**Reprinted from** *Proceedings of the 33rd ACM Symposium on the Principles of Programming Languages* **(POPL'06)**

# Staged Allocation: A Compositional Technique for Specifying and Implementing Procedure Calling Conventions

Reuben Olinsky
Division of Engineering
and Applied Sciences
Harvard University
olinsky@post.harvard.edu

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

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

## Abstract

We present *staged allocation*, a technique for specifying calling conventions by composing tiny allocators called *stages*. A specification written using staged allocation has a precise, formal semantics, and it can be executed directly inside a compiler. Specifications of nine standard C calling conventions range in size from 15 to 30 lines each. An implementation of staged allocation takes about 250 lines of ML or 650 lines of C++. Each specification can be used not only to help a compiler implement the calling convention but also to generate a test suite.

**Categories and Subject Descriptors** D.3.4 [*Programming Languages*]: Processors—Retargetable compilers

**General Terms** Algorithms, Design, Standardization, Languages

**Keywords** Calling conventions

## 1. Introduction

Calling conventions are tricky to specify precisely. Specifications found in architecture manuals are written in informal English, but such specifications can be long, self-contradictory, incomplete, and misunderstood. Using informal specifications, even a mature compiler can fail (Lindig 2005), and a compiler can fail even on an example found in a manual (Bailey and Davidson 1995).

Informal specifications also make implementation difficult. Difficulties mount when a compiler supports multiple conventions, as it must if it supports multiple machines. Many compilers implement more than one convention per machine, e.g., a language-specific calling convention and the C calling convention. In practice, each of these conventions is coded by hand, and the code is often error-prone and unsatisfying to write.

An alternative to hand coding is to specify the calling convention concisely using a domain-specific language, then generate an implementation (Bailey and Davidson 1995). This alternative seems attractive, but no suitable language exists. Bailey and Davidson's language, CCL, is concise, but it is defined only by example and by an implementation that is no longer maintained.



We propose a new alternative: to specify and implement calling conventions using *staged allocation*. The name comes from the two insights that drive the design:

- The convention's placement of parameters in registers and memory can be viewed as an allocation problem.
- We can build an allocator by composing small *stages*. Each stage may satisfy an allocation request or may pass a (possibly modified) request to a subsequent stage.

By passing each allocation request through a sequence of stages, we keep individual stages small and simple. For example, one stage can allocate registers while another allocates stack slots. And by making stages composable, we enable specification of many different conventions using relatively few primitive stages.

Staged allocation makes the following contributions:

- Specifications written using staged allocation are concise.
- Staged allocation specifies register-use conventions by using *counters* to "skip past" registers that should not be used. This technique is simple and can express complex calling conventions more easily than CCL's alternative of using logical rules to exclude registers.
- Staged allocation is lightweight. A specification written using staged allocation can be executed directly in a compiler, so configuration and installation are simpler than with a program generator. And it is easy to create a specialized calling convention for a single use, e.g., to call a garbage collector or to fork a new thread.
- Staged allocation has a precise, formal semantics, so if you need to implement it yourself, you can. The implementation should be small and simple: our implementations, in the Quick C-- and Machine SUIF compilers, are about 250 lines of ML and 650 lines of C++, respectively.

We have specified and tested standard calling conventions on five machines, all of which are shown in this paper.

## 2. Calling-convention background

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: each procedure needs stack space for saved registers and private data, the operating system may need stack space to hold state when a signal is delivered, and the 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 passing values between caller and callee; we consider run-time system or operating system only when they impose constraints on a procedure. A companion paper addresses the complementary issues of how to share the stack among all parties and how to lay out stack frames (Lindig and Ramsey 2004).

## 2.1 The difficulty of calling conventions

It is surprising that a contract so central to the implementation of programming languages is so hard to get right. Evidence of difficulty is primarily anecdotal: many compiler writers find the implementation of calling conventions tricky and unpleasant. There is also some hard evidence that calling conventions are a source of bugs: Bailey and Davidson (2003) report 23 faults in production-quality C compilers for the MIPS and SPARC, and Lindig (2005) reports 13 new bugs in more mature C compilers for four platforms.

Why do such bugs linger even in widely used compilers? Our best guess is that because the specifications of calling conventions are written in informal English, it is very hard to get all the corner cases right. Certainly common test and benchmark suites do not exercise corner cases; for example, in the "torture" test suite for `gcc` 4.0, 90% of arguments have type pointer, integer, or character, and 90% of functions return `void` or `int`. Very few tests pass compound (`struct` or `union`) or floating-point values. As another example, in the SPEC CPU 2000 benchmark, over 95% of values passed have simple types, and most functions have at most two arguments. Although the contract between caller and callee must cover all cases, bugs in corner cases can go undetected for a long time.

## 2.2 Underlying assumptions

The contract embodied by a calling convention must say where to place the value of each parameter or result. 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 registers and part in memory. As in prior work, we assume that all but finitely many parameters are 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 parameters can be placed by a sequence of *passes*, each of which considers one parameter at a time, starting with the leftmost parameter in the source code. A single pass suffices if the location in which parameter $k$ is passed depends only on the *types* (including sizes) of parameters $\leq k$. Because of variadic procedures (varargs), a single pass suffices for every C calling convention: it must be possible to extract variadic parameters one at a time using the `va_arg` macro. More ambitious conventions might require multiple passes; for example, a first pass might pack 64-bit floating-point parameters into aligned register pairs before allowing a second pass to put 32-bit parameters in single floating-point registers. Or in another convention, a first pass might put all 32-bit integer parameters into integer registers before allowing later passes to use integer registers for larger or smaller parameters.

## 2.3 Formal modeling

Bailey and Davidson (1995) first studied formal models of calling conventions, making these contributions:

- Given a calling convention, they use a Mealy (1955) automaton to allocate a location for each parameter. The parameter's type is presented to the automaton, and the automaton makes a state transition and emits a location for that parameter. Because the automaton must be capable of allocating arbitrarily many parameters, it must have infinitely many states. Bailey and Davidson invented a clever mapping of this infinite-state automaton onto a finite-state automaton they call a P-FSA. The finite-state P-FSA can't be used to place parameters, but because the mapping is homomorphic (preserving the structure of the transitions of the original automaton), the P-FSA can be used to analyze the convention.
- They use the P-FSA to detect inconsistency and incompleteness in calling conventions. (A convention is inconsistent if it allocates a single location to carry more than one parameter. A convention is incomplete if there is a sequence of parameters for which no location is specified.)
- They present a domain-specific *Calling Convention Language* (CCL) for describing calling conventions.
- They present an enumeration procedure that can be used to create a P-FSA from *any* executable specification of a calling convention, even one without a formal semantics. The enumeration procedure repeatedly executes the specification with different parameter lists, exhaustively finding the P-FSA's states and transitions. An enumerated P-FSA can be represented as a set of tables, which can be interpreted to place parameters at compile time.
- They use enumerated P-FSAs to develop target-specific test suites for calling conventions (Bailey and Davidson 1996; Bailey and Davidson 2003).

Because a P-FSA can be analyzed for incompleteness and inconsistency, and because it can also be used to generate a test suite, P-FSA models should be attractive to any compiler writer. But because it is defined only by example (and by its implementation), CCL is not so attractive. By contrast with CCL, staged allocation is precisely defined, simpler, and easier to engineer into a compiler (Section 5). And staged allocation, like any other deterministic algorithm for placing parameters, fully supports P-FSAs and the automated analysis and testing techniques that have been developed for P-FSAs.

# 3. Specifying automata for passing parameters

To specify an automaton, we must formalize three things: the types of parameters presented to the automaton, the locations produced by the automaton, and the behavior of the automaton itself. We start with types, then present two examples before moving on to locations and automata. But before we can do any of this, we must first explain how staged allocation fits into a surrounding context.

By itself, staged allocation is not a complete formal framework. It is a specification and implementation technique that is intended to be embedded in a surrounding formal language or programming language. The surrounding language must represent abstractions such as machine locations, and it must also describe or implement simple computations involving integers, strings, Booleans, and sequences. In this paper, we use a surrounding formal language that borrows notation from functional programming languages. Functions are written using $\lambda$ notation, and literal integers, strings, and Booleans use standard notations. An empty sequence is $[\,]$, and a nonempty sequence with first element $s$ and remaining elements $ss$ is $s :: ss$. We also use syntactic sugar: a finite sequence is $[x_1, \ldots, x_n]$, and one sequence followed by another is $ss +\!\!+\, ss'$. Machine registers are written using typewriter font, as `eax` or `f16`, for example.

We have implemented staged allocation using three languages: Objective Caml (Leroy et al. 2004), Lua (Ierusalimschy 2003; Ramsey 2005), and C++ (Stroustrup 1997). Although we use $\lambda$ notation in this paper, first-class functions are not required for an implementation; neither our C++ code nor our Lua code uses them.

## 3.1 Formalizing types

A calling convention is normally specified in terms of a particular high-level language, and the convention decides where to place parameters based on their types. Staged allocation abstracts away from the high-level language and its type system. Instead, it expects each high-level type to be mapped to a triple: a *width*, a *kind*, and an *alignment*.

- The width is the size in bits of a value of the type.
- The kind is a string that indicates what kind of location a value of the type might be passed in. For example, the `"float"` kind might indicate a floating-point register while the `"address"` kind indicates an address register. The empty kind `""` typically indicates a general-purpose or integer register. Another view of a kind is that a kind encapsulates just enough information about a high-level type to tell us in what sort of location a value of that type should be passed.
- The alignment is an integer that constrains the address of whatever memory location a parameter might occupy: the address must be a multiple of the alignment. The units of alignment are the addressing units of the target machine: normally 8-bit bytes, but larger units on word-addressed machines.

Mappings of high-level types are straightforward. For example, on many platforms a C `double` maps to a 64-bit parameter with kind `"float"`, aligned on an 8-byte boundary. As another example, most conventions map C structures and unions to the empty kind, but when structures and unions must be treated differently from integers of the same size, we use the kind `"struct"` to indicate a C structure or union.

In many cases a parameter's alignment is determined by its width and kind, but the alignment is not superfluous. For example, different struct types may have the same width and kind but different alignments: a struct containing two `int`s may have width 64, the empty kind, and alignment 4; and a struct containing a single `double` may have width 64, the empty kind, and alignment 8.

Type mapping has one fine point: staged allocation passes parameters by value. If a parameter should be passed by reference or by value-result, it is the front end's job to generate intermediate code that passes, e.g., the address of that parameter. Similarly, the C convention may require that a function returning a structure take the address of that structure as an extra, hidden parameter. If so, the front end must add the parameter before running staged allocation.

Width, kind, and alignment correspond to representations used in typical compilers. Width and alignment are often represented directly, as they are in Machine SUIF (Smith and Holloway 2000) and `lcc` (Fraser and Hanson 1995), for example. A kind often corresponds to an internal enumeration or abstraction; for example, it corresponds to the "type suffix" used in `lcc` version 4 and to the "type id" used in Machine SUIF.

## 3.2 Example specifications

Given width, kind, and alignment, let us temporarily take locations for granted in order to look at some example specifications. Figures 1 and 2 present specifications of standard C calling conventions on the Pentium and the Alpha. Each convention requires two automata: one to pass parameters and one to receive results. An automaton is specified by composing *stages*, which are defined precisely and discussed in detail in Section 3.4 below—here we show examples.

Figure 1 shows the standard C calling convention for a Pentium running Linux. A specification is a list of stages; it is created using the facilities of the surrounding language and the operations of staged allocation, which are shown in small caps. Every convention requires two specifications: one for parameters and one for results.

$$
\begin{array}{l}
\mathit{parms} = \\
\quad [\textsc{widen}(\lambda w.\mathrm{round\_up}(w,32)), \textsc{overflow}(c_o, \textsc{up}, 4)] \\
\mathit{results} = \\
\quad [\textsc{choice}( \\
\qquad [\lambda\langle w,k,\sigma\rangle.k = \texttt{"float"}, \\
\qquad\quad [\textsc{widen}(\lambda w.80), \textsc{useregs}([\texttt{fp\_stack\_top}])] \\
\qquad , \lambda\langle w,k,\sigma\rangle.\mathbf{true}, \\
\qquad\quad [\textsc{widen}(\lambda w.\mathrm{round\_up}(w,32)), \textsc{useregs}([\texttt{EAX},\texttt{EDX}])] \\
\qquad ]) \\
\quad ]
\end{array}
$$

**Figure 1.** Pentium calling conventions

$$
\begin{array}{l}
\mathit{parms} = \\
\quad [\textsc{widen}(\lambda w.\mathrm{round\_up}(w,64)) \\
\quad , \textsc{bitcounter}(\texttt{"bits"}) \\
\quad , \textsc{choice}( \\
\qquad [\lambda\langle w,k,\sigma\rangle.k = \texttt{"float"}, \\
\qquad\quad \textsc{regs\_by\_bits}(\texttt{"bits"}, [\texttt{f16},\ldots,\texttt{f21}]) \\
\qquad , \lambda\langle w,k,\sigma\rangle.\mathbf{true}, \\
\qquad\quad \textsc{regs\_by\_bits}(\texttt{"bits"}, [\texttt{r16},\ldots,\texttt{r21}]) \\
\qquad ]) \\
\quad , \textsc{overflow}(c_o, \textsc{up}, 16) \\
\quad ] \\
\mathit{results} = \\
\quad [\textsc{widen}(\lambda w.\mathrm{round\_up}(w,64)) \\
\quad , \textsc{choice}( \\
\qquad [\lambda\langle w,k,\sigma\rangle.k = \texttt{"float"}, \textsc{useregs}([\texttt{f0},\texttt{f1}]) \\
\qquad , \lambda\langle w,k,\sigma\rangle.\mathbf{true}, \textsc{useregs}([\texttt{r0}]) \\
\qquad ]) \\
\quad ]
\end{array}
$$

**Figure 2.** Alpha calling conventions

These appear in Figure 1 as the lists *parms* and *results*. When a parameter is placed, it is first widened to a multiple of 32 bits, so for example, a C `char` is promoted to an `int`. Then, space for the parameter is allocated on the stack (in the overflow block). The overflow block grows upward (UP), so earlier parameters are placed at lower addresses, and an address in the overflow block may be aligned on at most a 4-byte boundary. The $c_o$ in the OVERFLOW stage is a counter that tracks how many bytes have been allocated in the overflow block; in our implementations, this counter is not exposed to the client, but because it is needed for the formal semantics, we show it here.

The *results* automaton makes a choice based on the kind of the result. The CHOICE stage begins with a predicate that receives the triple $\langle w, k, al\rangle$ and makes a decision based on the value of the kind $k$. A floating-point result is widened to 80 bits and placed on the top of the floating-point stack, which consists of 80-bit registers. Any other result is widened to a multiple of 32 bits, then placed either in general-purpose register `EAX` or in the register pair `EAX:EDX`, depending on width. (For example, a C `long long` result is placed in the pair.) If the result is wider than 64 bits, the automaton halts with an error message. The C convention never returns a result on the stack, so there is no overflow block.

Figure 2 gives another example: the C calling convention for an Alpha running OSF/1. The Alpha is a 64-bit machine, and up to six words of parameters may be placed in registers; the remaining parameters are placed on the stack, in the overflow block. A floating-point parameter may be placed in one of the floating-point registers `f16` to `f21`; any other type of parameter may be placed in general-purpose registers `r16` to `r21`. If a floating-point parameter is placed in `f16`, the next integer parameter must go in `r17`; that is, it is necessary to leave a gap. We specify this gap by using a counter named `bits` to count the number of bits of parameters placed so far; each REGS_BY_BITS stage skips as many registers as account for that number of bits.

At the bottom of Figure 2, a floating-point result is returned in register `f0` (registers `f0` and `f1` if it is complex). Any other result is returned in register `r0`.

More examples appear in Section 4 and Appendix A.

### 3.3 Formalizing locations

In any calling convention, each parameter or result should be passed in a distinct *location*. Ideally locations would be simple, but in practice they aren't. For example, although a machine register or a block of memory is obviously a location, calling conventions require more complex locations as well:

- The least significant $k$ bits of an $n$-bit register could be a location. For example, a byte-sized parameter could be passed in the least significant 8 bits of a 32-bit register.
- A pair of registers could be a location. For example, a doubleword floating-point parameter could be passed in the MIPS floating-point register pair `f12`–`f13` (also called `d12`).
- More generally, a combination of registers and memory blocks could be a location. For example, a large structure could have its first 16 bytes passed in registers `r4`–`r7` and the remaining bytes in memory.

The many kinds of locations account for a significant fraction of the complexity in calling conventions.

We formalize a location as an abstraction that has a width and can be read or written. For purposes of this paper, we form locations using the grammar in Figure 3. A location notated $r$ is a machine register, and a location notated $\texttt{start} + n$ is a slot in the overflow block. These locations are atomic, have machine-defined read and write operations, and have widths that depend on the machine. There are also composite locations, which are notated "$\mathrm{combine}(\ell, \ell')$" or "$\mathrm{narrow}(\ell, \mathrm{w}, \mathrm{k})$," where $w$ is the width of the narrow location and $k$ is the kind of narrowing done. The meaning of composite locations is given by the equations in Figure 4. The bottom part of the figure lists the functions and operators used in the equations.

### 3.4 Specifying and formalizing automata

In staged allocation, we specify an automaton as a sequence of *stages*, written $ss$. A sequence of stages is formed according to the grammar in Figure 5. In the rest of this section, we show how such a sequence is used to compute the location of a parameter.

In a sequence $ss$, each stage can respond to a request $\langle w, k, al\rangle$, which asks for a location of width $w$ bits with kind $k$ and alignment $al$. The stage may satisfy the request or pass the (possibly modified) request on to the next stage. A stage may also count requests or bits allocated by using a *state variable* or *counter*, written $c$. In a specification, such a variable is referred to by its name. Its value is kept in a *store*, written $\sigma$; in the initial store, the value of every variable is zero. The store and the specification together form an automaton $\langle ss, \sigma\rangle$. The state variables are private to the automaton and are hidden from clients.

An automaton is used to place parameters by giving it an allocation request for each parameter. When automaton $\langle ss, \sigma\rangle$ gets a request $\langle w, k, al\rangle$, it responds with a location $\ell$. The automaton may also update counters, producing a new store $\sigma'$. We specify the automaton's behavior formally as a set of inference rules, which are shown in Figure 6. These rules use the judgment $\langle ss, \sigma\rangle @\langle w, k, al\rangle \overset{\ell}{\Longrightarrow} \sigma'$, which says that automaton $\langle ss, \sigma\rangle$ responds to request $\langle w, k, al\rangle$ by producing location $\ell$ and changing its state to $\sigma'$. Figure 6 uses notation and auxiliary functions which are summarized in Figures 7 and 8.

$$\ell ::= r \mid \texttt{start} + n \mid \mathrm{combine}(\ell, \ell') \mid \mathrm{narrow}(\ell, w, k)$$

**Figure 3.** Ways to form a location $\ell$

$$\begin{aligned}
\mathrm{read}(\mathrm{combine}(\ell, \ell')) &= \mathrm{read}(\ell) \ll \ell'.\texttt{width} + \mathrm{read}(\ell') \\
\mathrm{write}(\mathrm{combine}(\ell, \ell'), v) &= \mathrm{write}(\ell, v \gg \ell'.\texttt{width}); \\
&\quad\ \mathrm{write}(\ell', \mathit{lobits}_{\ell'.\texttt{width}}(v)) \\
\mathrm{read}(\mathrm{narrow}(\ell, w, \texttt{"float"})) &= \mathit{f2f}_{\ell.\texttt{width}\to w}(\mathrm{read}(\ell)) \\
\mathrm{write}(\mathrm{narrow}(\ell, w, \texttt{"float"}), v) &= \mathrm{write}(\ell, \mathit{f2f}_{w\to\ell.\texttt{width}}(v)) \\
\mathrm{read}(\mathrm{narrow}(\ell, w, _)) &= \mathit{lobits}_w(\mathrm{read}(\ell)) \\
\mathrm{write}(\mathrm{narrow}(\ell, w, _), v) &= \mathrm{write}(\ell, \mathit{sx}_{w\to\ell.\texttt{width}}(v))
\end{aligned}$$

$$\begin{aligned}
\mathrm{combine}(\ell, \ell').\texttt{width} &= \ell.\texttt{width} + \ell'.\texttt{width} \\
\mathrm{narrow}(\ell, w, k).\texttt{width} &= w
\end{aligned}$$

| | |
|---|---|
| $\mathit{lobits}_w$ | Extract least significant $w$ bits |
| $\mathit{sx}_{w\to w'}$ | Sign extend from $w$ bits to $w'$ bits |
| $\mathit{f2f}_{w\to w'}$ | Float-to-float conversion (change width) |
| $\ll$ | Shift left |
| $\gg$ | Shift right |

**Figure 4.** Reading and writing composite locations

$$\begin{aligned}
ss &::= s :: ss \mid [\,] \\
s &::= ss \\
&\mid \textsc{overflow}(c, g, \mathit{max_align}) \\
&\mid \textsc{widths}(ws) \\
&\mid \textsc{widen}(f) \\
&\mid \textsc{align_to}(f) \\
&\mid \textsc{argcounter}(c) \\
&\mid \textsc{bitcounter}(c) \\
&\mid \textsc{pad}(c) \\
&\mid \textsc{regs_by_args}(c, rs) \\
&\mid \textsc{regs_by_bits}(c, rs) \\
&\mid \textsc{choice}([p_1, s_1, \ldots, p_n, s_n]) \\
&\mid \textsc{first_choice}(c, [p_1, s_1, \ldots, p_n, s_n])
\end{aligned}$$

$\textsc{useregs}(rs) \equiv [\textsc{bitcounter}(c), \textsc{regs\_by\_bits}(c, rs)]$, where $c$ is a fresh counter.

**Figure 5.** Abstract syntax of stages

In principle, an allocation request could lead to a situation in which no rule in Figure 6 applies. This situation would indicate either an error in the mapping from high-level types to $\langle w, k, al\rangle$ or an error in the specification $ss$. In practice, we use Bailey and Davidson's (1995) enumeration procedure to guarantee that no such errors occur. The enumeration requires a specification $ss$ and a set of high-level types, and it ensures at compile-compile time that $ss$ is *complete*.

In the rest of this section, we explain the rules in Figure 6, working from the top down.

**Overflow** An overflow stage $\textsc{overflow}(c, g, \mathit{max\_align})$ satisfies every request by allocating from the overflow block; it never passes a request to its successor. Counter $c$ counts the number of bytes allocated in the block; direction $g$ says which way the over-

$$\frac{al \text{ divides } max\_align \qquad w \text{ is a multiple of } \texttt{mem\_size} \qquad n = \sigma(c) \qquad n' = \text{round\_up}(n, al)}{\langle \textsc{overflow}(c, \textsc{up}, max\_align) :: ss, \sigma\rangle @ \langle w, k, al\rangle \overset{\texttt{start}+n'}{\Longrightarrow} \sigma\{c \mapsto n' + w/\texttt{mem\_size}\}} \quad (\textsc{Overflow-Up})$$

$$\frac{al \text{ divides } max\_align \qquad w \text{ is a multiple of } \texttt{mem\_size} \qquad n = \sigma(c) \qquad n' = \text{round\_up}(n, al) + w/\texttt{mem\_size}}{\langle \textsc{overflow}(c, \textsc{down}, max\_align) :: ss, \sigma\rangle @ \langle w, k, al\rangle \overset{\texttt{start}-n'}{\Longrightarrow} \sigma\{c \mapsto n'\}} \quad (\textsc{Overflow-Down})$$

$$(\textsc{WidthsOK}) \quad \frac{w \in ws \qquad \langle ss, \sigma\rangle @ \langle w, k, al\rangle \overset{\ell}{\Longrightarrow} \sigma'}{\langle \textsc{widths}(ws) :: ss, \sigma\rangle @ \langle w, k, al\rangle \overset{\ell}{\Longrightarrow} \sigma'}$$

$$\frac{w \le f(w) \qquad \langle ss, \sigma\rangle @ \langle f(w), k, al\rangle \overset{\ell}{\Longrightarrow} \sigma' \qquad \ell' = \text{narrow}(\ell, w, k)}{\langle \textsc{widen}(f) :: ss, \sigma\rangle @ \langle w, k, al\rangle \overset{\ell'}{\Longrightarrow} \sigma'} \quad (\textsc{Widen})$$

$$(\textsc{AlignTo}) \quad \frac{\langle ss, \sigma\rangle @ \langle w, k, f(w)\rangle \overset{\ell}{\Longrightarrow} \sigma'}{\langle \textsc{align\_to}(f) :: ss, \sigma\rangle @ \langle w, k, al\rangle \overset{\ell}{\Longrightarrow} \sigma'}$$

$$\frac{\langle ss, \sigma\rangle @ \langle w, k, al\rangle \overset{\ell}{\Longrightarrow} \sigma' \qquad \sigma'(c) = n}{\langle \textsc{argcounter}(c) :: ss, \sigma\rangle @ \langle w, k, al\rangle \overset{\ell}{\Longrightarrow} \sigma'\{c \mapsto n+1\}} \quad (\textsc{Argcounter})$$

$$\frac{\langle ss, \sigma\rangle @ \langle w, k, al\rangle \overset{\ell}{\Longrightarrow} \sigma' \qquad \sigma'(c) = n}{\langle \textsc{bitcounter}(c) :: ss, \sigma\rangle @ \langle w, k, al\rangle \overset{\ell}{\Longrightarrow} \sigma'\{c \mapsto n+w\}} \quad (\textsc{Bitcounter})$$

$$(\textsc{Pad}) \quad \frac{n = \sigma(c) \qquad n' = \text{round\_up}(n, al \times \texttt{mem\_size}) \qquad \langle ss, \sigma\{c \mapsto n'\}\rangle @ \langle w, k, al\rangle \overset{\ell}{\Longrightarrow} \sigma'}{\langle \textsc{pad}(c) :: ss, \sigma\rangle @ \langle w, k, al\rangle \overset{\ell}{\Longrightarrow} \sigma'}$$

$$\frac{\sigma(c) = n \qquad \text{drop}(n, rs) = [\,] \qquad \langle ss, \sigma\rangle @ \langle w, k, al\rangle \overset{\ell}{\Longrightarrow} \sigma'}{\langle \textsc{regs\_by\_args}(c, rs) :: ss, \sigma\rangle @ \langle w, k, al\rangle \overset{\ell}{\Longrightarrow} \sigma'} \quad (\textsc{RegsByArgs-None})$$

$$\frac{\sigma(c) = n \qquad \text{drop}(n, rs) = \ell :: \ell s \qquad \ell.\texttt{width} = w}{\langle \textsc{regs\_by\_args}(c, rs) :: ss, \sigma\rangle @ \langle w, k, al\rangle \overset{\ell}{\Longrightarrow} \sigma} \quad (\textsc{RegsByArgs-Fits})$$

$$\frac{C = [p_1, s_1, \ldots, p_n, s_n] \qquad 1 \le i \le n \qquad p_i(w, k, \sigma) \qquad \forall j : 1 \le j < i : \neg p_j(w, k, \sigma) \qquad \langle s_i :: ss, \sigma\rangle @ \langle w, k, al\rangle \overset{\ell}{\Longrightarrow} \sigma'}{\langle \textsc{choice}(C) :: ss, \sigma\rangle @ \langle w, k, al\rangle \overset{\ell}{\Longrightarrow} \sigma'} \quad (\textsc{Choice})$$

$$\frac{\sigma(c) = 0 \qquad C = [p_1, s_1, \ldots, p_n, s_n] \qquad 1 \le i \le n \qquad p_i(w, k, \sigma) \qquad \forall j : 1 \le j < i : \neg p_j(w, k, \sigma) \qquad \langle s_i :: ss, \sigma\rangle @ \langle w, k, al\rangle \overset{\ell}{\Longrightarrow} \sigma'}{\langle \textsc{first\_choice}(c, C) :: ss, \sigma\rangle @ \langle w, k, al\rangle \overset{\ell}{\Longrightarrow} \sigma'\{c \mapsto i\}} \quad (\textsc{FirstChoice-Init})$$

$$\frac{\sigma(c) = i \qquad i > 0 \qquad C = [p_1, s_1, \ldots, p_n, s_n] \qquad \langle s_i :: ss, \sigma\rangle @ \langle w, k, al\rangle \overset{\ell}{\Longrightarrow} \sigma'}{\langle \textsc{first\_choice}(c, C) :: ss, \sigma\rangle @ \langle w, k, al\rangle \overset{\ell}{\Longrightarrow} \sigma'} \quad (\textsc{FirstChoice-Later})$$

$$\frac{\langle ss' \mathbin{+\!\!+} ss, \sigma\rangle @ \langle w, k, al\rangle \overset{\ell}{\Longrightarrow} \sigma'}{\langle ss' :: ss, \sigma\rangle @ \langle w, k, al\rangle \overset{\ell}{\Longrightarrow} \sigma'} \quad (\textsc{stages})$$

$$\frac{\sigma(c) = n \qquad \text{drop\_bits}(n, rs) = [\,] \qquad \langle ss, \sigma\rangle @ \langle w, k, al\rangle \overset{\ell}{\Longrightarrow} \sigma'}{\langle \textsc{regs\_by\_bits}(c, rs) :: ss, \sigma\rangle @ \langle w, k, al\rangle \overset{\ell}{\Longrightarrow} \sigma'} \quad (\textsc{RegsByBits-None})$$

$$\frac{\sigma(c) = n \qquad \text{drop\_bits}(n, rs) = \ell :: \ell s \qquad \ell.\texttt{width} = w}{\langle \textsc{regs\_by\_bits}(c, rs) :: ss, \sigma\rangle @ \langle w, k, al\rangle \overset{\ell}{\Longrightarrow} \sigma} \quad (\textsc{RegsByBits-Fits})$$

$$\frac{\begin{array}{c}\sigma(c) = n \qquad \text{drop\_bits}(n, rs) = \ell :: \ell s \qquad \ell.\texttt{width} < w \\ \langle \textsc{regs\_by\_bits}(c, rs) :: ss, \sigma\{c \mapsto n + \ell.\texttt{width}\}\rangle @ \langle w - \ell.\texttt{width}, k, al\rangle \overset{\ell'}{\Longrightarrow} \sigma' \qquad \ell'' = \text{combine}(\ell, \ell') \qquad \sigma'(c) = n'\end{array}}{\langle \textsc{regs\_by\_bits}(c, rs) :: ss, \sigma\rangle @ \langle w, k, al\rangle \overset{\ell''}{\Longrightarrow} \sigma'\{c \mapsto n' - \ell.\texttt{width}\}} \quad (\textsc{RegsByBits-Some})$$

**Figure 6.** Rules for allocating from an automaton

| | | | |
|---|---|---|---|
| $ss$ | sequence of stages | $\ell, \ell s$ | location(s) |
| $c$ | counter (state variable) | $f$ | function |
| $k$ | kind | $p$ | predicate |
| $al$ | alignment in bytes | $\sigma$ | state |
| $w, ws$ | width(s) in bits | $n$ | integer |
| $r, rs$ | register(s) | $C$ | list of choices |

**Figure 7.** Summary of notation

$$\begin{aligned}
\text{drop_bits}(0, rs) &= rs \\
\text{drop_bits}(n, [\,]) &= [\,] \\
\text{drop_bits}(n, r :: rs) &= \text{drop_bits}(n - r.\texttt{width}, rs) \\
&\qquad \text{when } n \ge r.\texttt{width} \\
\text{drop}(0, rs) &= rs \\
\text{drop}(n, [\,]) &= [\,] \\
\text{drop}(n, r :: rs) &= \text{drop}(n - 1, rs)
\end{aligned}$$

**Figure 8.** Auxiliary functions

flow block should grow; and $max\_align$ specifies the maximum alignment supported by the calling convention. As shown in the OVERFLOW-UP and OVERFLOW-DOWN rules at the top of Figure 6, the OVERFLOW stage allocates locations in contiguous memory, padding as needed to satisfy alignment requirements. Allocation starts at address `start`, which is a symbolic address that is resolved later, when the stack frame is frozen (Lindig and Ramsey 2004). When the overflow block grows up, `start` refers to the bottom of the block; when the block grows down, `start` refers to the top. The constant `mem_size` is the number of bits in the addressable unit of the target machine, so for a byte-addressed machine, `mem_size` is 8.

The direction of growth determines the order in which parameters appear on the stack. If the overflow block grows up, then parameters that are allocated earlier—which are normally the parameters that appear on the left in the source code—have lower addresses. We want *not* to characterize such parameters being "first on the stack," because even though they appear at lower addresses, on some machines they will have been *pushed* last. By separating the placement of the parameters from the instructions used to achieve that placement, we hope to avoid confusion.

**Selection and modification of width** We use two width-related stages that satisfy no requests themselves, but only check or modify requests before passing the requests to their successors. The WIDTHS($ws$) stage restricts the automaton to satisfy only requests for a width on the list $ws$. It is useful for detecting internal errors in the compiler, e.g., passing a 16-bit value when the convention supports only wider values.

Instead of halting with an error message, we can ask for a wider location. For example, we might embed a 16-bit value inside a 32-bit location. The stage WIDEN($f$) modifies a request for a width $w$ so it has a width $f(w)$, which must be at least as large as $w$. Common cases for $f$ include $\lambda w.n$, to widen a value to exactly $n$ bits; and $\lambda w.\mathrm{round\_up}(w, n)$, to widen a value to the nearest multiple of $n$ bits. Our C++ and Lua implementations provide some syntactic sugar for these cases.

As Figure 6 shows, the WIDEN($f$) stage requests a location $\ell$ of width $f(w)$ from its successor. The `widen` stage builds a new, narrower location $\ell' = \mathrm{narrow}(\ell, w, k)$, which it returns to its client. As shown in Figure 4, a read from $\ell'$ is implemented by reading the wide value in $\ell$ and narrowing the value to $w$ bits. A write to $\ell'$ is implemented by widening the value written and writing it into $\ell$. Widening and narrowing are done using either integer or floating-point operations, depending on $k$, the kind of the allocation request.

**Modification of alignment** The alignment of a request can be modified by the stage ALIGN_TO($f$), which uses the new alignment $f(w)$, where $w$ is the width of the request. The modified request is then passed to the successor of ALIGN_TO($f$).

Perhaps the functions used in ALIGN_TO and WIDEN stages should be generalized to use a request's kind, not just its width, to make their modifications. 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 $n$ arguments go in the first $n$ registers" and "the first $n$ bits of arguments go in the first $n$ bits of registers." We use separate stages to count $n$ and to allocate. The ARGCOUNTER($c$) and BITCOUNTER($c$) stages count arguments and bits, respectively. In our implementations, a counter $c$ is specified by giving its name; we allocate memory for each named counter and initialize each counter to zero.

Each of these stages simply passes each request to its successor, as shown in the ARGCOUNTER and BITCOUNTER rules in the middle left of Figure 6. Once the request is satisfied, the stage increments its counter. An ARGCOUNTER stage increments its counter by 1; a BITCOUNTER stage increments its counter by the width of the request. Counters are incremented *after* successor stages have run; when a stage is run, counters reflect state corresponding to parameters already allocated, not the current request.

The counters work with two other stages, REGS_BY_ARGS($c, rs$) and REGS_BY_BITS($c, rs$). The stage REGS_BY_ARGS($c, rs$) uses argument counter $c$ to implement the "$n$ arguments to $n$ registers" policy. Given a request, it uses the value $n$ of counter $c$ to drop the first $n$ registers from list $rs$. Depending on whether registers remain, it applies one of two rules at the lower left of Figure 6. If no registers remain (REGSBYARGS-NONE), the request is passed to the next stage. If registers remain (REGSBYARGS-FITS), the first remaining register is used to satisfy the request, provided its width is equal to the width of the argument. If the widths don't match, something has gone wrong, and the automaton halts with an error. This stage is seldom used because most conventions count bits of arguments, not arguments themselves. One exception is the MIPS R3000 (Section 3.6).

The stage REGS_BY_BITS($c, rs$) uses bit counter $c$ to implement the "$n$ bits of arguments to $n$ bits of registers" policy. Given a request, it uses the value $n$ of counter $c$ to drop enough registers from list $rs$ to account for the $n$ bits already allocated. Its behavior is similar to that of the "by arguments" stage, and it is described by two rules at the lower right of Figure 6. If no registers remain (REGSBYBITS-NONE), the request is passed to the next stage; otherwise the first remaining register is used to satisfy the request, provided the widths match (REGSBYBITS-FITS).

What if the width of the request is different from the width of the first remaining register?

- If the request is too wide for the first register, one could use a combination of registers to satisfy the request. For example, a 64-bit request might be satisfied using two 32-bit registers.

  If a request is large enough to exhaust registers, one could use registers to satisfy as much of the request as possible, then get 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.

  Both of these alternatives are covered by the REGSBYBITS-SOME rule, which appears at the bottom of Figure 6. The rule takes the first available register $\ell$, temporarily changes the value of the counter, and recursively requests a location $\ell'$ to hold the remaining bits. As described in Figure 4, the function combine takes the two narrow locations $\ell$ and $\ell'$ and returns a wide location $\ell''$.

- If the request is too narrow for the first register, the stage could halt the compiler with a bug report (an unsupported width) or could widen the request implicitly. 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 might 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 may need to increment the counter $c$ to indicate that more bits were allocated than were actually requested.

It is often necessary to split requests; for example, it is common to require that a large struct be passed partly in registers and partly in memory. But we have never seen a convention that needed to widen a request implicitly, perhaps because such a convention would be likely only if a datum were to require more bits when represented in a register than when represented in memory. Therefore the second alternative is not implemented in our system and not shown in Figure 6.

Named counters, although crucial for sharing state among stages, can be inconvenient. For the common case in which a BITCOUNTER($c$) is followed directly by REGS_BY_BITS($c, rs$), we provide the syntactic sugar USEREGS($rs$), which creates a fresh, private counter $c$:

$$\text{USEREGS}(rs) \equiv [\text{BITCOUNTER}(c), \text{REGS\_BY\_BITS}(c, rs)].$$

**Padding** The alignment of a request normally affects only its placement in the overflow block: bytes of padding are inserted as needed to be sure a parameter's address is a multiple of its alignment. But some calling conventions insert padding even when using registers. These conventions can be specified by using a stage PAD($c$), which rounds a bit counter up to respect alignment, as shown in the the left column of Figure 6. The PAD rule multiplies by `mem_size` in order to convert from alignment units to bits.

**Choice among stages** Many calling conventions pass different types of parameters in different kinds of registers. For example, the Alpha convention passes floating-point values in floating-point registers and other values in integer registers. We implement such a rule using a "choice" stage, which uses the kind and width of an allocation request to decide which alternative stage should satisfy the request. A choice stage operates on a list in which predicates and stages alternate; its form is CHOICE($[p_1, s_1, \ldots, p_n, s_n]$). As shown in the CHOICE rule at the upper right of Figure 6, 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. If no predicate is satisfied, no rule applies, and the automaton halts with an error.

A predicate $p$ is a function that takes a width, a kind, and the store; it returns a Boolean. Our C++ and Lua implementations provide extra support for many common cases, including predicates that check for a particular kind, a particular width, and a particular value for a given counter.

**Arbitrary state transition** Sometimes the location of a parameter depends on the width or kind 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: a stage that makes a permanent state transition on the relevant parameter. The stage FIRST_CHOICE($c, [p_1, s_1, \ldots, p_n, s_n]$) is like the stage CHOICE($[p_1, s_1, \ldots, p_n, s_n]$), except that the choice is made once, when the first request reaches the stage, instead of each time a request reaches the stage. After the first request chooses substage $s_i$, counter $c$ is set to $i$, and the FIRST_CHOICE($c, [p_1, s_1, \ldots, p_n, s_n]$) stage behaves like $s_i$ from then on. The implementation is shown in the FIRSTCHOICE-INIT and FIRSTCHOICE-LATER rules at the middle right of Figure 6, and an example appears in Section 3.6.

## 3.5 Implementing and using staged allocation

An implementation of staged allocation provides three operations on specifications:

- The `init` operation creates a fresh store $\sigma$ where $\sigma(c) = 0$ for all $c$, then forms the automaton $\langle ss, \sigma \rangle$. In addition to $ss$, `init` requires the `start` address of the overflow block and the byte order and `mem_size` of the target machine.
- The `allocate` operation takes a request $\langle w, k, al \rangle$ and an automaton, uses Figure 6 to compute a $\sigma'$ and $\ell$ satisfying $\langle ss, \sigma \rangle @ \langle w, k, al \rangle \overset{\ell}{\Longrightarrow} \sigma'$, mutates the internal state to replace $\sigma$ with $\sigma'$, and returns $\ell$.
- The `freeze` operation takes an automaton and returns the overflow block and the set of registers used by previous calls to `allocate`. The `freeze` operation is most easily formalized by generalizing the rules in Figure 6 to keep track of what locations are allocated.

To use staged allocation to implement a calling convention, a compiler needs two specifications: $parms$ and $results$. Both specifications are used in every procedure definition and at every call site. For all of the examples in this paper, parameters are allocated in a single pass, and a specification is simply a sequence of stages $ss$. We show how to generalize to multiple passes in step 2 below.

Given a specification and a list of formal or actual parameters, here is how a compiler computes the location of each parameter:

1. Pass the $parms$ specification to `init` to create a fresh automaton.
2. From the type of each parameter, compute a width, kind, and alignment. Call `allocate` once for each parameter, and remember the location returned.

   Most conventions can allocate parameters in a single pass, in which case `allocate` is simply called on each parameter in turn, in the order in which the parameters appear in the source code. But some conventions may require multiple passes. For such conventions, we use a specification that gives not only a sequence of stages $ss$ but also a sequence of predicates $[p_1, \ldots, p_n]$. Each predicate specifies one pass of the allocation: a parameter is allocated on pass $i$ if its width, kind, and alignment satisfy $p_i$ but did not satisfy any earlier predicate.
3. When all parameters have been allocated, call `freeze` to get the set of registers used and an overflow block. The overflow block becomes part of the stack frame; in our compiler, the overflow block is composed with other blocks using a simple declarative technique (Lindig and Ramsey 2004). The set of registers is used in liveness analysis: at a call site, this set is kept live by the call; in a procedure definition, this set is assumed to be defined on entry.

The $results$ specification is used in similar fashion at call sites and `return` statements.

## 3.6 More complex examples

The conventions and examples shown above are relatively simple. To show more of the stages described in Section 3, we present two somewhat more complex examples: the MIPS R3000 convention, which is notoriously complex and error-prone (Bailey and Davidson 1995), and the IA-64 convention.

**MIPS** On the MIPS, the first 16 bytes of parameters normally go in integer registers `r4`–`r7`, and the remaining parameters go on the stack. But floating-point parameters are subject to more complicated rules, and because these rules depend on the type of the first parameter, we need a FIRST_CHOICE stage, as shown in the upper part of Figure 9. The only fine point is the use of the PAD stage. A 64-bit floating-point parameter in the second position goes into integer register pair `r6`–`r7`, regardless of the size of the first parameter; register `r5` may go unused. Because the 64-bit floating-point parameter is the only parameter with an alignment of 8, we achieve this end using PAD.

When the first parameter is a floating-point parameter, we use the middle part of Figure 9. The first two floating-point parameters are placed by counting arguments, not bits: the first parameter occupies either floating-point register `f12` or floating-point register pair `f12`–`f13` (here called `d12`), depending on its size. The second parameter, if it is also a floating-point parameter, goes in either `f14` or `d14`. Remarkably, if a procedure takes four 32-bit floating-point parameters, the first two go in `f12` and `f14`, the next two go in

$$
\begin{array}{l}
\mathit{parms} = \\
\quad [\textsc{widen}(\lambda w.\mathrm{round\_up}(w, 32)) \\
\quad , \textsc{argcounter}(\texttt{"args"}) \\
\quad , \textsc{bitcounter}(\texttt{"bits"}) \\
\quad , \textsc{pad}(\texttt{"bits"}) \\
\quad , \textsc{first\_choice}(c_{fc}, \\
\qquad [\lambda\langle w, k, \sigma\rangle.k = \texttt{"float"}, \langle\textit{first parameter is floating-point}\rangle, \\
\qquad , \lambda\langle w, k, \sigma\rangle.\mathbf{true}, [\,] \\
\qquad ]) \\
\quad , \textsc{regs\_by\_bits}(\texttt{"bits"}, [\texttt{r4}, \ldots, \texttt{r7}]) \\
\quad , \textsc{overflow}(c_o, \textsc{up}, 16) \\
\quad ]
\end{array}
$$

$$
\begin{array}{l}
\langle\textit{first parameter is floating-point}\rangle \equiv \\
\quad \textsc{choice}( \\
\qquad [\lambda\langle w, k, \sigma\rangle.k = \texttt{"float"} \wedge w = 32, \\
\qquad\quad \textsc{regs\_by\_args}(\texttt{"args"}, [\texttt{f12}, \texttt{f14}]) \\
\qquad , \lambda\langle w, k, \sigma\rangle.k = \texttt{"float"} \wedge w = 64, \\
\qquad\quad \textsc{regs\_by\_args}(\texttt{"args"}, [\texttt{d12}, \texttt{d14}]), \\
\qquad , \lambda\langle w, k, \sigma\rangle.\mathbf{true}, \\
\qquad\quad [\,] \\
\qquad ])
\end{array}
$$

$$
\begin{array}{l}
\mathit{results} = \\
\quad [\textsc{widen}(\lambda w.\mathrm{round\_up}(w, 32)) \\
\quad , \textsc{widths}([32, 64, 128]) \\
\quad , \textsc{choice}( \\
\qquad [\lambda\langle w, k, \sigma\rangle.k = \texttt{"float"}, \textsc{useregs}([\texttt{f0}, \ldots, \texttt{f3}]) \\
\qquad , \lambda\langle w, k, \sigma\rangle.\mathbf{true}, \textsc{useregs}([\texttt{r2}, \texttt{r3}]) \\
\qquad ]) \\
\quad ]
\end{array}
$$

**Figure 9.** The standard MIPS R3000 calling convention

$$
\begin{array}{l}
\mathit{parms} = \\
\quad [\textsc{widen}(\lambda w.\mathrm{round\_up}(w, 64)) \\
\quad , \textsc{bitcounter}(\texttt{"bits"}) \\
\quad , \textsc{choice}( \\
\qquad [\lambda\langle w, k, \sigma\rangle.k = \texttt{"float"} \wedge \sigma(\texttt{"bits"}) < 512, \\
\qquad\quad \textsc{useregs}([\texttt{f8}, \ldots, \texttt{f15}]) \\
\qquad , \lambda\langle w, k, \sigma\rangle.\mathbf{true}, \\
\qquad\quad \textsc{regs\_by\_bits}(\texttt{"bits"}, [\texttt{out0}, \ldots, \texttt{out7}]) \\
\qquad ]) \\
\quad , \textsc{overflow}(c_o, \textsc{up}, 16)] \\
\mathit{results} = \\
\quad [\textsc{choice}( \\
\qquad [\lambda\langle w, k, al\rangle.k = \texttt{"float"} \wedge w \le 82, \\
\qquad\quad [\textsc{widths}([32, 64, 82]), \textsc{useregs}([\texttt{f8}])] \\
\qquad , \lambda\langle w, k, \sigma\rangle.\mathbf{true}, \\
\qquad\quad [\textsc{widen}(\lambda w.\mathrm{round\_up}(w, 64)) \\
\qquad\quad , \textsc{widths}([64, 128]) \\
\qquad\quad , \textsc{useregs}([\texttt{r8}, \texttt{r9}])]])]
\end{array}
$$

**Figure 10.** The standard IA-64 convention

`r6` and `r7`, and `f13` and `f15` are not used. More examples of MIPS parameter placements appear in Figure 13 in Appendix B.

**IA-64** The standard C convention for IA-64, shown in Figure 10, is also complex. This convention sets aside eight integer and eight floating-point registers for passing parameters. Like the Alpha convention in Figure 2, it uses at most eight of these sixteen registers, even if more than eight parameters are passed. Also like the Alpha convention, it leaves "gaps" in the integer registers to correspond to floating-point parameters. But unlike the Alpha convention, it leaves no gaps in the floating-point registers; instead, it leaves registers unused at the end. To place a floating-point parameter, we need the semantics, "if fewer than eight parameters have been passed, use the next available floating-point register; otherwise go to the overflow block." We implement this convention using the counter-testing predicate $\sigma(\texttt{"bits"}) < 512$.

The *results* convention shows one oddity: most floating-point results are returned in floating-point registers, but a 128-bit, quad-precision floating-point result is returned in the integer register pair `r8`–`r9`.

## 4. Variations and extensions

Section 3 tells only part of the story about allocating locations for parameters: there are a number of variations and possible extensions. Some we know to be useful; others might not be.

When a REGS_BY_BITS stage splits a large parameter across multiple locations, it uses the equation $\ell'' = \mathrm{combine}(\ell, \ell')$. This equation puts the most significant bits of the parameter in the first location, which is appropriate for a big-endian machine. On a little-endian machine, it makes more sense to use the equation $\ell'' = \mathrm{combine}(\ell', \ell)$. Our implementation uses the equation appropriate to the byte order of the target machine.

A convention might want to pass two small parameters together in a single large register. To do so, we could define a variation on REGS_BY_BITS that would split a register into two locations.

A variation on the BITCOUNTER stage could increment its counter not by the width of the request but by the total width of the machine locations that are used to satisfy that request. We have chosen to use the width of the request because that is how most calling conventions seem to be described. Usually the two widths are the same.

Our WIDEN stage stores a small parameter in the least significant bits of a larger location. But some conventions, including the Mac OS X Power PC convention (Apple 2003), store a small parameter in the *most* significant bits of a larger location.

Some conventions, including the OS X convention, reserve "shadow" space on the stack for parameters passed in registers. To support such conventions, we provide alternate versions of the stages above, called REGS_BY_BITS_RESERVE, USEREGS_RESERVE, and REGS_BY_ARGS_RESERVE. Even when such a stage finds a register, it also allocates an extra location from its successor, but it does not do anything with the extra location. In the REGSBYBITS and REGSBYARGS rules in Figure 6, this behavior amounts to adding a new premise

$$\langle ss, \sigma\rangle @ \langle w, k, al\rangle \overset{\hat{\ell}}{\Longrightarrow} \hat{\sigma},$$

ignoring $\hat{\ell}$, and threading $\hat{\sigma}$ appropriately. In our implementations, these "reserving" stages share code with the original versions of the same stages.

Using the RESERVE forms, Figure 11 shows a specification for the Mac OS X convention on the Power PC. This specification is suitable for passing parameters that are multiples of 32 bits in size. It is also suitable only for non-varargs functions, because the OS X convention requires that a parameter passed to the variable part of a varargs C function be passed in *both* floating-point and integer registers (Apple 2003, p55). To implement this convention would require a new kind of composite location $\ell = \mathrm{both}(\ell_1, \ell_2)$ such that writing $\ell$ writes both $\ell_1$ and $\ell_2$. We would also need a BOTH stage with suitable semantics. Finally, we would need the kind $k$ to distinguish a parameter passed to a varargs function from that same parameter passed to a function with a non-varargs prototype.

As these examples should make clear, staged allocation is not a complete, definitive language for specifying calling conventions.

$$
\begin{array}{l}
\mathit{parms} = \\
\quad [\textsc{widen}(\lambda w.\mathrm{round\_up}(w, 32)) \\
\quad , \textsc{bitcounter}(\texttt{"bits"}) \\
\quad , \textsc{choice}( \\
\qquad [\lambda\langle w, k, \sigma\rangle.k = \texttt{"float"}, \\
\qquad\quad [\textsc{widen}(\lambda w.64), \textsc{useregs\_reserve}([\texttt{f1}, \ldots, \texttt{f13}])] \\
\qquad , \lambda\langle w, k, \sigma\rangle.\mathbf{true}, \\
\qquad\quad \textsc{regs\_by\_bits\_reserve}(\texttt{"bits"}, [\texttt{r3}, \ldots, \texttt{r10}]) \\
\qquad ]) \\
\quad , \textsc{overflow}(c_o, \textsc{up}, 4) \\
\quad ] \\
\mathit{results} = \textsc{choice}( \\
\quad [\lambda\langle w, k, \sigma\rangle.k = \texttt{"float"}, \\
\qquad [\textsc{widen}(\lambda w.64), \textsc{useregs}([\texttt{f1}])] \\
\quad , \lambda\langle w, k, \sigma\rangle.\mathbf{true}, \\
\qquad [\textsc{widen}(\lambda w.32), \textsc{useregs}([\texttt{r3}, \texttt{r4}])] \\
\quad ])
\end{array}
$$

**Figure 11.** The OS X PowerPC convention

Rather, it is a framework for organizing the specification and implementation of calling conventions—a framework that covers most conventional techniques, but that can be expanded at need.

## 5. Results

We report on experience implementing and using staged allocation, and we compare staged allocation with CCL.

### 5.1 Implementation experience

Staged allocation is not just for formal specification; it is intended to be easy to implement. To evaluate the cost of implementation, we have added staged allocation to two different compilers: Quick C--, which is implemented in a combination of Objective Caml and Lua, and Machine SUIF, which is implemented in C++.

- Quick C-- is an implementation of C--, which is a language and a run-time interface whose primary mission is to support retargetable compilation of multiple programming languages (Peyton Jones, Ramsey, and Reig 1999; Ramsey and Peyton Jones 2000). The C-- language 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. When a value is passed to a separately compiled procedure, the C-- compiler needs help deciding what kind of register should hold it. A C-- program provides such help by attaching a kind to every actual and formal parameter; at any call site, the kind of each actual parameter must match the kind of the corresponding formal parameter at the declaration of the procedure called. Inside the compiler, kinds are passed directly to automata.
- Machine SUIF is a flexible, extensible compiler infrastructure whose primary mission is to support the development of machine-specific optimizations and profile-driven optimizations (Smith and Holloway 2000). It is also used to evaluate architectural ideas.

  Internally, Machine SUIF treats a source-language type as an abstraction that has a size and alignment. The abstraction can also be asked whether its values are Booleans, integers (signed or unsigned), floating-point numbers, pointers, enumeration literals, structures, unions, or arrays. The answer is used to compute a kind for staged allocation.

In both compilers, the compile-time costs of using staged allocation are negligible. For example, `gprof` reports that $0.1\%$ of Quick C--'s execution time is spent in staged allocation; this number is comparable to measurement error. We therefore focus on the programmer-time costs: the effort of writing the implementations.

Developing staged allocation required significant intellectual effort, but the results of that effort are captured concisely in Figure 6, which we have used to guide our implementations. The programming effort can be summarized by these statistics:

| Quick C-- (Caml) | | Machine SUIF (C++) | |
|---|---|---|---|
| 88 | lines glue code | | |
| 229 | lines main code | 621 | lines code |
| | | 20 | constructor methods |
| 45 | top-level functions | 63 | other methods |
| 14 | data types | 29 | classes |

It is difficult to make meaningful comparisons between Caml programs and C++ programs, but the main points appear to be these:

- Both implementations are small: 229 and 621 lines respectively. Only non-blank, non-comment lines are counted.
- Quick C-- uses an additional 88 lines of "glue code," which makes the Caml implementation callable from Lua scripts (Ramsey 2005). Using Lua has two advantages: it is consistent with the way we configure the whole Quick C-- compiler, and it enables us to experiment with (and debug!) calling conventions without rebuilding the compiler. The Machine SUIF compiler does not use Lua, so our C++ implementation has no glue code.
- Although both implementations follow Figure 6, the Caml code has significantly fewer top-level functions than the C++ code has methods, even when constructor methods are omitted. This is because Caml allows nested functions but C++ does not allow nested methods.
- The C++ code requires classes to do jobs that the Caml code handles using other language constructs. Of the C++ classes, 6 represent record types or exceptions, 3 represent abstractions used in the implementation (e.g., location), and 20 are used in the implementations of stages. Stages written in Caml use anonymous functions, which need no special declarations.

The effort of building our implementation for Quick C-- is inseparable from the effort of developing Figure 6. But with Figure 6 in hand, we were able to build an implementation for Machine SUIF fairly quickly: one of us (Olinsky), who had no prior experience with compiler back ends or with Machine SUIF, built a first implementation in one week. Then, part-time over three weeks, he embedded the implementation in Machine SUIF, debugged it, and simplified the code. The C++ code that implements calling conventions using staged allocation is significantly easier to read and maintain than the code it replaced:

- The calling convention is implemented only once, where before it had been implemented both at call sites and in the generation of a procedure's prolog.
- Structured arguments and scalar arguments are handled using the same code, where before structures had been handled specially and separately.

We conclude that the effort required to implement staged allocation is small and that the benefits are worth the effort.

### 5.2 Specification experience

As shown above, we have written specifications for standard C calling conventions on the Alpha, IA-64, MIPS, Pentium, and Power PC. The Alpha, IA-64, and Pentium back ends have passed Bailey and Davidson's (2003) tests for interoperation with the native C compiler. Our MIPS machine is not fast enough to run these tests, and on the Power PC, Bailey and Davidson's tests are im-

| Machine | Staged Allocators | CCL parms+res | CCL total |
|---|---|---|---|
| Alpha | 13+6 | — | — |
| IA-64 | 11+12 | — | — |
| MIPS R3000 | 20+7 | 37+11 | 63 |
| Pentium | 4+9 | — | — |
| PowerPC | 12+6 | — | — |
| SPARC* | 5+6 | 17+10 | 45 |
| VAX* | 2+3 | 14+10 | 37 |
| 68020* | 2+3 | 11+11 | 35 |
| 88100* | 5+3 | 21+10 | 42 |

**Table 1.** Number of lines in specifications (* means untested)

practical: even if we limit parameter types to `char`, `short`, `int`, `float`, and `double`, Bailey and Davidson's procedure generates over 70,000 tests. Instead, these platforms have passed a symbolic test, which symbolically evaluates assembly language from the *native* compiler and checks that registers are used as predicted by our specifications. They have also passed execution tests on functions with randomly generated prototypes (Lindig 2005).

We have also written specifications of SPARC, VAX, 68020, and 88100 calling conventions. We cannot test them at present, but to enable comparisons with CCL, we include them in Appendix A.

Most of the effort of writing a specification goes into understanding the calling convention. Writing the simple specifications for the Pentium, SPARC, VAX, 68000, and 88000 took just minutes each, for example. Writing the IA-64 specification required deeper thought; eventually we decided we wanted a predicate to test the value of a named counter. Once we used this predicate, the IA-64 specification worked on its first tests. The most difficult specification to write was the MIPS specification, because it is so difficult to understand the convention. We did so by reverse engineering `lcc`'s implementation (Fraser and Hanson 1995). This job was fairly easy because `lcc`'s `argreg` function matches up nicely with our allocation stages: it counts arguments (ARGCOUNTER), counts bytes of parameters allocated (related to BITCOUNTER), remembers whether the first argument was a floating-point argument (FIRST_CHOICE), and uses the byte count to index an array of registers (REGS_BY_BITS). By automatically comparing our code with code from the native C compiler, we debugged our specification in about half an hour.

Table 1 compares the sizes of our specifications and CCL specifications; it counts the number of non-blank, non-comment lines required to write each specification. For staged allocation, we show line counts from our Lua implementations in Quick C--. For CCL, we show counts from Appendix B of Bailey's (2000) PhD thesis. Each count for value passing, whether for staged allocation or for CCL, is reported as a sum: parameters+results. To make comparisons fair, we show two counts for each CCL specification: the "parms+res" count includes only those parts of the CCL specification that describe value passing; "view changes" and global aliases are omitted. The "total" count shows the total size of each CCL specification, including parts that are not used by the CCL tools. Even when these unused parts are omitted, our specifications are about half the size of CCL's.

## 5.3 Other comparisons with CCL

Before making further comparisons with CCL, we provide a brief summary based on Bailey 2000, §4.3. Unfortunately, CCL descriptions are too big to include examples in this paper.

**Summary of CCL** CCL treats each parameter, each result, and each location as a "resource." Like our machine locations, each resource has a width, but it also has two Boolean properties: whether it has been allocated to hold a parameter and whether it is available to be allocated. The values of these properties change as the state of an automaton changes. A resource that represents a parameter or result also has a type, which appears to be equivalent to a kind in staged allocation.

A CCL specification describes a mapping from each parameter to a location. Locations are arranged in ordered sets of ordered sets, and a CCL description includes code that indicates from which ordered set locations should be allocated. The CCL interpreter iterates through the parameters, at each step allocating the first location whose properties identify it as unallocated and available. If the location is too small, CCL appears to allocate more locations and combine them somehow. If the location is too big, CCL returns it, and the compiler must figure out what to do with the too-big location.[1]

A CCL description includes rules that trigger changes to properties as the state of the automaton changes. In practice, these rules are triggered by the allocation of a location, and they are used to mark other locations as unavailable. For example, in the CCL MIPS specification, if floating-point register `f12` is allocated, the rules mark integer registers `r4` and `r5` as unavailable. In other words, the rules are used to implement an *exclusion relation*.

CCL also provides other information. It lists the registers whose values are preserved across calls, and it also includes something called a *view change*, which serves two purposes: it accounts for changes in the names of stack locations as the stack pointer moves, and on machines that have register windows, it accounts for changes in the names of registers as register windows move. This information appears not to be used by the CCL tools.

**Comparison** The work based on CCL is largely complementary to staged allocation.

- Bailey and Davidson's work is all about P-FSAs: identifying this class of automata, devising an enumeration procedure for computing P-FSAa, analyzing P-FSAs, and using P-FSAs to generate test cases. The specification language, CCL, is almost an afterthought, which explains why it has no published syntax or semantics.
- Staged allocation is all about the specification language: defining the syntax and semantics, expressing standard calling conventions, repeatedly refining and simplifying the language, and implementing it in different compilers. The P-FSA is an afterthought; we have replicated Bailey and Davidson's algorithms for analysis and testing, but in our implementations, we never build a P-FSA or related tables.

**Implementation comparison** At 229 lines of Objective Caml and 621 lines of C++, our implementations of staged allocation are much smaller than the implementation of CCL. The program generator for CCL is about 2500 lines of Icon (Griswold and Griswold 1996). Because Icon is a very high-level language, a line of Icon is more nearly equivalent to a line of Objective Caml than to a line of C++. CCL's program generator emits a table that must be interpreted at compile time; the interpreter requires another 200 lines of C.

To port either staged allocation or CCL to a new compiler need not require writing any new code, provided the implementation language of the new compiler is Objective Caml or C++ (for staged allocation) or C (for CCL). But if the new compiler is written in a new language, like Java for example, significant reimplementation

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

would be required for either staged allocation or CCL. In the case of staged allocation, the rules in Figure 6 would have to be reimplemented in Java. In the case of CCL, the table interpreter would have to be rewritten in Java, and some part of the program generator would have to be rewritten to emit tables in Java. Because the implementation of CCL is no longer available, we don't know how much of CCL's program generator would have to be rewritten.

It might seem unfair to compare the implementations of CCL and staged allocation, because CCL talks about the stack pointer and about volatility of registers, and staged allocation does not. But as far as we can tell, CCL is used only to place parameters and results; its implementation does not generate code to deal with movement of the stack pointer or with volatility of registers.

## 6. Discussion

**Evaluating the design** There is no obvious yardstick by which to measure the quality of a language for describing calling conventions. The common "core-language" approach to design—in which the best language is the one that is the most expressive while having the fewest primitives—is not appropriate for calling conventions. The reason is that there are many conventions we would prefer not to express. Because *any* function that maps each list of parameter types to a list of distinct machine locations is a legitimate parameter-passing convention, the space of potential conventions is large. But there are far more "bad" conventions than "good" ones. (A convention is bad if it wastes machine resources or is too complicated.) So it is a bad idea to ask for the simplest language that expresses the most conventions.

A better approach is a "domain-specific" approach in which a good language *restricts* what can be said. But because the design space of good calling conventions is not well understood, a restrictive approach runs the risk that the domain-specific language may be unable to express some necessary convention. We have addressed that risk by designing staged allocation as an extensible framework, not just a language. The essence of the framework is captured by the form of the judgment $\langle ss, \sigma\rangle @ \langle w, k, al\rangle \stackrel{\ell}{\Longrightarrow} \sigma'$. This judgment, together with the stages we have designed, captures our considered opinions about a space of "good" conventions.[2] As language primitives, our stages are big by design; we hope that using big primitives will make it more likely not only that simple specifications describe good conventions but also that good conventions can be described by simple specifications. For this reason, for example, we have made FIRST_CHOICE a primitive even though it can be defined by some (slightly scary) syntactic sugar.

The centrality of the judgment $\langle ss, \sigma\rangle @ \langle w, k, al\rangle \stackrel{\ell}{\Longrightarrow} \sigma'$ has several consequences.

- New kinds of stages can be added by hand with relatively few constraints: a stage should not base decisions on information that is not part of the formal model (such as the names of parameters), and it should not maintain private state that is not in $\sigma$. Of course, if a new kind of stage is to be useful for specification as well as implementation, it must be given a formal semantics.

  These loose constraints leave room for bad new ideas as well as good. For example, it would be easy but incorrect to add a stage that allocates a single location to multiple parameters (although such an error would be caught by Bailey and Davidson's analysis of consistency). An idea we consider bad but could be done correctly would be to choose a counter in $\sigma$ based on information about parameters.

[2] FIRST_CHOICE is not "good," but it is necessary to support C calling conventions that are, in our opinion, poorly designed.

- Our framework precludes some attractive language-design possibilities. For example, given their close similarity, it would be satisfying if REGS_BY_BITS and REGS_BY_BITS_RESERVE could be expressed as different combinations of simpler primitives. The problem is that decomposing these operations into simpler primitive stages would require more communication between the primitives. We prefer to keep the communication simple and to live with the near-duplication of primitives.
- Our framework excludes not only many obviously bad conventions but also some that might be considered reasonable. For example, there is no way to express any convention of the form "pass parameters in registers if and only if *all* parameters can be passed in registers; otherwise, pass all parameters on the stack." Such a convention is certainly conceivable, but in the interest of keeping common cases simple, we have chosen to describe only conventions in which parameters can be allocated one at a time (in some order).

Returning to evaluation, we believe that, aside from purely internal questions about the perspecuity of its syntax and semantics, a language for describing calling conventions should be evaluated empirically. Does it describe common conventions? Can good conventions be specified simply? Does the language discourage bad conventions? Can it stretch to accommodate legacy conventions, however awkward? By these measures, we are satisfied with staged allocation.

**Completing the convention** Although the rules for passing parameters and results are the most complicated part of a calling convention, a convention also has other parts:

- The convention says where overflow parameters and results are expected; stack-frame layout must put them there. Frame layout must also put saved registers where the convention says the run-time system expects to find them.
- The convention says how a return address is passed to a callee.
- The convention says what registers the compiler may use and which of those registers must be preserved across calls.
- The convention says which way the stack grows, which register is the stack pointer, and how the stack pointer is aligned.
- The convention says whether deallocating stack space used to pass parameters is the job of the caller or the callee. In typical C calling conventions, the caller deallocates, but in a convention that supports proper tail calls, the callee must deallocate (Ramsey and Lindig 2002, §4).

These parts of a calling convention are trivial to specify and easy to implement, except for frame layout. We lay out stack frames using a declarative, constraint-based technique, which computes the location of each slot in a stack frame by accumulating and solving a set of simultaneous linear equations (Lindig and Ramsey 2004).

**Other techniques for specifying automata** One might think that automata naturally lead to regular expressions. But regular expressions and the tools based on them are designed only to *accept* certain sequences of inputs and to reject others. A parameter-passing automaton must not only accept a sequence of inputs (parameter types); it must also produce a location for each input. And it must accept *every* sequence of inputs; if a parameter-passing automaton rejects any sequence of inputs, the convention it specifies is *incomplete* and therefore incorrect. Regular expressions are not much use for specifying calling-convention automata.

We can imagine specifying an automaton by giving its nodes and edges. Any set of nodes and edges can be written using FIRST_CHOICE, USEREGS, and OVERFLOW. But a direct specification of nodes and edges would be long, hard to write, and hard to read: Bailey and Davidson (1996) report 9 nodes and 90 edges for the relatively simple SPARC convention; the more complex MIPS convention takes 70 nodes and 772 edges. Our own measurements show that the Mac OS X Power PC convention takes at least 2,815 nodes and 28,150 edges. Direct specification would be impractical.

The existing technique most closely related to staged allocation is *combinator parsing* (Hutton 1992), which could be used to map a sequence of types to a sequence of locations. The main difference in feel would be in handling choices: where staged allocation uses explicit predicate functions, classic combinator parsing handles choice using a success/failure model and a choice operator that takes two parsers and returns a parser. We could adopt this model for staged allocation, but we think our CHOICE stage and predicates will be easier for a compiler writer who has not seen combinator parsing.

**Applications** Our primary goal has been to build a retargetable compiler that supports standard C calling conventions. But staged allocation is good for more than just standard conventions. For example, we use staged allocation to define a special convention that is used to start a new user-level thread. To start a thread, we take a function $f$ and a value $x$, create a stack, and return a program counter that, when jumped to, calls $f(x)$ on that stack. Ordinarily, the program counter we return would point to a snippet of code written in assembly language—this code would get $f$ and $x$ off the stack and call $f(x)$. By defining a special-purpose calling convention, we make it possible to write this code in our source language, instead. The code is written using the `"C-- thread"` calling convention, which looks for parameters on the stack, not in registers. The calling convention is named using the `foreign` keyword.

```
foreign "C-- thread"
Cmm_start_thread_helper(bits32 f, bits32 x) {
  f(x);
  foreign "C" abort();
}
```

The call to $f(x)$ is not supposed to return, so if it does return, we call the C function `abort`.

Using staged allocation to define new calling conventions may have other benefits. In Quick C--, it is especially easy to define a new calling convention, because the Lua specifications of calling conventions are read at compile time. By pointing to different Lua code on the compiler's command line, we can change calling conventions without rebuilding the compiler. By making changes so easy, we hope to enable more extensive experiments along lines set out by Davidson and Whalley (1991).

There are also advantages to specifying new calling conventions to be used only at certain call sites. For example, many compilers inline the fast path of allocation (Appel 1992), which means that statically, there are many calls to the garbage collector, but dynamically, these calls are rare. It therefore makes sense to design a special "GC convention" with the goal of minimizing code size. One candidate is a convention in which as many registers as possible are preserved across the call.[3] In particular, if the call to the garbage collector passes no parameters, the convention should not set aside any registers for passing parameters—registers that hold parameters in other conventions should be callee-saves in the GC convention. With similar goals in mind, specialized GC conventions are used in production compilers, such as Standard ML of New Jersey (George 1999).

[3] This convention implies unnecessary saves at most calls, but the cost of saving some registers unnecessarily is tiny compared to the cost of garbage collection.

**Conclusion** Staged allocation is simple, readable, precisely defined, and easy to implement. It can describe a variety of useful conventions with a minimum of notation. It can be packaged in a configuration language, like Lua, or it can be implemented in a compiler's native language, like C++. We hope these properties will make it the specification technique of choice for future conventions and the implementation technique of choice for future compilers.

ACKNOWLEDGEMENTS

Our implementation in Machine SUIF would have been impossible without the enthusiasm and support of Glenn Holloway. Sukyoung Ryu found many errors in a draft of this paper. João Dias, Simon Peyton Jones, Andreas Rossberg, and Mike Smith made helpful suggestions. This paper has also benefited from the comments, criticisms, and suggestions of many anonymous referees. We are especially grateful to the POPL referees for their difficult and insightful questions.

This work has been supported by NSF grants CCR-0096069 and ITR-0325460, by an Alfred P. Sloan Research Fellowship, and by a gift from Microsoft.

## References

Appel, Andrew W. 1992. *Compiling with Continuations*. Cambridge: Cambridge University Press.

Apple Computer. 2003. *Mach-O Runtime Architecture*.

Bailey, Mark W. 2000. *CSDL: Reusable Computing System Descriptions for Retargetable Systems Software*. PhD thesis, University of Virginia, Dept of Computer Science.

Bailey, Mark W. and Jack W. Davidson. 1995. 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. 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 (May): 249–257.

———. 2003. Automatic detection and diagnosis of faults in generated code for procedure calls. IEEE Transactions on Software Engineering 29 (November): 1031–1042.

Davidson, Jack W. and David B. Whalley. 1991. Methods for saving and restoring register values across function calls. Software—Practice & Experience 21 (2): 149–165.

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

George, Lal. 1999. SMLNJ: Garbage collection API. As of November 2005, available from `http://www.smlnj.org/compiler-notes/gc-api.ps`.

Griswold, Ralph E. and Madge T. Griswold. 1996. *The Icon Programming Language*. Third edition. San Jose, CA: Peer-to-Peer Communications.

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

Ierusalimschy, Roberto. 2003. *Programming in Lua*. Lua.org. ISBN 85-903798-1-7.

Leroy, Xavier, Damien Doligez, Jacques Garrigue, Didier Rémy, and Jérôme Vouillon. 2004. *The Objective Caml system release 3.08: Documentation and user's manual*. INRIA. Available at `http://pauillac.inria.fr/ocaml/htmlman`.

Lindig, Christian. 2005. Random testing of C calling conventions. In *Sixth International Symposium on Automated and Analysis-Driven Debugging (AADEBUG)*, pages 3–11.

Lindig, Christian and Norman Ramsey. 2004. Declarative composition of stack frames. In *13th International Conference on Compiler Construction (CC 2004)*, Vol. 2985 of *LNCS*, pages 298–312.

Mealy, George H. 1955. A method for synthesizing sequential circuits. Bell System Technical Journal 34 (5): 1045–1079.

Peyton Jones, Simon L., Norman Ramsey, and Fermin Reig. 1999. `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.

Ramsey, Norman. 2005. Embedding an interpreted language using higher-order functions and types. Journal of Functional Programming. To appear. A preliminary version of this paper appeared in *Proceedings of the ACM Workshop on Interpreters, Virtual Machines, and Emulators*, June 2003.

Ramsey, Norman and Christian Lindig. 2002. Custom calling conventions in a portable assembly language. Unpublished paper available at `http://www.eecs.harvard.edu/~nr/pubs/custom-abstract.html`.

Ramsey, Norman and Simon L. Peyton Jones. 2000. 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 (May): 285–298.

Smith, Michael D. and Glenn Holloway. 2000. An introduction to Machine SUIF and its portable libraries for analysis and optimization. See `http://www.eecs.harvard.edu/machsuif/software/nci/overview.html`.

Stroustrup, Bjarne. 1997. *The C++ Programming Language*. Third edition. Reading, MA: Addison-Wesley.


## A. More examples

For illustrative purposes, Figure 12 shows example specifications for several machines that are not supported by our compiler. These specifications are untested, but at least they give an idea of size and complexity.

## B. Details of the MIPS R3000 convention

Here we explore some details of the calling convention for the MIPS R3000. Figure 13 shows where parameters are placed by calls to some 4-argument C functions. The left column shows the type of each parameter: `d` for `double`, `i` for `int`, and `f` for `float`. The right column shows the location in which each parameter is passed. The notation $k$`(sp)` indicates a location on the stack, in the overflow block.

Figure 14 shows the exclusion relation for today's MIPS R3000 convention as implemented by `lcc` (Fraser and Hanson 1995), version 4.2. We computed the relation by running `lcc` on all C procedures that pass `int`, `float`, or `double` parameters in registers. We then parsed the assembly output and identified the sets in the figure, which are sets of registers that are never used in the same call.

The same exclusion relation can be expressed in the style of CCL using rules; for example, one rule says "if both `f13` and `f14` are used, then `r6` may not be used." To express the relation requires 23 such rules. This example, brief though it is, suggests why we prefer to specify complex conventions using CHOICE and FIRST_CHOICE constructs, not using exclusion.

$$
\begin{aligned}
&\mathit{VAX\ parms} = \\
&\quad [\textsc{overflow}(c_o, \textsc{up}, 4)] \\
&\mathit{VAX\ results} = \\
&\quad [\textsc{widen}(\lambda w.\mathrm{round\_up}(w, 32)), \textsc{useregs}([\mathtt{r0}, \mathtt{r1}])]
\end{aligned}
$$

$$
\begin{aligned}
&\mathit{68020\ parms} = \\
&\quad [\textsc{overflow}(c_o, \textsc{up}, 8)] \\
&\mathit{68020\ results} = \\
&\quad [\textsc{widen}(\lambda w.\mathrm{round\_up}(w, 32)), \textsc{useregs}([\mathtt{d0}, \mathtt{d1}])]
\end{aligned}
$$

$$
\begin{aligned}
&\mathit{88100\ parms} = \\
&\quad [\textsc{widen}(\lambda w.32) \\
&\quad , \textsc{useregs}([\mathtt{r2}, \ldots, \mathtt{r9}]) \\
&\quad , \textsc{overflow}(c_o, \textsc{up}, 8) \\
&\quad ] \\
&\mathit{88100\ results} = \\
&\quad [\textsc{widen}(\lambda w.\mathrm{round\_up}(w, 32)), \textsc{useregs}([\mathtt{r2}, \mathtt{r3}])]
\end{aligned}
$$

$$
\begin{aligned}
&\mathit{SPARC\ parms} = \\
&\quad [\textsc{widen}(\lambda w.\mathrm{round\_up}(w, 32)) \\
&\quad , \textsc{useregs}([\mathtt{r8}, \ldots, \mathtt{r13}]) \\
&\quad , \textsc{overflow}(c_o, \textsc{up}, 8) \\
&\quad ] \\
&\mathit{SPARC\ results} = \\
&\quad [\textsc{widen}(\lambda w.\mathrm{round\_up}(w, 32)) \\
&\quad , \textsc{choice}( \\
&\qquad [\lambda\langle w, k, \sigma\rangle.k = \texttt{"float"}, \\
&\qquad\quad \textsc{useregs}([\mathtt{f0}, \mathtt{f1}]) \\
&\qquad , \lambda\langle w, k, \sigma\rangle.\mathbf{true}, \\
&\qquad\quad \textsc{useregs}([\mathtt{f8}]) \\
&\qquad ]) \\
&\quad ]
\end{aligned}
$$

**Figure 12.** Untested specifications for which we have no corresponding back ends

| | |
|---|---|
| d·d·i·f | f12–f13 · f14–f15 · 16(sp) · 20(sp) |
| d·i·d·i | f12–f13 · r6 · 16(sp) · 24(sp) |
| d·i·i·f | f12–f13 · r6 · r7 · 16(sp) |
| i·i·i·i | r4 · r5 · r6 · r7 |
| i·i·i·d | r4 · r5 · r6 · 16(sp)–20(sp) |
| i·i·d·i | r4 · r5 · r6–r7 · 16(sp) |
| i·d·i·i | r4 · r6–r7 · 16(sp) · 20(sp) |
| d·d·i·i | f12–f13 · f14–f15 · 16(sp) · 20(sp) |
| f·f·f·f | f12 · f14 · r6 · r7 |
| f·i·f·i | f12 · r5 · r6 · r7 |
| d·f·f·i | f12–f13 · f14 · r7 · 16(sp) |
| f·f·d·i | f12 · f14 · r6–r7 · 16(sp) |
| i·f·i·f | r4 · r5 · r6 · r7 |
| i·f·i·i | r4 · r5 · r6 · r7 |
| i·i·f·i | r4 · r5 · r6 · r7 |

**Figure 13.** Example parameter placements on MIPS R3000

$$
\{\{\mathtt{r4}, \mathtt{f12}\}, \{\mathtt{r4}, \mathtt{f13}\}, \{\mathtt{r4}, \mathtt{f14}\}, \{\mathtt{r4}, \mathtt{f15}\}, \{\mathtt{r5}, \mathtt{f13}\}, \{\mathtt{r5}, \mathtt{f14}\}, \{\mathtt{r5}, \mathtt{f15}\}, \{\mathtt{r6}, \mathtt{f13}, \mathtt{f14}\}, \{\mathtt{r6}, \mathtt{f15}\}, \{\mathtt{r7}, \mathtt{f15}\}\}
$$

**Figure 14.** Exclusion sets for the MIPS R3000 convention