Staged Allocation: Engineering the Specification and Implementation of Procedure Calling Conventions

Reuben Olinsky
Christian Lindig
and
Norman Ramsey

TR-02-04

Computer Science Group
Harvard University
Cambridge, Massachusetts

Staged Allocation: Engineering the Specification and Implementation of Procedure Calling Conventions

Reuben Olinsky Christian Lindig Norman Ramsey
Division of Engineering and Applied Sciences
Harvard University
{olinsky,lindig,nr}@eecs.harvard.edu

Abstract

We present *staged allocation*, a new technique for specifying calling conventions. A specification written using staged allocation has a precise, formal semantics, and it can be executed directly inside a compiler. Specifications of nine 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++.

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, a compiler can fail even on an example 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 Bailey and Davidson's language, CCL, is concise, but it is defined only by example and by an implementation that is no longer maintained. Also, a program that generates code from specifications can be hard to reuse, because the generated code must be compatible with your compiler's implementation language, intermediate representation, and so on.

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 be more powerful than the 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 one-off calling conventions, 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 two implementations, in the Quick C-- and Machine SUIF compilers, are about 250 lines of ML and 650 lines of C++, respectively.

We have used our implementations to specify and test standard calling conventions on five machines. Three are shown in this paper; the others appear in Appendix A, which is on the Web.[1]

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. Stack-frame layout, which is closely related to the *implementation* of calling conventions, is discussed in a companion paper (Lindig and Ramsey 2004).

2.1 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

[1] At http://www.eecs.harvard.edu/~nr/pubs/eng-abstract.html, you can find a paper identical to this one except that it includes two appendices. But it may not be worth the trouble.

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.

We also assume that parameters can be placed 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 inherent in the nature of procedure calls, but because of variadic procedures (varargs), it *is* inherent in the C calling convention. 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, a more general model (in which the type of a later parameter could affect the placement of an earlier parameter) appears to offer no advantages.

2.2 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 the infinite-state automaton onto a finite-state automaton they call the P-FSA. The mapping is homomorphic: it preserves the structure of the transitions of the original automaton.
- 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. A CCL description is used to create a P-FSA via an enumeration procedure, which repeatedly calls a CCL interpreter and exhaustively finds the P-FSA's states and transitions. The P-FSA is represented as a set of tables, which are interpreted by a second, smaller interpreter that places parameters at compile time.
- They exploit the structure of P-FSAs to develop target-specific test suites for calling conventions (Bailey and Davidson 1996).

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. Staged allocation fully supports P-FSAs and their associated analysis and testing techniques. And by contrast with CCL, staged allocation is precisely defined, simpler, and easier to engineer into a compiler, as we show below. A comparison appears in Section 5.

3 Specifying parameter-passing automata

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 discuss types here, then present two examples before moving on to locations and automata.

```
X86.cc["C"].call =
  { A.widen(32, "multiple")
  , A.overflow { growth = "up", max_alignment = 4 }
  }

X86.cc["C"].results =
  A.choice
     { "float" , { A.widen(80)
                 , A.useregs { X86.fp_stack_top }
                 }
     , A.is_any, { A.widen(32, "multiple")
                 , A.useregs { X86.eax, X86.edx }
                 }
     }
```

Figure 1: Pentium calling conventions

3.1 Formalizing types

Staged allocation is not tied to any particular high-level language or 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.

There is 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.

```
Alpha.cc["C"].call =
  { A.widen(64, "multiple")
  , A.bitcounter("bits")
  , A.choice
        { "float",
             A.regs_by_bits("bits", f(16)..f(21))
        , A.is_any,
             A.regs_by_bits("bits", r(16)..r(21))
        }
  , A.overflow { growth = "up", max_alignment = 16 }
  }

Alpha.cc["C"].results =
  { A.widen(64, "multiple")
  , A.choice { "float" , A.useregs { f(0), f(1) }
             , A.is_any, A.useregs { r(0) }
             }
  }
```

Figure 2: Alpha calling conventions

$$
\begin{aligned}
\text{read}(\text{narrow}(\ell, w, \texttt{"float"})) &= \mathit{f2f}_{\ell.\texttt{width}\to w}(\text{read}(\ell)) \\
\text{write}(\text{narrow}(\ell, w, \texttt{"float"}), v) &= \text{write}(\ell, \mathit{f2f}_{w\to\ell.\texttt{width}}(v)) \\
\text{read}(\text{narrow}(\ell, w, \_)) &= \mathit{lobits}_w(\text{read}(\ell)) \\
\text{write}(\text{narrow}(\ell, w, \_), v) &= \text{write}(\ell, \mathit{sx}_{w\to\ell.\texttt{width}}(v))
\end{aligned}
$$

$$
\begin{aligned}
\text{read}(\text{combine}(\ell, \ell')) &= \text{read}(\ell) \ll \ell'.\texttt{width} + \text{read}(\ell') \\
\text{write}(\text{combine}(\ell, \ell'), v) &= \text{write}(\ell, v \gg \ell'.\texttt{width}); \\
&\qquad \text{write}(\ell', \mathit{lobits}_{\ell'.\texttt{width}}(v))
\end{aligned}
$$

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

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 3: Reading and writing composite locations

An automaton is specified by composing *stages*, which are defined precisely and discussed in detail in Section 3.4 below—here we show examples.

In the examples, as in our Quick C-- compiler, the specifications are written in the embedded programming language Lua (Ierusalimschy 2003; Ramsey 2004). However, staged allocation does not require Lua; it is easily implemented in a functional, imperative, or object-oriented language. For example, our implementation in Machine SUIF uses C++ directly; Lua is not involved.

Figure 1 shows the standard C calling convention for a Pentium running Linux. The code in the figure, when evaluated, produces two values: one for parameters and one for results. The value `X86.cc["C"].call` specifies the automaton used to place parameters. First every parameter is 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, 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 value `X86.cc["C"].result` specifies the automaton used to place a result. This automaton makes a choice based on the kind of the result. 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. A result of any other kind 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; the `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 can be read or written. Registers and memory are atomic locations with primitive read and write operations. Composite locations are created using the functions "narrow" and "combine," which are defined in Figure 3. The six equations at the top of Figure 3 specify what happens when we read from a composite location or write a value v into a composite location; narrow locations are built using floating-point operations for the kind `"float"` and using integer operations for other kinds. The two equations in the middle give the widths of composite locations. 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. Such a sequence is produced by evaluating code, which may be Caml code, C++ code, or Lua code like that shown in Figures 1 and 2. In this section, we present an internal representation of ss, we show how evaluation works, and we show how ss is used to compute the location of a parameter.

The internal representation of ss is given by the grammar in Figure 4. To write sequences, we use notation borrowed from functional programming languages. 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 $ss +\!\!+ ss'$.

$$
\begin{aligned}
ss &::= s :: ss \mid [] \\
s &::= ss \mid \textsc{overflow}(c, g, \mathit{max\_align}) \mid \textsc{widths}(ws) \mid \textsc{widen}(f) \mid \textsc{align\_to}(f) \\
&\quad \mid \textsc{argcounter}(c) \mid \textsc{bitcounter}(c) \mid \textsc{pad}(c) \mid \textsc{regs\_by\_args}(c, rs) \mid \textsc{regs\_by\_bits}(c, rs) \\
&\quad \mid \textsc{choice}([p_1, s_1, \ldots, p_n, s_n]) \mid \textsc{first\_choice}([p_1, s_1, \ldots, p_n, s_n])
\end{aligned}
$$

(N.B. `useregs`(rs) is syntactic sugar for $[\textsc{bitcounter}(c), \textsc{regs\_by\_bits}(c, [rs])]$, where c is a fresh counter.)

Figure 4: Abstract syntax of stages

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

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

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

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

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

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

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

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

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

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

$$\frac{\langle ss' +\!\!+ ss, \sigma \rangle @ \langle w, k, al \rangle \overset{\ell}{\Longrightarrow} \sigma'}{\langle ss' :: ss, \sigma \rangle @ \langle w, k, al \rangle \overset{\ell}{\Longrightarrow} \sigma'}$$

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

$$\frac{\sigma(c) = n \qquad \text{drop}(n, rs) = [] \qquad \langle ss, \sigma \rangle @ \langle w, k, al \rangle \overset{\ell}{\Longrightarrow} \sigma'}{\langle \textsc{regs\_by\_args}(c, rs) :: ss, \sigma \rangle @ \langle w, k, al \rangle \overset{\ell}{\Longrightarrow} \sigma'}$$

$$\frac{\sigma(c) = n \qquad \text{drop\_bits}(n, rs) = [] \qquad \langle ss, \sigma \rangle @ \langle w, k, al \rangle \overset{\ell}{\Longrightarrow} \sigma'}{\langle \textsc{regs\_by\_bits}(c, rs) :: ss, \sigma \rangle @ \langle w, k, al \rangle \overset{\ell}{\Longrightarrow} \sigma'}$$

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

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

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

Figure 5: 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	σ	state
w, ws	width(s) in bits	n	integer
r, rs	register(s)	C	list of choices

Figure 6: Summary of notation

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

Figure 7: Auxiliary functions

A specification *ss* is produced by an evaluation step, which can take place at compile-compile time, as in our implementation in Machine SUIF, or can be deferred to compile time, as in our implementation in Quick C--. We write an evaluation step as

$$\texttt{stage}(args) \hookrightarrow \text{STAGE}(args'),$$

where the typewriter font `stage` on the left indicates code in the compiler, and the small-caps font STAGE on the right indicates the internal form of the specification. Evaluation might change the arguments *args*—for example, by introducing a counter—but usually the source-language *args* appear unchanged as *args* in the internal representation. Evaluation might produce a sequence of stages instead of a single stage, which does no harm because a sequence of stages may be used anywhere a stage is expected.

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*. The values of these variables are kept in a *store*, written σ; 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 ℓ. The automaton may also update counters, producing a new store σ'. We specify the automaton's behavior formally as a set of inference rules, which are shown in Figure 5. These rules use the judgment $\langle ss,\sigma\rangle@\langle w,k,al\rangle \stackrel{\ell}{\Longrightarrow} \sigma'$, which says that automaton $\langle ss,\sigma\rangle$ responds to request $\langle w,k,al\rangle$ by producing location ℓ and changing its state to σ'. Figure 5 uses notation and auxiliary functions which are summarized in Figures 6 and 7.

In principle, an allocation request could lead to a situation in which no rule in Figure 5 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 that *ss* is *complete*.

In the rest of this section, we explain the evaluation step and the rules in Figure 5, working from the top down.

Overflow An overflow stage satisfies every request by allocating from the overflow block; it never passes a request to its successor. There are two parameters in the specification of an overflow stage: the direction *g* in which the overflow block should grow and the maximum alignment *max_align* of the overflow block supported by the calling convention.

$$\texttt{overflow}(g, max\_align) \hookrightarrow \text{OVERFLOW}(c, g, max\_align)$$

The counter *c* is a private state variable, which counts the number of bytes allocated in the overflow block. As shown in the first two rules at the top of Figure 5, 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.

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` 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 WIDTHS(*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.

$$\texttt{widths}(ws) \hookrightarrow \text{WIDTHS}(ws)$$

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*.

$$\begin{aligned}
\texttt{widen}(f) &\hookrightarrow \text{WIDEN}(f)\\
\texttt{widen}(n[, \texttt{"exact"}]) &\hookrightarrow \text{WIDEN}(\lambda w.n)\\
\texttt{widen}(n, \texttt{"multiple"}) &\hookrightarrow \text{WIDEN}(\lambda w.\text{round_up}(\text{w},\text{n}))
\end{aligned}$$

In a specification, we provide syntactic sugar for two common functions: an integer *n* alone or with the word `"exact"` stands for the function that maps its argument to *n*, and an integer *n* with the word `"multiple"` stands for the function that rounds its argument up to the nearest multiple of *n*.

As Figure 5 shows, the WIDEN(f) stage requests a location ℓ of width $f(w)$ from its successor. The `widen` stage builds a new, narrower location $\ell' = \text{narrow}(\ell, w, k)$, which it returns to its client. As shown in Figure 3, a read from ℓ' is implemented by reading the wide value in ℓ and narrowing the value to *w* bits. A write to ℓ' 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 *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).

$$\texttt{align\_to}(f) \hookrightarrow \text{ALIGN\_TO}(f)$$

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 and BITCOUNTER stages count arguments and bits, respectively.

$$\begin{aligned}
\texttt{argcounter}(c) &\hookrightarrow \text{ARGCOUNTER}(c)\\
\texttt{bitcounter}(c) &\hookrightarrow \text{BITCOUNTER}(c)
\end{aligned}$$

A counter *c* is specified simply by giving its name; our implementation allocates memory for each named counter and initializes each counter to zero.

Each of these stages simply passes each request to its successor, as shown in the middle left of Figure 5. 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 and REGS_BY_BITS. The stage REGS_BY_ARGS(c, rs) uses argument counter c to implement the "n arguments to n registers" policy.

$$\texttt{regs\_by\_args}(c, rs) \hookrightarrow \text{REGS\_BY\_ARGS}(c, rs)$$

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 5. If no registers remain, the request is passed to the next stage. If registers remain, 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 (Appendix A).

The stage REGS_BY_BITS(c, rs) uses bit counter c to implement the "n bits of arguments to n bits of registers" policy.

$$\texttt{regs\_by\_bits}(c, rs) \hookrightarrow \text{REGS\_BY\_BITS}(c, rs)$$

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. As in the "by arguments" case, if no registers remain, the request is passed to the next stage; otherwise the first remaining register is used to satisfy the request, provided the widths match. This behavior is described by two rules at the lower right of Figure 5.

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 rule at the bottom of Figure 5. The rule takes the first available register ℓ, temporarily changes the value of the counter, and recursively requests a location ℓ' to hold the remaining bits. As described in Figure 3, the function combine takes the two narrow locations ℓ and ℓ' and returns a wide location ℓ''.
- 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.

Although it is often necessary to split requests across registers, we have never seen a convention that needed to widen a request implicitly. Therefore the second alternative is not implemented in our system and not shown in Figure 5.

Named counters, although crucial for sharing state among stages, can be inconvenient. For the common case in which `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.

$$\texttt{useregs}\ \{rs\} \hookrightarrow [\text{BITCOUNTER}(c), \text{REGS\_BY\_BITS}(c, [rs])]$$

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 PAD stage, which rounds a bit counter up to respect alignment, as shown in the left column of Figure 5 just below the BITCOUNTER rule. The PAD rule multiplies by `mem_size` in order to convert from alignment units to bits.

$$\texttt{pad}(c) \hookrightarrow \text{PAD}(c)$$

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. We create a choice stage by passing the `choice` function a list in which predicates and stages alternate.

$$\texttt{choice}\ \{p_1, s_1, \ldots, p_n, s_n\} \hookrightarrow \text{CHOICE}([p_1, s_1, \ldots, p_n, s_n])$$

As shown at the upper right of Figure 5, 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. We use lots of syntactic sugar:

- A string s stands for the predicate $\lambda(w, k, \sigma).k = s$, i.e., the predicate that says the kind is equal to s.
- An integer n stands for the predicate $\lambda(w, k, \sigma).w = n$, i.e., the predicate that says the width is equal to n.
- A list of predicates stands for the conjunction of the predicates.
- The value `A.is_any` is the predicate $\lambda(w, k, \sigma).$**true**, i.e., the predicate that is always true.
- When n is an integer, the value `A.counter_is`(c, n) is the predicate $\lambda(w, k, \sigma).\sigma(c) = n$, i.e., the predicate that says counter c has value n.
- When f is a function, the value `A.counter_is`(c, f) is the predicate $\lambda(w, k, \sigma).f(\sigma(c))$, i.e., the predicate that says the value of counter c satisfies function f.

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.

$$\texttt{first\_choice}\ \{p_1, s_1, \ldots, p_n, s_n\} \hookrightarrow \text{FIRST\_CHOICE}(c, [p_1, s_1, \ldots, p_n, s_n])$$

Here c is a fresh counter that is private to the stage. The stage FIRST_CHOICE(c, C) is like CHOICE(C), 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, the FIRST_CHOICE(c, C) stage behaves as the chosen stage from then on. An example appears in Appendix A. The implementation, shown in two rules at the middle right of Figure 5, behaves just like CHOICE(C) when $\sigma(c) = 0$, but afterwards it sets c to the index of the stage chosen. Once $\sigma(c) = i > 0$, the FIRST_CHOICE stage behaves as stage s_i.

3.5 Using a specification

Every calling convention includes specifications for two automata: one for parameters and one for results. Both specifications are used at every procedure definition and at every call site.

Given a specification *ss*, a client creates an automaton by passing *ss* to the `Automaton.at` operation, along with the `start` address and the byte order and `mem_size` of the target machine. `Automaton.at` is part of the implementation of staged allocation; it is implemented directly in Objective Caml (or C++), and when called in a compiler, it returns a mutable abstraction that contains $\langle ss, \sigma \rangle$, where initially $\sigma(c) = 0$ for all c. The abstraction provides two operations: `allocate` and `freeze`.

The `allocate` operation takes a request $\langle w, k, al \rangle$, uses Figure 5 to compute a σ' and ℓ satisfying $\langle ss, \sigma \rangle @ \langle w, k, al \rangle \stackrel{\ell}{\Longrightarrow} \sigma'$, mutates the internal state to replace σ with σ', and returns ℓ. A compiler with a list of parameter types, either from a call site or a procedure's prolog, calls `allocate` once for each type. Each call uses the state produced by the previous call, and the calls accumulate a list of locations: one for each parameter. Once locations have been accumulated, the compiler generates code either to store actual parameters in the locations (for a caller) or to load formal parameters from the locations (for a callee).

The `freeze` operation is called once the compiler has all the locations. The `freeze` operation returns the overflow block and the set of registers used by previous calls to `allocate`. The overflow block becomes part of the stack frame (Lindig and Ramsey 2004), and the set of registers is used in liveness analysis. Freezing is most easily formalized by generalizing the rules in Figure 5 to keep track of what locations are allocated. In the interests of brevity and simplicity, we avoid such a generalization here.

4 Examples, variations, and extensions

The conventions and examples shown above are relatively simple. To show more of the stages described in Section 3, Appendix A (on the Web) presents the standard C conventions for the MIPS R3000 and IA-64 architectures. But 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 $\ell'' = \text{combine}(\ell, \ell')$. 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 $\ell'' = \text{combine}(\ell', \ell)$. 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. Our implementations of REGS_BY_BITS and REGS_BY_ARGS take an optional parameter `"reserve"`, which enables this behavior.

```
PPC.cc["C"].call =
 { A.widen(32, "multiple")
 , A.bitcounter("bits")
 , A.choice {
     "float",  { A.widen(64)
               , A.useregs(f(1)..f(13), "reserve")
               },
      A.is_any, A.regs_by_bits("bits", r(3)..r(10),
                               "reserve")
   }
 , A.overflow { growth = "up", max_alignment = 4 }
 }

PPC.cc["C"].results =
 A.choice
   { "float" , { A.widen(64), A.useregs { f(1) } }
   , A.is_any, { A.widen(32),
                 A.useregs { r(3), r(4) } }
   }
```

Figure 8: The OS X PowerPC convention

The effect is that even when the 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 lower rules in Figure 5, this behavior amounts to adding a new premise $\langle ss, \sigma \rangle @ \langle w, k, al \rangle \stackrel{\hat{\ell}}{\Longrightarrow} \hat{\sigma}$, ignoring $\hat{\ell}$, and threading $\hat{\sigma}$ appropriately.

Using `"reserve"`, Figure 8 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 $\ell = \text{both}(\ell_1, \ell_2)$ such that writing ℓ writes both ℓ_1 and ℓ_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.

5 Results

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

5.1 Implementation experience

We have implemented staged allocation in two different compilers: Quick C--, which is implemented in Objective Caml, 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; a kind at a call site must match the corresponding kind 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 5, 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" so that we can write descriptions in Lua, as shown in this paper. 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 5, 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 5. But with Figure 5 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.

	Staged	CCL	
Machine	Allocators	parms+res	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: Sizes of specifications (* means untested)

5.2 Specification experience

We have written specifications for standard C calling conventions on the Alpha, IA-64, MIPS, Pentium, and Power PC. Three of these specifications appear above; the others appear in Appendix A. The Alpha, IA-64, and Pentium back ends have passed Bailey and Davidson's (1996) 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 impractical: 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.

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 the `counter_is` predicate. But 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. 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. 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 passing of parameters and results; the "total" count shows the total size of each CCL specification, including parts that are not used by the CCL tools. Each count for value passing, whether for staged allocation or for CCL, is reported as a sum: parameters+results. Our specifications are about half the size of CCL's. (The counts for CCL are taken from Appendix B of Bailey's (2000) PhD thesis. The counts for staged allocation are taken from our Lua implementations in Quick C−−; some of these counts are slightly inflated because we have packed specifications into 50 columns for display in this paper. Our C++ code is larger; for example, for the Alpha, the au-

tomata themselves are 8+6 lines, but C++ needs another 15 lines of boilerplate and supporting code to do things like manage memory.)

5.3 Other comparisons with CCL

Before making further comparisons with CCL, we provide a brief summary based on Bailey 2000, §4.3. CCL descriptions are too big to include an example in the body of this paper, but we provide one in Appendix B.

Summary of CCL CCL uses a single abstraction, the "resource," to describe parameters, results, and locations. Each resource has not only a width but also 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. Resources 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.[2]

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 Looking at the big picture, 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 would be required for either staged allocation or CCL. In the case of staged allocation, the rules in Figure 5 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

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: which registers are saved across calls, where they are saved, which way the stack grows, and how the stack frame is laid out. Except for frame layout, these parts are trivial to specify and easy to implement. 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 easier for a compiler writer who has not seen combinator parsing.

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

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 open-code allocation, 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).

Conclusion Staged allocation is simple, readable, precisely defined, and easy to implement. 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, Mike Smith, and anonymous referees made helpful suggestions.

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.

[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.

References

Appel, Andrew W. 1998. *Modern Compiler Implementation*. Cambridge, UK: Cambridge University Press. Available in three editions: C, Java, and ML.

Apple Computer. 2003 (February). *Mach-O Runtime Architecture*.

Bailey, Mark W. 2000 (May). *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 (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.

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 (May). SMLNJ: Garbage collection API. As of November 2004, 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 (July). Higher-order functions for parsing. *Journal of Functional Programming*, 2(3):323–343.

Ierusalimschy, Roberto. 2003 (December). *Programming in Lua*. Lua.Org. ISBN 85-903798-1-7.

Lindig, Christian and Norman Ramsey. 2004 (April). 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 (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.

Ramsey, Norman. 2004. 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 (November). 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 (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.

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`.

A More examples

This appendix presents examples of more calling conventions. The MIPS R3000 convention is one that has historically been hard to get right. The IA-64 convention has an unusual way of using floating-point registers. Finally, we present several simple conventions that can be compared with CCL conventions.

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.

⟨*mips automata*⟩≡
```
Mips.cc["C"].call =
 { A.widen (32, "multiple")
 , A.argcounter("args")
 , A.bitcounter("bits")
 , A.pad("bits")
 , A.first_choice {
     "float" , ⟨first parameter is floating-point⟩ ,
     A.is_any, { }
   }
 , A.regs_by_bits("bits", r(4)..r(7))
 , A.overflow {max_alignment = 16, growth = "up"}
 }
```

In the normal case, 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, 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`.

⟨*first parameter is floating-point*⟩≡
```
A.choice {
  {"float", 32},
      A.regs_by_args("args", {f(12),f(14)}),
  {"float", 64},
      A.regs_by_args("args", {d(12),d(14)}),
  A.is_any,
      { }
}
```

Remarkably, if a procedure takes four 32-bit floating-point parameters, the first two go in `f12` and `f14`, the next two go in `r6` and `r7`, and `f13` and `f15` are not used.

More examples of MIPS parameter placements appear in Figure 9. These examples show placements for 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 corresponding location in which each parameter is passed. The notation *k*`(sp)` indicates a location on the stack, in the overflow block.

IA-64 Figure 10 shows a specification for the IA-64. 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 an `A.counter_is` predicate.

When describing the convention for results, we define a new predicate, `fpreturn`. This predicate is necessary because only some floating-point results are returned in floating-point registers; a 128-bit, quad-precision floating-point result is returned in the integer register pair `r8`–`r9`.

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 9: Parameter placement on MIPS R3000

```
function lt512(n) return n < 512 end

Ia64.cc["C"].call =
  { A.widen(64, "multiple")
  , A.bitcounter("bits")
  , A.choice {
       {"float", A.counter_is("bits", lt512)},
            A.useregs(f(8)..f(15)),
       A.is_any,
            A.regs_by_bits("bits",out(0)..out(7))
    }
  , A.overflow { growth = "up", max_alignment = 16 }
  }

function fpreturn(_, k, w)
  return k == "float" and w <= 82
end

Ia64.cc["C"].results =
  A.choice { fpreturn, { A.widths { 32, 64, 82 }
                       , A.useregs { f(8) }
                       }
           , A.is_any, { A.widen(64, "multiple")
                       , A.widths { 64, 128 }
                       , A.useregs { r(8), r(9) }
                       }
           }
```

Figure 10: The standard IA-64 convention

Other conventions For illustrative purposes, Figure 11 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. The comments in Figure 11 record some constraints on stack-frame layout.

```
Vax.cc["C"].call =
  A.overflow { max_alignment = 4, growth = "up" }

Vax.cc["C"].results = { A.widen(32, "multiple")
                      , A.useregs { r(0), r(1) }
                      }


M68020.cc["C"].call =
  A.overflow { max_alignment = 8, growth = "up" }

M68020.cc["C"].results = { A.widen(32, "multiple")
                         , A.useregs { d(0), d(1) }
                         }

-- if the overflow block is nonempty, frame
-- layout must add an extra 32 bytes

M88100.cc["C"].call =
  { A.widen(32)
  , A.useregs(r(2)..r(9))
  , A.overflow { max_alignment = 8, growth = "up" }
  }

M88100.cc["C"].results = { A.widen(32, "multiple")
                         , A.useregs { r(2), r(3) }
                         }

-- frame layout must reserve stack space
-- for 6 words of registers

Sparc.cc["C"].call =
  { A.widen(32, "multiple")
  , A.useregs(r(8)..r(13))
  , A.overflow { max_alignment = 8, growth = "up" }
  }

Sparc.cc["C"].results =
  { A.widen(32, "multiple")
  , A.choice { "float" , A.useregs(f(0)..f(1))
             , A.is_any, A.useregs{r(8)}
             }
  }
```

Figure 11: Untested specifications for which we have no corresponding back ends

B Details of CCL and MIPS

This appendix reproduces a detailed example of CCL from Bailey (2000). To facilitate comparisons based on a concrete problem, it also explores some details of the calling convention for the MIPS R3000.

Figure 14 (following page) shows the most recent CCL specification for the MIPS R3000, from Bailey (2000). Typography is significant in CCL, and this example does not duplicate Bailey's typography exactly, but it should give the flavor of a CCL description. Of special interest are the exclusion rules, which set the "unavailable" bits on integer registers `r4`–`r7` and floating-point registers `f12` and `f14`. These exclusion rules are not consistent with the requirements in the MIPS assembly-language programmer's guide or with the `lcc` or `gcc` compilers. It is not clear whether the rules are inconsistent because there is a bug in the CCL description or because the convention has been revised since the description was written.

```
Mips.cc["C"].call =
 { A.widen (32, "multiple")
 , A.argcounter("args")
 , A.bitcounter("bits")
 , A.pad("bits")
 , A.first_choice {
    "float" ,
       A.choice {
         {"float", 32},
             A.regs_by_args("args", {f(12),f(14)}),
         {"float", 64},
             A.regs_by_args("args", {d(12),d(14)}),
         A.is_any,
             { }
       },
     A.is_any, { }
   }
 , A.regs_by_bits("bits", r(4)..r(7))
 , A.overflow {max_alignment = 16, growth = "up"}
}

Mips.cc["C"].results =
  { A.widen  (32, "multiple")
  , A.widths { 32, 64 }
  , A.choice { "float" , A.useregs(f(0)..f(3))
             , A.is_any, A.useregs(r(2)..r(3))
             }
  }
```

Figure 12: MIPS R3000 convention using staged allocation

Figure 12 shows a specification for the MIPS R3000 using staged allocation, part of which is explained above in Appendix A. This specification does not have the same semantics as the CCL specification—in particular, it is consistent with current compilers for the MIPS. Also, when passing small structures, the CCL specification appears to do something special, but we haven't figured out what that is, so we use the same rules for passing structures as for any other non-floating-point data.

Figure 13 illustrates some of the complexity of exclusion rules. It shows the exclusion relation for today's MIPS R3000 convention as implemented by `lcc` (Fraser and Hanson 1995), version 4.2. We computed this exclusion relation by running `lcc` on all C procedures that pass `int`, `float`, or `double` parameters in registers. We then parsed the assembly language and identified sets of registers that are never used in the same call. These sets appear at the bottom of Figure 13. The top of Figure 13 shows the same information expressed as rules. There are more than three times as many rules as in Figure 14; this example shows the limitations of specifying conventions by means of such rules.

```
external SPILL_SIZE, LOCALS_SIZE
persistent {r^1, r^16:23, r^26:31}
alias REG_ARGS ≡ 16
alias sp ≡ r^29
caller prologue
    view change
        ∀offset ∈ {−∞ : ∞}
            M[sp + offset] becomes M[sp + offset + ⌈ARG_BLOCK_SIZE⌉^8]
    end view change
    data transfer (asymmetric)
        alias rindex ≡ 4 : 7
        alias fpindex ≡ 12, 14
        alias mstart ≡ sp + REG_ARGS
        alias mindex ≡ mstart : ∞
        resources {<r^rindex, M^mindex>, <f^fpindex, M^mindex>, <M^mindex>}
        ∀mem ∈ {M[sp(REG_ARGS)]} set mem.assigned ← true
        internal ARG_BLOCK_SIZE ← ∑(<M[addr].size | addr ∈ <mindex> ∧ M[addr].assigned>)
        class intregs ← <<r^x> | x ∈ <rindex>>
        class intfpregs ← <<r^x> | x ∈ <rindex> ∧ x mod 2 = 0>
        class fpfpregs ← <<f^x> | x ∈ <fpindex>>
        class mem ← <<M^loc> | loc ∈ <mindex> ∧ loc mod 4 = 0>
        class aligned_mem ← <<M^loc> | loc ∈ <mindex> ∧ loc mod 8 = 0>
        class struct_mem ← <<r^x, M^mstart> | x ∈ <rindex>>
        class aligned_struct_mem ← <<r^x, M^mstart> | x ∈ <rindex> ∧ x mod 2 = 0>
        ∃reg ∈ {reg | reg ∈ {f^12} ∧ reg.assigned} ⇒ set r^4.unavailable ← true
        ∃reg ∈ {reg | reg ∈ {f^12} ∧ reg.assigned} ⇒ set r^5.unavailable ← true
        ∃reg ∈ {reg | reg ∈ {f^14} ∧ reg.assigned} ⇒ set r^6.unavailable ← true
        ∃reg ∈ {reg | reg ∈ {f^14} ∧ reg.assigned} ⇒ set r^7.unavailable ← true
        ∃reg ∈ {reg | reg ∈ {r^4:5} ∧ reg.assigned} ⇒ set f^12.unavailable ← true
        ∃reg ∈ {reg | reg ∈ {r^6:7} ∧ reg.assigned} ⇒ set f^14.unavailable ← true
        ∀argument ∈ <ARG^1:ARG_TOTAL>
            map argument → argument.type ⊥ {
                byte, word, longword : <intregs, mem>,
                struct : argument.size ⊥ {
                    1, 2, 3, 4, 5, 6, 7 :   <struct_mem, mem>,
                    default:                <aligned_struct_mem, aligned_mem>,
                },
                float, double : ARG^1.type ⊥ {
                    struct, byte, word, longword : <intfpregs, aligned_mem>,
                    float, double :         <fpfpregs, aligned_mem>
                }
            }
    end data transfer
end caller prologue
callee prologue
    view change
        ∀offset ∈ {−∞ : ∞}
            M[sp + offset] becomes M[sp + offset + ⌈SPILL_SIZE + LOCALS_SIZE + NVSIZE⌉^8]
    end view change
end callee prologue
callee epilogue
    data transfer (asymmetric)
        resources {<r^2>, <f^0>}
        ∃return ∈ <RVAL^1:RVAL_TOTAL> ⇒
            map return → return.type ⊥ {
                byte, word, longword :  <<<r^2>>>,
                float, double :         <<<f^0>>>,
                struct :                ↑ (<<<r^2>>>)
            }
    end data transfer
end callee epilog
```

Figure 14: CCL description of MIPS R3000 calling convention from Bailey (2000)

if $\mathbf{r}^4$.assigned $\Rightarrow$ **set** $\mathbf{f}^{12}$.unavailable $\leftarrow$ true
if $\mathbf{r}^4$.assigned $\Rightarrow$ **set** $\mathbf{f}^{13}$.unavailable $\leftarrow$ true
if $\mathbf{r}^4$.assigned $\Rightarrow$ **set** $\mathbf{f}^{14}$.unavailable $\leftarrow$ true
if $\mathbf{r}^4$.assigned $\Rightarrow$ **set** $\mathbf{f}^{15}$.unavailable $\leftarrow$ true
if $\mathbf{r}^5$.assigned $\Rightarrow$ **set** $\mathbf{f}^{13}$.unavailable $\leftarrow$ true
if $\mathbf{r}^5$.assigned $\Rightarrow$ **set** $\mathbf{f}^{14}$.unavailable $\leftarrow$ true
if $\mathbf{r}^5$.assigned $\Rightarrow$ **set** $\mathbf{f}^{15}$.unavailable $\leftarrow$ true
if $\mathbf{r}^6$.assigned $\Rightarrow$ **set** $\mathbf{f}^{15}$.unavailable $\leftarrow$ true
if $\mathbf{r}^6$.assigned $\wedge$ $\mathbf{f}^{13}$.assigned $\Rightarrow$ **set** $\mathbf{f}^{14}$.unavailable $\leftarrow$ true
if $\mathbf{r}^6$.assigned $\wedge$ $\mathbf{f}^{14}$.assigned $\Rightarrow$ **set** $\mathbf{f}^{13}$.unavailable $\leftarrow$ true
if $\mathbf{r}^7$.assigned $\Rightarrow$ **set** $\mathbf{f}^{15}$.unavailable $\leftarrow$ true
if $\mathbf{f}^{12}$.assigned $\Rightarrow$ **set** $\mathbf{r}^4$.unavailable $\leftarrow$ true
if $\mathbf{f}^{13}$.assigned $\Rightarrow$ **set** $\mathbf{r}^4$.unavailable $\leftarrow$ true
if $\mathbf{f}^{13}$.assigned $\Rightarrow$ **set** $\mathbf{r}^5$.unavailable $\leftarrow$ true
if $\mathbf{f}^{13}$.assigned $\wedge$ $\mathbf{f}^{14}$.assigned $\Rightarrow$ **set** $\mathbf{r}^6$.unavailable $\leftarrow$ true
if $\mathbf{f}^{14}$.assigned $\Rightarrow$ **set** $\mathbf{r}^4$.unavailable $\leftarrow$ true
if $\mathbf{f}^{14}$.assigned $\Rightarrow$ **set** $\mathbf{r}^5$.unavailable $\leftarrow$ true
if $\mathbf{f}^{15}$.assigned $\Rightarrow$ **set** $\mathbf{r}^4$.unavailable $\leftarrow$ true
if $\mathbf{f}^{15}$.assigned $\Rightarrow$ **set** $\mathbf{r}^5$.unavailable $\leftarrow$ true
if $\mathbf{f}^{15}$.assigned $\Rightarrow$ **set** $\mathbf{r}^6$.unavailable $\leftarrow$ true
if $\mathbf{f}^{15}$.assigned $\Rightarrow$ **set** $\mathbf{r}^7$.unavailable $\leftarrow$ true

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

Figure 13: Exclusion rules and sets for MIPS R3000 convention