Reprinted from *Proceedings of the ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, January 1995.

A Formal Model and Specification Language for Procedure Calling Conventions[1]

Mark W. Bailey
mark@virginia.edu

Jack W. Davidson
jwd@virginia.edu

Department of Computer Science
University of Virginia
Charlottesville, VA 22903, U.S.A.

Abstract

Procedure calling conventions are used to provide uniform procedure-call interfaces. Applications, such as compilers and debuggers, which generate, or process procedures at the machine-language abstraction level require knowledge of the calling convention. In this paper, we develop a formal model for procedure calling conventions called P-FSA's. Using this model, we are able to ensure several completeness and consistency properties of calling conventions. Currently, applications that manipulate procedures implement conventions in an *ad-hoc* manner. The resulting code is complicated with details, difficult to maintain, and often riddled with errors. To alleviate the situation, we introduce a calling convention specification language, called CCL. The combination of CCL and P-FSA's facilitates the accurate specification of conventions that can be shown to be both consistent and complete.

1 Introduction

Procedures, or functions, in programming languages work in concert to implement the intended function of programs. To facilitate this cooperation between procedures, we must accurately specify the procedure-call interface. This interface must define how to pass actual parameters and describe function return values, and which *machine resources*, such as registers, the called procedure must preserve. This understanding between the *caller*[2] and *callee*[3] is known as the *procedure calling convention*. Because of the machine-specific nature of the calling convention, calling conventions vary widely from machine-to-machine, programming-language-to-programming-language, and, language-implementation-to-language-implementation.

1.1 Why a Calling Convention Specification?

Currently, information about a particular calling convention can be found by: looking in the programmer's reference manual for the given machine, or reverse-engineering the code generated by the compiler. Reverse-engineering the compiler has many obvious shortcomings. Using the programmer's reference manual may be equally problematical. As with much of the information in the programmer's manual, the description is likely to be written in English and is liable to be ambiguous, or inaccurate. For example, in the MIPS programmer's manual [KANE92] the English description is so difficult to understand that the authors provide fifteen examples, several of which are contradictory[FRAS93]—and this is the *second* edition. Furthermore, the convention, once understood, is difficult to implement. For example, the GNU ANSI **C** compiler fails on an example listed in the manual. Digital, in recognizing the problem, has published a calling standard document for their new Alpha series processors [DEC93] that exceeds 100 pages[4]. Thus, it should be clear that there is a need for an accurate, concise description of procedure calling conventions.

1.2 Applications

Any application that must process or generate procedures at the machine-language abstraction level is likely to need to know about a procedure calling convention. Examples of such uses include compilers, debuggers, evaluation tools such as profilers, and documentation. The code that implements the calling convention in these applications lends itself to automatic generation. In many cases, the convention itself is not difficult to understand, or implement for a given instance of a procedure. However, the implementation of the general case is complicated with details that are difficult to implement correctly for all cases.

Compilers, perhaps would benefit most from specification of the calling convention. The calling convention is exhibited in the calling sequence the compiler uses when generating code. A *calling sequence* is a sequence of instructions that implements the calling convention. Thus, a calling sequence is an instantiation of the more general calling convention. Frequently, a compiler will use a calling convention that differs from the one used by the native compiler for the machine. In such cases, it is desirable to be able to call procedures that were generated using the native compiler. System library functions, which would be compiled using the native compiler, are such an example. It therefore would be convenient for a compiler to cope with more than one calling convention. In many compilers, the portion of code that implements the calling convention is lengthy, detailed, and therefore difficult to modify or parameterize by the calling convention.

The existence of a method for accurately specifying calling conventions also makes it possible to experiment with different conventions. Johnson and Richie have identified the issues in pro-

1. This work was supported in part by National Science Foundation grant CCR-9214904.
2. The calling procedure is known as the *caller.*
3. The called procedure is known as the *callee*.

4. Although this document also includes information on exception handling and information pertinent to multithreaded execution environments, more than 42 pages are devoted to documenting the calling convention.

POPL'95 1/95 San Francisco CA USA

viding an efficient calling sequence after one has already defined a calling convention [JOHNSON]. However, the convention makes many choices that directly affect the efficiency of calling procedures. We therefore feel that it is important to experiment with different conventions on each to tune the convention to the machine. Davidson and Whalley have performed a limited experiment in investigating different calling conventions [DAVI91]. However, due to the enormous amount of work required to change their compiler from one calling convention to another, their experiment was limited to several different methods of saving and restoring registers.

1.3 Contributions

This paper makes several contributions. It provides a formal model for procedure calling conventions that can be used in a variety of system software. The paper presents a specification language that, when used in conjunction with the formalism, can provide accurate convention information to an application. Further, it shows that by modeling a convention in this manner, several desirable properties about calling conventions can be established. It also shows how conventions that are not complete, or are inconsistent can be automatically identified. Finally, the paper shows how this formalism can be used by an optimizing compiler to automatically generate procedure calling sequences.

2 The Language Concepts

This section describes the underlying model for the convention descriptions. Many features of the description language have their foundation in the underlying model.

2.1 Convention vs. Sequence

When one first tries to model the procedure call interface, one would likely consider—as we did—simply modeling the calling sequence. This is natural since compiler writers are most familiar with calling sequences. Traditionally, the terms calling sequence and calling convention have been used interchangeably in the literature to refer to the calling sequence. However, after some thought, the subtle differences between the convention and the sequence become apparent.

The calling convention defines how two procedures, on either side of a procedure call interface, interact. It is an agreement between the caller and the callee about where information is found and how to manage machine resources. Choosing which registers retain their values across a procedure call, or the order and location of procedure arguments, or where the return address is found, are all decisions that one makes when defining a procedure calling convention. One can think of the calling convention as a definition of what is done by *whom*.

The calling sequence, on the other hand, is an implementation of the calling convention. There may be many calling sequences for given calling convention. In particular, since the calling sequence implements the calling convention, it is impossible for the caller to determine if the callee is using the same sequence, and vice versa. Thus, while it is imperative that a caller and a callee use the same calling convention, it is not necessary that they use the same calling sequence.

2.2 Interfaces and Agents

So far, we have referred to the procedure call interface. In fact, there are two interfaces: the procedure call interface and the procedure return interface. On each side of these interfaces, there is an agent. An *agent* ensures that *that* side of the interface satisfies the requirements of the calling convention. These agents are the *whom* in the definition of the calling convention. For the procedure call interface, there are the *caller prologue* and *callee prologue* agents that are responsible for correctly passing the procedure arguments and constructing an environment that the callee can execute in. For the procedure return interface, there are the *callee epilogue* and *caller epilogue* that are responsible for correctly passing the procedure return values and restoring the environment of the caller. The responsibilities of each of the four agents are closely related. The caller prologue and callee prologue agents must agree on how to pass information, as do the caller epilogue and callee epilogue. Additionally, actions of the epilogue agents must be symmetric to the actions of the prologue agents to properly restore the environment (*e.g.*, if the call decrements the stack pointer, the return must increment it). It is precisely these restrictions that make it difficult correctly construct a calling sequence.

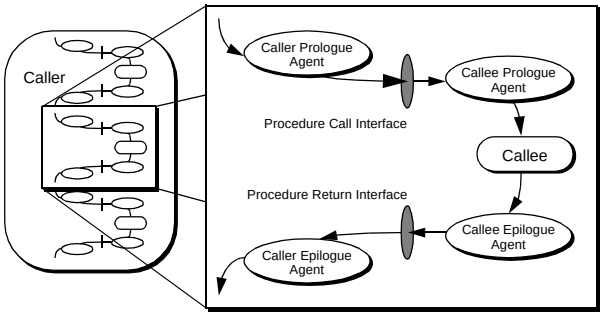

Figure 1: The Role of Agents in Procedure Call and Return Interfaces.

2.3 Defining the Interface

The procedure call interfaces are defined in terms of two concepts: data placement and view change. These abstractions are all that one needs to accurately specify procedure calling conventions.

2.3.1 Data Placement

Data placement specifies where information should be placed/found as well as who is to place it there. This mechanism is used primarily for defining where information is to be placed to pass across an interface (procedure arguments and return values) and where to save information to restore later (contents of registers). In the former, this is an agreement between two agents on opposite sides of an interface. For the latter, this is an agreement between agents in the caller or agents in the callee.

Abstractly, data placement definitions are functions that map values onto machine resources. The functions take a value and corresponding attributes (such as data type) and decide where the value belongs. More precisely, placement definitions are finite state machines, since the mapping is order-dependent. Figure 2 illustrates an application of a placement definition to place procedure arguments. In this example, floating-point values are placed in even/odd register pairs, structures are placed on the stack, and integers are placed in the next available register. When argument registers are exhausted, the stack is used. The placement is complicated by restrictions. An example restriction is registers are that are passed over (*i.e.*, an odd numbered register when placing a floating-point value) cannot be subsequently used. Such restrictions are common in real calling conventions, and must therefore be captured in the data placement definition.

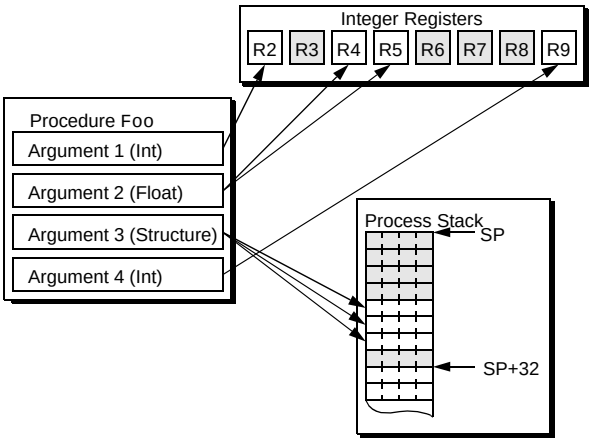

Figure 2: Mapping from arguments to machine resources.

2.3.2 View Change

View change indicates something has happened that caused locations to *appear* to move. The register window mechanism on the SPARC microprocessor is an example. When the register window slides, the contents of the registers appear to move because the names of the registers have changed. We wish to indicate this change without causing the move to actually occur. The change of view indicates how the names of locations have changed. View change is used more commonly when describing that a frame must be pushed on the stack. When a push occurs, all locations referenced by the stack pointer appear to shift.

3 The Language

In this section, we present CCL (Calling Convention Language), the language that we use to capture the concepts described in the previous section.

3.1 A Simple Calling Convention

The calling convention is the set of rules to which the caller and callee must conform. Figure 3 contains the calling convention rules for a hypothetical machine. Consider the following ANSI **C** prototype for a function `foo`:

```
int foo(char p1, int p2, int p3, double p4);
```

For the purpose of transmitting procedure arguments for our simple convention, we are only interested in the *signature* of the procedure. We define a procedure's signature to be the procedure's name, the order and types of its arguments, and its return type. This is analogous to ANSI **C**'s abstract declarator, which for the above function prototype would be:

```
int foo(char, int, int, double);
```

which defines a function that takes four arguments (a `char`, two `int`'s, and a `double`), and returns an `int`.

With `foo`'s signature, we can apply the calling convention in Figure 3 to determine how to call `foo`. `foo`'s arguments would be placed in the following locations:

- `p1` in register $\mathbf{a}^1$
- `p2` in register $\mathbf{a}^2$
- `p3` in register $\mathbf{a}^3$
- `p4` on the stack in **M**[sp:sp + 7] (**M** denotes memory)

1. Registers $\mathbf{a}^1$, $\mathbf{a}^2$, $\mathbf{a}^3$, and $\mathbf{a}^4$ are 32-bit argument-transmitting registers.
2. Arguments may be passed on the stack in increasing memory locations starting at the stack pointer (**M**[sp]).
3. An argument may have type `char` (1 byte), `int` (4 bytes), or `double` (8 bytes).
4. An argument is passed in registers (if enough are available to hold the entire argument), and then on the stack.
5. Arguments of type `int` are 4-byte aligned on the stack.
6. Arguments of type `double` are 8-byte aligned on the stack.
7. Stack elements that are skipped over cannot be allocated later.
8. Return values are passed in registers $\mathbf{a}^1$ and $\mathbf{a}^2$.
9. Values of registers $\mathbf{a}^6$, $\mathbf{a}^7$, $\mathbf{a}^8$, and $\mathbf{a}^9$ must be preserved across a procedure call.

Figure 3: Rules for a simple calling convention.

Notice that although register $\mathbf{a}^4$ is available, `p4` is placed on the stack since it cannot be placed completely in the remaining register (rule 4). Such restrictions are common in actual calling conventions.

Now that we have seen how arguments are transmitted for a simple example, we can describe the objects in our model. The primary objects of interest are machine resources. A machine resource is simply any location that can store a value. Examples include registers and memory locations, such as the stack. Defining where required values are located is accomplished by specifying a mapping from one resource to another. We call such a mapping a *placement*. Although a procedure's arguments and its return value are technically not machine resources by the above definition, we consider them as special resources in our model.

We partition a machine's resources into two categories: finite and infinite. Resources such as register sets that can easily be enumerated are considered finite. Resources that are conceptually "unbounded" such as the stack are considered infinite. Although the stack is finite for any particular implementation of a machine, we model it as infinite since the programmer considers it, for all intents and purposes, to be infinite. This distinction is important since we must treat infinite resources in a special way.

3.2 Typographical Extensions

Figure 4 contains the complete CCL specification for the simple calling convention. The first thing to notice about CCL descriptions is prevalent use of typographical extensions. We extend the standard ASCII character set used in most machine-readable languages to include multiple fonts, super/subscript's, and variations in font angle (italic) and weight (bold). This approach helps accomplish two of our goals in the language design: conciseness and naturalness. Since information can be encoded in the fonts, we can reduce the size of the descriptions. Second, in contrast to simple ASCII text, it provides a more natural way to describe many data types used in CCL. The following is a list of many expressions used in CCL:

- Sets: {2:9} ≡ {2,3,4,5,6,7,8,9}
- Ordered sets: <2,8,3,9,4,10>, <0:∞>
- Labeled sets: {char: 1, short: 2, longword: 4, float: 4, double: 8}
- Arrays: $\mathbf{M}[14] \equiv \mathbf{M}^{14}$, $<\mathbf{M}[\mathbf{r}^{14}:\mathbf{r}^{14}+31]> \equiv <\mathbf{M}[\mathbf{r}^{14}(32)]>$
- Operators: mod, $\sum$, $\wedge$, $\in$, $\perp$
- Keywords: **external**, **alias**, **call prologue**, **resources**, **map**, **set**
- Comments: *This is a comment*

```
1   external NVSIZE, SPILL_SIZE, LOCALS_SIZE
2   non-volatile {a^6, a^7, a^8, a^9}
3   alias sp ≡ a^5
4   caller prologue
5       view change
6           ∀ offset ∈ {−∞:∞}
7               M[sp + offset] becomes M[sp + offset + ARG_SIZE]
8       end view change
9       data transfer (asymmetric)
10          alias mindex ≡ <sp:∞>
11          alias argregs ≡ <a^1:4>
12          resources {argregs, <M^mindex>}
13          internal ARG_SIZE ← ∑(<M[addr].size | addr ∈ mindex ∧ M[addr].assigned>)
14          class regs ← <<register> | register ∈ argregs>
15          class imem ← <<M[addr]> | addr ∈ mindex ∧ addr mod 4 = 0>
16          class dmem ← <<M[addr]> | addr ∈ mindex ∧ addr mod 8 = 0 >
17          ∀ argument ∈ <ARG^1:ARG_TOTAL>
18              map argument → argument.type ⊥ {
19                  char:       <regs, M^mindex>,
20                  int:        <regs, imem>,
21                  double:     <regs, dmem>,
22              }
23      end data transfer
24  end caller prologue
25  callee prologue
26      view change
27          ∀ offset ∈ {-∞:∞}
28              M[sp + offset] becomes M[sp + offset + SPILL_SIZE + LOCALS_SIZE + NVSIZE]
29      end view change
30  end callee prologue
31  callee epilogue
32      data transfer (asymmetric)
33          resources {a^1:2}
34          map RVAL^1 → <<<a^1>>>
35      end data transfer
36  end callee epilogue
37  caller epilogue
38  end caller epilogue
```

Figure 4: A Complete Simple Example.

An advantage of using typographical extensions is that a simple, concise convention indicates the portions of descriptions that are literals, meta-symbols, and predefined elements. Comments are clearly offset from the remaining description because they are both italic and set in a different font. Sets are used heavily in the language and adhere to their natural syntax in mathematics. Keywords are in bold making them easy to identify.

There are two minor disadvantages of typographical extensions. One is that descriptions cannot be edited with existing text editors (*e.g.*, emacs, vi, etc.), rather it requires the use of tools such as a specialized editor and postscript viewer. However, such tools are widely available as are postscript printers for printing descriptions. Indeed, such a tool was used to develop the CCL descriptions in this paper. A second disadvantage is the tools that process CCL are slightly more complicated as they must deal with an intermediate representation that has typographical information included. Our initial experiments show that this is not be a major obstacle. Consequently, the benefits of this approach far outweigh the minor disadvantages.

3.3 Outer Environment

CCL is a part of a larger description system we are developing at the University of Virginia. CCL is part of the compiler-specific description. Although CCL is used to capture all information about a calling convention, a CCL description does not contain all necessary information to produce a calling sequence. Indeed, CCL descriptions are not complete by themselves. CCL descriptions require information from the outer environment to complete the descriptions. Information about the machine and language, such as the size of registers, the base data types and local procedure information, such as the amount of space needed for temporary variables, and which registers are used, must be provided by the outer environment. Four variables that are always defined by the outer environment are the special resources **ARG**, **RVAL**, and the corresponding special resource sizes ARG_TOTAL and RVAL_TOTAL. Since these values are always defined, they are implicitly declared as external values. All other variables whose values are provided by the outer environment are declared using the **external** statement.

A CCL description is typically language dependent as well. This is, in part, because the language definition influences the calling convention. For example, the **C** language [KERN78] defines a slightly different calling convention than its successor ANSI **C**

[KERN88]. One difference is that **C** always promotes arguments of type float to type double, while ANSI **C** does not. These differences are part of the calling convention, and are, therefore, present in the resulting CCL descriptions. Although ANSI **C** is now the standard, all of the examples in this paper assume the traditional **C** language calling convention since it presents more interesting examples.

3.4 Placement of Procedure Arguments

First, we examine the placement of procedure arguments. We use the simple calling convention specification shown in Figure 4. For placement of arguments, we focus on the **data transfer** statement within the caller prologue section of the description (lines 9-23). We use the **alias** statement to introduce the name 'argregs' as a name for the parameter passing registers and 'mindex' as a set of stack addresses ($\mathbf{a}^5$ is the stack pointer). Line 12 defines the set of possible destinations for data placement, which we call the resources. Lines 14-16 specify classes that each defines a subset of these resources where placements may start. Since the convention has two different alignment restrictions for memory, which are based on argument type, there is a corresponding class for each restriction as well as a class for the argument registers. The language requires classes to be ordered sets of ordered sets. Classes simply partition the resources into sets of valid locations to place values. The outer set indicates the order in which to consider placing the arguments. In this example, when passing arguments in memory, we consider memory locations in low-to-high address order. The inner set typically contains a single element (the starting location). More complicated conventions make more use of the inner set as we will see later.

The remaining lines (17-22) of the data transfer contain the argument placement description. The universal quantifier ($\forall$) operator iterates over the set, each time binding the variable *argument* to an element of the set. Here, the set is ordered, ensuring that *argument* will take values in the set in order. The resource **ARG** is a special resource that is provided by the outer environment. It contains information such as the type and size of the arguments for the call.

The two operators on line 18 complete the placement description. The placement operator ($\rightarrow$) is invoked for each value *argument* is assigned. The placement operator takes a value (here an argument) and a list of classes. The classes are searched, in order, for an available resource to place the given value. When a resource is found, the location is marked as used, by setting the 'assigned' attribute, to ensure unique locations for each placed value. The selection operator ($\perp$) is used on labeled sets. This is simply a case expression. Based on the value of *argument*'s type attribute, one expression from the labeled set is selected.

3.4.1 Placement of Procedure Return Values

Specifying the locations of procedure return values is similar to procedure arguments. To determine the return value placement, we examine the **data transfer** statement within the callee epilogue section for Figure 4. Here, we see **RVAL**, the other special resource defined by the outer environment, which refers to the list of return values (in most languages, there is only one). The resources used for returning values are the registers $\mathbf{a}^1$ and $\mathbf{a}^2$. These registers are used for returning values of all types. Recall that the registers have size 4 bytes, integers are 4-byte quantities and doubles are 8-byte quantities. So, this specification indicates only $\mathbf{a}^1$ will be used for chars and ints, but $\mathbf{a}^1$ and $\mathbf{a}^2$ will be used for double values. This level of conciseness is achieved by indicating only the starting location rather than indicating the size, which can be attained from the type.

3.4.2 Non-Volatile Registers

Non-volatile registers are registers that contain values that the caller expects to be preserved during a procedure call. This expectation is part of the calling convention. If the callee wishes to use a non-volatile register, the register's value must be preserved by the callee, and restored to its original value prior to returning to the caller. Registers whose values are non-volatile are listed in line 2 of Figure 4.

Two important details about non-volatile registers are missing from this specification. These are where the registers values are saved, and how they saved. The former is defined by the frame layout, while the latter is defined by the calling sequence. Although these details are important for the callee's implementation, they are of no concern of the caller. Since they are of no concern of the caller, they are not part of the calling convention. Thus, while we could easily include this information in our CCL descriptions, we have chosen not to include it to avoid unnecessary restrictions in our calling convention specifications.

3.4.3 Putting it All Together

So far, we have examined the specification of each aspect of our simple convention in isolation. We now broaden our view to the entire description shown in Figure 4. A description is divided into five sections: one section for each agent in our model, and a global declaration section. We place **data transfer** and **view change** statements within agent sections. Finally, we place the two data placement schemes discussed above in their corresponding locations in the description.

First, let's examine the **caller prologue** section. This section specifies the responsibilities of the caller prologue agent. Most **data transfer** statements have been described previously. Line 13, however, has not. It computes the amount of space that was assigned by the placement operator. Although this computation has been placed before the placement operation, its value will not actually be computed until after, since the computation is dependent on the results of the placement. The result of this computation is then used in the above **view change** statement.[1] The **view change** indicates that the value in location **M**[sp] will now be found in location **M**[sp + ARG_SIZE]. Such a change of view corresponds to a decrement of the stack pointer (a push) of precisely the amount needed to pass the arguments.

Although the location of the procedure arguments must be known for both the caller prologue and the callee prologue, the placement description only resides in the caller prologue section. This is because the callee prologue can determine the locations of the arguments by applying the appropriate view change to the description located in the caller prologue section. Hence, describing the change of view makes it unnecessary to restate where the procedure arguments are located when the view changes.

A final note about this description. The two **data transfer** statements (for passing arguments and return values) are tagged with the keyword **(asymmetric)**. This indicates that the transfer is done by the agent, but not undone (values transferred back) by the symmetric agent (callee epilogue for callee prologue, caller epilogue for caller prologue). However, for all of the view changes, the lack of the **(asymmetric)** keyword indicates that a symmetric action takes place in the symmetric agent. For example, in this specification, the caller epilogue is empty. However, the caller epilogue performs the symmetric view change shown in the callee

1. There is no set ordering for **view change** and **data transfer** statements. However, since the **view change** occurs before the **data transfer**, all references to resource **M** are in terms of the new view. Had the **view change** been after the **data transfer**, this would not be the case.

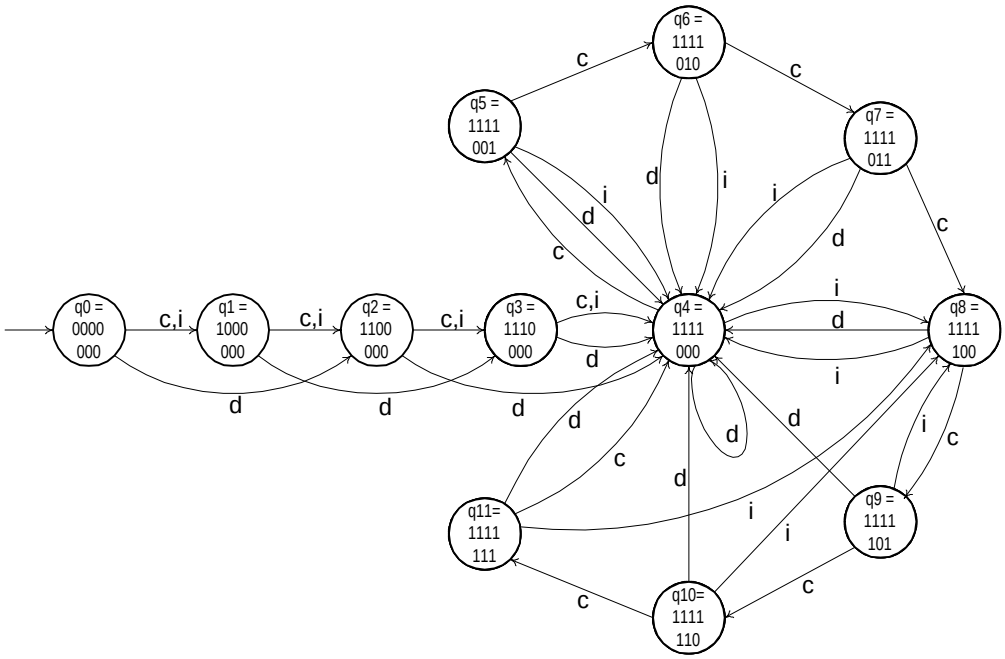

Figure 5: P-FSA for transmission of parameters for a simple calling convention.

prologue and has access to the procedure return value **data transfer** statement in the callee epilogue. Without this concept of symmetry, the description in Figure 4 would be considerably more involved.

Our simple calling convention illustrates how many common features in calling conventions are described. However, real-world examples tend to have additional constraints that complicate the descriptions. Appendix A contains a complete specification for the MIPS R3000 calling convention and a brief English description.

4 The Formal Model

This section presents the formal model that we use as a foundation for implementing procedure calling conventions.

4.1 P-FSA Representation

We use finite state automata to model each placement in the calling convention. One such FSA is shown in Figure 5. This FSA models the placement of procedure arguments for the simple calling convention. The placement FSA (P-FSA) takes a procedure's signature as input and produces locations for the procedure's arguments as output. The automaton works by moving from state to state as the location of each argument is determined. During transition, information about the current parameter is read from the input, and the resulting placement is written to the output.

The states of the machine represent that state of allocation for the machine resources. For example, the state labeled q_2 represents the fact that register $\mathbf{a}^1$ and $\mathbf{a}^2$ have been allocated, but that $\mathbf{a}^3$, $\mathbf{a}^4$ and stack locations have not been allocated. The transitions between states represent the placement of a single argument. Since arguments of different types and sizes impose different demands on the machine's resources, we may find more than one transition leaving a particular state. In our example, q_8 has three transitions even though two of them (`int` and `double`) have the same target state (q_4). This duplication is required since the output from mapping an `int` is different from the output from mapping a `double`.

Modeling the allocation of an infinite resource, such as the stack, using an FSA poses a problem, however. As stated above, the state indicates which resources have been allocated. For finite resources, this is easily accomplished by maintaining a bit vector. When a resource no longer may be used, the associated bit is set to indicate this. For an infinite resource this scheme cannot work if we hope to use an FSA, since this would require a bit vector of infinite length. To simplify the problem, we impose a restriction on infinite resources: their allocation must be contiguous. Thus, for an infinite resource $I = \{i_1, i_2, \ldots\}$, we can store the allocation state by maintaining an index p whose value corresponds to the index of the first available resource in I. Because the allocation of I must be contiguous, p partitions the resources, since a resource i_j is unavailable if $j < p$ or available if $j \geq p$. For instance, if the stack is the infinite resource, p can be considered the stack pointer.

Nevertheless, we still have a problem. Although for a particular machine, the value of p must be finite, the resulting FSA could have as many as 2^{32} stack allocation states for a 32-bit machine. However, we can significantly reduce this number by observing that the decision of where to place a parameter in memory is not based on p, but rather on alignment restrictions. For our example, we care only if the next available memory location is one-, four-, or eight-byte aligned. Consequently, we can capture the allocation state of the machine with three bits that distinguish the memory allocation states. We call these the *distinguishing* bits for infinite resource allocation.

Handling passing structures by value creates a complimentary problem. Since only the "alignment state" of the stack is of interest, structures that affect the state of the P-FSA differently must use different transitions. So for a convention that requires struc-

λ	q_0	q_1	q_2	q_3	q_4	q_5	q_6	q_7	q_8	q_9	q_{10}	q_{11}
char	$\mathbf{a}^1$	$\mathbf{a}^2$	$\mathbf{a}^3$	$\mathbf{a}^4$	000	001	010	011	100	101	110	111
int	$\mathbf{a}^1$	$\mathbf{a}^2$	$\mathbf{a}^3$	$\mathbf{a}^4$	mem_1[a]	mem_2[b]	mem_2	mem_2	mem_2	mem_1	mem_1	mem_1
double	$\mathbf{a}^1\mathbf{a}^2$	$\mathbf{a}^2\mathbf{a}^3$	$\mathbf{a}^3\mathbf{a}^4$	mem_3[c]	mem_3	mem_3	mem_3	mem_3	mem_3	mem_3	mem_3	mem_3

Table I: Definition of λ for example P-FSA.

a. mem_1 = 000 001 010 011
b. mem_2 = 100 101 110 111
c. mem_3 = 000 001 010 011 100 101 110 111

tures to be passed in 8-byte aligned memory locations, all structures of size *n* where *n* mod 8=1 share the same transition out of a given state. Therefore, number of transitions leaving a state is limited by the alignment restrictions of the machine.

4.2 P-FSA Definition

To generalize our approach, we have the set of finite machine resources $R = \{r_1, r_2, \ldots, r_n\}$, infinite resource $I = \{i_1, i_2, \ldots\}$[1], and *selection criteria* $C = \{c_1, c_2, \ldots, c_m\}$. The selection criteria correspond to characteristics about arguments (such as their type and size) that the calling convention uses to select the appropriate placement for an argument. We encode the *signature* of a procedure with a string $w \in C^*$. Each state q in the automaton is labeled according to the allocation state that it represents. The label includes a bit vector v of size n that encodes the allocation of each of the finite resources in R. Additionally, to express the state of allocation for an infinite resource, we include d, the distinguishing bits of index p. So, a state label is a string vd that indicates the resource allocation state. In our example, $n = 4$, and $\|d\| = 3$. So, each state is labeled by a string from the language $\{0, 1\}^4\{0, 1\}^3$. The output of M is a string $s \in P$, where $P = R \cup \{0, 1\}^{\|d\|}$, which contains the placement information. So, from our example in Figure 5, state q_8 is labeled 1111 100 to indicate that each of the argument registers has been used, and that the first available stack location is four-byte aligned.

From the above discussion, we have the following values that are pertinent to defining a finite state machine:

- a set of finite resources $R = \{r_1, r_2, \ldots, r_n\}$.
- an infinite resource $I = \{i_1, i_2, \ldots\}$.
- d, the distinguishing bits of p.
- selection criteria $C = \{c_1, c_2, \ldots, c_m\}$.
- bit vector $v = \{b_1, b_2, \ldots b_n\}$, where b_i is set if resource r_i is used.
- the set of placement strings $P = R \cup \{0, 1\}^{\|d\|}$.

We now formalize our definition of a P-FSA for modeling placement. Since the P-FSA produces output on transitions, we have a Mealy machine [MEAL55]. We define the P-FSA as a six-tuple[2] $M = (Q, \Sigma, \Delta, \delta, \lambda, q_0)$, where:

- Q is the set of states with labels $\{0, 1\}^n\{0, 1\}^{\|d\|}$ representing the allocation state of machine resources,
- the input alphabet $\Sigma = C$, is the set of selection criteria,
- the output alphabet $\Delta = P$, is the set of placement strings,
- the transition function $\delta: Q \times \Sigma \rightarrow Q$,
- the output function $\lambda: Q \times \Sigma \rightarrow \Delta^+$,
- q_0 is the state labeled by $0nw$ where $\|w\| = \|d\|$ is the initial state of d.

We also define $\hat{\delta}: Q \times \Sigma^* \rightarrow Q$ and $\hat{\lambda}: Q \times \Sigma^* \rightarrow \Delta^*$ which are just string versions (defined by Hopcroft and Ullman [HOPC79]) of δ and λ, respectively. So, for our example, we have $M = (Q, \{$`char`, `int`, `double`$\}, \{\mathbf{a}^1, \mathbf{a}^2, \mathbf{a}^3, \mathbf{a}^4\} \cup \{0, 1\}^3, \delta, \lambda, q_0)$, where Q and δ are pictured in Figure 5 and λ is defined in Table I. Note that we have modified the traditional definition of λ to allow multiple symbols to be output on a single transition. This reflects the fact that arguments can be located in more than one resource. For example, in state q_5 on an `int`, Table I indicates that M produces the string of four symbols 100 101 110 111 that indicates four bytes that are four-byte aligned, but are not eight-byte aligned.

The signature:

```
int phred(double, double, char, int);
```

will take the P-FSA in Figure 5 from state q_0 to q_4 producing the string $(\mathbf{a}^1\ \mathbf{a}^2)\ (\mathbf{a}^3\ \mathbf{a}^4)$ (000) (100 101 110 111) along the way. The parentheses in the output string are required to determine where the placement of one argument ends and the next argument's placement begins. Although these are necessary, we have omitted them from our automaton definition to simplify its presentation. From the string, we can derive the placement of the `phred`'s arguments. The first `double` is placed in registers $\mathbf{a}^1$ and $\mathbf{a}^2$, the second in registers $\mathbf{a}^3$ and $\mathbf{a}^4$, the `char` at the first stack location and the `int` starting in the fifth stack location. The padding on the stack between the `char` and the `int` is indicated by the omission of locations 001, 010 and 011 that correspond to the pad locations.

4.3 Automatic P-FSA Construction

In this section, we present an algorithm for automatically constructing automata to model placement computations. For the moment, we assume the existence of a function $f: \Sigma^* \rightarrow \Delta^*$. f computes the same value as M. Since f and M are equivalent, why construct M at all? The answer is that f may have undesirable properties. For instance, M may be used in a context, such as a compiler, where performance is an issue. If f is implemented as an interpreter, the time it takes to compute a placement may not satisfy the performance constraints. Additionally, by using a P-FSA, there are several properties (such as an upper bound on M's execution time) we can prove about the P-FSA that we cannot prove about f. We present such properties in Section 5.

We construct the P-FSA by performing a depth-first-traversal of the states in Q to determine the set of reachable states from q_0. At each state q, the states that are reachable from q in one step are determined by using each element of $\{wc \mid c \in C\}$ as input to f. Each newly reachable state q' is added to Q and is subsequently visited by BUILD-P-FSA (The algorithms are included in Appendix A). Finally, the appropriate additions to δ and λ are made for q'.

1. This can easily be extended to model more than one infinite resource.
2. In this paper, we use the notation of Hopcroft and Ullman for finite state automata and regular expressions [HOPC79]. We use letters early in the alphabet (a, b, c) to denote single symbols. Letters late in the alphabet (w, x, y, z) will denote strings of symbols.

BUILD-P-FSA also uses an auxiliary function STATE-LABEL:$P \rightarrow Q$. STATE-LABEL takes an output string from M and computes the label for the state that M was in when the input was exhausted.

Our construction is now complete, except the definition of the function f. We supply f's definition using an interpreter. We have designed and implemented a language for specifying procedure calling conventions. The language has an interpreter that takes as input a calling convention specification, information about a procedure's signature and some additional information about the target machine, and produces the necessary mapping information to properly call the given procedure. Thus, this interpreter can be used to implement f in our algorithm above. In Section 6, we present the interpreter's use in an implementation.

5 Completeness and Consistency in P-FSA's

In this section, we consider a number of different properties of procedure calling conventions. But first we identify several implementation difficulties that one might encounter when dealing with a calling convention.

5.1 Common Difficulties

Applications, such as compilers and debuggers, which generate, or process procedures at the machine-language level require knowledge of the calling convention. Until now, the portion of such an application's implementation that concerned itself with the procedure call interface was constructed in an ad-hoc manner. The resulting code is complicated with details, difficult to maintain, and often incorrect. In our experience, we have encountered many recurring difficulties in the calling convention portion of a retargetable compiler. There are three sources for these problems: the convention specification, the convention implementation, and the implementation process. We address each of these in the following paragraphs.

Many problems arise from the method of convention specification. Often, no specification exists at all. Instead the native compiler uses a convention that must be extracted by reverse-engineering it. In the cases where a specification exists, it typically takes the form of written prose, or a few general rules (*e.g.*, our example description in Figure 3). Such methods of specification have obvious deficiencies. Furthermore, even if we have an accurate method for specifying a convention, it still may be possible to describe conventions that are internally inconsistent, or incomplete. For example, the convention may require that more than one procedure argument be placed in a particular resource. Another possibility is that the specification may omit rules for a particular data type, or combination of data types.

Those problems that do not stem from the specification result from incorrect implementation of the convention. Many of the same problems in the specification process also plague the implementation. Many conventions have numerous rules, and exceptions that must be reflected in the implementation. Another difficulty is that the implementation may require the use of the convention in several different locations. Maintaining a correspondence between the various implementations can itself be a great source of errors. Finally, this problem is exacerbated by the fact that the implementation frequently undergoes incremental development. Rather than taking on the chore of implementing the entire convention at once, a single aspect of the convention, such as providing support for a single data type, is tackled. After successfully implementing this subset, the next increment is tackled. In doing so, some aspect of the first stage may break due to the interactions between the two pieces.

The result of these observations is that there are several properties that we would like to ensure about a specification and implementation. The above discussion motivates the following categories of questions:

1. Completeness:
   a. *Does the specified convention handle any number of arguments?*
   b. *Does the convention handle any combination of argument types?*
2. Consistency:
   a. *Does the convention map more than one argument to a single machine resource?*
   b. *Do the caller and callee's implementations agree on the convention?*

Many questions like these can be answered using P-FSA's. The following sections show how we can prove certain properties about conventions that ensure desirable responses to the above questions.

5.2 Completeness

The completeness properties address how well the convention covers the possible input cases. A convention must handle any procedure signature. If we could guarantee that the convention was complete, or covered the input set, then we could answer the completeness questions posed in the previous section. We can determine if a convention is complete by looking at the resulting P-FSA. For example, will the convention work for any combination of argument types? The answer lies in the P-FSA transitions. For the convention to be complete, each state $q \in Q$ must have $\delta(q, c)$ defined for all $c \in C$.

Using P-FSA's, we can guarantee that no incomplete convention will go undetected. For an incomplete convention K to not be detected, it would first have to be constructed using our algorithm. Assume such a P-FSA M exists for K. Then there must be some state q_k that is reachable from q_0 but does not have $\delta(q_k, a)$ defined for some $a \in C$. Let W_k denote the set of all strings x such that $\delta(q_0, x) = q_k$. That is, W_k is the set of strings that take M from state q_0 to q_k. Thus, for all strings x such that x e W_k, xa represents a signature that K does not cover. However, during construction, BUILD-P-FSA visited state q_k with some string w such that $\delta(q_0, w) = q_k$. Thus, w must be in W_k and must not be covered by K. Since BUILD-P-FSA calls $f(wc)$ for all $c \in C$, f will be called using $f(wa)$. Since wa is not covered by K, $f(wa)$ will be undefined. At this point the construction process will signal that K is incomplete.

5.3 Consistency

The consistency properties address whether the convention is internally and externally consistent. A convention is internally consistent if there is no machine resource that can be assigned to more than one argument. A convention is externally consistent if the caller and callee agree on the locations of transmitted values. In our model, we *detect* internal inconsistency, and *prevent* external inconsistency.

To detect internal inconsistencies, we again turn to the P-FSA. If the convention only used finite resources, detecting a cycle in the P-FSA would be sufficient to detect the error. However, when infinite resources are introduced, so are cycles. We cannot have an internal inconsistency for an infinite resource since p is defined to be monotonically increasing. We detect finite resource inconsistencies in the following manner. An inconsistency can occur when there is a transition from some state q_j to q_k where bit i in the finite bit vector is 1 in q_j, but 0 in q_k. At this point, M has lost the information that resource r_i was already allocated. We can detect this change by comparing all pairs of bit vectors v_1, v_2 such that v_1 labels q_j, v_2 labels q_k and $\delta(q_j, c) = q_k$ for some $c \in C$. To do

the comparison, we compute $v_3 = (v_1 \oplus v_2) \wedge v_1$. $v_1 \oplus v_2$ selects all bits that differ between v_1 and v_2. We logically and ($\wedge$) this with v_1 to determine if any set bits change value. Thus, if v_3 has any bit set, we have an inconsistency.

Our convention specification language prevents external inconsistencies in the calling convention. A convention specification only defines the argument transmission locations once. Although both the caller and the callee must make use of this information, the specification does not duplicate the information. Since we only have a single definition of argument locations, we only construct a single P-FSA to model the placement mapping. This single P-FSA is used in both the caller and callee. By doing so, we prevent external inconsistencies by requiring the caller and callee use the same implementation for the placement mapping.

6 The Implementation

6.1 The Interpreter

We have implemented an interpreter for the CCL specification language. The interpreter's source is approximately 2500 lines of Icon code [GRIS90]. The interpreter takes as input the CCL description of a procedure calling convention, a procedure's signature, and some additional information about the target architecture, and produces locations of the values to be transmitted, in terms of both the callee and the caller's frame of reference.

We have developed CCL specifications for the following machines: MIPS R2000, SPARC, DEC VAX-11, Motorola M68020, and Motorola M88100. Each of these CCL specifications is approximately one page in length. Using the specification for the MIPS, and the CCL interpreter, we constructed a P-FSA that implements the MIPS calling convention. The MIPS P-FSA uses only 16 out of a possible 512 states (the state label has 9 bits), but requires nine transitions for each state to implement the selection criteria for the **C** programming language. Since the MIPS convention has more machine resource classes and alignment requirements than any of the other machines, it represents the most complicated convention we have. Therefore, we would expect P-FSA's for the other architectures to be significantly smaller. For machines that pass procedure arguments on the stack with no alignment restrictions, such as the VAX-11, would only be a few states.

For comparison purposes, we have examined the calling convention specific code for a retargetable compiler. The MIPS implementation requires 781 lines of **C** code, while the SPARC implementation has 618 lines. This code is one of the most complex sections of the machine-dependent code. This code is replaced by the P-FSA tables and a simple automaton interpreter.

6.2 Realizing the Calling Sequence

In this section, we present how the information from our CCL descriptions can be used to generate calling sequences for the *vpcc/vpo* optimizing compiler[BENI88][BENI94].

In our compiler, the code for the procedure bodies is generated without knowledge of the calling convention. For a callee, the optimizer treats formal parameters as local variables. It assigns each parameter either a register or a memory location, based on the parameter's predicted reference frequency. Thus, although an established convention for where values cross the procedure call interface exists, the code generated by our compiler for a procedure's body may not conform to the convention.

To correct this problem, instructions are placed before and after the callee's body, and before and after the call site in the caller. We call these instructions the caller/callee prologue/epilogue sequences. It is these sequences of instructions that are collectively called the calling sequence. The sequences introduce four new interfaces shown as ▭ in Figure 6. In each sequence, the instructions transform a convention interface to a code body interface or vice versa. Since these sequences of instructions are used to "glue" the procedure bodies to the convention interfaces, they correspond to the agents, shown in Figure 1, of our high-level model.

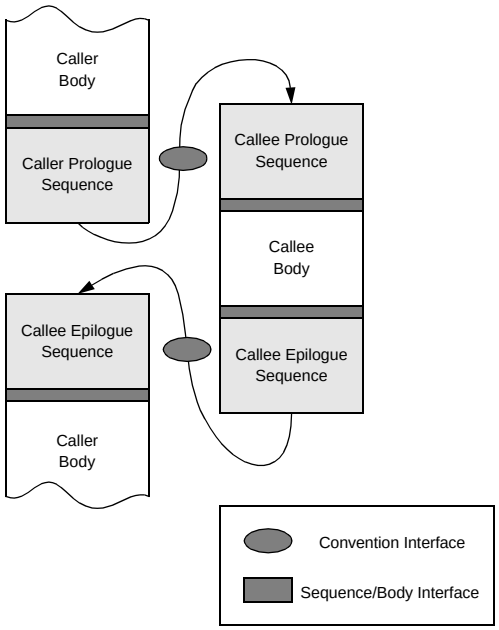

Figure 6: Calling Sequence Locations

An agent's responsibilities fall into one of three categories: allocation or deallocation of storage space, movement of values from their locations in the first interface to locations in the second interface, and the construction/restoration of procedure execution environments. Hence, to generate an agent's actions, we must have information about where the calling convention expects values, what space to allocate or free, and the procedure's environment structure. We can automatically generate the first two.

To illustrate our technique, we show how to generate the instruction sequence for one agent. The instruction sequences that correspond to the other three agents are generated exactly the same way. For our example, we focus on the prologue callee agent for the procedure `foo` introduced earlier.

Recall that for our hypothetical machine, `foo`'s arguments are placed by the caller in locations $\mathbf{a}^1$, $\mathbf{a}^2$, $\mathbf{a}^3$, $\mathbf{M}$[sp:sp+7]. The frame layout on the stack just before control passes to `foo` is shown in Figure 7a. Assume that in generating the `foo`'s body, the optimizer uses two non-volatile registers, allocates 12 bytes of memory for local variables (including `foo`'s arguments) and uses 8 bytes of spill space. One possible frame layout for `foo` is shown in Figure 7b. The relative locations of the temporary spill space, local variable space and non-volatile register save space are determined by the optimizer. The optimizer provides the locations where the callee body expects values. These are listed in the second column of Table II. These locations represent an agreement between the callee body and the callee prologue agent.

The optimizer calls the P-FSA interpreter with the `foo`'s signature and values of the external variables:

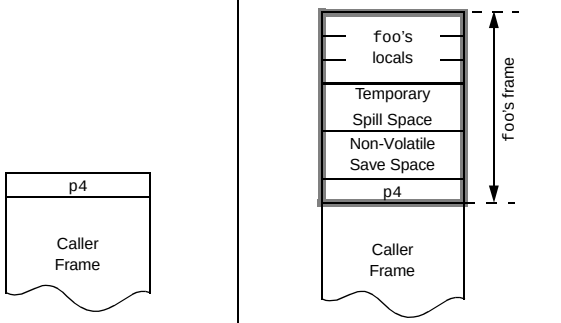

Figure 7a: Frame before call. **Figure 7b:** `foo`'s frame layout.

Figure 7: Frame layouts at different stages of procedure call.

[SPILL_SIZE=8, LOCALS_SIZE=12, NVSIZE=8,
(**ARG**^{1}, type:char, size:1),
(**ARG**^{2}, type:int, size:4),
(**ARG**^{3}, type:int, size:4),
(**ARG**^{4}, type:double, size:8)]

The P-FSA returns view changes, a list of argument locations that correspond to the calling convention, and a list of non-volatile registers:

[(∀ *offset* ∈ {-∞:∞}, **M**[sp + *offset*] **:** **M**[sp + *offset* + 28]),
[(**ARG**^{1}, **a**^{1}),
(**ARG**^{2}, **a**^{2}),
(**ARG**^{3}, **a**^{3}),
(**ARG**^{4}, **M**[sp+28:sp+35]),
[non-volatile**:** **a**^{6}, **a**^{7}, **a**^{8}, **a**^{9}]]

In this example, the view change occurred before the list of locations. Therefore, the locations reflect this fact.

View change information corresponds to the allocation or deallocation of storage space. This view change indicates that any memory location's address, that contains a valid value for offset, shifts down by 28 bytes. Since offset can take on any positive or negative value (-∞:∞), this corresponds to all addresses relative to the stack pointer. Thus, a decrement of the stack pointer by 28 bytes is needed. This allocation of stack space will appear as a view change since it changes the names of all locations referenced by the stack pointer. A table is consulted for each view change in the CCL description. The table maps all view changes to valid machine instructions.

After the view change has been performed, the necessary moves must be made to transform the agreement between the caller prologue agent and callee prologue agent to the agreement between the callee prologue agent and the callee body. Table II summarizes the location information. Column 1 shows the locations returned by the P-FSA. Column 2 shows the locations that the optimizer supplies. Column 3, which can be trivially derived from columns 1 and 2, indicates the necessary actions. Each of these moves is a register/memory to register/memory move. A table of available move instructions is consulted to determine the necessary instructions to be inserted into the callee prologue's sequence.

After the agent's actions are determined, the list of sources and destinations must be examined to determine if there are any dependencies. If a source is also a destination, the move containing the source must be performed before the move containing the destination, otherwise the source value will be lost. It is not uncommon for a circularity to exist. For example, if **a**^{1}→**a**^{2} and **a**^{2}→**a**^{1}, we must introduce a third location to break the circularity: **a**^{1}→temp, **a**^{2}→**a**^{1}, temp→**a**^{2}. Either an available register or a memory location must be used to temporary hold one of the values. In our optimizer, we usually have a register available.

	Convention	Callee Prologue Agent/Callee Agreement	Callee Prologue Agent Actions
Arguments	p1:**a**^{1}	p1:**a**^{3}	**a**^{1}→**a**^{3}
	p2:**a**^{2}	p2:**M**[sp+4:sp+7]	**a**^{2}→**M**[sp+4:sp+7]
	p3:**a**^{3}	p3:**a**^{4}	**a**^{3}→**a**^{4}
	p4:**M**[sp+28:sp+35]	p4:**a**^{1},**a**^{2}	**M**[sp+28:sp+35]→**a**^{1},**a**^{2}
Non-Volatile	**a**^{6}	**M**[sp+20:sp+23]	**a**^{6}→**M**[sp+20:sp+23]
	a^7	**M**[sp+24:sp+27]	**a**^{7}→**M**[sp+24:sp+27]
	a^8	**a**^{8}	—
	a^9	**a**^{9}	—

Table II: Summary indicating how callee prologue agent actions are determined from placement information from both interfaces.

For our implementation of the **C** language, the callee prologue has no other responsibilities. However, in other implementations, or other languages, special environment initialization might be required. For example, in PASCAL, the variable "display" that is used for addressing outer-block variables might need to be setup. Although this would probably be performed in the callee prologue sequence, the initialization is not part of the calling convention and is, therefore beyond the scope of this system.

At this point, the callee prologue instruction sequence is complete. So far, we have not addressed instruction sequence efficiency. Because of the frequency of procedure calls, generating efficient instruction sequences is an important feature of optimizing compilers. In our compiler, the resulting instruction sequences are processed by the optimizer. Thus, although the instruction sequences that are initially generated by this process are naive, they benefit from thorough optimization just as other code does.

6.3 Related Issues

Providing support for procedures that may receive a varying number of arguments is always difficult. In the **C** language, the mechanism used is *varargs* which is more a convention than a language feature. Johnson and Ritchie spend considerable time explaining the ramifications that *varargs* has on the calling sequence [JOHNSON]. In fact, providing support for **C**'s *varargs* frequently has profound influence on the calling convention. However, in **C**, procedures that receive variable numbers of arguments still adhere to the defined calling convention. While *varargs* must be considered when developing a particular calling sequence, information about *varargs* is not present in the definition of the calling convention.

An important decision when designing a calling convention is deciding which registers retain their value across a procedure call. If some registers retain their value, it is the responsibility of the callee to restore the original values of any such register that is used. Rather than define the mechanism employed in the conven-

tion as caller or callee save, we simply define who is responsible for the save. This is accomplished by indicating that registers are non-volatile. Volatile register values must be saved by the caller, while non-volatile register values must be saved by the callee if it uses them.

The specifications in this paper, and the implementation that we have presented are for the **C** language. We have not, as yet, considered how CCL could be used for languages that are drastically different from **C**. However, we anticipate that CCL could handle features such as heap-based parameter passing without modification.

7 Related Work

What little work there has been in calling sequences has been ad-hoc. For example, Johnson and Richie discuss some rules of thumb for designing and implementing a calling sequence for the **C** programming language [JOHNSON]. Davidson and Whalley experimentally evaluated several different **C** calling conventions [DAVI91]. No attempts have been made to formally analyze calling conventions.

On the other hand, the use of FSA for modeling parts of a compiler, and as an implementation tool has a long and successful history. For example, Johnson et al. [JOHN68] describe the use of FSA's to implement lexical analyzers. More recently, Proebsting and Fraser [PROE94], and Muller [MULL93] have used finite state automata to model and detect structural hazards in pipelines for instruction scheduling.

8 Summary

Current methods of procedure call specification are frequently imprecise, incomplete, contradictory or inconsistent. This comes from the lack of a formal model, or specification language that guarantee these properties. We have presented a formal model, called P-FSA's, for procedure calling conventions that can ensure these properties. Furthermore, we have developed a language and interpreter for the specification of procedure calling conventions. With the interpreter, a P-FSA that models a convention can be automatically constructed from the convention's specification. During construction, the convention can be analyzed to determine if it is complete and consistent. The resulting P-FSA can then be directly used as an implementation of the convention in an application.

9 Acknowledgments

We express our thanks to John Reppy and Sanjay Jinturkar for their extensive comments on earlier drafts of this paper. We should mention in particular Ricky Benitez who provided many insightful conversations and the implementation of the optimizing compiler used in this work. Finally, we would also like to thank the reviewers for their helpful suggestions.

10 References

[BAIL93] Bailey, M.W. and Davidson, J.W. *A Formal Specification for Procedure Calling Conventions*. Technical Report CS-93-59. University of Virginia, 1993.

[BAIL94] Bailey, M.W. and Davidson, J.W. *A Formal Model for Procedure Calling Conventions*. Technical Report CS-94-57. University of Virginia, 1994.

[BENI88] Benitez, M.E. and Davidson, J.W. A Portable Global Optimizer and Linker. *In Proceedings of the SIGPLAN '88 Conference on Programming Language Design and Implementation*, Atlanta, Georgia, June, 1988, 329-338.

[BENI94] Benitez, M.E. and Davidson, J.W. The Advantages of Machine-Dependent Global Optimization. In *Proceedings of the 1994 Conference on Programming Languages and Systems Architectures*, Zurich, Switzerland, March 1994, 105-124.

[DAVI91] Davidson, J.W. and Whalley, D.B. Methods for Saving and Restoring Register Values across Function Calls. *Software—Practice and Experience* 21(2):149–165 February 1991.

[DEC78] Digital Equipment Corporation. *VAX Architecture Handbook*. Digital Equipment Corporation, 1978.

[DEC93] Digital Equipment Corporation. *Calling Standard for AXP Systems*. Digital Equipment Corporation, July 1993.

[FRAS93] Fraser, C.W. Personal Communication, November, 1993.

[GRIS90] Griswold, R.E. and Griswold, M.T. *The Icon Programming Language*, 2nd edition, Prentice-Hall, 1990.

[HOPC79] Hopcroft, J.E. and Ullman, J.D. *Introduction to Automata Theory, Languages, and Computation*. Addison-Wesley, 1979.

[JOHNSON] Johnson, S.C. and Ritchie, D.M. *The C Language Calling Sequence*. Bell Labs.

[JOHN68] Johnson, W.L., J.H. Porter, S.I. Ackley, and D.T. Ross. Automatic generation of efficient lexical processors using finite state techniques, *Communications of the ACM*, 11:(12), 805–813.

[KANE92] Kane, G. and Heinrich, J. *MIPS RISC Architecture*. Prentice Hall, 1992.

[KERN78] Kernighan, B.W. and Ritchie, D.M. *The C Programming Language*. Prentice-Hall, 1978.

[KERN88] Kernighan, B.W. and Ritchie, D.M. *The C Programming Language*, 2nd edition. Prentice-Hall, 1988.

[MEAL55] Mealy, G.H. A method for synthesizing sequential circuits, *Bell System Technical Journal*, 34(5):1045–1079, 1955.

[MULL93] Muller, T. Employing Finite Automata for Resource Scheduling. In *Proceedings of the 26th Annual International Symposium on Microarchitecture*, 1993, 12–20.

[PROE94] Proebsting, T.A. and Fraser, C.W. Detecting Pipeline Structural Hazards Quickly. In *Proceedings 21st ACM SIGPLAN-SIGACT Symposium on the Principles of Programming Languages*, 1994, 280–286.

Appendix A

A.1 Construction Algorithms

We define the algorithm BUILD-P-FSA in Figure 8. The algorithm starts with the initial state q_0 as the only element of Q. Since there are no transitions yet, λ and δ have no rules. A call to BUILD-P-FSA takes three parameters, q, w, and x. q represents the state for BUILD-P-FSA to visit, while w represents the input string such that (q_0, w) yields (q, ε), and x is output string upon reaching q. From this definition, the initial call to BUILD-P-FSA must be BUILD-P-FSA(q_0, ε, ε).

```
function BUILD-P-FSA(q, w, x)
    // q ∈ Q, w ∈ Σ*, x ∈ Δ* | λ̂(w) = x
    for each criterion c ∈ C do
        // compute placement for signature wc
        y ← f(wc);
        // compute state label from placement
        q' ← STATE-LABEL(y);
        if q' ∉ Q then
            Q ← Q ∪ {q'};
            BUILD-P-FSA(q', wc, y);
        end if
        // set a as the suffix of y not in x
        a ← b | xb = y;
        add λ(q, c) = q';
        add δ(q, c) = a;
    end for
end function
```

Figure 8: Algorithm to build the P-FSA

The algorithm for STATE-LABEL is simple. We start with state q_0. As STATE-LABEL reads each symbol from the string, it encounters either the name of a finite resource, or a symbol representing the distinguishing bits of p. In the finite case, the bit corresponding to the resource is set in the finite resource vector. In the infinite case, the distinguishing bits of the state are set to the input symbol that was read. At the end of the input, all finite resources that have been read have their bits set to indicate they are unavailable, and the distinguishing bits indicate the last set of distinguishing bits read. To complete the computation, we need to move the infinite resource index to the next available resource (it currently points to the last unavailable one)[1]. The result of this computation is precisely the label for the final state of M for output w since it indicates which resources are available for allocation. The complete algorithm is shown in Figure 9.

A.2 A Complex Example

We now present a significantly more complex example: the MIPS R3000. The MIPS is a RISC machine with both integer and floating-point registers. Unlike most machines, the MIPS convention designates that not only some integer registers but also some floating-point registers are to be used for passing arguments. Figure 10 contains the complete convention specification.

Although the MIPS convention is more complicated, the description is quite similar to our previous example—with a few additional restrictions. First, notice that the resource list (line 14) now includes the floating-point registers. Each resource set is ordered to indicate that the resources within them must be assigned in sequence. This prevents the subsequent placement operator from using element n after element $n + 1$ has been assigned. Second, we

1. An ordered list of values for p's distinguishing bits is known so that we can perform this calculation, although this is usually just an increment.

```
function STATE-LABEL(w)          // w ∈ Δ*
    z ← 0^n;                     // z is the finite resource vector
    while w ≠ ε do
        // extract the first symbol from w
        define a and x such that ax = w;
        w ← x;                   // set w to the rest of w
        if a ∈ R then            // for finite resources:
            // mark it as used
            set a's corresponding bit in z;
        else                     // for infinite resources:
            d ← a;               // keep the last one encountered
        end if
    end while
    // set d to the next resource (first available)
    d ← d + 1;
    // return the state label made up of z and d
    return zd;
end function
```

Figure 9: Definition of STATE-LABEL

have added several new classes. These reflect the addition of registers for passing arguments and alignment constraints placed on the registers and stack. For example, the class 'intfpregs' is the set of starting points in the integer register set that have even register numbers. The class 'amem' is the set of stack locations that are 8-byte aligned. Finally, the class 'smem' contains a set of starting-point pairs. The pair is used to indicate that if the first resource exhausts, the placement continues using the second resource starting point. This class is used in passing structure arguments and indicates that a single structure argument may span the argument registers and stack.

After properly defining the classes, the placement (lines 27-34) is straightforward. For each type, a list of classes to use is specified. In each case, a register class is first, followed by the corresponding stack class. This reflects the convention that registers are used until exhausted, followed by stack use. The placement is slightly complicated in the floating-point case since the register class to use is dependent on the type of the first argument. When the first argument is a floating-point value, the floating-point registers are used. When the first value is any other type, the integer registers are used to pass floating-point values.

The MIPS convention has two other features we must convey. The first requires that the initial 16 bytes of the frame, which correspond to the argument registers, must be reserved so the callee can save the register arguments if necessary. This is specified on line 15 by setting the 'assigned' attribute for these resources. The second constraint is that floating-point argument registers are associated with the integer registers ($\mathbf{f}^6$ with $\mathbf{r}^4$ and $\mathbf{r}^5$, $\mathbf{f}^7$ with $\mathbf{r}^6$ and $\mathbf{r}^7$). The association requires that if a register in one class is assigned, the associated register in the other class cannot be assigned. Each of the four associations is specified, on lines 23-26, using the existential quantifier ($\exists$) which is simply a conditional expression. These restrictions complete the calling convention for the MIPS. The remaining details are similar to the simple example presented earlier.

```
1   external NVSIZE, SPILL_SIZE, LOCALS_SIZE
2   alias REG_ARGS ≡ 16
3   alias sp ≡ r^29
4   non-volatile {r^1:3, r^8:11, r^16:31}
5   caller prologue
6       view change
7           ∀ offset ∈ {–∞:∞}
8               M[sp + offset] becomes M[sp + offset + ⌈ARG_SIZE⌉^8]
9       end view change
10      data transfer (asymmetric)
11          alias rindex ≡ <4:7>
12          alias fpindex ≡ <6:7>
13          alias mindex ≡ <sp:∞>
14          resources {<r^rindex>, <f^fpindex>, <M^mindex>}
15          ∀ register ∈ {M[sp(REG_ARGS)]} set register.assigned ← true
16          internal ARG_SIZE ← ∑(<M[addr].size | addr ∈ mindex ∧ M[addr].assigned>)
17          class intregs ← <<r^rindex>>
18          class intfpregs ← <<r^x> | x ∈ rindex ∧ x mod 2 = 0>
19          class fpfpregs ← <<f^x> | x ∈ fpindex ∧ x mod 2 = 0>
20          class mem ← <<M[addr]> | addr ∈ mindex ∧ addr mod 4 = 0>
21          class amem ← <<M[addr]> | addr ∈ mindex ∧ addr mod 8 = 0 >
22          class smem ← <<r^rindex, M[addr]> | addr ∈ mindex ∧ addr mod 8 = 0>
23          ∃ reg ∈ {reg | reg ∈ {f^6} ∧ reg.assigned} ⇒ set r^4:5.assigned ← true
24          ∃ reg ∈ {reg | reg ∈ {f^7} ∧ reg.assigned} ⇒ set r^6:7.assigned ← true
25          ∃ reg ∈ {reg | reg ∈ {r^4:5} ∧ reg.assigned} ⇒ set f^6.assigned ← true
26          ∃ reg ∈ {reg | reg ∈ {r^6:7} ∧ reg.assigned} ⇒ set f^7.assigned ← true
27          ∀ argument ∈ <ARG^1:ARG_TOTAL>
28              map argument → argument.type ⊥ {
29                  byte, word, longword:                <intregs, mem>,
30                  struct:                              <smem, amem>,
31                  float, double:ARG^1.type ⊥ {
32                      struct, byte, word, longword:    <intfpregs, amem>,
33                      float, double:                   <fpfpregs, amem>
34                  }
35              }
36      end data transfer
37  end caller prologue
38  callee prologue
39      view change
40          ∀ offset ∈ {-∞:∞}
41              M[sp + offset] becomes M[sp + offset + ⌈SPILL_SIZE + LOCALS_SIZE + NVSIZE⌉^8]
42      end view change
43  end callee prologue
44  callee epilogue
45      data transfer (asymmetric)
46          resources {r^2,f^0}
47          map RVAL^1 → RVAL^1.type ⊥ {
48              byte, word, longword: <<<r^2>>>,
49              float, double:        <<<f^0>>>,
50              struct:               <↑(<<r^2>>)>
51          }
52      end data transfer
53  end callee epilogue
```

Figure 10: The MIPS R3000 Specification