* node, which changes cs, is implemented by instructions that move values into or out of callee-saves registers. Such instructions include spills, reloads, and register shuffles. The translation of C-- to Abstract C-- does not use callee-saves registers or the *CalleeSaves* node, but by including *CalleeSaves* node in Abstract C--, we enable Abstract C-- to represent the results of a code improvement that puts values in callee-saves registers. Of course, such code improvements much take into account control flow along `also cuts to` edges; such flow destroys values stored in callee-saves registers.

The value-passing area A is an abstraction representing those registers that are set aside for passing values and results, as well as overflow areas that may be reserved on the stack. This abstraction may have different concrete representations at different call sites. C-- supports multiple named calling conventions, and each calling convention may dictate a different representation of A. We also intend that C-- optimizers be free to choose customized value-passing mechanisms when possible, e.g., for passing parameters to procedures all of whose call sites are known.

Finally, the abstract stack corresponds to the machine's stack of activation records.[7] On the stack, a continuation bundle can be represented just by its program counter PC, which in general points to a branch table (Section 4.2). If the run-time system needs to find $\vec{p}_u$, it can use PC to look them up in a table.

The *uid* exists only to ensure that a program which tries to use a dead continuation goes wrong. Because using a dead continuation is an *unchecked* run-time error, the *uid* need not be represented an implementation. It may nevertheless be useful to represent it explicitly in order to debug front-end compilers that generate code which goes wrong.

Values of the form $Bits_k\ n$ are, of course, the basic values of machines. A value of the form $Code\ p$ is represented by a pointer to the instructions for p. There are several ways to implement a continuation value $Cont(p, u)$. One possible implementation is to allocate two words in the current activation record, and to represent $Cont(p, u)$ as a pointer to this pair. With a bit more cleverness, it may be possible to allocate only the program counter in the current activation record, and use the pointer to that location as the initial stack pointer for the continuation. In either case, some continuation values will need to encapsulate locations in which to store parameters that cannot fit in registers. This information, too, can be made implicit.

6. OPTIMIZING C-- PROGRAMS

Table 3 gives rules for adding dataflow information to a C-- procedure, in terms of definitions, uses, copies, and kills.[8] This information is enough to enable standard optimizations like common-subexpression elimination, partial-redundancy elimination, constant propagation, copy propagation, dead-code elimination, code motion, etc. The optimizer can perform all the usual rearrangements, provided it respects the dataflow and it doesn't insert code after *Exit*, *Jump*, *CutTo*, or the *abort* part of a continuation bundle.

Figure 5 shows an example C-- procedure, and Figure 6 shows its translation into Abstract C-- and the dataflow information attached to the Abstract C--. The dataflow information is expressed as a static single-assignment (SSA) numbering of the variables (Alpern, Wegman, and Zadeck 1988; Rosen, Wegman, and Zadeck 1988; Appel 1998). The elements of the A array may be mapped onto different hardware registers depending on which nodes define and use them and what the conventions for passing parameters and return values are. For example, in Figure 6, variables $A[1]_1$ and $A[1]_2$ should be mapped to the register holding the first parameter to a call, but variables $A[1]_3$, $A[1]_5$, and $A[1]_7$ should be mapped to the register holding the first result from a call.

[7] We do not intend that C-- itself be compiled using continuation-passing style, although of course C-- can easily represent high-level programs that are compiled using continuation-passing style. A front end would perform a CPS transformation and build explicit closures to represent continuations. The code in these closures would be represented by C-- procedures, to which control would be transferred using `jump`. The weak C-- continuations are useful only for nonlocal exits, not for representing first-class continuations.

[8] The operation **kill** v may be defined as assigning a bogus value, e.g., $v := \bot_{wrong}$, and any computation that depends on $\bot_{wrong}$ may be defined to have gone wrong.

```
f( bits32 a ) {
  bits32 b, c, d;

  b = a;
  c = a;
  b, c = g( c )
    also unwinds to k;
  c = b + c + a;
  return( c );

 continuation k( d ):
  return( b + d );
}
```

Figure 5: Example procedure.

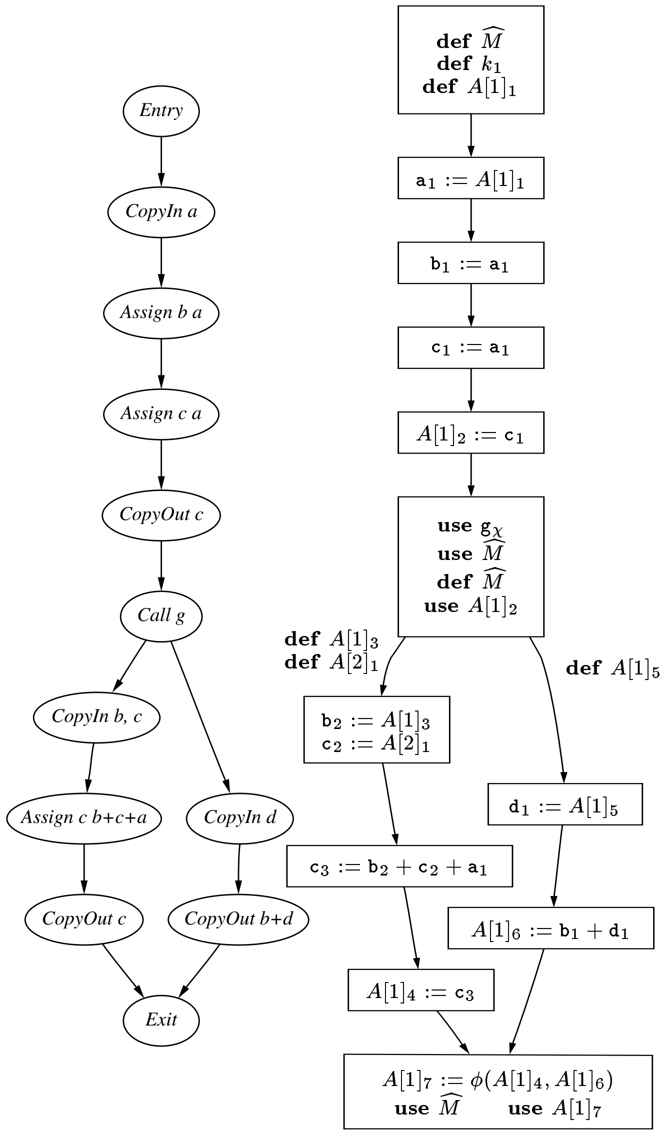

Figure 6: The example procedure's translation in Abstract C--, and its dataflow graph in SSA form.

7. RELATED WORK

This paper discusses the low-level *implementation* of exceptions; Drew and Gough (1994) complements this paper by presenting a taxonomy of high-level *designs* of exception-handling models.

Chase (1994a) and (1994b) provide helpful, clear explanations of the techniques required to implement both synchronous and asynchronous exceptions. Liskov and Snyder (1979) discusses both the programming methodology in which exceptions should be used and the efficiency of their implementation.

Drew, Gough, and Ledermann (1995) discusses implementation of the stack-unwinding technique, in which there is zero dynamic overhead to enter the scope of an exception handler. Their register allocator is unaware of the control-flow edges from call sites to exception handlers, so any variable that is mentioned in a body and a handler must be allocated on the stack. C--'s `also unwinds to` annotation should enable optimizers to avoid this performance penalty.

Hennessy (1981) discusses errors that can occur if the optimizer does not know about the additional control-flow edges introduced by exceptional termination. It presents dataflow equations that a front-end compiler can use to compute the annotations it must place at each C-- call site. It also discusses interprocedural analyses that compute potential effects on global variables. Such analyses could be accommodated in C-- by splitting $\widehat{M}$. This scheme would require additional annotations for stores, fetches, calls, jumps, and invocations.

The intermediate form used in the Vortex compiler (Dean *et al.* 1996) includes explicit control flow edges that represent the effects of exceptions. These edges enable the rest of the Vortex optimizer to work correctly without having to treat exceptions as a special case. It is hard to find out how other real optimizing compilers deal with exceptions, but we believe that most either make pessimistic assumptions or implement *ad hoc* rules that are tightly bound to the particular exception model of the language being compiled. In either case, all the standard optimizations must be reconsidered in light of the particular semantics chosen for exceptions. The approach we advocate, which is used in Vortex, allows well-developed optimization technology to be applied to a program that uses exceptions. Appel (1992) does something similar in a functional setting—by identifying exception handlers with continuations, Appel's compiler can simply use the existing body of optimizations known to be supported by continuation-passing style. (This style could be expressed in C-- in a manner much like that of the Appendix, except that `jump` would be used instead of `cut to`.)

Bruggeman, Waddell, and Dybvig (1996) introduces "one-shot continuations," which are like general Scheme continuations except that they can be invoked at most once. One-shot continuations would certainly suffice to implement C-- continuations, but the implementation is part of the Chez Scheme system, and it is not immediately obvious whether the tradeoffs that are good for Scheme would also be good for C--. The results on segmented stacks, at least, appear highly relevant to a concurrent version of C--.

Acknowledgments

Richard Black, Kent Dybvig, Michael Ernst, Lal George, Thomas Johnsson, Xavier Leroy, Greg Morrisett, Nikhil, John Reppy, Olin Shivers, David Watt, and the referees for PLDI'99 and PLDI 2000 have all given valuable feedback.

This work has been supported in part by NSF grants CCR-9733974 and ASC-9612756, by DARPA Contract MDA904-97-C-0247, and by a generous gift from Microsoft.

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. 1992. *Compiling with Continuations.* Cambridge: Cambridge University Press.

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

Atkinson, Russell R., Barbara H. Liskov, and Robert W. Scheifler. 1978 (December). Aspects of implementing CLU. In *Proceedings of the ACM 1978 Annual Conference*, pages 123–129. ACM.

Bruggeman, Carl, Oscar Waddell, and R. Kent Dybvig. 1996 (May). Representing control in the presence of one-shot continuations. *ACM SIGPLAN '96 Conference on Programming Language Design and Implementation, in SIGPLAN Notices*, 31(5):99–107.

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.

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

Dean, Jeffrey, Greg DeFouw, David Grove, Vassily Litvinov, and Craig Chambers. 1996 (October). Vortex: An optimizing compiler for object-oriented languages. *OOPSLA '96 Conference Proceedings,* in *SIGPLAN Notices*, 31(10):83–100.

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

Drew, Steven J., K. John Gough, and J. Ledermann. 1995. Implementing zero overhead exception handling. Technical Report 95-12, Faculty of Information Technology, Queensland U. of Technology, Brisbane, Australia. See `http://www.dstc.qut.edu.au/~gough/zeroex.ps`.

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.

Hsieh, Cheng-Hsueh A., John C. Gyllenhaal, and Wen-mei W. Hwu. 1996 (December). Java bytecode to native code translation: the Caffeine prototype and preliminary results. In *Proceedings of the 29th annual IEEE/ ACM International Symposium on Microarchitecture.*

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

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

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.

Peyton Jones, Simon L., Dino Oliva, and T. Nordin. 1997. C--: A portable assembly language. In *Proceedings of the 1997 Workshop on Implementing Functional Languages*, Vol. 1467 of *LNCS*, pages 1–19. Springer Verlag.

Peyton Jones, Simon L. and Norman Ramsey. 1998 (August). Machine-independent support for garbage collection, debugging, exception handling, and concurrency (draft). Technical Report CS-98-19, Department of Computer Science, University of Virginia. See `http://www.eecs.harvard.edu/~nr/pubs/c--rti-abstract.html`.

Peyton Jones, Simon L., Norman Ramsey, and Fermin Reig. 1999 (September). C--: a portable assembly language that supports garbage collection. In Nadathur, G, editor, *International Conference on Principles and Practice of Declarative Programming*, number 1702 in Lecture Notes in Computer Science, pages 1–28, Berlin.

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.

APPENDIX

A. IMPLEMENTING EXCEPTIONS

We believe that C-- can support a rich variety of source-language exception semantics. Demonstrations would provide supporting evidence, but unfortunately any demonstration necessarily fixes both the semantics of the source language and the cost model used in the implementation. To save space, we sketch only two potential exception dispatchers. Both dispatchers implement Modula-3 exceptions, but they use two different cost models.

A.1 Modula-3 exceptions with zero normal-case overhead

Figure 7 shows a fragment from a game-playing program written in 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 `BadMove` 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`.

Using an implementation based on run-time stack unwinding, this code might be translated into the C-- procedure shown in Figure 8.

```
PROCEDURE TryAMove() =
BEGIN
  TRY
    makeMove(getMove(player));
    next := (next + 1) MOD NUMBER(players);
  EXCEPT
  | BadMove(why) => player.badmove(why);
  | NoMoreTiles  => player.badmove("too few tiles");
  END;
  INC(movesTried);
END TryAMove;
```

Figure 7: Example Modula-3 procedure.

```
TryAMove() {
    bits32 s, t;

    t = getMove(player) also unwinds to k1, k2
                        also aborts;
    makeMove(t)         also unwinds to k1, k2
                        also aborts;
    t = bits32[players];       /* load size of array
                                  from its descriptor */
    next = (next + 1) mod t;

  finish:
    movesTried = movesTried + 1;
    return;

  continuation k1( s ):
    t = bits32[bits32[player]+12]; /* load address of
                                      badmove method */
    t( s );
    goto finish;

  continuation k2:
    t = bits32[bits32[player]+12]; /* load address of
                                      badmove method */
    t( "Not enough tiles" );
    goto finish;
}
```

Figure 8: C-- implementation of Modula-3 `TryAMove`, using run-time stack unwinding.

To see how exception dispatch works, let us suppose that `getMove` terminates normally, but `makeMove` discovers that the move cannot be made because it goes off the board. `makeMove` would contain the Modula-3 statement

```
RAISE BadMove("off board");
```

which might be translated into a `yield` to awaken the front end runtime and request exception-dispatching service. The details of the particular exception would be pushed onto a global "exception stack."

```
push_exn_info(Exn_BadMove, "off board");
yield( EXCEPTION );
```

The front-end runtime would invoke the exception dispatcher, a simplified version of which appears in Figure 9. The dispatcher would get the exception information, then call `FirstActivation(tcb, &a)` to get the activation handle for the topmost activation on the stack. Next it would map the activation handle to a statically allocated *exception descriptor* for `TryAMove`; mechanisms for implementing this mapping are discussed in Peyton Jones and Ramsey (1998).

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

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

  pop_exn_info(&exn_tag, &arg);
  FirstActivation(tcb, &a);
  for (;;) {
    struct exn_descriptor *d;
    d = ...a... ; /* Map activation to exn
                     descriptor, somehow */
    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 */
          SetUnwindCont(tcb,
                      d->handlers[i].cont_num);
                             /* choose handler */
          if (d->handlers[i].takes_arg) {
            /* exn expects value */
            void **result = FindContParam(tcb, 0);
            *result = arg;      /* Assign result */
          }
          return;
        }
    }
    if (!NextActivation(&a))
      abort();   /* unhandled exception: dump core */
  }
}
```

Figure 9: A simplified exception dispatcher for Modula-3, written in C.

If the exception raised were not handled by any of the handlers in the descriptor, the dispatcher would then call `NextActivation(&a)` to get the next frame. Eventually it would find the activation for `TryAMove`, whose exception descriptor states that continuation 0 handles the Modula-3 exception `BadMove`. (For purposes of `SetUnwindCont`, we number continuations, starting at zero, in the order in which they appear in the `also unwinds to` annotation for the call site at which the activation is suspended.) The dispatcher would then use `SetActivation` to establish the activation to resume and `SetUnwindCont` to cause resumption at the proper continuation. Finally it would use `FindContParam(tcb, 0)` to find the location in which to put the argument to the handler. (Continuation parameters are also numbered starting at zero.)

As recommended in the Modula-3 manual, this implementation requires zero dynamic overhead for entering the scope of an exception handler, but the cost of dispatching an exception may be considerable. 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.

```
register bits32 exn_top;
        /* top of exn stack */

TryAMove() {
    bits32 t, exn_tag, arg, k1;
    exn_top += sizeof(k);
        /* put k on the dynamic exception stack */
    bits32[exn_top] = k;
    t = getMove(player) also cuts to k;
    makeMove(t)         also cuts to k;
    t = bits32[players];
        /* load size of array from its descriptor */
    next = (next + 1) mod t;
    exn_top -= sizeof(k)
        /* leave TRY-EXCEPT-END */

  finish:
    movesTried = movesTried + 1;
    return;

  continuation k (exn_tag, arg):
    if (exn_tag == BadMove) {
      t = bits32[bits32[player]+12];
        /* load address of badmove method */
      t( arg );
      goto finish;
    } else if (exn_tag == NoMoreTiles) {
      t = bits32[bits32[player]+12];
        /* load address of badmove method */
      t( "Not enough tiles" );
      goto finish;
    } else {
      k1 = bits32[exn_top];
      exn_top -= sizeof(k1);
      cut to k1(exn_tag, arg);
    }
}
```

Figure 10: C-- implementation of Modula-3 `TryAMove`, using stack cutting.

A.2 Modula-3 exceptions in constant time

Some compilers use a different implementation trade-off. A small overhead is added to every `TRY-EXCEPT-END`, but in exchange, exception dispatch is very efficient—typically a few instructions. The high-level language maintains a stack of handlers, perhaps pointed to by a register. Every exception is dispatched 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.

Using this style of implementation, the procedure `TryAMove` from Figure 7 could be compiled into the C-- code shown in Figure 10. (This example assumes that the machine's native data-pointer type is `bits32`.) The code to raise an exception:

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

would be compiled into this C--:

```
k = bits32[exn_top];    /* fetch current handler
                           from stack */
exn_top -= sizeof(k);   /* pop stack */
invoke k(exn, val);     /* invoke the handler */
```

There is no equivalent to the exception dispatcher in Figure 9; instead, the propagation of exceptions is implemented in the handler itself.