C--: A Portable Assembly Language

Simon Peyton Jones[1], Thomas Nordin[2], and Dino Oliva[2]

[1] Department of Computing Science, University of Glasgow
[2] Pacific Software Research Centre, Oregon Graduate Institute

Abstract. Of late it has become very common for research compilers to emit C as their target code, relying on a C compiler to generate machine code. In effect, C is being used as a portable compiler target language. It offers a simple and effective way of avoiding the need to re-implement effective register allocation, instruction selection, and instruction scheduling, and so on, all for a variety of target architectures.

The trouble is that C was designed as a *programming* language not as a compiler target language, and is not very suitable for the latter purpose. The obvious thing to do is to define a language that *is* designed as a portable target language.

This paper describes C--, a portable compiler target language, or assembler. C-- has to strike a balance between being high-level enough to allow the back end a fair crack of the whip, while being low level enough to give the front end the control it needs. It is not clear that a path exists between these two rocks; the ghost of UNCOL lurks ominously in the shadows [6]. Yet the increasing popularity of C as a compiler target language (despite its unsuitability) suggests strong demand, and provides an existence proof that something useful can be done.

This paper appears in the Proceedings of the 1997 Workshop on Implementing Functional Languages, St Andrews, ed C Clack, Springer Verlag LNCS, 1998.

1 Introduction

The author of a new compiler often wants to generate good code, but does not want to duplicate the effort involved in writing a good code generator. One approach is to use C as a portable assembler, relying on a collection of tricks and non-standard extensions to persuade C to generate the sort of code they want. This approach has become quite common among compilers for functional and logic languages [8, 19, 11, 21, 3], including our own compiler for Haskell, the Glasgow Haskell Compiler.

In the light of our experience we have become more and more dissatisfied with using C as a compiler target language. (That is not a criticism of C — it was not designed for that purpose.) For example, a particular difficulty with using C for a garbage-collected language is that it is difficult for the garbage collector to find pointers that are manipulated by C-compiled code. Another is the lack of unrestricted jumps. We elaborate on these difficulties in Section 3.

A possible way to resolve these problems is to design a language that is specifically intended as a compiler target language for garbage-collected languages. This paper sketches the design for just such a language, `C--`. Despite its name `C--` is by no means a strict subset of C. The name arose from the above-noted popularity of C as a compiler target language, suggesting that a good design approach would be to delete as many features as possible from C, and add as few new features as possible. Whether `C--` is sufficiently better than C to justify the switching costs is an open question — but it is a question that we can only answer by debating a particular concrete design.

The paper gives only an informal overview of `C--`, concentrating on design choices, rather than giving a complete definition.

2 Goals and non-goals

The goals of `C--` are these:

`C--` **is a portable assembler, intended particularly for garbage-collected source languages**, such as Prolog, Lisp, Smalltalk, ML, Erlang, and Haskell. However, `C--` does not *embody* a particular garbage collector. The garbage-collection field is too active, and the storage management loads imposed by different languages too diverse, to make the collector part of the `C--` implementation. Nor does `C--` force the use of a conservative garbage collector (Section 3.2). Instead, it makes enough information available to support an accurate garbage collector.

`C--` **generates high-performance code.** A conventional assembler does not guarantee good code, but by definition it does not stand in the way of the highest possible performance — it simply exposes the bare machine. That is not true of programming languages in general: the more a language hides the machine behind tractable and portable abstractions, the harder it is to generate really fast code. `C--` sits near the bare-machine end of the spectrum. It is designed to gain a reasonable degree of architecture independence for an extremely modest cost in performance.

`C--` **exploits existing code generators.** The *raison d'etre* of `C--` is the desire to exploit the tremendous amount of research and implementation has been done in code generation technology. The tasks that should be largely or completely done by the code generator are:

- Register allocation, both local and inter-procedural.
- Instruction selection.
- Instruction scheduling.
- Jump elimination and other local optimisations.

In particular, `C--` should support the abstraction of an infinite number of named "registers" — usually called "local variables" in a programming-language context — rather than requiring the front end to map its values onto a finite set (no matter how large). For example, the front end should not have to worry about re-using a register for two different purposes; it can just use two different local variables.

`C--` **is independent of any particular code generator.** Despite all the work on code generation, actual implementations of this technology are surprisingly inaccessible to compiler writers. Usually the back-end implementation is presented in the form of some data types and procedures for building them, rather than as a *language*. (`gcc`'s RTL, XIL [18], and ML-Risc [7] are examples.) This approach forces the back-end user to adopt the language of back-end provider. It also forces commitment to a particular back end.

`C--` is intended to be independent of any particular back end, by the simple expedient of being defined as a language with a concrete ASCII syntax, and a semantics independent of any particular implementation. The hope is that a relatively simple parser should suffice to impedance-match `C--` to a variety of different back ends.

`C--` **is inter-operable.** It is possible to call C from `C--` (and hence C++, COM, and so on), and for a `C--` procedure to be called by C. (Perhaps Pascal calling conventions should also be supported; we're not sure.)

On the other hand, when a `C--` procedure is calling, or jumping to, another `C--` procedure, there is no requirement that standard C parameter passing conventions should be used — it's up to the `C--` implementation.

`C--` **is largely independent of the target architecture.** That is, a `C--` program contains very little architecture-specific code. It would be ideal if all `C--` programs would run unchanged on every architecture. As we will see, that is a very hard goal to meet without sacrificing efficiency, so we accept that the front-end compiler may need to know a few facts about the target architecture (such as its word size).

`C--` **is human writable, and readable.** While most `C--` will be written by compilers, some will be written by human beings (such as fragments of run-time system code), and some will be read by compiler writers when debugging their compilers, so `C--` should not be impossibly inconvenient for human programmers.

There are some important non-goals too.

`C--` **is not a distribution format.** There are now many machine-independent distribution formats on the market, including `mcode` [14], ANDF [4], Java byte codes [9], and Omniware [15]. Their war-cry is complete machine-independence, safety, and compactness. They can still be executed rather quickly using just-in-time compilation. Some of these properties come at the price of having to adopt a higher-level execution model than one would want for a compiler target language.

`C--` is not a competitor in this market place. Complete machine independence, checkable safety, and compactness are not goals.

`C--` **is an assembler, not a full programming language.** While we expect programmers to write a little `C--` code, `C--` lacks many features you would expect to find in a language intended for large-scale programming. For example, it has virtually no type system (Section 7).

Apart from C, which we discuss next, the Java Virtual Machine byte codes are `C--`'s most obvious competitor. Why not compile into JVM, and rely on JVM compilers for execution speed? The difficulty is that the JVM is much too high level. To encode Haskell or ML or Prolog into the JVM can be done, but it requires heroic optimism to believe that the resulting performance will ever be good. For a start, these languages allocate like crazy, and Java implementations are not built for that load; and even if they were, every object carries too much baggage. The JVM is not a portable assembler; it is a compact representation for Java.

3 Why C is not suitable as a portable assembler

An obvious question is this: why not (continue to) use C as a portable assembler? Quite a few papers have appeared in the last few years describing ways of achieving this effect [8, 19, 11, 21, 3], but all are unsatisfactory in one way or another. More precisely, we can identify the following difficulties:

- Some source languages require the so-called *tail call optimisation*. This means that a procedure call whose result is returned to the caller of the current procedure is executed in the stack frame of the current procedure. In turn, this allows iteration to be implemented efficiently using recursion.

  C's generality makes it very difficult to implement the tail call optimisation, and no C compiler known to us does so across separately compiled modules. This makes it difficult to map source-language procedures onto C procedures (Section 3.1).
- A C compiler is at liberty to lay out its stack frames as it pleases. This makes it difficult for a garbage collector to find the live pointers. In fact, the use of C procedures more or less forces the use of a conservative garbage collector (Section 3.2).
- A C compiler has to be very conservative about the possibility of memory aliasing. This seriously limits the ability of the instruction scheduler to move loads earlier in the instruction stream, perhaps past preceding stores, where there is less chance of the load causing a stall. The front-end compiler often knows that aliasing cannot occur, but there is no way to convey this information to the compiler.
- C 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 (Section 5).

– Many language implementations require frequent global access to certain variables, such as the stack pointer(s), allocation pointer, environment pointer, and so on. To accommodate this, most implementations rely on non-standard `gcc` extensions to C that nail specified global variables into particular registers. The wide availability of `gcc` means that this is not a pressing problem.

3.1 Tail calls

The ability to make tail calls requires the ability to jump to (rather than call) an arbitrary address. In higher-order languages this address might be fetched from a data structure rather than statically known. In an assembler we take for granted the ability to branch arbitrarily, but not so in C.

Several ways of mapping tail calls onto C have become popular:

1. One can embed all of the program inside a single C procedure, using local labels and gotos for control transfer. This does not work with separate compilation, and some C compilers choke on multi-thousand-line procedures.
2. One can treat parameter-less C procedures as extended basic blocks, using a so-called "trampoline" to transfer control between them [20, 21, 11]. The idea is simple: to "jump" to the next basic block the C procedure *returns* the address of the next C procedure. A tiny top-level loop simply calls the returned address:

   ```
   while TRUE { addr = (*addr)(); }
   ```

   This portable trick avoids growing stack, at the cost of a call and return instead of a jump.
3. One can trampoline more sparingly, by calling the next extended basic block (rather than jumping to it), allowing the C stack to grow, and periodically resetting it when it gets too big [2].
4. A gruesome but effective alternative to a trampoline is to post-process the assembly code to remove procedure prologues and epilogues, and use `asm` statements to generate real jumps.
5. `gcc` provides "first-class labels" as a non-standard extension. At first sight these might seem to solve the problem, but there are significant difficulties in practice. Notably, `gcc` says that "totally unpredictable things will happen" if control is transferred to a computed label in a different function body. Not only that, but some separate mechanism must be used to transfer parameters from the jump site to the destination. With considerable care, the Mercury compiler does, nevertheless, use this technique [8].

The bottom line seems to be this. The lack of tail calls is not an insuperable obstacle to using C, especially if `gcc` is used, but to generate efficient code usually leads to a complex and fragile implementation that relies on un-specified aspects of the C compiler.

3.2 Garbage collection

Our particular interest is in source languages that require a garbage-collected heap. That goal places quite complex requirements on the code generation technology.

One way of classifying garbage collectors is as follows:

- A *conservative* collector treats any word that *looks* as if it is a pointer (e.g. it points into the heap) as if it *were* a pointer. It might actually be an integer that just happened to address the heap, but if so all that happens is that some heap structure is retained that is not actually needed. A conservative collector cannot, therefore, move live objects, because altering the apparent "pointer" to it might instead alter the value of an integer.
- An *accurate* collector accurately follows live pointers. It never treats an integer (say) as a pointer. An accurate collector therefore needs a lot more information than a conservative collector. Using C as a code generator is effectively incompatible with accurate garbage collection, because the C compiler may save heap pointers on the stack, using a layout known only to itself. The only way out is to avoid letting C ever save a live pointer *by never calling a C procedure*. Instead, one uses tail calls exclusively, saving all live pointers explicitly on a separate, explicitly-managed stack.

The main reason that accurate garbage collection is desirable is because it allows compaction. Compaction improves locality, eliminates fragmentation, and allows allocation from a single contiguous area rather than from a free list or lists. This in turn makes allocation much cheaper, amortises the cost of the heap-exhaustion check over multiple allocations (where these occur together), and reduces register pressure (because multiple allocations can all be addressed as offsets from the allocation pointer).

A second reason that accurate garbage collection is desirable is because it does not follow pointers that are dead, even though they are valid pointers. A conservative collector would follow a dead pointer and retain all the structure thereby accessible. Plugging such "space leaks" is sometimes crucially important [10, 1]. Without an accurate garbage collector, the mutator must therefore "zap" dead pointers by overwriting them with zero (or something). Frustratingly, these extra memory stores are usually redundant, since garbage collection probably will not strike before the zapped pointer dies.

However, accurate garbage collection also imposes mutator costs, to maintain the information required by the garbage collector to find all the pointers. Whether accurate garbage collection is cheaper than conservative collection depends on a host of factors, including the details of the implementation and the rate of allocation.

Our goal for `C--` is to *permit* but not *require* accurate collection.

4 Possible back ends

The whole point of `C--` is to take advantage of existing code-generation technology. What, then, are suitable candidates? The most plausible-looking back ends we have found so far are these:

- ML-Risc [7].
- The Very Portable Optimiser (VPO) [5].
- The `gcc` back end, from RTL onwards.

Some of these (e.g. VPO) have input languages whose *form* is architecture independent, but whose details vary from one architecture to another. The impedance-matcher, that reads `C--` and stuffs it into the code generator, would therefore need to be somewhat architecture-dependent too.

5 The main features of `C--`

The easiest way to get a flavour of `C--` is to look at an example program. Figure 1 gives three different ways of writing a procedure to compute the sum and product of the numbers $1..N$. From it we can see the following design features.

Procedures. `C--` provides ordinary procedures. The only unusual feature is that `C--` procedures may return multiple results. For example, all the `sp` procedures return two results, the sum and the product. The `return` statement takes zero or more parameters, just like a procedure call; and a call can assign to multiple results as can be seen in the recursive call to `sp1`. This ability to return multiple results is rather useful, and is easily implemented.

The number and type of the parameters in a procedure call must match precisely the number and type of parameters in the procedure definition; and similarly for returned results. Unlike C, procedures with a variable number of arguments are not supported.

Types. `C--` has a very weak type system whose sole purpose is to tell the code generator how big each value is, and what class of operations can be performed on it. (In particular, floating point values may well be kept in a different bank of registers.) Furthermore, the size of every type is explicit — for example, `word4` is a 4-byte quantity. We discuss the reasons for these choices in Section 7.

Tail calls. `C--` guarantees the tail call optimisation, even between separately compiled modules. The procedure `sp2_help`, for example, tail-calls itself to implement a simple loop with no stack growth. The procedure `sp2` tail-calls `sp2_help`, which returns directly to `sp2`'s caller. A tail call can be thought of as "a jump carrying parameters".

```
-- sp1, sp2, sp3 all compute the sum 1+2+...+n
--                         and the product 1*2*...*n

-- Ordinary recursion
sp1( word4 n ) {
  word4 s, p;
  if n == 1 {
      return( 1, 1 );
  } else {
      s, p = sp1( n-1 );
      return( s+n, p*n );
} }

-- Tail recursion
sp2( word4 n ) {
  jump sp2_help( n, 1, 1 );
}

sp2_help( word4 n, word4 s, word4 p ) {
  if n==1 {
      return( s, p );
  } else {
      jump sp2_help( n-1, s+n, p*n )
} }

-- Loops
sp3( word4 n ) {
      word4 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 C-- sum-and-product functions

Local variables. `C--` allows an arbitrary number of local variables to be declared. The expectation is that local variables are mapped to registers unless there are too many alive at one time to keep them all in registers at once. In `sp3`, for example, the local variables `s` and `p` are almost certainly register-allocated, and never even have stack slots reserved for them.

Labels. `C--` provides local labels and `goto`s (see `sp3`, for example). We think of labels and `goto`s simply as a textual description for the control-flow graph of a procedure body. A label is not in scope anywhere except in its own procedure body, nor is it a first class value. A label can be used only as the target of a `goto`, and only a label can be the target of a `goto`. For all other jumps, tail calls are used.

Conditionals. `C--` provides conditional statements, but unlike C there is no boolean type. (In C, `int` doubles as a boolean type.) Instead, conditionals syntactically include a comparison operation, such as "`==`", "`>`", "`>=`", and so on. Like C, `C--` also supports a `switch` statement. The only difference is that `C--` allows the programmer to specify that the scrutinised value is sure to take one of the specified values, and thus omit range checks.

6 Procedures

Most machine architectures (and hence assembler) support a jump instruction, and usually a call instruction. However, these instructions deal simply with control flow; it is up to the programmer to pass parameters in registers, or on the stack, or whatever. In contrast, `C--` supports parameterised procedures like most high-level languages. Why?

If `C--` had instead provided only an un-parameterised call/return mechanism, the programmer would have to pass parameters to the procedure through explicit registers, and return results the same way. So a call to `sp1` might look something like this, where `R1` and `R2` register names:

```
R1 = 12;       /* Load parameter into R1 */
call sp1;
/* Results returned in R1, R2 */
```

This approach has some serious disadvantages:

- Different code must be generated for machines with many registers than for machines with few registers. (Presumably, some parameters must be passed on the stack in the latter case.) This means that the front end must know how many registers there are, and generate a calling sequence based on this number. Matters are made more complicated by the fact that there are often two sorts of registers (integer and floating point) – this too must be exposed to the front end.
- If the front end is to pass parameters on the stack, responsibility for stack management must be split between the front end and the back end. That

is quite difficult to arrange. Is the front end or the back end responsible for saving live variables across a call? Is the front end or the back end responsible for managing the callee-saves registers? Does the call "instruction" push a return address on the stack or into a register? What if the target hardware's call instruction differs from the convention chosen for `C--`? And so on. We have found this question to be a real swamp. It is much cleaner for *either* the front end *or* the back end to have complete responsibility for the stack.

- If the mapping between parameters and registers is not explicit, then it may be possible for the code generator to do some inter-procedural register allocation and use a non-standard calling convention. It is premature for the front end to fix the exact calling convention.
- Finally, it seems inconsistent to have an infinite number of named "virtual registers" available as local variables, but have to pass parameters through a fixed set of real registers.

For these reasons, `C--` gives complete control of the stack to the back end, and provides parameterised procedures as primitive[3].

However, `C--`'s procedures are carefully restricted so that they can be called very efficiently:

- The types and order of parameters to a call completely determine the calling convention for a vanilla call. (After inter-procedural register allocation `C--` might decide to use non-vanilla calling conventions for some procedures, but that is its business.)
- The actual parameters to a call must match the formal parameters both in number and type. No checks are made. All the information needed to compile a vanilla call is apparent at the call site; the `C--` compiler need know nothing about the procedure that is called. There is no provision for passing a variable number of arguments.
- Procedure calls can only be written as separate statements. They cannot be embedded in expressions, like this:

  ```
  x = f(y) + 1;
  ```

  The reason for this restriction is partly that the expression notation makes no sense when `f` returns zero or more than one result, and partly that `C--` may not be able to work out the type of the procedure's result. Consider:

  ```
  g( f( x ) );
  ```

  This is illegal, and must instead be written:

  ```
  float8 r;
  ...
  r = f( x );
  ```

[3] Of course, a `C--` implementation is free not to use a control stack at all, provided it implements the language semantics; but we will find it convenient to speak as if there were a stack.

```
g( r );
```

Now the type of the intermediate is clear.

6.1 Parameterised returns

The `return` statement takes as arguments the values to return to the caller. In fact a `return` statement looks just like a call to a special procedure called "`return`".

At a call site, the number and type of the returned values must match precisely the actual values returned by `return`. For example, if `f` returns an `word4` and a `float8` then every call to `f` must look like:

```
r1, r2 = f( ... );
```

where `r2` is an lvalue of type `word4` and `r2` of type `float8`. It is *not* OK to say

```
f( ... );
```

and hope that the returned parameters are discarded.

6.2 Tail calls

A tail call is written

```
jump f( ...parameters... )
```

Here, we do not list the return arguments; they will be returned to the current procedure's caller.

No special properties are required of `f`, the destination of the tail call[4]. It does not need to take the same number or type of parameters as the current procedure (the tail call to `sp2_help` in `sp2` in Figure 1 illustrates this); it does not need to be defined in the same compilation unit; indeed, `f` might be dynamically computed, for example by being extracted from a heap data structure.

Tail calls are expected to be cheap. Apart from getting the parameters in the right registers, the cost should be about one jump instruction.

Every control path must end in a `jump` or a `return` statement. There is no implicit `return()` at the end of a procedure. For example, this is illegal:

```
f( word4 x ) {
      word4 y;
      y = x+x;
}
```

[4] This situation contrasts rather with C, where `gcc`'s manual says of the `-mtail-call` flag: "Do (or do not) make additional attempts (beyond those of the machine-independent portions of the compiler) to optimise tail-recursive calls into branches. You may not want to do this because the detection of cases where this is not valid is not totally complete."

7 Data types

`C--` has a very weak type system, recognising only the following types:

- `word1`, `word2`, `word4`, `word8`.
- `float4`, `float8`.

The suffix indicates the size of the type in bytes. This list embodies two major design choices that we discuss next: the paucity of types (Section 7.1), and the explicit size information (Section 7.2).

7.1 Minimal typing

The main reason that most programming languages have a type system is to detect programming errors, and this remains a valid objective even if the program is being generated by a front-end compiler. However, languages that use dynamic allocation are frequently going to say "I know that this pointer points to a structure of this shape", and there is no way the `C--` implementation is going to be able to verify such a claim. Its truth depends on the abstractions of the original source language, which lie nakedly exposed in its compiled form. Paradoxically, the lower level the language the more sophisticated the type system required to gain true type security, so adding a secure type system to an assembler is very much a research questions [17, 16].

So, like a conventional assembler, `C--` goes to the other extreme, abandoning any attempt at providing type security. Instead, types are used only to tell the code generator the information it absolutely has to know, namely:

- The kind of hardware resource needed to hold the value. In particular, floating point values are often held in different registers than other values.
- The size of the value.

For example, signed and unsigned numbers are not distinguished. Instead, like any other assembler, it is the *operations* that are typed. Thus, "`+`" adds signed integers, while "`+u`" adds unsigned integers, but the operands are simply of type `word1`, `word2` etc. (The size of the operation is, however, implicit in the operand type.)

Similarly, `C--` does not have a type corresponding to C's "`*`" pointer types. In C, `int*` is the type of a pointer to `int`. In `C--` this type is just `word4` (or `word8`). When doing memory loads or stores, the size of the value to be loaded or stored must therefore be given explicitly. For example, to increment the integer in memory location `foo` we would write:

```
word4[foo] = word4[foo] + 1;
```

The addressing mode `word4[foo]` is interpreted as a memory load or store instruction, depending on whether it appears on the left or right of an assignment.

7.2 Type sizes

Since different machines have different natural word sizes, it is tempting to suggest that C-- should abstract away from word-size issues. That way, an unchanged C-- program could run on a variety of machines. C does this: an `int` is 32 bits wide on some machines and 64 bits on others.

While this is somewhat attractive in a programming language, it seems less appropriate for an assembler. The front-end compiler may have to generate huge offset-calculation expressions. Suppose the front-end compiler is computing the offset of a field in a dynamically-allocated data structure containing two floats, a code pointer, and two integers. Since it does not know the actual size of any of these, it has to emit an offset expression looking something like this:

```
2*sizeof(float) + sizeof(codepointer) + 2*sizeof(int)
```

Apart from the inconvenience of generating such expressions (and implementing arithmetic over them in the front-end compiler) they produce a substantial increase in the size of the C-- program.

Another difficulty is that the front-end compiler cannot calculate the alignment of fields within a structure based on the alignment of the start of the structure.

One way of mitigating these problems would be to introduce `struct` types, as in C; offsets within them would then be generated by field selectors. However, this complicates the language, and in a Haskell or ML implementation there might have to be a separate `struct` declaration for each heap-allocated object in the program. Offset calculations using `sizeof` would still be required when several heap objects were allocated contiguously. And so on.

C-- instead takes a very simple, concrete approach: *each data type has a fixed language-specified size.* So, for example, `word4` is a 4-byte word, while `word8` is an 8-byte word.

This decision makes life easy for the C-- implementation, at the cost of having to tell the front-end compiler what C-- data types to use for the front end's various purposes. In practice this seems unlikely to cause difficulties, and simplifies some aspects (such as arithmetic on offsets).

8 Memory

8.1 Static allocation

C-- supports rather detailed control of static memory layout, in very much the way that ordinary assemblers do. A data block consists of a sequence of: labels, initialised data values, uninitialised arrays, and alignment directives. For example:

```
data {
  foo:  word4{10};      /* One word4 initialised to 10 */
```

```
        word4{1,2,3,4}; /* Four initialised word4's */
        word4[80];      /* Uninitialised array of 80 word4's */
  baz1:
  baz2: word1           /* An uninitialised byte */
  end:
}
```

Here `foo` is the address of the first `word4`, `baz1` and `baz2` are both the address of the `word1`, and `end` is the address of the byte after the `word1`.

All the labels have type `word4` *or* `word8`*, depending on the particular architecture.* On a Sparc, `foo`, `baz1`, `baz2` and `end` all have type `word4`. You should think of `foo` as a pointer, not as a memory location.

Unlike C, there is no implicit alignment nor padding. Furthermore, the address relationships between the data items in a data block is guaranteed; for example, `baz1` $=$ `foo` $+ 340$.

8.2 Dynamic allocation

The `C--` implementation has complete control over the system stack; for example, the system stack pointer is not visible to the `C--` programmer. However, sometimes the `C--` programmer may want to allocate chunks of memory (not virtual registers) on the system stack. For this, the `stack` declaration is provided:

```
f( word4 x ) {
  word4 y;
  stack {
      p : word4;
      q : float8;
      r : word4[30];
  }
  ...
```

Just as with static allocation, `p`, `q`, and `r` all have type `word4` (or `word8` on 64-bit architectures). Their address relationship is guaranteed, regardless of the direction in which the system stack grow; for example, `q` = p+4.

`stack` declarations can only occur inside procedure bodies. The block is preserved across `C--` calls, and released at a `return` *or a tail call.* (It is possible that one might want to allocate a block that survives tail calls, but we have not yet found a reasonable design that accommodates this.)

8.3 Alignment

Getting alignment right is very important. `C--` provides the following support.

- Simple alignment operations, `align`*i*, are provided for static memory allocation. For example, the top-level statements

```
foo:  word4;
      align8;
baz:  float8;
```

ensures that baz is the address of a 8-byte aligned 8-byte memory location. There is no implicit alignment at all.

– Straightforward memory loads and stores are assumed aligned to the size of the type to be loaded. For example, given the addressing mode `float8[ptr]`, `C--` will assume that `ptr` is 8-byte aligned.

Sometimes, however, memory accesses may be mis-aligned, or over-aligned:

– When storing an 8-byte float in a dynamically allocated object, it may be convenient to store it on a 4-byte boundary, the natural unit of allocation.

– Byte-coded interpreters most conveniently store immediate constants on byte boundaries.

– Sometimes one might make a byte load from pointer that is known to be 4-byte aligned, for example when testing the tag of a heap-allocated object. Word-oriented processors, such as the Alpha, can perform byte accesses much more efficiently if they are on a word boundary, so this information is really worth conveying to the code generator.

Coding up mis-aligned accesses in a language that does not support them directly is terribly inefficient (lots of byte loads and shifts). This is frustrating, because many processors support mis-aligned access reasonably well. For example, loading a 8-byte float with two 4-byte loads is easy on a Sparc and MIPS R3000; the Intel x86 and PowerPC support pretty much any unaligned access; and the Alpha has canned sequences using the `LDQ_U` and `EXTxx` instructions that do unaligned accesses reasonably efficiently.

For these reasons, `C--` also supports explicitly-flagged mis-aligned or over-aligned accesses.

– The load/store addressing mode is generalised to support an alignment assumption: *type*{*align*}[*ptr*]. For example:

  - `float8{align4}[ptr]` does a 8-byte load, but only assumes that `ptr` is aligned to a 4 byte boundary.
  - `word4{align1}[ptr]` does a 4-byte load from a byte pointer.
  - `word1{align4}[ptr]` does a 1-byte load from a pointer that is 4-byte aligned.

8.4 Endian-ness

There is no support varying endian-ness. (No language provides such support. At best, the type system prevents one reading a 4-byte integer one byte at a time, and hence discovering the endian-ness of the machine, but that would be inappropriate for an assembler.)

8.5 Aliasing

It is often the case that the programmer knows that two pointers cannot address the same location or, even stronger, no indexed load from one pointer will access the same location as an indexed load from the other.

The `noalias` directive supports this:

```
noalias x y;
```

is a directive that specifies that no memory load or store of the form *type*`[x` *op* *e*`]` can conflict with a load or store of similar form involving `y`. Here *op* is `+` or `-`.

9 Garbage collection

How should `C--` support garbage collection?

One possibility would be to offer a storage manager, including a garbage collector, as part of the `C--` language. The trouble is that every source language needs a different set of heap-allocated data types, and places a different load on the storage manager. Few language implementors would be happy with being committed to one particular storage manager. We certainly would not.

Another possibility would be to provide no support at all. The `C--` user would then have the choice of using a conservative collector, or of eschewing `C--` procedures altogether and instead using tail calls exclusively (thereby avoiding having `C--` store any pointers on the stack). This alternative has the attraction of clarity, and of technical feasibility, but then `C--` would be very little better than C.

The third possibility is to require `C--` to keep track of where the heap pointers are that it has squirreled away in its stack, and tell the garbage collector about them when asked. A possible concrete design is this. The garbage collector calls a C procedure `FindRoots( mark )`, passing a C procedure `mark`. `FindRoots` finds all the heap pointers in the `C--` stack, and calls `mark` on each of them. Which of the words stored in the stack are reported as heap pointers by `FindRoots`? The obvious suggestion is to have a new `C--` type, `gcptr`, which identifies such pointers.

This is easily said, but not quite so easily done. How might `C--` keep track of where the `gcptr` values are in the stack? The standard technique is to associate a stack descriptor (a bit pattern, for example) with each return address pushed on the stack. After all, the code at the return address "knows" the layout of the stack frame, and it is only a question of making this information explicitly available. One can associate the descriptor with the code either by placing it just before the code, or by having a hash table that maps the code address to the descriptor information.

The trouble with this approach is that it is somewhat specific to a particular form of garbage collection. For example, what if there is more than one kind of heap pointer that should be treated separately by the collector? Or, what if the pointer-hood of one procedure parameter can depend on the value of another, as is the case for the TIL compiler [22]. We are instead developing other ideas that provide much more general support for garbage collection, and also provide support for debuggers and exception handlers, using a single unified mechanism [13].

10 Other features of C--

There are several features of C-- that we have not touched on so far:

- Global registers.
- Arithmetic operations and conversions between data types.
- Interfacing to C.
- Separate compilation.

They are described in the language manual [12].

11 Status and future directions

C-- is at a very early stage. We have two implementations of a core of the language, one using VPO and one using ML-Risc. Each was built in a matter of weeks, rather than months, much of which was spent understanding the code generator rather than building the compiler. These implementations do not, however, cover the whole of C-- and are far from robust.

The support for tail calls raises an interesting question: does it make sense for the implementation to use callee-saves registers, as do most implementations of conventional languages? The trouble is that the callee-saves registers must be restored just before a tail call, and then perhaps immediately saved again by the destination procedure. Since each procedure decides independently which callee-saves registers it needs to save it is not at all obvious how to avoid these multiple saves of the same register. Perhaps callee-saves registers do not make sense if tail calls are common. Or perhaps some inter procedural analysis might help.

As modern architectures become increasingly limited by memory latency, one of C--'s biggest advantages should be its ability to provide detailed guidance about possible aliasing to the code generator, and thereby allow much more flexibility in instruction scheduling. We have not yet implemented or tested such facilities.

Many garbage-collected languages are also concurrent, so our next goal is to work out some minimal extensions to C-- to support concurrency. We have in mind that there may be thousands of very light-weight threads, each with a heap-allocated stack. This means that C-- cannot assume that the stack on which it is currently executing is of unbounded size; instead it must generate stack-overflow checks and take some appropriate action when the stack does overflow, supported by the language-specific runtime system.

Acknowledgements

This document has benefited from insightful suggestions from Chris Fraser, Dave Hanson, Fergus Henderson, Xavier Leroy, Norman Ramsay, John Reppy, and the IFL referees. We are very grateful to them.

References

1. AW Appel. *Compiling with continuations.* Cambridge University Press, 1992.
2. H Baker. Cons should not cons its arguments, Part II: Cheney on the MTA. *SIGPLAN Notices*, 30(9):17–20, Sept 1995.
3. JF Bartlett. SCHEME to C: a portable Scheme-to-C compiler. Technical Report RR 89/1, DEC WRL, 1989.
4. ME Benitez, P Chan, JW Davidson, AM Holler, S Meloy, and V Santhanam. ANDF: Finally an UNCOL after 30 years. Technical Report TR-91-05, University of Virginia, Department of Computer Science, Charlottesville, VA, March 1989.
5. ME Benitez and JW Davidson. The advantages of machine-Dependent global optimization. In *International Conference on Programming Languages and Architectures (PLSA'94)*, pages 105–123, 1994.
6. ME Conway. Proposal for an UNCOL. *Communications of the ACM*, 1(10):5–8, October 1958.
7. L George. MLRISC: Customizable and Reusable Code Generators. Technical report, Bell Laboratories, Murray Hill, 1997.
8. Fergus Henderson, Thomas Conway, and Zoltan Somogyi. Compiling logic programs to C using GNU C as a portable assembler. In *Postconference Workshop on Sequential Implementation Technologies for Logic Programming (ILPS'95)*, pages 1–15, Portland, Or, 1995.
9. H McGilton J Gosling. The Java Language Environment: a White Paper. Technical report, Sun Microsystems, 1996.
10. R Jones. Tail recursion without space leaks. *Journal of Functional Programming*, 2(1):73–80, Jan 1992.
11. Simon L. Peyton Jones. Implementing Lazy Functional Languages on Stock Hardware: The Spineless Tagless G-machine. *Journal of Functional Programming*, 2(2):127–202, April 1992.
12. SL Peyton Jones, T Nordin, and D Oliva. The `C--` manual. Technical report, Department of Computing Science, University of Glasgow, 1998.

13. SL Peyton Jones, N Ramsey, and JW Davidson. Portable support for garbage collection, debugging, and exceptions. Technical report, In preparation, Department of Computing Science, University of Glasgow, 1998.
14. Brian T Lewis, L Peter Deutsch, and Theodore C Goldstein. Clarity MCode: A retargetable intermediate representation for compilation. *ACM SIGPLAN Notices*, 30(3):119–128, March 1995.
15. S Lucco, O Sharp, and R Wahbe. Omniware: a universal substrate for Web programming. In *Fourth International World Wide Web Conference, Boston: The Web Revolution*, Dec 1995.
16. G Morrisett, K Crary, N Glew, and D Walker. Stack-based typed assembly language. In *Proc Types in Compilation, Osaka, Japan*, March 1998.
17. G Morrisett, D Walker, K Crary, and N Glew. From system f to typed assembly language. In *Proc 25th ACM Symposium on Principles of Programming Languages, San Diego*, Jan 1998.
18. Kevin O'Brien, Kathryn M. O'Brien, Martin Hopkins, Arvin Shepherd, and Ron Unrau. XIL and YIL: The intermediate languages of TOBEY. *ACM SIGPLAN Notices*, 30(3):71–82, March 1995.
19. M Pettersson. Simulating tail calls in C. Technical report, Department of Computer Science, Linkoping University, 1995.
20. GL Steele. Rabbit: a compiler for Scheme. Technical Report AI-TR-474, MIT Lab for Computer Science, 1978.
21. D Tarditi, A Acharya, and P Lee. No assembly required: compiling Standard ML to C. *ACM Letters on Programming Languages and Systems*, 1(2):161–177, 1992.
22. D Tarditi, G Morrisett, P Cheng, C Stone, R Harper, and P Lee. Til: A type-directed optimizing compiler for ml. In *Proc SIGPLAN Symposium on Programming Language Design and Implementation (PLDI'96), Philadelphia*. ACM, May 1996.