Tail Calling Between Code Generated by C and Native Backends

Laurent Huberdeau
DIRO
Université de Montréal
Canada
laurent.huberdeau@umontreal.ca

Marc Feeley
DIRO
Université de Montréal
Canada
feeley@iro.umontreal.ca

Abstract

Compiler backends are typically incompatible; it is not possible to call the code generated by one backend from code generated by another. When writing a new backend, not being able to use the runtime library of an existing backend makes the development process longer and more complex than it should be as everything has to be rewritten from scratch. Having interoperability of code generated by different backends allows code sharing for non-performance critical parts and simplifies the development of new backends. This paper presents an approach used in the Gambit Scheme compiler to seamlessly mix the execution of code generated by C and native backends, by making tail calls and non-tail calls between the two worlds possible. Our approach uses a dual purpose control point representation and *bridges* which are executed when control jumps from one world to the other. The technique is presented in the context of the ongoing development of Gambit's native backend which targets the x86-32, x86-64 and ARM architectures.

CCS Concepts • **Software and its engineering** → *Retargetable compilers*; *Source code generation*; *Runtime environments*;

Keywords Compiler, Backend, Interoperability, Tail Call, Scheme

1 Introduction

A popular implementation approach for programming languages consists in designing an abstract instruction set tailored to the source language and implementing this abstraction as a virtual machine (VM). This usually simplifies the compiler, it offers portability when the VM is based on an interpreter, and the abstract instruction set can be used as the intermediate representation (IR) of a compiler with multiple target languages.

This approach is particularly interesting for Scheme because the language has constructs that have no direct equivalent in other languages; tail calls and continuations among others. The Bit [1], Picobit [8], Guile [7], Racket [6] and Gambit [3] systems are all based on this approach.

SFPW'18, Sept. 28, 2018, Saint Louis, MO, USA
2018.

The Gambit system uses the Gambit Virtual Machine (GVM) [4] as the IR of a compiler that generates code for several target languages, including C, JavaScript, PHP, Python, Ruby, and Java. In a nutshell, each GVM instruction is translated into a sequence of instructions of equivalent effect in the target language.

Gambit's C backend is the most mature. It handles all of the Gambit Scheme features and is in a sense the Gambit Scheme *reference implementation*. However, the C language imposes code generation constraints that affect performance, notably in regard to the handling of tail calls and continuations. This has prompted an effort to develop a family of backends that target the x86-32, x86-64 and ARM architectures at the machine code level. They are collectively called the *native backend*, as these backends share a large part of their logic.

Because the GVM is a relatively high-level abstraction that, for example, does not specify how objects are represented and how garbage collection is performed, implementing the native backend could involve redesigning and rewriting the whole runtime system. Not only would this approach easily require a few man-years of effort, but the support for the various Gambit Scheme features would progress incrementally over the course of the project, hindering large-scale experimentation and adoption by users until all features are supported. We have instead chosen to design the native backend so that the code generated interoperates transparently with the code generated by the C backend. In other words, the native backend implements the same object representation as the C backend (allowing data to be shared transparently) and it is possible to jump back and forth between the existing runtime system (including library procedures and garbage collector) compiled using the C backend and code generated by the native backend. This way the availability of the Gambit Scheme features comes early in the development process because their implementation using the C backend can be used as a fallback until the moment they are handled directly, and hopefully more efficiently, by the native backend.

Having early access to the C backend's runtime system simplifies the development process of the native backend.

Allocation procedures, the garbage collector, operating-system interface procedures, complex library procedures including `display` and `pretty-print`, ressource and run time measurement procedures, exception handlers (stack/heap overflow, wrong number of arguments, etc), first-class continuations, can all be used in test programs. These features are particularly useful in debugging the machine code generation and to assess the performance of various code generation approaches (choice of instructions, registers, etc).

This paper explains the design and implementation of the mechanism to jump between code generated by the C backend and code generated by the native backend (cross jumps). Three important design objectives that are achieved by our design are:

1. Cross jumps are transparent. A backend does not know statically if a jump is a cross jump or not. It is only at run time that the destination is known. A library procedure compiled using the C backend could be called with return addresses that cause it to return to code generated by the C backend sometimes and the native backend other times. Moreover, cross jumps that are tail calls must not consume space, on the stack or elsewhere.
2. Non-cross jumps are performed as efficiently as if the other backend did not exist. For the native backend, this means that it can implement procedure values and return addresses as pointers to machine code, and a plain *branch to address* machine instruction can be used for procedure calls and returns.
3. Cross jumps have a cost that is similar to non-cross jumps within code generated by the C backend, which uses trampolines for implementing jumps.

The paper is organized as follows. Section 2 presents the GVM and how the C and native backends implement tail calls and continuations. The technique and its implementation are explained in detail in Section 3. Section 4 evaluates the performance overhead of the technique and its applicability to the development of the native backend.

2 Background

2.1 Gambit Virtual Machine

The Gambit Virtual Machine is the abstract machine targeted by the compiler's frontend. The machine uses a set of locations that can store any Scheme object reference. There are general purpose registers numbered starting at 0 (e.g. *r1*), frames that are stored on a stack (e.g. *frame[1]* is the first slot of the current frame) and global variables (e.g. *global[a]* to access the global variable *a*). The backends can parameterize the generation of code for the GVM by specifying the number of available registers, the set of inlined primitives and the procedure calling convention. In the existing backends, only 5 registers are used, 2 of which have a special purpose. The first register, *r0*, contains the return address of the current procedure and the last, *r4*, which we call the *self* register, is used when calling closures. One of the general purpose registers, *r1*, is also used to contain the return value of procedures. The frontend generates a control flow graph of basic blocks that contain code for the GVM.

The GVM defines seven instructions: *label*, *jump*, *ifjump*, *switch*, *copy*, *close* and *apply*. The *label* instruction is the first instruction of each basic block and gives the block's identifier, which is a small number prefixed by # (e.g. *#2*), in addition to its kind. There are two kinds of basic blocks: first-class blocks and local blocks. First-class blocks can be used like any other Scheme object reference and are for control flow between procedures. There are two types of first-class blocks: entry points and return points, which correspond to procedure values and return addresses respectively. When called, entry points verify the number of arguments before proceeding with the execution of the basic block. Local basic blocks are not first-class objects as they are only used for control flow inside the procedure. The label instruction also indicates the current size of the stack frame. This is used to access locations on the stack with the correct offset from the top of the stack.

Basic blocks always end with a branch instruction indicating the frame size at the end of the basic block. This allows the backend to adjust the stack pointer only once as explained below. There are two different types of branches: conditional and unconditional. Conditional branches are generated for `if`, `case` and `cond` Scheme expressions and correspond to the *ifjump* and *switch* GVM instructions. Unconditional branches correspond to the GVM's *jump* instruction. Conditional and unconditional branches have an important difference in regard to their destinations. Conditional branches may only jump to local blocks and unconditional branches may jump to local blocks as well as first-class blocks. We call *jump* instructions to first-class blocks external *jump*s as the control goes to a label that may be in another procedure. For external *jump*s, the *jump*'s operand may be a reference to a closure, an entry point or a return point.

The *jump* instruction optionally specifies a number of arguments. This is used by the callee to verify that the number of arguments passed to the procedure is correct and to support rest and optional parameters. When the value is given, the destination of the external *jump* must be a procedure (closure or entry point), otherwise, it must be a return point. This is used to generate efficient code for the *jump*.

```
(define (display-length lst)
  (display (length lst)))

#1 fs=0 entry-point nparams=1
  frame[1] = r0  ; Create continuation frame containing r0
  jump fs=1 global[length] r0=#2 nargs=1  ; Non tail call length procedure
#2 fs=1 return-point
  r0 = frame[1]  ; Extract return address from continuation frame
  jump fs=0 global[display] nargs=1  ; Tail call display procedure
```

Figure 1. Scheme and GVM code of `display-length`

The remaining instructions only appear between the label and branch instructions of the basic blocks.

The *copy* instruction assigns the value of the source operand to the destination operand.

The *close* instruction creates one or more closure objects and assigns them to the destination operands. The closure object contains a reference to the entry point that is executed when the closure is jumped to and the value of its free variables. When called, a reference to the closure is assigned to the self register, allowing the closure's code to access its free variables.

The *apply* instruction executes a primitive (a simple operation like `##car`) and stores the result in the destination operand. For example `r1 = (##cons r2 r3)` allocates a pair whose car and cdr are initialized from *r2* and *r3*, and stores the reference to this pair in *r1*. The set of primitives that are usable in *apply* instructions, *inlined primitives*, may vary from one backend to another, but there is always a procedure of the same name in the runtime library performing the same operation. This means that a given primitive operation can always be performed by a call to the runtime library using a *jump* instruction, but if the backend handles its inlining, an *apply* instruction can be used, usually with much lower execution time.

The GVM's stack is composed of frames. There are two types of stack frames: *activation* frames and *continuation* frames. The first is created before calling a procedure. It contains the arguments of the call that are not passed in the registers. It is empty if the arguments all fit in the registers, a fairly common situation. When tail calling, the topmost frame can be reused to store the arguments of the destination procedure. When a non-tail call is made, a continuation frame is created and the destination procedure's activation frame is added on top of it. The continuation frame is used to save the values of the registers that are needed at the return point of the call.

The stack is managed implicitly by the *label* and *branch* instructions. The *label* instructions indicate the current frame size (e.g. fs=0) before the execution of the basic block and the *branch* instructions indicate the frame size at the end of the basic block. The frame size difference is how many slots have been pushed (or popped if negative). Using the initial frame size, backends can access the slots of the frame with the correct offset. Writes to frame slots with index higher than the current frame size are equivalent to push operations and pops are done at the end of the basic block when the frame size decreases.

As an example, Figure 1 shows the Scheme and GVM code of the `display-length` procedure that displays the length of a list.

Basic block *#1* is the `display-length` procedure's entry point. The label instruction specifies that 1 argument is expected (nparams=1). Because this argument and the return address are passed in registers the activation frame is empty (fs=0). The second instruction creates a continuation frame by saving the return address (*r0*) to the stack (it is the only value needed at the return point of the `length` procedure). The last instruction of the basic block is a jump to the `length` procedure with basic block *#2* as the return point passed in register *r0*. The number of arguments of the call is 1. Because the first argument is always passed in *r1*, *r1* already contains `lst`, the argument for the `length` procedure. The result of `length` is also in *r1*.

Basic block *#2* is a return point as it is used as the continuation of the call to `length`. Before its execution, the topmost frame is a continuation frame with only 1 slot: the return point of the `display-length` procedure. The return point is restored in *r0* to execute a tail call to `display`. Again, the argument of `display`, which is the result of `length`, is already in *r1* and is left untouched. Note that when tail calling `display`, the continuation frame is reclaimed (fs=0).

2.2 Object Representation

We describe here the object representation used before Gambit's C backend was modified to accommodate the native backend. A Scheme object reference is a machine

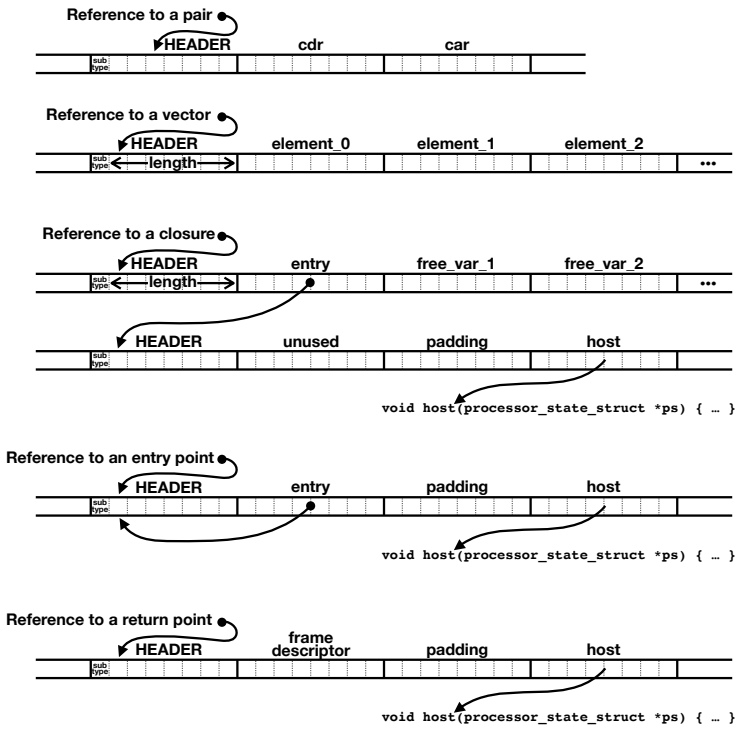

Figure 2. C backend Scheme object representation for memory allocated objects

word, either 32 or 64 bits wide depending on the architecture. A low-bit tagging is used with the two lowest bits containing one of the 4 possible tags:

- `00` – fixnum value in the remaining 30 or 62 bits
- `01` – reference to a memory-allocated object other than a pair
- `10` – other immediate value (characters, `#f`, `#t`, `()`, `#!eof`, `#!void`, ...)
- `11` – reference to a pair

Memory-allocated objects are word-aligned, so they start at an address that is a multiple of 4 or 8, leaving enough space in the object reference to store the 2 tag bits. The first word of all objects, including pairs, is a word-size header. It contains other type information in the least-significant 8 bits (the *subtype*) and the length of the object in the remaining bits of the word (useful for variable length objects such as vectors, strings and closures).

The type tests `(fixnum?` x`)` and `(pair?` x`)` can be done quickly by checking the tag bits of x. Type tests for other memory-allocated objects, like `(flonum?` x`)` and `(vector?` x`)`, require checking that the tag bits are `01` and then checking the subtype field of the header.

Figure 2 shows the memory layout of some types of objects for a 64 bit architecture. The top two show the pair and vector objects. In this drawing, arrows starting from a dot are object references. Note that a reference to a pair points to the fourth byte of the header (tag = `11`) and a reference to a vector, like all other memory allocated objects, points to the second byte of the header (tag = `01`). When accessing the content of the object, the tag can easily be cancelled in the offset used to indirect memory. For example, if x is a reference to a pair, the car can be extracted by reading the word at address $x + 13$ ($13 = 2 \times 8 - 3$). Modern architectures, including x86 and ARM, can typically do this address calculation at no cost, so the untagging is often free. Similarly, the choice of `00` as the tag for fixnums simplifies fixnum addition and subtraction, rather frequent operations. Indeed, adding or subtracting two fixnums can be done with the machine instructions yielding a correctly tagged fixnum result.

2.3 First-Class Control Points and the Trampoline

Figure 2 also shows the memory layout of entry points, return points and closures. The flat closure representation is used, containing a pointer to the closure's entry point followed by the closure's non-global free variables. Note that both entry points and closures count as procedures, but entry points can only have free variables that are global variables. For that reason entry points

```
typedef struct processor_state_struct {
  WORD r0, r1, r2, r3, r4;   /* GVM registers */
  WORD *fp;                  /* frame pointer */
  WORD *hp;                  /* heap pointer */
  WORD na;                   /* number of arguments */
  WORD pc;                   /* tagged pointer to a label_struct */
  ...
} processor_state_struct;

typedef struct label_struct {
  WORD HEADER;
  WORD entry_or_descr;                      /* extracted with GET_ENTRY(pc) */
  WORD unused;
  void (*host)(processor_state_struct *ps);  /* extracted with GET_HOST(pc) */
} label_struct;

void trampoline(processor_state_struct *ps) {
  while (TRUE) GET_HOST(ps->pc)(ps);
}

/* below is the code generated by the C backend */

label_struct labels[];    /* forward declaration */

void host(processor_state_struct *ps) {

  WORD start = 1+(WORD)&labels[0]; /* start control point of this host */

  jump:
    switch ((ps->pc - start) / sizeof (label_struct)) {
      case 0: ...;     /* control point 0 of this host */
      case 1: ...;     /* control point 1 of this host */
      case 2: ...;     /* control point 2 of this host */
      default: return; /* destination is in another host */
    }
}

label_struct labels[] = {
  { ..., 1+(WORD)&labels[0], 0, host } /* label_struct of control point 0 */
, { ..., 1+(WORD)&labels[1], 0, host } /* label_struct of control point 1 */
, { ..., 1+(WORD)&labels[2], 0, host } /* label_struct of control point 2 */
};
```

Figure 3. Overall structure of the trampoline and the generated C code

and return points are statically allocated, and closures are in general dynamically allocated in the heap.

Gambit's C backend trampoline mechanism has evolved since its original implementation [5]. The following description has been slightly simplified from the actual implementation to avoid getting caught up with minor details. Note that other Gambit backends (e.g. targeting JavaScript) also use a trampoline mechanism, as explained in [2]. Figure 3 shows the overall structure of the trampoline and the generated C code (`host` function and `labels` array).

Entry points and return points are represented with a `label_struct` structure that contains 4 word-size fields: the header, the entry, an unused word, and the host which is a pointer to a C function. This structure conceptually represents a control point in the code that can be jumped to from any point in the code, essentially analogous to a *label* in assembly code. These correspond to either an entry point or a return point, which are called first-class control points (simply *control points* from here on). The trampoline implements the jumping to the control point corresponding to a `label_struct` structure.

The `processor_state_struct` structure stores the state of the GVM. Among other things, it contains the state of the GVM registers (`r0` to `r4`), the frame pointer (`fp`), the heap pointer (`hp`), the number of arguments

(`na`), and the current control point (named `pc` because of the analogy with the program counter of a machine).

The executable code generated by the C backend is contained in *host* C functions. In general a host contains more than one control point and a program has more than one host (this is useful for separate compilation, in particular of the runtime library, and dynamically loadable compiled modules). Gambit offers two compilation modes: in *multiple host* mode a host C function is generated for each top level procedure of a compiled file, and in *single host* mode all the code of a file is enclosed in a single host C function. In addition to the host C function(s) generated for a file, an array of `label_struct` structures is generated by the compiler, one for each control point in the file. Each of these structures has a host field that points to the host containing the control point. The structures are ordered so that those that pertain to a given host are contiguous in the array. For each host the compiler knows at what index in the array the first `label_struct` structure of that host is located, so it can derive the object reference to this structure, which we will call the *start* control point of that host. It is an easy matter for a host to substract *start* from `pc` and divide by the size of the `label_struct` structure to obtain an index usable in a C `switch` statement to dispatch to one of several `case`s each corresponding to one of that host's control points. Note that the division is actually implemented as a shift because the `label_struct` structure is padded to force its size to be a power of two.

Jumping between hosts is the responsibility of the trampoline C function. In an endless loop, it extracts the host field of the processor state's `pc` and calls that function, passing the processor state as the sole parameter. When the host function returns it will have put in the processor state's `pc` the destination control point of the jump. At the next iteration of the loop, the corresponding host will be called. The program's execution is started with an initialization of the processor state's `pc` field and a call to the trampoline such as `ps->pc = 1+(WORD)&labels[0]; trampoline(ps);`.

The `GET_HOST` macro extracts the host field of the `label_struct` structure referenced by `ps->pc`. The trampoline's loop appears to be infinite, but it is actually exited directly from code in hosts using the C function `longjmp`. Because this happens rarely it is more efficient to use `longjmp` than to slow down the trampoline's loop with a condition.

To jump to a destination control point x the C code executes `ps->pc =` x`; goto jump;`. This will repeat the dispatch of the `switch` statement with the new destination. This optimizes the case where the destination happens to be in the same host (for example a recursive Scheme function returning within itself). When the destination's `label_struct` structure belongs to another host the `default` case of the `switch` will be executed and the trampoline will take care of jumping to that host.

Gambit implements a few additional optimizations not shown here that improve the execution speed of the trampoline. A host C function copies the relevant parts of the processor state (`pc`, `fp`, GVM registers, ...) into local variables to improve the likelihood the C compiler will allocate these to machine registers. Before returning to the trampoline the host copies the modified variables back to the processor state. Moreover, when the destination control point is statically known and in the same host, for example when calling a Scheme procedure locally defined or one compiled with the special declaration `(declare (block))` and single host mode, a simple `goto` is used. Finally, when compiling with `gcc` and `clang`, the `switch` statement is replaced by a faster *indirect* `goto`, which is a C extension supported by these compilers.

2.4 Jumping to Procedures and Return Points

Procedures are either entry points or closures. In both cases, the first field after the header is *entry*, which contains a reference to a `label_struct` structure. Procedures that are entry points have an entry field that contains a reference to itself. To jump to procedure x, the code `ps->pc = GET_ENTRY(ps->r4 =` x`); goto jump;` is executed. This jumps to the entry point whether x is a closure or not. In addition, this puts in the self register, `r4`, a reference to the called procedure, which is useful in the case x is a closure to access its free variables. To jump to return point x, the simpler `ps->pc =` x`; goto jump;` is executed. In the GVM *jump* instruction these cases can be distinguished because in the first case the number of arguments is specified in the *jump* instruction, and not in the second case.

3 Jumping between C and native code

3.1 Bridges

The way the GVM's state is stored is an important difference between the C and native backends. In the C backend, parts of the GVM state are copied to the host's local variables for efficiency and it is the C compiler's option to assign these to machine registers of its choosing. In the native backend, most of the state is stored in registers and fixed machine registers are assigned to the GVM registers (for example, on x86-64, `rdi`/`rax`/`rbx`/`rdx`/`rsi` are assigned to *r0..r4*). This means that cross jumps can't be done with a simple branch instruction. The different assignment of registers makes this impossible. Instead, whenever there is a cross jump, a *bridge* routine is used to move the GVM's state to where the other backend expects it. There are two bridges, one for C to native

cross jumps (`to_native_bridge`), and one for native to C cross jumps (`from_native_bridge`).

3.2 Control Point Representation

To achieve our objective that the native backend uses simple and fast *branch to address* instructions to perform jumps, the representation of control points had to be changed. The reference to a control point must be a pointer to the executable machine code. For example, a reference to an entry point must point to the first machine instruction of that basic block. Thus a control point reference can no longer point to the header of the `label_struct`, which contains data. The layout of a `label_struct` was changed so that the `host` field is now the first, just before the header. The second field after the header, which was unused, now contains executable machine code (the `code` field). Finally an object reference is now a tagged pointer to the second field after the header. This means that a control point reference points to the second byte of the `label_struct`'s `code` field. These changes are illustrated in Figure 4 for the x86-64, which has 64 bit words.

For `label_struct`s generated by the C backend, the `code` field contains machine code that jumps to the `from_native_bridge`. This is a *call* instruction, on x86, or a *branch-and-link* instruction on ARM. When code generated by the native backend jumps to a control point in code generated by the C backend, the branch instruction first transfers control to the executable code in the `code` field which then transfers control to the bridge. A side effect of the call (or branch-and-link) instruction is to store on the stack (or register) the address of the `label_struct`. This information is then used by the bridge to know which `label_struct` is the destination of the jump. The pointer will then be stored in the `pc` field of the processor state before returning to the trampoline, causing the jump to the appropriate C host function.

To support cross jumps in the other direction, the native backend generates `label_struct`s before each control-point (see bottom part of Figure 4) such that:

1. The host field stores a pointer to `to_native_bridge`.
2. The reference to the control point is a pointer to the first machine instruction of the basic block.

When code generated by the C backend jumps to a control point in code generated by the native backend, the trampoline will naturally cause a call to the host function `to_native_bridge` and the processor state's `pc` field will have a reference to the destination control point. The bridge then copies the state of the GVM from the processor state structure to the appropriate machine registers, and then branches to the machine code at the destination control point.

The representation of closures must also be changed to accommodate the native backend. The native backend can jump to closures like control points using a *branch to address* instruction to the closure reference. To support this, a `code` field is also added to the closures (both those generated by the C and native backends). This field contains the machine code that reads the closure's `entry` field and branches to that address using a *call* (or *branch-and-link*) instruction. This has the side-effect of saving to the stack (or register) the address of the closure, which will be used to setup the self register either by the `from_native_bridge` if it is a native to C cross jump, or by the first few instructions at the entry point if it is a native to native non-cross jump.

3.3 Continuations

Because control may freely go between code generated by the C and native backends, the stack may contain a mix of continuation frames of procedures compiled by both backends. This is not an issue as the backends implement the GVM in the same way with the same stack, the same frame structure and the same frame descriptors. This allows garbage collection and first-class continuations to work seamlessly without modification. For the same reasons, serialization of procedures and continuations compiled by the native backend is also possible.

3.4 Implementation Details

The native backend reserves one of the machine registers to point to the processor state, thus making it easy to access the GVM state that has not been copied to machine registers, such as the number of arguments (`na` field) and the stack and heap limit pointers. This register is `rcx` on x86-64, `ecx` on x86-32, and `r5` on ARM. Moreover, to allow compact instructions to call `from_native_bridge`, the address of this routine is stored in a word just before the processor state, so the x86-64 can use a `call [rcx-8]` and the x86-32 can use a `call [ecx-4]`, both 3 byte instructions that fit in a machine word. On ARM negative offsets on memory accesses are not encoded as compactly as positive offsets, so `r5` is made to point to the word just before the actual processor state, allowing `from_native_bridge` to be called with the 4 byte instruction sequence `ldr r7,[r5]; blx r7` (`r7` is a temporary register). Figure 5 shows the machine instructions in the `code` fields of `label_struct`s and closures. Note that for x86-32 and ARM, which both have 32 bit words, two words are needed in closures to store the machine instructions. The implementation of the bridges for x86-64 is given in Appendix A (moderately edited to make it easier to read).

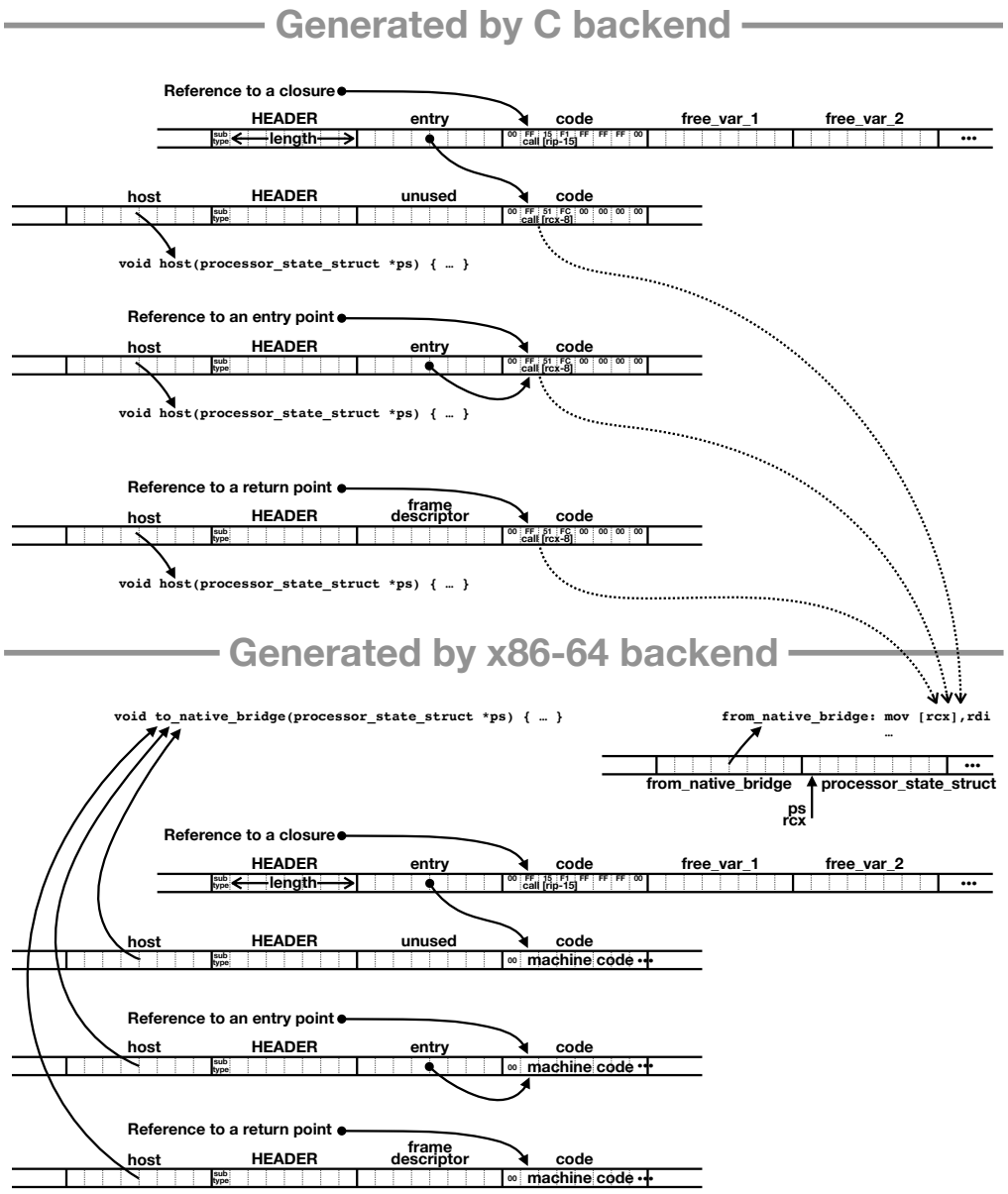

Figure 4. Scheme object representation for memory allocated objects for C and x86-64 backends

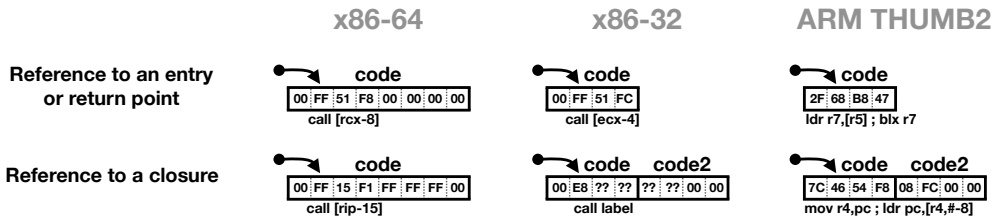

Figure 5. Machine instructions in the `code` fields of `label_structs` and closures, for the x86-64, x86-32 and ARM architectures

```
(declare (standard-bindings) (not safe))

(define (dec x)  ;; variant of program defines dec in the runtime library
  (fx- x 1))

(define (run n f)
  (let loop ((n n))
    (if (fx> n 0)
        (loop (f n)))))  ;; call/return dec

(time (run 100000000 dec))
```

Figure 6. Scheme program for measuring the cost of jumps

	Kind of jumps for procedure call and return to `dec` procedure			
Architecture	native to native	C to C (intra host)	C to C (inter host)	native to C to native (cross jumps)
x86-64	0.107 rsd=0.2% **0.07**×	0.238 rsd=0.0% **0.17**×	1.436 rsd=0.1% **1**×	2.264 rsd=0.0% **1.58**×
x86-32	2.947 rsd=0.2% **0.17**×	6.787 rsd=0.1% **0.40**×	16.854 rsd=0.1% **1**×	19.605 rsd=0.1% **1.16**×
ARM	2.758 rsd=0.3% **0.24**×	4.054 rsd=0.2% **0.35**×	11.431 rsd=0.2% **1**×	11.680 rsd=0.2% **1.02**×

Figure 7. Execution time in seconds of 10^8 calls to the `dec` procedure, the relative standard deviation over 100 runs, and in bold the relative execution time (relative to the C to C inter host jumps)

4 Evaluation

4.1 Use Cases

There are a few use cases for our cross jump mechanism. First, assuming the native backend yields higher performance than the C backend, Scheme code kernels whose performance is critical can be compiled with the native backend and called from a main program compiled with the C backend. Another use case is allowing programs compiled with the native backend to call complex runtime library procedures. These runtime library procedures are currently compiled with the C backend as they are mostly written in Scheme with low-level parts written in C. The Scheme part can't just be compiled using the native backend because many primitives are not yet inlined and the FFI needed to interface to the C parts is also not yet ported to the native backend. In both of these use cases the cost of cross jumps is not critical because each procedure call represents a considerable amount of work. A low cost for cross jumps is more important for runtime library procedures that don't represent much work, such as type tests and structure accessors. It is expected that the native backend will eventually inline all of these, for performance reasons, but as the development progresses there will always be some subset of the primitives that are not yet inlined and that must be implemented with a cross jump to the runtime library. In this use case the cross jump mechanism is a crutch that allows compiling and testing all types of programs, but with reduced performance when non inlined primitives are often called. This is particularly convenient when debugging the performance of the native backend's code generation for specific parts of code within a big program using lots of primitives. The debugging can focus on the specific parts that only use inlined primitives. For this to work in practice, the cost of cross jumps must be low.

4.2 Performance of Cross Jumps

To evaluate the performance of cross jumps we have used the program shown in Figure 6. The `run` procedure performs 10^8 calls to the procedure `dec`, a quick procedure that simply decrements its fixnum argument. Two jumps are executed per call to `dec`, one to its entry point and one to return back to the `run` procedure. The run time is measured using the `time` special form. A variant of this program is identical except it defines `dec` in the Scheme runtime library compiled by the C backend. The variant and original program are then compiled with the C backend using *single host* mode and with the native backend. Consequently, the jumps to and from `dec` are performed differently in all four configurations: (1) from C to C in different hosts (inter host), (2) from C to C within the same host (intra host), (3) from native to C to native (i.e. two cross jumps), and (4) from native to native. Configurations (1) and (2) use the trampoline, configuration (3) uses the trampoline and both bridges, and configuration (4) uses neither.

Benchmark	C backend	x86-64 backend
`(ack 3 12)`	0.895 rsd=0.4% **1**×	0.818 rsd=0.9% **0.91**×
`(fib 40)`	0.304 rsd=0.0% **1**×	0.277 rsd=0.4% **0.91**×
`(tak 40 20 11)`	1.112 rsd=0.1% **1**×	1.001 rsd=0.7% **0.90**×

Figure 8. Execution time in seconds for highly recursive benchmarks compiled with the C and x86-64 backends, the relative standard deviation over 100 runs, and in bold the execution time relative to the C backend

We used Gambit v4.9.0 and the following hardware, all running Debian: x86-64 = Intel i7-7700K @ 4.2GHz, x86-32 = Coppermine Pentium III @ 0.7GHz ARM = Quad Cortex A7 @ 0.9GHz. The programs were run 100 times. The mean execution times and standard deviation are shown in Figure 7. The table also shows the execution times relative to the pure trampoline (C to C inter host jumps), which corresponds to the situation where a program compiled with the C backend calls a runtime library procedure.

The execution time for the program includes the execution of the loop in `run`, the creation of the continuation for the call to `dec`, the execution of `dec`'s body, and the execution of the jumps to and from `dec`. The first three parts represent few machine instructions (unsafe fixnum operations and tail recursive loop) compared to one iteration of the trampoline, so we can use the execution time as a rough measure of the cost of the jump (it is an overestimation of the cost of native to native jumps which represent relatively little time compared to the overhead of the loop).

We see that the cost of cross jumps is comparable to the pure trampoline, between 2% and 58% higher depending on the architecture. We feel that this is a reasonably low overhead compared to the C to C inter host jumps.

4.3 Performance of Native to Native Jumps

Another interesting result is the very low cost of native to native jumps, at least an order of magnitude faster than pure trampoline jumps on x86-64. This means that there may be a considerable improvement in performance when Scheme code performing frequent procedure calls is compiled with the native backend (either kernels called from code compiled with the C backend, or complete programs). To explore this aspect we used the well known highly recursive benchmarks `ack`, `fib`, and `tak` with fixnum arithmetic, which have the virtue of requiring very few inlined primitives (all of which are inlined by the native backend). When compiled with the C backend, all jumps use the trampoline and are intra host jumps. When compiled with the x86-64 backend, all jumps use the *branch to address* instruction.

The benchmarks were run 100 times. The mean execution times and standard deviation are shown in Figure 8. We can see that the x86-64 backend generates code that is 9-10% faster than the C backend for these benchmarks. An examination of the machine code generated by both backends suggests that the execution time difference is mostly due to how the jumps are implemented. We can thus expect that the native backend will bring some execution time improvement to programs with frequent procedure calls.

4.4 Sample Uses

As a demonstration of the usefulness of the bridge, the native backend can be used to compile the `triangl` benchmark from the R7RS benchmark suite. Two procedures used by this benchmark, `vector->list` and `list->vector`, are currently only defined in the Scheme runtime library compiled by the C backend. Still, the native backend manages to compile `triangl` by using cross jumps for calling these procedures and the resulting program executes only 10% slower than when the C backend is used.

Another use for the bridge is calling the `##exec-stats` runtime library procedure. This procedure evaluates the thunk it receives and collects statistics on the execution. This is useful for evaluating the performance of the code generated by the backends and verify that it executes correctly. This procedure is called using the bridge as it requires the FFI which is only implemented by the C backend.

5 Conclusion

This paper has presented a technique used in the C and native backends of Gambit Scheme that supports transfers of control between the code generated by these backends. It is tranparent and supports tail calls and continuations. This allows easy reuse of parts of the Gambit runtime library currently compiled from Scheme to C by the C backend or hand written in C, such as the garbage collector and I/O subsystem.

Experiments show that the overhead of our approach is comparable to that of the trampoline. Our approach is particularly useful for the development of the native backend as it makes available the debugging features of Gambit and allows the compilation of complex programs to test the correctness and performance of the machine code generation.

Acknowledgments

This work was supported by the Natural Sciences and Engineering Research Council of Canada.

References

[1] Danny Dubé and Marc Feeley. 2005. Bit: A very compact Scheme system for microcontrollers. *Higher-order and symbolic computation* 18, 3-4 (2005), 271–298.

[2] Marc Feeley. 2015. Compiling for Multi-Language Task Migration. In *ACM Dynamic Languages Symposium*. 63–77.

[3] Marc Feeley. 2018. Gambit Scheme Compiler v4.9.0. (Sept. 2018). http://gambitscheme.org/

[4] Marc Feeley and James S. Miller. 1990. A parallel virtual machine for efficient Scheme compilation. In *ACM SIGPLAN Conference on Lisp and Functional Programming*. 119–130.

[5] Marc Feeley, James S. Miller, Guillermo J. Rozas, and Jason A. Wilson. 1997. *Compiling Higher-Order Languages into Fully Tail-Recursive Portable C*. Technical Report 1078. Université de Montréal, DIRO.

[6] Casey Klein, Matthew Flatt, and Robert Bruce Findler. 2012. The Racket virtual machine and randomized testing. *Higher-Order and Symbolic Computation* 25, 2-4 (2012), 209–253.

[7] Thomas Lord. 1995. An Anatomy of Guile: The Interface to Tcl/Tk. In *Tcl/Tk Workshop*. 95–114.

[8] Vincent St-Amour and Marc Feeley. 2009. PICOBIT: a compact Scheme system for microcontrollers. In *International Symposium on Implementation and Application of Functional Languages*. 1–17.

A Implementation of Bridges for x86-64

```
// C to native bridge
void to_native_bridge(processor_state_struct *ps) {
  __asm__ __volatile__ (

  "mov  %0, %%rcx"  // copy ps into %rcx

  "mov  %%rsp, -2*8(%%rcx)"  // save C sp

  // setup handler for returning from native code
  "lea  from_native_bridge(%%rip), %%rax"
  "mov  %%rax, -1*8(%%rcx)"  // setup handler

  // setup frame pointer and heap pointer registers
  "mov  5*8(%%rcx), %%rsp"  // rsp = ps->fp
  "mov  6*8(%%rcx), %%rbp"  // rbp = ps->hp

  // setup self register
  "mov  4*8(%%rcx), %%rsi"  // rsi = ps->r4

  "mov  8*8(%%rcx), %%rax"  // rax = ps->pc
  "cmpq $0x100000,-1-2*8(%%rax)"  // closure?
  "jl   setup_other_registers"
  "add  $3, %%rsi" // handle closures
  "push %%rsi"
  "add  $-3, %%rsi"

  "setup_other_registers:"
  "mov  (%%rcx), %%rdi"     // rdi = ps->r0
  "mov  1*8(%%rcx), %%rax"  // rax = ps->r1
  "mov  2*8(%%rcx), %%rbx"  // rbx = ps->r2
  "mov  3*8(%%rcx), %%rdx"  // rdx = ps->r3

  "jmp  * 8*8(%%rcx)" // jump to ps->pc


  // native to C bridge
  "from_native_bridge:"

  "mov  %%rdi, (%%rcx)"     // ps->r0 = rdi
  "mov  %%rax, 1*8(%%rcx)"  // ps->r1 = rax
  "mov  %%rbx, 2*8(%%rcx)"  // ps->r2 = rbx
  "mov  %%rdx, 3*8(%%rcx)"  // ps->r3 = rdx

  // recover destination control point in ps->pc
  "pop  %%rax"
  "add  $-3, %%rax"  // rax = destination ctrl pt
  "mov  %%rax, 8*8(%%rcx)"   // ps->pc = rax

  "cmpq $0x100000,-1-2*8(%%rax)" // closure?
  "jl   store_self_register"
  "pop  %%rsi"  // handle closures
  "add  $-6, %%rsi"

  "store_self_register:"
  "mov  %%rsi, 4*8(%%rcx)"  // ps->r4 = rsi
  "mov  %%rsp, 5*8(%%rcx)"  // ps->fp = rsp
  "mov  %%rbp, 6*8(%%rcx)"  // ps->hp = rbp

  "mov  -2*8(%%rcx), %%rsp"  // restore C sp

  : // no outputs
  : // inputs
    "m" (ps)
  : // clobbers
    "%rdi","%rax","%rbx","%rdx","%rsi",
    "%rcx","%rbp"
  );
}
```