Foreign Language Interoperability and Annotations for Code Generation in C-- (DRAFT)

The title should be improved

Fermín J. Reig Galilea[*]
Department of Computing Science
University of Glasgow
reig@dcs.gla.ac.uk

April 24, 2001

Abstract

Keywords:

Problem with listings package:1)beginning of text 2) keywords

1 Introduction

Give a little bit of background about C-- ([JRR99, RPJ00]). Emphasis on the decoupling of the back end, since this is a recurring theme in other parts of the paper.

Contributions:

- C-- provides interoperability with C via `foreign` but C--'s stack walking interface is inadequate for situations where C-- interoperates with foreign code, or foreign runtime systems. We explain the limitations and propose an alternative. Implemented and tested in a C-- back end for OCaml[?].
- Monolithic compilers share information between different phases via constructs of the intermediate language(s), symbol tables, files (for cross-module optimizations), or other means. When the back end is decoupled from the rest of the compiler—as is the case in a C-- back end—the only way to communicate program properties to the back end is via C-- language constructs. At present C-- lacks the syntax to do this. We propose constructs (annotations) to convey several program properties to the back end. Some of them are implemented in cmmc, but the OCaml front end makes only limited use of them.

2 Stack Scanning and Unwinding (Background)

(General background, not specific to C--)

[*]Funded by Gobierno de Navarra.

Definition 1 Stack scanning. Traverse the stack of activation frames, inspecting and possibly modifying their contents.

Garbage collection The local variables of a procedure form most of the *root set* of pointers to heap-allocated objects.[1] Local variables are assigned to registers or frame locations by the compiler. The garbage collector scans the stack of activation frames looking for roots. Copying collectors[Jon96] also modify the frames.

Definition 2 Stack unwinding. Unroll (?) the stack of activation frames, possibly resuming execution at a previous point in the call chain.

Exception handling When an exception is raised, we unwind until the first activation where a handler is in scope and transfer control to the handler[RPJ00]. Depending on the language, other actions have to be executed as the stack is unwound. In C++ we need to invoke destructors for objects that go out of scope.

Add a reference for C++ EH and destructors

To unwind the stack we must know how to 1) navigate from an activation to the activation of its caller 2) decide whether a handler is in scope.

To scan the stack we need to know 1) how to position ourselves at the topmost activation 2) how to navigate to the the caller 3) the exact location in the frame (or registers) of roots (for accurate collectors only).

2.1 Garbage Collection

To inspect activation frames conservative collectors[BW88] need little or no support from the compiler: they scan the whole stack.

> *They do need to know about the stack limits, though. Q: How you tell the Boehm collector about this?*

Type-accurate and liveness-accurate collectors[ADM98]scan only those locations that are known to contain roots at the point where a procedure is *suspended for garbage collection.* A procedure is suspended when it makes a call, and is resumed when the callee returns. In multithreaded languages like Java, threads may be stopped for garbage collection at other points than calls[SLC99]. An accurate collector needs to know the liveness and exact location of roots (registers, frame slots).

ToDo:Briefly describe advantages of accurate collection.

The compiler can emit *frame descriptors* that contain root locations. It emits a descriptor for every point where a procedure can be suspended for garbage collection. The back end knows this information.

2.2 Exception Handling

Descriptors for exception handling tell whether a handler is in scope and, if so, where to transfer control to. They may contain other information, like what destructors have to be called. The front end (or the middle end) of the compiler knows this information.

In compilers for languages with GC and exceptions, descriptors (for call sites) can contain both GC and EH information. Or we can have separate descriptors.

> *Q: What is typical: unified or separate descriptors for GC and EH? This is relevant for `C--`, since the back end is decoupled.*

[1]Others are: global variables, foreign roots.

3 Garbage Collection and Exception Handling in C--

[JRR99, RPJ00] propose a C-- *run-time system* so that a garbage collector can scan the stack or an exception dispatcher can unwind it.

The following two procedures are provided:

Have a figure with procedure names and descriptions.

`cmm_first_activation`

- , to position ourselves at the topmost C-- activation.

`cmm_next_activation`

- , to move to the caller of the current activation.

I actually prefer the names cmm_topmost_activation and cmm_prev_activation. . .

This is the descriptor (other fields are not exported, since the GC does not need them).

```
struct cmm_gc_descriptor {
  unsigned root_count;
  unsigned root_offsets[1];
};
```

An accurate garbage collector would scan the stack like this:

```
a = cmm_first_activation()

for(;;) {
  descr = cmm_get_descriptor(a);

  /* Scan roots in this activation */
  for(i=0; i < descr->root_count; i++) {
    <... descr->root_offsets[i] ...>

  /* Stop if there are no more C-- activations */
  if(a = cmm_next_activation(a) == NULL) break;
} /* for */
```

(`cmm_get_descriptor` finds the descriptor corresponding to the current activation.)

3.1 Dealing with Language Interoperability

However, this interface does not suffice for all front end needs. In particular:

> There is an implicit assumption that the C-- stack is one single, contiguous region of memory. Several modern programming languages allow foreign calls and callbacks. A certain implementation may choose to use a single stack, and then we have interleaving chunks of C-- and foreign frames. For example OCaml implements it this way (How about Mercury? Others?).

Insert picture of a stack here.

When we are unwinding the C-- stack and we make a transition from a C-- stack section to a foreign stack section, the caller's frame is not a C-- frame, and we cannot unwind directly to it using `cmm_next_activation`. It is not clear what the behaviour of `cmm_next_activation` would be: maybe it gets stuck (crash), or maybe it signals that we have reached the bottom `C--` frame, when this is not true. It is up to the front end runtime to decide what is the correct thing to do: this is a policy decision and `C--` provides mechanisms. Two alternatives:

1. Skip the foreign stack chunk.[2]
2. Scan the foreign chunk via a call to a foreign runtime.

   It may even be the case that `C--` is the foreign code! (maybe `main` is Java, and via a FFI we call OCaml code —that happens to be compiled via `C--`): the "master" garbage collector starts on a non-`C--` stack chunk. In this case, there never was a call to `cmm_first_activation`! Nor to `cmm_yield`, for that matter; when the collector encounters the first foreign stack chunk, it lets the OCaml/`C--` collector take over for a while. At this point, all we have is a pointer to the `C--` frame in the call stack, and a return address to some `C--` code (that we can use to get a `C--` descriptor).

And similarly for exception handling (skip or foreign unwind). The point is that the front end decides what action is appropriate (`C--` does not know).

N.B. The focus of this discussion is on language interoperation where we have two separate garbage collectors or the exception handling mechanisms differ. GCJ and g++ share the same code generator and use the same EH mechanism, so they have an easier task. In particular the GCJ pages say:

> "You can throw a Java exception from C++ using the ordinary throw construct, and this exception can be caught by Java code. Similarly, you can catch an exception thrown from Java using the C++ catch construct.
>
> Note that currently you cannot mix C++ catches and Java catches in a single C++ translation unit. We do intend to fix this eventually."

Like GCJ, MS .net provides a unified mechanism for inter-language cooperation. *This could be discussed here, but maybe all of this can go to related work.*

3.2 Let the Front End Decide

Need a better section title.

`cmm_next_activation`
tries to hide too much. Proposal: let the front end have more control of the unwinding process.

Other than by `cmm_next_activation` failing, how do we know that we have reached the bottom activation of a `C--` stack section? I can think of four ways:

- The return address stored in the current frame is to a special callback routine in the (front end) runtime.
- The descriptor of the caller is somehow marked as special. OCaml does this: the descriptor contains a negative frame size.

[2]This is what OCaml does for garbage collection. Foreign calls and callbacks happen via runtime stubs. OCaml has an opportunity to do some bookkeeping at these points, and it records the beginning and end of the foreign stack chunk.

`cmm_get_descriptor`

- fails to find the descriptor of the caller of the current activation (returns `NULL`).
- By inspecting the value of the sp. OCaml uses this method to determine which is the bottom activation.

So, we will let the front end runtime figure it out itself. What does `cmm_next_activation` do?

1. Get the descriptor for the caller of the current activation (using the current descriptor, locate the return address in the current frame; using this return address as key, lookup descriptor in table).
2. Move pointer from current frame to caller's frame (using the frame size stored in the current descriptor).

So that the runtime can implement this itself, we expose more of the `C--` descriptor:

```
unsigned saved_retaddr;
unsigned frame_size;
```

And we also expose the fact that descriptors are keyed by the address where procedures are suspended for garbage collection, and instead of

```
cmm_gc_descriptor *cmm_get_descriptor(activation *a);
```

we provide this

```
cmm_gc_descriptor *cmm_get_descriptor(code_addr ra);
```

Now the front end runtime implements `cmm_next_activation` itself thus:

```
ra = *(sp + descr->saved_retaddr);
sp = sp + descr->frame_size;
descr = cmm_get_descriptor(ra);
```

and `cmm_first_activation` like this:

```
ra = <...>
sp = <...>
descr = cmm_get_descriptor(ra);
```

And here is `cmm_next_activation` when we have foreign stack sections:

```
ra = *(sp + descr->saved_retaddr);
if (<end of C-- section>) {
  <...>
  ra = <...>
  sp = <...>
} else {
  /* Normal case */
  sp = sp + descr->frame_size;
}
descr = cmm_get_descriptor(ra);
```

Exposing `saved_retaddr` has other uses. Generational stack collection[CHL98] aims to reduce the cost of scanning deep stacks in in generational collection. The collector patches the stored return address with the address of a runtime stub. (OCaml uses a different technique, but it also involves patching return addresses).

> *A detail that [JRR99] omits: Most likely there is a (hash) table of descriptors indexed by return address. When is this table*

built? Either at program start, or lazily, before the first garbage collection cycle. Now, only C-- knows the exact format of the descriptors, so there has to be a call for this in C--'s RTS, say cmm_init_gc_descriptors.

If the front end runtime manages unwinding itself, it also has to deal with callee-saves registers (update location as we unwind). In OCaml there are no callee-saves, so I have not had to deal with this. We could devote a subsection to discuss this.

BTW, I don't use first_activation, etc in the OCaml back end, but the stuff described above. I haven't even implemented first_activation and friends.

4 C-- Annotations

To tame the complexity of compiler construction, compilers are often organized as a pipeline of passes that operate on one or more intermediate languages[KH89] (other references). [3]Different phases of the compiler can gather information useful for optimizations—the result of program analysis phases, profile-driven compilation, or programmer annotations. In monolithic compilers, this is passed from early to later phases via constructs of the intermediate language(s), symbol tables, files (for cross-module optimizations), or other means. (A piece of information can be useful to the very last phases of code generation, like register allocation).

When the back end is decoupled from the rest of the compiler—as is the case in a C-- back end—the only way to communicate program properties to the back end is via C-- language constructs. Right now, C-- lacks adequate syntax for this. We think it is essential that we can express this in C--, so that aggressive optimizers for C-- can be built.

Another example: with VPO all the information you can pass to the code generator/optimizer is whatever can be expressed as RTLs. Norman can give more details.

Criterion 1 *Do not throw away information that can be exploited for low-level optimizations and that the C-- compiler cannot (easily) recover.*

We are interested in widely-applicable optimizations, like register allocation. Not interested at this stage in more specialized (less applicable) things like, say, loop vectorization.

Although it would be a large effort to build a code generator that has all the optimizations of gcc, in C-- we can compensate by exploiting any information that the front end is able to collect.

Propose C-- syntax (annotations) for: calls to procedures that do not return normally, calls to procedures that are guaranteed not to start a garbage collection, calls to procedures that have no side effects, branch probability, aliasing.

For each: what optimizations are enabled; why can't C-- recover the same information.

1. Some procedures do not return normally to their caller: they terminate the program abruptly, or they raise an exception. The standard libraries of many programming languages usually contain some.

[3]Example: OCaml 3 uses seven different intermediate languages.

Examples of the former: C's `exit, abort`, Java's `System.exit`, Haskell's `error`. Examples of the latter: OCaml `failwith`. To name just a few. A typical use:

```
if (some error condition) { abort(); }
```

Another example: in languages with run-time error checking the compiler generates calls to non-terminating procedures (We may end up with many of them in the `C--` program). Ex:

```
if (array index > limit) {
  /* Print error message and raise exception */
  array_bounds_error();
}
```

What optimizations are possible? 1) Better register allocation: variables that are live across the call can be allocated to scratch (caller-saves) registers 2) Any instructions immediately following the non-returning call are unreachable code and can be removed.[4]

A procedure never returns if all its possible control paths end in a statement that does not return: raise an exception[5], call or tail call a procedure that never returns. gcc[Sta92] allows this annotation. Proposal for `C--`: `no_return` for call statements.

(Often (almost always) the non-returning procedure is defined somewhere else, so the `C--` compiler sees call sites but not definitions. Without annotations, it cannot infer whether a procedure returns.)

2. (Relevant to garbage-collected languages only) Many procedures never trigger a garbage collection—and for some of them this can be determined statically. What procedures are guaranteed to never start a collection? Those that do not allocate and only call or tail call procedures that do not start a garbage collection. The compiler emits a descriptor for every call site. However, for some of these calls we are emitting an unnecessary descriptor. This 1) makes the object code larger 2) slows down garbage collection: we have to build a larger hash table (once), and we have to query a larger table (each call to `cmm_get_activation` 3) wastes compilation time for each useless descriptor that we emit (probably very little, but who knows?). Proposal for `C--`: `no_gc` for call statements (Many functions in the standard library of OCaml and other languages do not start a garbage collection; foreign functions in OCaml can be annotated as "noalloc").

3. Branch probability. Some branches are not taken most of the time. A good example: the run-time error checking examples of above (array index, ...). Another one:

   ```
   if (!allocate_memory(sz)) {
     garbage_collection();
   }
   ```

   Optimization: register allocation. When the register allocator needs to spill a variable or compiler-generated temporary, it selects a victim according to a spill cost function that estimates the cost of adding extra loads and stores. Statements that are executed more often

[4]For example, in the alpha, after a call instruction you may need an instruction to reload a global pointer register.

[5]We include C's `longjmp` in this category.

contribute more cost to the cost function. Having precise information about branch probability helps to estimate spill costs accurately. Optimization: modern processors predict forward jumps as not taken. When we emit code for ifthenelse statements, we can invert the condition and switch the branches. Optimization: instruction prefetching. Before the branch is taken (better, before the condition is evaluated), prefetch the first instruction of the likely branch. Branch probability in `C--`: if (expr prob), goto expr [label prob]*, switch.

4. Some procedures do not perform side effects[JG91, BK00]. Optimization: calls to these procedures are subject to code motion optimizations (loop invariants, etc). What procedures are pure? Those that only contain effect-free statements, where a call or tail call to a procedure that does not have side effects is also a side effect free statement. Examples: many in the standard library (mathematical functions, string manipulation, etc). gcc allows this annotation. Proposal for `C--`: `pure` for call statements.

5. Memory aliasing. Optimizations: instruction scheduling[NHN95], dead store elimination. Quite important for architectures like IA64, where the compiler has a lot of weight in the final performance of the code. Annotate loads, stores and calls[CK88]. If we want to annotate calls, we need to come up with a good syntax for defining *sets* of memory regions.

6. Data prefetching. Several architectures have data prefetch instructions (prefetch, prefectch for write). sgiCC allows programmer annotations for data prefetching. C9X has constructs that a compiler can exploit for prefetching. Example. In C9X, the use of the `static` keyword in this function prototype

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

   guarantees that x and y point to arrays containing at least six floats. A C9X compiler can schedule a prefetch early within `dot_product` or even before a call to this function. Proposal for `C--`: A statement `prefetch addr size [read/write]` where `addr` is a `C--` expression of type native data pointer and `size` is a `C--` expression (known at compile-time?) of type int.

This list is not closed: as new architectures and new compiler optimizations emerge, there will be new sensible annotation for `C--`.

cmmc understands `no_gc, no_return`.

5 Related work

MLRISC accepts memory region usage annotations in load expressions, store and call statements. MLRISC does not try to interpret the region annotations: it is up to the client to decide whether two regions may alias. Also, the MLTree data structure is parametrized with client extensions. For example, you can add a prefetch statement. The client has to give the translation of the extension into target instructions. This is more flexible than `C--` annotations, but requires the client to do all the work to generate the optimized code.

[Tar00] describes techniques to minimize the space occupied by GC descriptors.

Acknowledgement

Lal George and Allen Leung have provided invaluable help with MLRISC. Xavier Leroy.

References

[ADM98] Ole Agesen, David Detlefs, and J. Eliot B. Moss. Garbage collection and local variable type-precision and liveness in Java Virtual Machines. In *Proceedings of the ACM SIGPLAN'98 Conference on Programming Language Design and Implementation (PLDI)* [PLD98], pages 269–279. *SIGPLAN Notices* 33(5), May 1998.

[BK00] Nick Benton and Andrew Kennedy. Monads, effects and transformations. In Andrew Gordon and Andrew Pitts, editors, *Electronic Notes in Theoretical Computer Science*, volume 26. Elsevier Science Publishers, 2000.

[BW88] Hans-Juergen Boehm and Mark Weiser. Garbage collection in an uncooperative environment. *Software Practice and Experience*, 18(9):807–820, 1988.

[CHL98] Perry Cheng, Robert Harper, and Peter Lee. Generational stack collection and profile-driven pretenuring. In *Proceedings of the ACM SIGPLAN'98 Conference on Programming Language Design and Implementation (PLDI)* [PLD98], pages 162–173. *SIGPLAN Notices* 33(5), May 1998.

[CK88] Keith D. Cooper and Ken Kennedy. Interprocedural side-effect analysis in linear time. In *Proceedings of the ACM SIGPLAN'88 Conference on Programming Language Design and Implementation (PLDI)*, pages 57–66, Atlanta, Georgia, 22–24 June 1988. *SIGPLAN Notices* 23(7), July 1988.

[FH95] Chris W. Fraser and David R. Hanson. *A Retargetable C Compiler: Design and Implementation.* Benjamin/Cummings Pub. Co., Redwood City, CA, USA, 1995.

[JG91] Pierre Jouvelot and David K. Gifford. Algebraic reconstruction of types and effects. In *Conference Record of the Eighteenth Annual ACM Symposium on Principles of Programming Languages*, pages 303–310, Orlando, Florida, January 21–23, 1991. ACM SIGACT-SIGPLAN, ACM Press.

[Jon96] Richard E. Jones. *Garbage Collection: Algorithms for Automatic Dynamic Memory Management.* Wiley, July 1996. With a chapter on Distributed Garbage Collection by R. Lins.

[JRR99] Simon Peyton Jones, Norman Ramsey, and Fermin Reig. C--: a portable assembly language that supports garbage collection. In *International Conference on Principles and Practice of Declarative Programming*, pages 1–28. LNCS 1702, September 1999. Invited paper.

[KH89] Richard Kelsey and Paul Hudak. Realistic compilation by program transformation – detailed summary. In *Conference Record of the Sixteenth Annual ACM Symposium on Principles of Programming Languages*, pages 281–292, Austin, Texas, January 11–13, 1989. ACM SIGACT-SIGPLAN, ACM Press.

[NHN95] Steven Novack, Joseph Hummel, and Alexandru Nicolau. A simple mechanism for improving the accuracy and efficiency of instruction-level disambiguation. In *Languages and Compilers for Parallel Computing*, volume 1033 of *LNCS*, 1995.

[PLD98] *Proceedings of the ACM SIGPLAN'98 Conference on Programming Language Design and Implementation (PLDI)*, Montreal, Canada, 17–19 June 1998. *SIGPLAN Notices* 33(5), May 1998.

[RPJ00] Norman Ramsey and Simon Peyton Jones. A single intermediate language that supports multiple implementations of exceptions. In *Proceedings of the ACM SIGPLAN'00 Conference on Programming Language Design and Implementation (PLDI)*, pages 285–298, Vancouver, British Columbia, 18–21 June 2000. *SIGPLAN Notices* 35(5), May 2000.

[SLC99] James M. Stichnoth, Guei-Yuan Lueh, and Michal Cierniak. Support for garbage collection at every instruction in a Java compiler. In *Proceedings of the ACM SIGPLAN'99 Conference on Programming Language Design and Implementation (PLDI)*, pages 118–127, Atlanta, Georgia, 1–4 May 1999. *SIGPLAN Notices* 34(5), May 1999.

[Sta92] Richard M. Stallman. *Using and Porting GNU CC (Version 2.0).* Free Software Foundation, February 1992.

[Tar00] David Tarditi. Compact garbage collection tables. In Tony Hosking, editor, *Proceedings of the Second International Symposium on Memory Management*, Minneapolis, MN, October 2000. ACM Press. ISMM is the successor to the IWMM series of workshops.