Custom Calling Conventions in a Portable Assembly Language

Norman Ramsey   Christian Lindig
Division of Engineering and Applied Sciences
Harvard University
nr@eecs.harvard.edu   lindig@eecs.harvard.edu

Abstract

Calling conventions are so difficult to implement and maintain that people rarely experiment with alternatives. The primary sources of difficulty appear to be parameter passing and stack-frame layout. To implement parameter passing, we use the automata developed by Bailey and Davidson, but we have developed a new specification language that is implementable directly in the compiler. To implement stack-frame layout, we have developed an applicative, composable abstraction that cleanly decouples the layout of the stack frame from the order in which compiler phases execute. We hope these abstractions will so simplify the implementation problem that compiler writers will routinely customize calling conventions to improve performance.

1 Introduction

The C calling convention on your favorite platform may have been carefully crafted to make effective use of registers while also supporting procedures with a variable number of arguments. Such a convention can be hard to implement, and even a mature compiler can harbor lingering bugs (Bailey and Davidson 1996). If a compiler supports multiple targets, tail calls, and calls to foreign functions, the effort is compounded, because one calling convention is not enough. Such a compiler is required for C--.

C-- is a portable assembly language that is intended to be generated by a language-dependent *front end* and compiled to efficient machine code (Peyton Jones, Oliva, and Nordin 1997; Peyton Jones, Ramsey, and Reig 1999). Our hope is to make C-- as easy to generate as C, while making the resulting code perform almost as well as code from a custom code generator. To this end, C-- must be expressive enough so that the author of a front end can control cost tradeoffs. A previous paper discusses mechanisms an author can use to control tradeoffs in exception dispatch (Ramsey and Peyton Jones 2000); this paper describes mechanisms used to control cost tradeoffs in calling conventions.

One of the cost tradeoffs in a calling convention concerns use of registers. For example, how many registers are used as callee-saves, caller-saves, and parameter registers can have a significant effect on performance (Davidson and Whalley 1991; Appel and Shao 1992). As an example where use of registers matters, we present code that might be used to create a cons cell in a functional language. In this example, `hp` is a "heap pointer" that points to space that can be used for allocation, and `hplim` marks the end of that space. We might form a cons cell containing `a` and `d` using this C-- code:

```
if (hp + 12 > hplim) {  // if no space is left
  gc();                 // garbage collect
}
bits32[hp]   = TAG_FOR_CONS_CELL;
bits32[hp+4] = a;  // store the car
bits32[hp+8] = d;  // store the cdr
p  = hp +  4;      // p is the new object
hp = hp + 12;      // set hp to next free word
```

The code checks to see if there is enough space, and if not, it calls the garbage collector. Allocation and initialization are done inline; the `bits32[...]` notation is C-- for a reference to memory.

What sort of calling convention should be used to call `gc()`? Statically, call sites to `gc()` are likely to be common (there is at least one for every extended basic block that allocates), but actual dynamic calls are likely to be rare (almost every allocation succeeds without garbage collection). We therefore want `gc` to have as many callee-saves registers as possible—this convention would minimize code size at each call site and thus minimize total code size. If we were to save some registers unnecessarily at a garbage collection, the dynamic cost would be dominated by the cost of the collection. To maximize the number of callee-saves registers, we can't simply tune an existing convention to change the number of registers saved across calls—we must also be sure that no register is reserved to hold a parameter or result, because an unused parameter register is typically treated as caller-saves (Chow 1988). The mechanisms described in this paper can be used to define an appropriate convention, which can be called using something like

```
call "max-callee-save" gc();
```

in place of the call above.

The contribution of this paper is a set of abstractions that a compiler can use to describe calling conventions.

- We have developed new abstractions for specifying parameter passing (Section 3). Our specifications decouple parameter passing from other parts of the calling convention, are no more complex than those used in previous work (Bailey and Davidson 1995), and eliminate the need for a program generator apart from the compiler.
- We have developed a declarative technique for specifying the layout of a procedure's activation record or *frame* (Section 5). This technique makes it easy to decouple frame layout from the order in which different phases of the compiler execute, and it makes a frame-layout specification correspond closely to the sort of picture of

frame layout one finds in a typical architecture manual or calling-convention manual.

What if you don't care about customizing calling conventions? Our abstractions are simple enough that they should be useful even for implementing one calling convention on one platform, and they can describe existing, standard calling conventions cleanly. Although we have not yet performed experiments with custom calling conventions, we have used our abstractions to implement standard C calling conventions on the Alpha, Intel *x*86, and MIPS platforms. Implementations range from 90–130 lines of code each, divided about evenly among parameter passing, frame layout, and basic infrastructure such as importing modules and naming registers. We have tested these implementations using Bailey and Davidson's (1996) techniques, which our abstractions support. We have also used the same techniques to confirm that our compiler interoperates with the native C compiler, at least for procedure calls that can be expressed in the type systems of both compilers.

2 Background

Before presenting automata and frame layout, we provide some background on calling conventions and on C--.

Calling conventions

A calling convention is a contract among four parties: a calling procedure (the *caller*), a called procedure (the *callee*), a run-time system, and an operating system. All four parties must agree on how space will be allocated from the stack and how that space will be used: a procedure needs stack space for saved registers and private data, an operating system needs stack space to hold machine state ("signal context") when an interrupt occurs, and a run-time system needs to walk a stack to inspect and modify the state of a suspended computation. In addition to sharing the stack with the other two parties, a caller and callee must agree on how to pass parameters and results. This paper focuses on caller and callee; we consider run-time system or operating system only when they impose constraints on a procedure.

The most significant previous work is by Bailey and Davidson (1995, 1996), who first used automata to describe parameter passing, showed how to use such automata to test compilers (even compilers built without regard to automata), and presented a specification language, CCL, from which automata (and other information) could be derived. We found automata and automaton-based testing extremely helpful. CCL, however, did not meet our needs. It is fairly complex and is used before compilation to generate parts of the compiler. We prefer something simple enough to embed in the compiler and run at compilation time, so that the author of a front end can experiment with a new calling convention without having to rebuild the C-- compiler.

Another difficulty with CCL is that much of stack-frame layout lies outside its scope. For example, in the specification of the M88100 calling convention, the compiler writer must know the meaning of the special constants SPILL_SIZE, LOCALS_SIZE, NVSIZE, and ARG_BLOCK_SIZE, must supply values for three of the four, and must know that the frame size is *round_up*(SPILL_SIZE + LOCALS_SIZE + NVSIZE + ARG_BLOCK_SIZE, 8). Rather than adopt CCL, we use Bailey and Davidson's automata and testing methods, but we provide a new specification language and a new way of implementing frame layout.

C-- and Quick C--

C-- is intended as a target for a high-level–language compiler: the *front end.* C-- encapsulates a code generator—technology that is well understood but difficult to implement. The primary goal of C-- is to enable a code generator to be *reused.* Effective reuse requires that the author of a front end be able to control cost tradeoffs.

C-- comprises a language and a run-time interface; the language is used by the front end to request generated code, and the run-time interface is used by the front end's run-time system to support such language features as exception dispatch and garbage collection. Only the language is relevant to this paper.

The C-- language is designed around a register-transfer model of computation. This model is a refinement (Ramsey and Davidson 1998) of a model first used for compilation by Davidson and Fraser (1980); a similar model is used in the popular compiler `gcc`. C-- also provides a form of procedure, which may accept multiple parameters and return multiple results. Although there is no limit to the number of parameters or results any one procedure may accept or return, this number is fixed at compile time; C-- does not support varargs as it is understood in C.[1] C-- provides control flow between procedures using a construct called `cut to`, which is a bit like C's `longjmp` but can pass multiple parameters (Ramsey and Peyton Jones 2000).

C-- has just enough of a type system to help a compiler put values in machine registers: the type of a value is its width in bits. Some machines have more than one kind of register, and a good C-- compiler should put a value in the kind of register that best fits the way the value is used. For example, an intermediate result in a floating-point computation should be put in a floating-point register.

When a value is passed to a separately compiled procedure, the C-- compiler can't see how it is used, so the compiler needs help deciding what kind of register should hold it. We provide such help by using a *hint* on every actual and formal parameter in a C-- program. The hint enables the caller and callee not only to agree on what register should be used to pass the value but also to agree on an appropriate *kind* of register. The hints at a call site must match the hints at the declaration of the procedure called. The set of meaningful hints depends on what kinds of registers are available and is therefore machine-dependent. On the platforms our compiler supports, we use three hints: the `"float"` hint, which identifies a floating-point number; the `"address"` hint, which identifies a pointer; and the empty hint, which identifies an integer or other non-pointer data. For purposes of a calling convention, the type of a parameter is its width together with its hint.

Our work takes place in the context of Quick C--, a new compiler for C--. Quick C-- is modeled after *vpo*, the Very Portable Optimizer (Benitez and Davidson 1988), with one small difference. *Vpo* requires a front end to include a "code expander" that translates the front end's intermediate form into a register-transfer computation in which each register-level assignment can be expressed as a *single* instruction on the target machine. The Quick C-- com-

[1] A planned extension to C-- will support a variable number of arguments, but using a mechanism other than ordinary parameters. Unlike in C, an ordinary call site will not interoperate with a variadic procedure.

piler establishes the single-instruction property on its own, so a front end can simply translate its intermediate form into an arbitrary register-level computation without regard to the target instruction set. There is also a minor difference in implementations. *Vpo* is implemented entirely in C, but Quick C-- is implemented in two languages: Objective Caml (Leroy et al. 2001) is the main implementation language of the compiler, but the compiler and optimizer are configured using the embedded language Lua (Ierusalimschy, De Figueiredo, and Celes Filho 1996), and part of each calling convention is specified using Lua. Both languages appear in examples in this paper. Finally, *vpo* is a mature tool that produces excellent code. Quick C-- is immature and produces naïve code, but it does run hello world and the kinds of tests described by Bailey and Davidson (1996). Quick C-- may be downloaded from `www.cminusminus.org`.

3 Passing values between procedures

The contract embodied by a calling convention must say where to put the value of each parameter or result when control is passed from one procedure to another. In a C calling convention, a parameter is typically passed either in a register or in memory, but a large parameter may be split, passing part in register and part in memory, and some conventions may require that two copies of a parameter be passed: one in a register and one in memory. We therefore represent a *location* as an abstract object that supports read and write operations.

As do Bailey and Davidson (1995), we assume that all but finitely many parameters must be passed in contiguous, sequentially allocated locations in memory. Intuitively, these parameters are the parameters that don't fit in registers. We call the area from which they are allocated the *overflow block*. The assumption may seem restrictive, but it is satisfied by all calling conventions we know of, and it could easily be relaxed to accommodate multiple overflow blocks.

We also assume that it is possible to place parameters in locations by working from left to right. More formally, we assume that the location in which parameter k is passed depends only on the types (including sizes) of parameters $\leq k$. This assumption is not required by our translator or inherent in the nature of procedure calls, but it *is* inherent in the C calling convention because of variadic procedures (varargs). In a variadic C procedure, it must be possible to extract parameters one at a time using the `va_arg` macro. A convention that supports varargs must therefore work left to right. Even for other conventions, however, working left to right is convenient for both specification and implementation, and a more general model (in which the type of a later parameter can affect the placement of an earlier parameter) appears to offer no particular advantages.

Under these assumptions, we can follow Bailey and Davidson and use an *automaton* to allocate a location for each parameter. Each allocation request presents a parameter's type to the automaton, and we present the types left to right. An automaton includes a finite-state control plus an overflow block. The overflow block has infinitely many possible states, because its state includes the number of bytes allocated. The overflow block is the *only* place from which one can allocate arbitrarily many locations.

We use automata not only for allocation but also to generate a test suite (Bailey and Davidson 1996).

Client's view of an automaton

We represent an automaton as a value of type `Automaton.t`, which is a mutable abstraction. An `Automaton.t` supports two operations: `allocate` and `freeze`. The `allocate` operation takes a width and hint (which amount to a type in C--) and returns an abstract location. The `freeze` operation is called after all parameters are allocated. It returns the overflow block, the set of registers disbursed by previous calls to `allocate`, and information about memory locations used and the internal state of the overflow block. The overflow block is used in laying out the stack (Section 5), and the set of registers is used in liveness analysis. The other information is not used in the compiler but is used to implement Bailey and Davidson's testing technique.

A client of an automaton, either at a call site or in a procedure's prolog, starts with a list of parameter types and gets back a list of locations and an overflow block. Such a client presents the types to `allocate` one at a time, then calls `freeze`.

How to specify an automaton

We could specify any automaton by giving its nodes and edges. Such a specification would be both hard to write and hard to read, however: Bailey and Davidson (1996) report 9 nodes and 90 edges for a simple convention like the SPARC convention; the more complex MIPS convention takes 70 nodes and 772 edges. An alternative is to invent an abstract specification language such as CCL and to write a program generator that analyzes a specification and generates an automaton. CCL uses a few very powerful operators to talk about parameter passing, and the translation from CCL to an automaton is not obvious. We steer a middle course: we use abstraction, but our abstractions are simple constructors that build automata directly. Our specifications seem slightly larger than CCL specifications but still relatively readable. One benefit is that we can implement our constructors directly in the compiler, which obviates the need for a program generator and makes it possible to specify new conventions at compile time. Building the implementation into the compiler also means that if you must write special-purpose code for an unusual calling convention, you have all the power of your chosen implementation language. Whatever that language, it should be easy to re-implement our constructors in it.

Our idea is to build an automaton in *stages*. Each stage may satisfy an allocation request or pass it on to a succeeding stage. It is also possible for a stage to satisfy *part* of a request, e.g., when splitting a parameter between registers and memory. Each stage implements the `allocate` and `freeze` operations, but in a slightly different form than that used by a client.

- An internal `allocate` request includes not only the width and hint of a parameter but also the alignment needed for the parameter if it should be allocated to memory.
- An internal `freeze` request accumulates all the registers and memory locations used by preceding stages and passes this information to succeeding stages.

Because most automata include an overflow block, we begin with the overflow stage.

Overflow An overflow stage satisfies every request by allocating from the overflow block. Because it satisfies *every* request, it never passes a request to its successor, and therefore it is useless to give it a successor. To create an overflow stage, we supply the direction in which the overflow block should grow and the maximum alignment of the overflow block supported by the calling convention.

⟨*constructors for automaton stages*⟩≡

```
val overflow :
  growth:Memalloc.growth -> max_alignment:int ->
  stage
```

An automaton without an overflow stage can satisfy only finitely many allocation requests and so can support passing only finitely many values.

Selection and adjustment of width We define two width-related stages that satisfy no requests themselves, but only check or change requests before passing the requests to their successors. The `widths` stage restricts the automaton to satisfy only requests for one of an enumerated list of widths. It is useful for detecting internal errors in the compiler, e.g., passing a 16-bit value when the convention supports only wider values. When it receives a request, the stage `width ws` checks to see if the requested width is in the list `ws`. If so, it passes the request to its successor; if not, it halts the compiler with an error message.

⟨*constructors for automaton stages*⟩+≡

```
val widths : width list -> stage
```

Instead of halting with an error message, we can alter an allocation request to ask for a wider location. For example, we might embed a 16-bit value inside a 32-bit location. The `widen f` stage alters a request for a width `w` so it has a width `f w`, which must be at least as large as `w`.

⟨*constructors for automaton stages*⟩+≡

```
val widen : (width -> width) -> stage
```

The `widen f` stage requests a location l of width `f w` from its successor. If `f w` is larger than `w`, the `widen` stage builds a new, narrower location n that it returns to its client. A read from n is implemented by reading the wide value in l and narrowing the value read. A write to n is implemented by widening the value written and writing it into l. Widening and narrowing are done using either integer or floating-point operations, depending on the hint in the original allocation request.

Adjustment of alignment As noted above, a value that is allocated in memory needs an alignment. The alignment is 1 by default, but if the stage `align_to f` is used, the alignment is adjusted to `f w`, where `w` is the width of the request. The adjusted request is then passed to the successor of `align_to f`.

⟨*constructors for automaton stages*⟩+≡

```
val align_to : (width -> int) -> stage
```

It is possible that the functions used to build `align_to` and width stages should be generalized to use a request's hint, not just its width, to make their adjustments. We have not yet needed such generality.

Allocation of registers The most interesting stages are those that place arguments in registers. We have identified two policies that are used by common calling conventions: "the first n bits of arguments go in the first n bits of registers" and "the first n arguments go in the first n registers." We use separate stages to count bits or arguments and to allocate registers. A counter and allocator normally share a mutable cell containing an integer; such a cell has type `int ref`.

⟨*constructors for automaton stages*⟩+≡

```
val bitcounter : int ref -> stage
val argcounter : int ref -> stage
```

The stage `bitcounter n` doesn't allocate; given a request, it increments `n` by the width of the request and then passes the request to its successor. Similarly, `argcounter n` increments `n` by one and passes the request to its successor.

The stage `regs_by_bits n rs` uses the bit counter `n` to implement the "n bits of arguments to n bits of registers" policy. It allocates registers from list `rs`.

⟨*constructors for automaton stages*⟩+≡

```
val regs_by_bits : int ref -> Register.t list -> stage
```

Given a request, `regs_by_bits n rs` subtracts the width of the request from `n` to compute the number of bits already allocated. It skips as many registers from `rs` as are needed to account for bits already allocated, then uses the first remaining register to satisfy the allocation request. If no registers remain, it passes the request to the next stage.

What if the width of a request is different from the width of the first available register? If the request is too narrow for the first register, the stage could halt the compiler with a bug report (semantic checking failed to detect an unsupported width) or could widen the request. Widening the request is not the same as inserting a preceding `widen` stage; for example, to pass a 64-bit floating-point value on the Pentium, we want to use an 80-bit floating-point register if one is available, but request a 64-bit memory slot from the successor stage if no register is available. If the stage widens a request, it increases `n` to indicate that more bits were allocated than were actually requested.

If a request is too wide for the first register, it is reasonable to see if a combination of registers can satisfy the request. For example, a 64-bit request might be satisfied by splitting it into two 32-bit requests and using two 32-bit registers. Splitting requires a byte order to tell whether the first register carries the most significant or the least significant bits of the request.

It is also reasonable, if a request is large enough to exhaust registers, to use registers to satisfy as much of the request as possible, then request the remaining space from the next stage. For example, a 64-bit request might be satisfied using one 32-bit register and a 32-bit area in the overflow block.

The stage `regs_by_args n rs` uses the arg counter `n` to implement the "n arguments to n registers" strategy.

⟨*constructors for automaton stages*⟩+≡

```
val regs_by_args : int ref -> Register.t list -> stage
```

Here, it still seems sensible to widen a request that is too narrow, but it does not appear sensible to split a request across multiple registers.

It is sometimes possible to make a counter implicit. For the case in which `bitcounter n` is immediately followed by `regs_by_bits n rs`, we provide the shorthand `useregs rs`.

⟨*constructors for automaton stages*⟩+≡

```
val useregs : Register.t list -> stage
```

Choice among stages Many calling conventions pass different types of parameters in different kinds of registers. For example, according to the 32-bit MIPS calling convention for C, the first parameter is passed in integer register 4, unless it is a floating-point parameter, in which case it is passed in the floating-point register pair 12–13. We implement such a rule by providing a "choice" stage, which uses the hint and width of an allocation request to decide which alternative stage should satisfy the request. We create a choice stage by passing the `choice` function a list of (predicate, stage) pairs.

⟨*constructors for automaton stages*⟩+≡
```
val choice :
  ((width -> hint -> bool) * stage) list -> stage
```

A choice stage works a bit like a Lisp `cond`; when a request reaches the stage, it evaluates the predicates one at a time, and it behaves as the first stage whose predicate is satisfied. For convenience in creating predicates, we provide functions `is_width`, `is_hint`, and `is_any`.

Arbitrary state transition Sometimes the location of a parameter may depend on the width or hint of a previous parameter. On the MIPS, for example, if the second parameter is a floating-point parameter, its placement depends on the type of the *first* parameter. We solve this problem by introducing history: the automaton makes a permanent state transition on the relevant parameter.

⟨*constructors for automaton stages*⟩+≡
```
val first_choice :
  ((width -> hint -> bool) * stage) list -> stage
```

The stage `first_choice l` is like `choice l`, except that the choice is made once, at the time of the first request that reaches the stage, instead of each time a request reaches the stage. After the first request, the `first_choice l` stage behaves as the chosen stage from then on. An example appears in Appendix B.

Having `first_choice` makes our system general: because stages can be chained and because parameter-passing automata do not contain cycles, we can define any needed node-edge structure using `first_choice` and `useregs`.

Example automata

We present automata used to implement standard C calling conventions. We build each automaton by composing stages of type `stage`. To connect stages, we use the composition operator `*>`; this infix associative operator takes two stages and produces a stage. When all stages have been combined with `*>`, we build a full automaton using `Automaton.at start stages`, where `start` is the starting address of the overflow block and `stages` are the combined stages. In our examples, we show only the combined stages.

On the Pentium, arguments are passed on the stack, in the overflow block, which is aligned on a 4-byte boundary. Each request is rounded up to a multiple of 32 bits.

⟨*x86 automata*⟩≡
```
let c_arguments () =
 widen (Aux.round_up_to ~multiple_of:32) *>
 overflow ~growth:Memalloc.Up ~max_alignment:4
```

The ~ is Objective Caml syntax that names an actual parameter in a function call.

The Pentium's return convention requires a choice operation: a floating-point result is returned in floating-point register 0 (written `f 0`), but an integer result is returned in integer register EAX. There is no overflow block.

⟨*x86 automata*⟩+≡
```
let c_results () =
 choice
  [ is_hint "float",
     widen (Aux.round_up_to ~multiple_of:80) *>
     widths [80] *> useregs [f 0];
   is_any,
     widen (Aux.round_up_to ~multiple_of:32) *>
     widths [32] *> useregs [eax]
  ]
```

On the Alpha, the first six words' worth of parameters go in registers, and the rest go in the overflow block. Floating-point parameters go in floating point registers 16–21; other parameters go in integer registers 16–21. Choice of register is by offset of the relevant parameter; because both integer and floating registers use the same offset, we need an explicit counter.

⟨*alpha automata*⟩≡
```
let c_arguments () =
 let ctr = ref 0 in (* allocate a fresh counter *)
 bitcounter ctr *>
 choice
  [ is_hint "float",
     widths [64] *>
     regs_by_bits ctr
      [f 16; f 17; f 18; f 19; f 20; f 21];
   is_any,
     widen (Aux.round_up_to ~multiple_of:64) *>
     regs_by_bits ctr
      [r 16; r 17; r 18; r 19; r 20; r 21]
  ] *>
 overflow ~growth:Memalloc.Up ~max_alignment:16
```

This convention, like many C conventions, leaves some registers unused. For example, suppose a procedure takes two floating-point parameters and an integer parameter, each 64 bits wide. When the first request comes in, with hint `float` and width 64, the `bitcounter` stage adds 64 to `ctr`. The `choice` stage directs the request its first alternative, which verifies the request is 64 bits wide and passes it to the first `regs_by_bits` stage. This stage subtracts the current width from `ctr` to determine that 0 bits have been allocated so far, so it drops nothing from its list of registers and returns floating-point register `f 16`. The second parameter, also a floating-point parameter, takes the same path, but the `regs_by_bits` stage drops `f 16` and returns `f 17`. The third parameter is an integer parameter, with an empty hint, so it takes the *second* choice. The `widen` stage has no effect because the 64-bit request is already a multiple of 64 bits, and the second `regs_by_bits` stage, still using the same `ctr`, determines that 128 bits have been allocated already, so it drops integer registers 16 and 17, which are not used, and it returns integer register `r 18`.

We have also implemented the standard C convention for the MIPS, which uses `regs_by_args` for floating-point parameters and `regs_by_bits` for integer parameters. The MIPS argument automaton appears in Appendix B.

4 Frame-layout folklore

A stack frame includes not only overflow parameters but also user variables, saved registers, and other data. Locations must be allocated and laid out in a way that is consistent with the calling convention. There are simple textbook techniques that work well for C, but these techniques don't support proper tail calls. If proper tail calls are supported, the stack pointer moves, and care is required to avoid dedicating two registers to point to the stack frame. We have found surprisingly little discussion of these issues in the literature. This section sketches the problems and our solutions.

Managing overflow blocks

To describe a stack, we use terminology that is independent of the direction of stack growth. If f calls g, f's frame is next to g's frame on the stack, but f's frame is older. We therefore refer to the two ends of the stack as the *old* and *young* ends. The young end is the end at which the stack grows and shrinks. We also refer to the old or young end of an individual frame.

When a procedure f calls another procedure g, passing parameters, they share the space that holds the overflow block. This space is allocated and initialized by the caller, f, but read by the callee, g. The shared space appears at the young end of f's frame and the old end of g's frame. The caller, f, *allocates* this space, but who *deallocates* it? One possibility is for g to deallocate this space, perhaps after using it for private purposes. The other possibility is for g to let f deallocate or reuse the space. The choice depends on the calling convention.

Most conventions for C leave it to f to deallocate this space, because a caller can easily reuse the *same* space to pass outgoing overflow parameters at multiple call sites. (A space used in this way is sometimes called an *argument-build area.*) The caller sets the stack pointer *once*, on entry, and does not adjust it afterward. Unfortunately, caller-deallocates is incompatible with *proper tail calls*.

A proper tail call is a bit like a return and call in one. If f calls g and g properly tail-calls h, g's frame disappears, and when h returns it returns directly to f. A proper tail call recovers g's stack space before passing control to h, so an arbitrarily long sequence of proper tail calls uses only constant stack space (Clinger 1998).

If f calls g and g properly tail-calls h, h must deallocate any overflow parameters it receives from g. To see why, suppose that f passes only a few overflow parameters to g, but g passes many overflow parameters to h. When h returns to f, the best f can do is to deallocate the small overflow area it passed to g, not the large area that g passed to h. In other words, if a caller makes a tail call, it is no longer around to deallocate anything, so the job of deallocation must fall to the callee. This means when control returns to f, the stack pointer has *moved.* Probst (2001) discusses this issue at length and presents a nonstandard convention for C that supports both tail calls and varargs functions.

Even when the stack pointer moves, we find it convenient to have the stack pointer return to a single location between calls. In Quick C`--`, we call this location a *stable* location.

The problem of overflow parameters has a dual: overflow results. Overflow results occur when a procedure returns more values than can fit in registers. (Unlike C, C`--` permits a procedure to return arbitrarily many results in registers.) Like incoming overflow parameters, outgoing overflow results appear at the old end of a stack frame. The overflow-result space must be *deallocated* by the caller, but it may be *allocated* either by caller or callee. The choice appears to be independent of tail calls.

Addressing and allocating slots

We call a location in the stack frame a *slot.* How can we refer to a slot? If the stack pointer moves, the textbook solution is to express a slot's address as an offset from a *frame pointer.* The frame pointer is an extra register that provides a fixed base through which to address slots.

Every compiler must have a way to allocate a slot and provide its address. A typical strategy, which is used in the `lcc` compiler (Fraser and Hanson 1995), is to allocate and deallocate slots last in, first out. The compiler uses the *order* of allocation and deallocation requests to signal both the *layout* of slots in the frame and the *lifetime* of each slot. Because deallocation is last in, first out, lifetimes must nest, and the compiler must be structured to make the order of requests match the lifetimes. This requirement is not too onerous for a simple C compiler: for example, the lifetimes of variables in an inner compound statement (i.e., declared within `{`$\cdots$`}`) nest within the lifetimes of other variables.

Last-in, first-out allocation is easy to implement using two pointers: the *cursor* tracks the location of the next free slot, and the *high-water mark* tracks the maximum number of slots ever allocated at one time. C conventions typically require the cursor to start at the old end of the frame, near where the frame pointer points, and move toward the young end, where the stack pointer points.

The main drawback of last-in, first-out slot allocation is that when the calling convention constrains the order in which parts of the stack frame must be laid out, those constraints also affect the order in which different parts of the compiler may execute. Again, these constraints are not too onerous for a simple C compiler, because C calling conventions are designed with this compilation strategy in mind. Another drawback is that the frame-layout code can be difficult to understand and to change, and it is not obvious how to check whether it is consistent with the pictures of frame layout one sees in manuals. These drawbacks become problematic when one wants to support many calling conventions in one compiler. In Section 5, we present an alternative that decouples the order of execution from the layout of the stack frame.

Eliminating the frame pointer

Using two registers to point to one frame is costly; most serious compilers use a *virtual frame pointer* instead. A virtual frame pointer supports last-in, first-out allocation: the cursor starts at the virtual frame pointer and moves toward the young end of the fame.

If the stack pointer does not move, the virtual frame pointer is simply the stack pointer plus the frame size. Because the frame size is not known until frame layout is complete, it is represented as a symbolic constant. We call such a constant a *late compile-time constant.* When the stack pointer moves, we need to define the virtual frame pointer differently. In Quick C`--`, the virtual frame pointer is the value of the stack pointer at procedure entry. This definition allows the stack pointer to move.

Because the virtual frame pointer is not a real register, we cannot emit code that refers to it. If the virtual frame pointer is VFP and the stack pointer is SP, we can always replace VFP by $\mathit{SP} + k$ for some k, but the k may be different from instruction to instruction. Computing k is a simple forward dataflow problem. On entry, k is zero, and k changes only when the stack pointer is modified. To solve the dataflow problem, we rely on the following invariants:

- Every addressing expression that refers to a stack slot is written $\mathit{VFP} + n$, where n is a (possibly late) compile-time constant. The stack pointer is never used in an addressing expression.
- The virtual frame pointer is immutable, so VFP never appears in an lvalue context.
- The stack pointer is modified in only two ways: absolute and relative. An absolute modification has the form $\mathit{SP} \leftarrow \mathit{VFP} + k$; a relative modification has the form $\mathit{SP} \leftarrow \mathit{SP} + k$.

Given these invariants, it is easy to write code that, in one pass, computes k such that $VFP = SP + k$ and replaces VFP by $SP + k$. The same k should be computed on each edge into a join point; if not, there is a bug in the compiler.

The virtual frame pointer enables us to refer to each slot by an expression of the form $VFP + n$. To help compute n, we introduce the *block* abstraction.

5 Frame layout by block composition

We want to make it easy for different parts of the compiler to allocate slots in any order, independent of frame layout. For each procedure, we create a set of *allocation areas*, each of which will hold a collection of related slots. For example, an area may hold spilled registers, user data stored on the stack, exception-handling information, or any other data that is private to the back end of the compiler. Our compiler uses at least one area for each of these categories. Each area is implemented using a simple cursor; no slot is ever deallocated. Any part of the compiler can allocate a slot in any area at any time, and the allocator returns a slot address that is an expression involving a late compile-time constant and the virtual frame pointer.

When slot allocation is complete, we convert each area into a *block*. A block is simply a contiguous region of memory that holds a collection of slots. Our Bailey/Davidson automata also produce blocks, which hold overflow parameters or results.

Once we have a procedure's blocks, we compose them to form the stack frame. At this point, the location of each block is known, and we can compute the exact offset of each slot. To explain how blocks are composed, we present two views of blocks: the *external* view, which is used by parts of the compiler that create blocks, and the *internal* view, which is used to lay out the stack frame.

In the external view, each block has a *size*, an *alignment*, and a *base address*. The size and alignment are integer values, but the base address is an expression, not a value; it typically has the form $VFP+n$, where n is a late compile-time constant.[2]

Blocks may be composed by *concatenation* or *overlapping*. If two blocks are concatenated, they occupy contiguous but distinct locations in the stack frame, and they may be live at the same time. It is typical, for example, to concatenate blocks for spilled registers and for user data stored on the stack. If two blocks are overlapped, they *share* locations in the stack frame, and they must never be live at the same time. It is typical, for example, to overlap blocks for outgoing overflow parameters at different call sites.

The result of composing two blocks is itself a block; the rules are straightforward. For example, if two blocks b_{hi} and b_{lo} are concatenated, the resulting block has the address of b_{lo}, an alignment that is the least common multiple of the alignments of b_{hi} and b_{lo}, and a size that is the sum of the sizes of b_{hi} and b_{lo}, or possibly larger if padding is needed to satisfy requirements for alignment. Equally simple rules apply to overlapping, which comes in two forms: line up either the high ends or the low ends of the overlapped blocks. Complete rules are given in Appendix A.

In the internal view of a block, everything is as in the external view, plus each block is augmented with a set of *constraints*. These constraints, which take the form of equations, relate addresses as prescribed by any overlapping or concatenation operations that have been used to form the block. In the example above, if we concatenate b_{hi} and b_{lo}, the resulting block includes the constraint $address(b_{hi}) = address(b_{lo}) + round\_up(size(b_{lo}), alignment(b_{hi}))$.

To compose blocks into a stack frame, we need to know what blocks we are working with. For each procedure, Quick C-- creates these blocks:

- List `oldblocks` is a list of blocks that occur at the old end of the stack frame. These are typically incoming overflow parameters and outgoing overflow results.
- List `youngblocks` is a list of blocks that occur at the young end of the stack frame. These are typically outgoing overflow parameters and incoming overflow results.
- Block `vfp` is an empty block whose address is the virtual frame pointer, and block `stable_sp` is an empty block whose address is the "stable" location of the stack pointer.
- Block `stackdata` contains user data that is allocated on the stack. Blocks `continuations` and `spills` hold the compiler's private data: one for the exception mechanism and one for the register allocator.

To compose these blocks, we use Lua, which is a simple imperative language with roughly Pascal-like syntax (Ierusalimschy, De Figueiredo, and Celes Filho 1996). The Lua versions of our block-composition primitives support lists of blocks, not just pairs of blocks.

`Block.relative(`b`, `$name$`, `s`, `a`)`

Make a new block whose base address is $p+n$, where p is the base address of block b and n is a fresh, late compile-time constant whose name contains the string *name*. The parameters s and a give the size and alignment of the new block.

`Block.overlap(`w`, `$which\_end$`, `bs`)`

Overlap the blocks in the list bs at either their high or low ends as specified by *which_end*. If bs is empty, create an empty block whose address is w bits wide.

`Block.cat(`w`, `bs`)`

Concatenate the blocks in the list bs. If bs is empty, create an empty block whose address is w bits wide.

As an example, Figure 1a shows the part of our compiler-configuration code that specifies stack layout for a C calling convention on the Intel $x86$ platform. The $x86$ layout function creates and uses the nonstandard block `blocks.ra` to mark the location of the return address. The call to `Block.relative` creates a new block with size and alignment 4 and with an address relative to `blocks.vfp`, i.e., an address that is the sum of a fresh late compile-time constant and the address of `blocks.vfp`. The function then overlaps the blocks that can safely be overlapped and concatenates the rest using the variable `layout`; the curly braces and commas are Lua syntax for a list literal. The resulting block looks like the picture in Figure 1b. We make this block the stack frame by calling `Stack.freeze`.

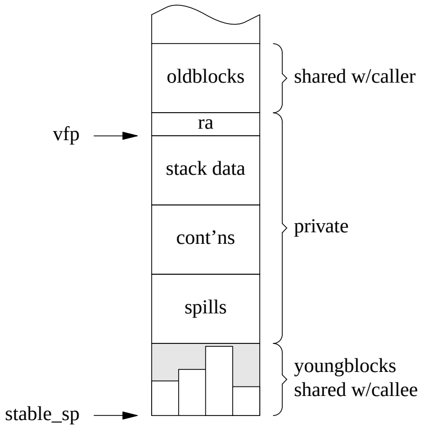
The function `Stack.freeze` takes a single block and passes the block's constraints to an equation solver in the style of Ramsey (1996) or Knuth (1986, §585). Subject to constraints of overlapping, concatenation, alignment, and size, the solver computes the position of each block in the

[2] A block may also be used to represent exception-handling information. In the frame where the exception is handled, this is an ordinary block that is part of the stack frame, and its address is relative to the virtual frame pointer. In the frame where the exception is raised, however, the block is part not of the current frame but of the faraway frame where the handler is. In this case, the address of the block is the value that points to the handler, and this address is *not* relative to the virtual frame pointer.

```
⟨x86 stack layout⟩≡
  function X86.layout["C"](_,proc)
      local blocks = Stack.blocks(proc)
      blocks.ra = Block.relative(blocks.vfp, "return address", 4, 4)

      blocks.oldblocks   = Block.overlap(32, "low", blocks.oldblocks)
      blocks.youngblocks = Block.overlap(32, "low", blocks.youngblocks)
      ⟨code to dump blocks if Debug.stack is set⟩
      local layout =
                  { blocks.oldblocks               -- <-- high addresses
                  , blocks.ra
                  , blocks.vfp
                  , blocks.stackdata
                  , blocks.continuations
                  , blocks.spills
                  , blocks.youngblocks
                  , blocks.stable_sp               -- <-- low addresses
                  }
      local block = Block.cat(32, layout)
      Stack.freeze(proc,block)
      return 1
  end
```

(a)



(b)

Figure 1: Stack-layout specification and picture for Intel $x86$

stack frame, and it computes the values of all the late compile-time constants. Since the location of each slot is expressed using such a constant, `Stack.freeze` then substitutes the correct value of each constant in each instruction throughout the procedure.

The example in Figure 1 is for a standard C calling convention; for example, the young blocks are overlapped at the low end and are placed above the stable location of the stack pointer, which indicates that outgoing overflow parameters are deallocated by the caller. The layout for a convention that supports tail calls is somewhat different. The full C-- language permits a procedure to contain call sites for many different kinds of conventions; in this case, the details are more complex because we have to split the young blocks into two groups depending on which conventions do and do not support tail calls.

6 Completing the convention

Value passing and stack layout are not all there is to a calling convention. A complete specification requires additional information. In Quick C--, we specify a calling convention using a record containing these fields:

- Three value-passing automata, which are described by constructors from Section 3: one automaton each for call, return, and `cut to`
- A stack-layout function, such as is shown in Figure 1a, that composes the blocks of the stack frame
- The set of registers managed by the register allocator and the subset that must be preserved across a call
- A function used to save a register that must be preserved, as described below
- The direction of stack growth, the register used as the stack pointer, and the alignment required of the stack pointer at a call
- The identity of the party (caller or callee) that deallocates overflow parameters and the identity of the party that allocates overflow results
- Information about where the return address is passed and saved

Adding a new calling convention is as easy as defining such a record. The compiler looks up the record by name at each call site and procedure definition. In the record, we use a convention-dependent function to save registers: although the compiler can save registers anywhere, the run-time system may need for them to be saved in conventional locations.

The contract with the run-time system A C-- run-time system must be able to walk the stack and recover values of callee-saves registers. C-- actually has a stronger requirement: the run-time system must be able to find the value of any live variable, not just a callee-saves register. We use a single mechanism for both; this mechanism remembers the (PC-dependent) location of each variable and callee-saves register. But in a compiler for a simpler language, such as C, the calling convention requires that callee-saves registers be saved in conventional locations. We have not implemented this convention, but we sketch an implementation based on blocks. We would use a table, indexed by callee-saves register, to store for each register both a one-slot block and a Boolean that indicates if the register is spilled. When a callee-saves register is spilled, we would use this table to find the spill location. We would also record in the table the fact of the spill. Finally, we would expose the table to the Lua code in the stack-layout function, which would include in the stack frame, in the conventional places, only those blocks that are actually used to hold spilled callee-saves registers.

7 Discussion

Although we expected parameter passing and stack layout to be the central problems of implementing a calling convention, we also expected that they would be separate problems requiring different solutions. We have been surprised by the similarities. Both problems require allocation of locations, and each allocator is best expressed as an imperative abstraction. Both kinds of allocation result in fixed-sized blocks, which are best composed into a stack layout using a declarative abstraction. The major differences are in the complexity of the allocators. The stack-slot allocators are so simple that we barely describe them in this paper. The parameter-passing allocators—the automata—are so complicated that it is not obvious how best to create them.

Automaton constructors We create an automaton as a sequence of stages, where each stage is built using a constructor. The constructors `first_choice` and `useregs` are general in the sense that if we have an automaton in mind, we can use `first_choice` and `useregs` to specify the nodes and edges. (For full generality we would need a constructor that allocates a memory location *outside* the overflow block, but we have not found a convention that needs such a constructor.) But automata for C conventions have many nodes and edges, and automata for conventions not constrained to support varargs have even more nodes and edges. The constructors `choice`, `regs_by_bits`, and `regs_by_args` make specifications easier to read, write, and understand.

Our other significant automaton constructor is `widen`. The `widen` stage does not affect the node/edge structure of the automaton; it is used only to adapt a large location so it can hold a small value. (It is the *request* that is widened; the location used to satisfy the request is narrowed.) We include the widening operation directly in the automaton, but this operation could be pushed into another part of the compiler, or in a system that does not keep track of widths, ignored completely. For example, in *vpo*, the parameter-passing automaton provides a location that is at least as wide as is needed to hold the parameter. If the location is wider than is needed, it is up to another part of the compiler to figure out how to widen the value passed.[3]

Our automaton constructors may be viewed as *parsing combinators* whose input is a sequence of types. While we specify choice using explicit predicate functions, classic parsing combinators handle choice using a success/failure model and a choice operator that takes two parsers and returns a parser (Hutton 1992). We could easily adopt this model, but we think our `choice` operator and predicates will be more accessible to a compiler writer who has not seen parsing combinators.

We also tried constructors that use an *exclusion relation.* Such a relation might say, for example, that when integer register 5 is allocated, floating-pointer register 12 is marked unavailable. Although Bailey and Davidson (1995) use exclusion to describe the original MIPS calling convention, we were unable to make exclusion work for today's MIPS calling convention. One case that is especially difficult is a C procedure expecting three parameters with types `double`, `int`, and `double`. Our experience suggests that exclusion is less useful than `regs_by_bits` and `regs_by_args`.

Block composition The block abstraction is, as far as we know, new. It works because it is simple, because it is applicative rather than imperative, and because it supports specifications that correspond closely to pictures of stack layout found in programming manuals. The key idea is to separate imperative allocation of slots from applicative composition of blocks; an earlier version that supported both imperative and applicative operations was much more difficult to understand. Because each phase of our compiler has its own imperative allocator and its own block(s), phases can easily execute in any convenient order.

Our compiler reuses stack slots only when it finds entire blocks that don't interfere. A more aggressive optimizing compiler might use register-allocation techniques to maximize reuse of slots. Such a compiler could still use blocks, but the stack-slot allocator would probably use a single allocation area, which would get get turned into a single block.

The trickiest part of programming with blocks is giving each block a suitable base address. We give each block an address of the form $VFP{+}n$, where n is a *fresh* late compile-time constant with an automatically generated name. We carefully avoid writing code that uses the name of such a constant. In an earlier version of our compiler, we often shared names of such constants; for example, two different modules in the compiler might both use the address VFP + `stackdata`. Such code led to frequent bugs in the compiler, perhaps because there is no programming-language mechanism to control access to such names. In our current compiler, when we need to share information among modules, we give all modules access to a relevant block, and the shared information is the address of that block. The new style seems to result in fewer bugs.

When stack layout does go wrong, the equation solver used in `Stack.freeze` is unable to solve the block equations: either there are more unknown late compile-time constants than there are equations, or the equations are inconsistent. Debugging these problems requires care, since a late compile-time constant is a bit like a global variable: it is hard to tell where it comes from. Good naming helps; each constant's name has a distinctive root that is mnemonic and can be searched for in the compiler's sources. It is also useful for the frame-layout function to print all blocks, with constraints, and for the constraint solver to indicate unconstrained or overconstrained variables. If all else fails, we selectively turn off phases of the compiler or remove blocks from the layout until the problem is identified.

Related work Our work is inspired and informed by Bailey and Davidson's (1995) work on automata for parameter passing. We have not adopted their specification language, CCL, because it is hard to apply to compiler construction—a separate program generator is needed, and the semantics of CCL are difficult to reconstruct without studying that program generator. Our contribution has been to find a new way to specify a Bailey/Davidson automaton: a way that is easily included in a compiler and needs no program generator. For comparison, our constructors are implemented in about 450 lines of Objective Caml; the interpreter and program generator for CCL are about 2500 lines of Icon.

Our approach to frame layout is different from Bailey and Davidson's. One significant difference is how we account for allocation of a new stack frame at a call. CCL treats the movement of the stack pointer as a "view change," which renames all locations on the stack. In our solution, there is no view change. Instead, we give special status to the value of the stack pointer at procedure entry, calling it the virtual frame pointer. Because the virtual frame pointer never changes, the meaning of a reference relative to the virtual frame pointer is fixed: no matter how the stack

[3] Private communication from Jack Davidson, 2 Oct 2002.

pointer moves, our view never changes. We find this model easier to work with than the view-change model.

The other significant difference is in the frame-size computation. In CCL, this computation must be written explicitly by the specifier. It is not clear whether it can be connected, even in principle, to the code that allocates slots inside the compiler—instead, the compiler writer must enforce the conventional layout constraints and guarantee they are consistent with the frame-size computation. In our solution, the frame-size computation is implicit in the composition of the blocks that make up the frame. Moreover, because each slot lies in a particular block, the relative positions of slots in the frame are automatically guaranteed to conform to the layout specified by the composition. Both layout and size are described and can be changed in exactly one place.

We get extra flexibility by writing the stack-layout specification in an embedded interpreted language. We plan for this language to support automaton constructors and calling-convention records as well, so we will be able to use Lua to specify a new calling convention at compile time without touching the Quick C-- compiler. We hope that the flexibility of Lua will make it easy for authors of front ends to undertake comparative experiments in the style of Davidson and Whalley (1991) and thereby to find custom calling conventions that maximize performance.

Acknowledgements John Dias, Dan Grossman, Glenn Holloway, Andi Krall, Simon Peyton Jones, Mike Smith, and David Whalley provided helpful comments on this manuscript.

References

Appel, Andrew W. and Zhong Shao. 1992. Callee-save registers in continuation-passing style. *Lisp and Symbolic Computation*, 5(3):189–219.

Bailey, Mark W. and Jack W. Davidson. 1995 (January). A formal model and specification language for procedure calling conventions. In *Conference Record of the 22nd Annual ACM Symposium on Principles of Programming Languages*, pages 298–310.

———. 1996 (May). Target-sensitive construction of diagnostic programs for procedure calling sequence generators. *Proceedings of the ACM SIGPLAN '96 Conference on Programming Language Design and Implementation,* in *SIGPLAN Notices*, 31(5):249–257.

Benitez, Manuel E. and Jack W. Davidson. 1988 (July). A portable global optimizer and linker. *Proceedings of the ACM SIGPLAN '88 Conference on Programming Language Design and Implementation,* in *SIGPLAN Notices*, 23(7):329–338.

Chow, Fred C. 1988 (July). Minimizing register usage penalty at procedure calls. *Proceedings of the ACM SIGPLAN '88 Conference on Programming Language Design and Implementation,* in *SIGPLAN Notices*, 23(7):85–94.

Clinger, William D. 1998 (May). Proper tail recursion and space efficiency. *Proceedings of the ACM SIGPLAN '98 Conference on Programming Language Design and Implementation,* in *SIGPLAN Notices*, 33(5):174–185.

Davidson, Jack W. and Christopher W. Fraser. 1980 (April). The design and application of a retargetable peephole optimizer. *ACM Transactions on Programming Languages and Systems*, 2(2):191–202.

Davidson, Jack W. and David B. Whalley. 1991. Methods for saving and restoring register values across function calls. *Software—Practice & Experience*, 21(2):459–472.

Fraser, Christopher W. and David R. Hanson. 1995. *A Retargetable C Compiler: Design and Implementation.* Redwood City, CA: Benjamin/Cummings.

Hutton, Graham. 1992 (July). Higher-order functions for parsing. *Journal of Functional Programming*, 2(3):323–343.

Ierusalimschy, Roberto, Luis H. De Figueiredo, and Waldemar Celes Filho. 1996 (June). Lua — an extensible extension language. *Software—Practice & Experience*, 26(6):635–652.

Knuth, Donald E. 1986. *Computers & Typesetting.* Volume D, METAFONT*: The Program.* Addison-Wesley.

Leroy, Xavier, Damien Doligez, Jacques Garrigue, Didier Rémy, and Jérôme Vouillon. 2001. *The Objective Caml system release 3.04: Documentation and user's manual.* INRIA. Available at `http://pauillac.inria.fr/ocaml/htmlman/`.

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

Peyton Jones, Simon L., Norman Ramsey, and Fermin Reig. 1999 (September). C--: a portable assembly language that supports garbage collection. In *International Conference on Principles and Practice of Declarative Programming*, Vol. 1702 of *LNCS*, pages 1–28. Springer Verlag.

Probst, Mark. 2001 (February). Proper tail recursion in C. Diplomarbeit Thesis, Institute for Computer Languages, Technical University of Vienna.

Ramsey, Norman. 1996 (April). A simple solver for linear equations containing nonlinear operators. *Software—Practice & Experience*, 26(4):467–487.

Ramsey, Norman and Jack W. Davidson. 1998 (June). Machine descriptions to build tools for embedded systems. In *ACM SIGPLAN Workshop on Languages, Compilers, and Tools for Embedded Systems (LCTES'98)*, Vol. 1474 of *LNCS*, pages 172–188. Springer Verlag.

Ramsey, Norman and Simon L. Peyton Jones. 2000 (May). A single intermediate language that supports multiple implementations of exceptions. *Proceedings of the ACM SIGPLAN '00 Conference on Programming Language Design and Implementation,* in *SIGPLAN Notices*, 35(5):285–298.

A Equational specifications for blocks

A block is created by specifying its base address, alignment, and size; a newly created block has no constraints. Figure 2 shows the equations that govern composition of blocks by concatenation and overlapping.

B C argument convention for the MIPS

The MIPS has a complex calling convention. To add to the confusion, the convention has changed over the lifetime of the architecture; Bailey and Davidson (1995) describe the old convention, for example.

Floating-point parameters are allocated by argument number, but other parameters are allocated by bit offset. We therefore need both a `bitcounter` and an `argcounter` stage. Because the type of the first parameter affects the rules for passing later parameters in registers, we also need a `first_choice` stage.

```
⟨mips automata⟩≡
  let arguments () =
    let bits = ref 0 in
    let arg  = ref 0 in
    widen (Aux.round_up_to ~multiple_of: 32) *>
    bitcounter bits *>
    argcounter arg  *>
```

$$
\begin{aligned}
\mathit{address}(\mathit{cat}(\mathit{hi},\mathit{lo})) &= \mathit{address}(\mathit{lo})\\
\mathit{alignment}(\mathit{cat}(\mathit{hi},\mathit{lo})) &= \mathit{lcm}(\mathit{alignment}(\mathit{hi}),\mathit{alignment}(\mathit{lo}))\\
\mathit{size}(\mathit{cat}(\mathit{hi},\mathit{lo})) &= \mathit{round\_up}(\mathit{size}(\mathit{lo}),\mathit{alignment}(\mathit{hi})) + \mathit{size}(\mathit{hi})\\
\mathit{constraints}(\mathit{cat}(\mathit{hi},\mathit{lo})) &= \mathit{constraints}(\mathit{hi}) \cup \mathit{constraints}(\mathit{lo}) \cup\\
&\qquad \{\mathit{address}(\mathit{hi}) = \mathit{address}(\mathit{lo}) + \mathit{round\_up}(\mathit{size}(\mathit{lo}),\mathit{alignment}(\mathit{hi}))\}
\end{aligned}
$$

$$
\begin{aligned}
\mathit{address}(\mathit{overlap}(\mathtt{Low},x,y)) &= \mathit{address}(x)\\
\mathit{alignment}(\mathit{overlap}(\mathtt{Low},x,y)) &= \mathit{lcm}(\mathit{alignment}(x),\mathit{alignment}(y))\\
\mathit{size}(\mathit{overlap}(\mathtt{Low},x,y)) &= \mathit{max}(\mathit{size}(x),\mathit{size}(y))\\
\mathit{constraints}(\mathit{overlap}(\mathtt{Low},x,y)) &= \mathit{constraints}(y) \cup \mathit{constraints}(x) \cup \{\mathit{address}(y) = \mathit{address}(x)\}
\end{aligned}
$$

$$
\begin{aligned}
\mathit{address}(\mathit{overlap}(\mathtt{High},x,y)) &= \mathit{address}(x),\ \text{if } \mathit{size}'(x) > \mathit{size}'(y)\\
\mathit{address}(\mathit{overlap}(\mathtt{High},x,y)) &= \mathit{address}(y),\ \text{if } \mathit{size}'(x) \le \mathit{size}'(y)\\
\mathit{alignment}(\mathit{overlap}(\mathtt{High},x,y)) &= \mathit{lcm}(\mathit{alignment}(x),\mathit{alignment}(y))\\
\mathit{size}(\mathit{overlap}(\mathtt{High},x,y)) &= \mathit{max}(\mathit{size}'(x),\mathit{size}'(y))\\
\mathit{constraints}(\mathit{overlap}(\mathtt{High},x,y)) &= \mathit{constraints}(y) \cup \mathit{constraints}(x) \cup\\
&\qquad \{\mathit{address}(y) + \mathit{size}'(y) = \mathit{address}(x) + \mathit{size}'(x)\}
\end{aligned}
$$

where $\mathit{size}'(b) = \mathit{round\_up}(\mathit{size}(b), \mathit{alignment}(b))$ and $\mathit{lcm}(n, m)$ is the least common multiple of n and m.

Figure 2: Equations for block composition

```
    first_choice
    [ is_hint "float",
        ⟨automaton F: first parameter is floating-point⟩
    ; is_any,
        ⟨automaton R: first parameter is not floating-point⟩
    ] *>
    overflow ~growth:Memalloc.Up ~max_alignment:16
```

When the first parameter is *not* a floating-point parameter, we mostly pack parameters into integer registers 4–7, regardless of type. The exception is that a 64-bit floating-point parameter in the second position goes into integer register pair 6–7, regardless of the size of the first parameter.

⟨*automaton R: first parameter is* not *floating-point*⟩≡

```
  choice
  [ (fun width hint -> hint = "float" && width = 64),
      regs_by_args arg [rpair 4; rpair 6];
        (* r4/r5 is a placeholder; uses pair r6/r7 *)
    is_any,
      regs_by_bits bits [r 4; r 5; r 6; r 7]
  ]
```

When the first parameter *is* a floating-point parameter, things are more sane and simple. The first two floating-point parameters go into floating-point registers 12 and 14. Depending on the size of the parameter, we use either a single register or a register pair. Remaining floating-point parameters, as well as any other kinds of parameters, are placed first in integer registers and then in the overflow block.

⟨*automaton F: first parameter is floating-point*⟩≡

```
  choice
  [ is_hint "float",
      choice
      [ is_width 64, regs_by_args arg [d 12; d 14];
        is_any     , regs_by_args arg [f 12; f 14]
      ];
    is_any,
      unit (* a stage that always passes to successor *)
  ] *>
  regs_by_bits bits [r 4; r 5; r 6; r 7]
```

This specification is 28 lines of Objective Caml; for comparison, the CCL description of the older, simpler MIPS convention is 27 lines of CCL.