A Formal Model for Procedure Calling Conventions

Mark W. Bailey
Jack W. Davidson

Computer Science Report No. CS-94-27
July 22, 1994

A Formal Model of Procedure Calling Conventions

Mark W. Bailey
mark@virginia.edu
(804) 982-2296

Jack W. Davidson
jwd@virginia.edu
(804) 982-2209

Department of Computer Science
University of Virginia
Charlottesville, VA 22903

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 a number of 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, how to return values, and define which *machine resources*, such as registers, the called procedure must preserve. This understanding between the *caller*[1] and the *callee*[2] is known as the *procedure calling convention*.

Any application that must process or generate assembly language code is likely to need to know about the procedure calling convention. Examples of such applications include compilers, debuggers, and evaluation tools such as profilers. The code in such applications that concerns itself with the calling convention is complicated with details, is frequently riddled with errors, and often only covers a subset of the possible cases. By introducing a formal model of the procedure calling convention, we can simplify the specification, scrutinize the convention to identify common errors, and produce more robust implementations.

[1]The calling procedure is termed the *caller.*
[2]The procedure that is called is termed the *callee*.

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 a MIPS programmer's manual [KH92] the English description is so difficult to understand that the authors provide fifteen examples, several of which are contradictory [Fra93] —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[1]. Thus, it should be clear that there is a need for an accurate, concise description of procedure calling conventions, and a supporting model for their evaluation.

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

2 The Model

When one first tries to model the procedure call interface, one undoubtedly would consider—as we did—modeling the calling *sequence*. However, after some thought, it becomes clear the calling *convention* is what is important. The convention 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. The calling sequence is simply one of many possible implementations of the calling convention.

To illustrate our method, we will focus on one aspect of the procedure calling convention: determining transmission locations for a procedure's parameters. Although we will only present the argument placement, the return value placement can also be modeled in this manner. Other features of the calling convention, including frame layout, stack allocation and register windows are not addressed in this paper.

2.1 A Simple Calling Convention

The calling convention is just a set of rules that the caller and callee must conform to. Figure 1 contains the calling convention rules for a hypothetical machine. Consider the following ANSI **C** prototype for a function `foo`:

```
int foo (char parm1, int parm2, int parm3, double parm4);
```

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:

[1]This document also includes information on exception handling and information pertinent to multithreaded execution environments.

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

which defines a function that takes three arguments (a `char`, an `int`, and a `double`), and returns an `int`.

1. Registers $\mathbf{a}^1$, $\mathbf{a}^2$, $\mathbf{a}^3$, $\mathbf{a}^4$ 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.

Figure 1: Rules for a simple calling convention.

With `foo`'s signature, we can apply the calling convention in Figure 1 to determine how to call `foo`. `foo`'s arguments would be placed in the following locations:

- parm1 in register $\mathbf{a}^1$
- parm2 in register $\mathbf{a}^2$
- parm3 in register $\mathbf{a}^3$
- parm4 on the stack in **M**[sp:sp + 7]

Notice that although register $\mathbf{a}^4$ is available, `parm4` 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 not technically 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.

2.2 P-FSA Representation

We use finite state automata to model each placement in the calling convention. One such FSA is shown in Figure 2. This FSA models the placement of procedure arguments for our 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 machine works by moving from state to state as the location of an argument is determined. On each 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

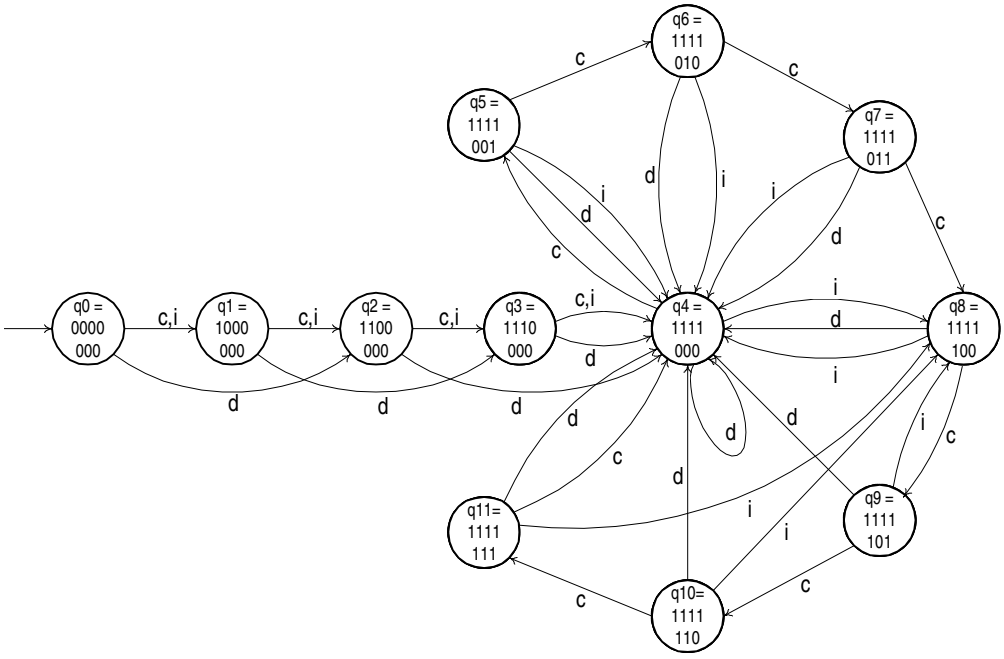

Figure 2: P-FSA for a transmission of parameters for a simple calling convention.

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 of 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 $I = \{i_1, i_2, ...\}$, 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 $j \leq p$. For instance, if the stack is the infinite resource, p can be considered to be 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. Nevertheless, 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.

2.3 P-FSA Definition

To generalize our approach, we have the set of finite machine resources $R = \{r_1, r_2, ..., r_n\}$, infinite resource $I = \{i_1, i_2, ...\}$[1], and *selection criteria* $C = \{c_1, c_2, ..., c_m\}$. 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 2, 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 $R = \{r_1, r_2, ..., r_n\}$.
- an infinite resource $I = \{i_1, i_2, ...\}$.
- d, the distinguishing bits of p.
- selection criteria $C = \{c_1, c_2, ..., c_m\}$.
- bit vector $v = \{b_1, b_2, ... b_n\}$, 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 [Mea55]. 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 $0^n w$ 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 [HU79]) of δ and λ, respectively. So, for our example, we have $M = (Q, \{\texttt{char}, \texttt{int}, \texttt{double}\}, \{\mathbf{a}^1, \mathbf{a}^2, \mathbf{a}^3, \mathbf{a}^4\} \cup \{0, 1\}^3, \delta, \lambda, q_0)$, where Q and δ are pictured in Figure 2 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 goo(double, double, char, int);
```

[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 [HU79]. 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.

will take the P-FSA in Figure 2 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 `goo`'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.

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[†]	mem_2[‡]	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[††]	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.

[†]mem_1 = 000 001 010 011
[‡]mem_2 = 100 101 110 111
[††]mem_3 = 000 001 010 011 100 101 110 111

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 constants. Additionally, by using a P-FSA, there are a number of 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 4.

We define the algorithm BUILD-P-FSA in Figure 3. 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).

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. Finally, the appropriate additions to δ and λ are made for q'. 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.

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

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

Figure 3: Algorithm to build the P-FSA

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 4.

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

Figure 4: Definition of STATE-LABEL

Our construction is now complete, except for 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 5, we present the language and its interpreter.

[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.

4 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.

4.1 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 a number of 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. In many cases, 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 1). 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 a number of 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.

4.2 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 $\hat{\delta}(q_0, x) = q_k$. 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 $\hat{\delta}(q_0, w) = q_k$. 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.

4.3 Consistency

The consistency properties address whether the convention is internally and externally consistent. A convention is internally consistent if there is no machine resource 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 inconstancy, and *prevent* external inconstancy.

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 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. In doing so, we prevent external inconsistencies by requiring that the caller and callee use the same implementation for the placement mapping.

5 CCL

After reading the previous sections, one my ask: why not use a finite state automaton to specify the placement of procedure arguments? Although it is true that by using finite automata for *specification* one reaps benefits from everything discussed in Section 4, it would exhibit many of the same shortcomings that using a procedural language, like C, does. Such a method of specification would suffer in an incremental development environment. Referring again to our simple example, the P-FSA has twelve states and three data types, resulting in a 36 entry table for both δ and λ. Some convention rules, such as alignment requirements, that are overlooked, or subsequently modified can cause changes to most entries in the δ and λ tables. Further, although the P-FSA is a powerful

tool for modeling the placement functions, it is not an intuitive method of specification. It is for these reasons that we have chosen to provide our Calling Convention Language (CCL). In this section, we briefly[1] present CCL, and its interpreter.

5.1 The Language

Figure 5 shows the excerpt of a CCL specification that corresponds to the placement rules from our running example. The first thing to notice about CCL descriptions is the prevalent use of typographical extensions. In contrast to simple ASCII text, the typographical extensions prove to be a more natural way to describe many of the data types used in CCL.

```
caller prologue
    data transfer (asymmetric)
        resources {<a^1:4, M[sp:∞]>}
        ∀argument ∈ <ARG^1:ARG_TOTAL>
            map argument → argument.type ⊥ {
                char: <<<a^1:4>>, <<M[sp:∞]>>>,
                int:<<<a^1:4>>, <<M[addr] | addr ∈ <sp:∞> ∧ addr mod 4 = 0>>>,
                double: <<<a^1:4>>, <<M[addr] | addr ∈ <sp:∞> ∧ addr mod 8 = 0>>>
                }
    end data transfer
end caller prologue
```

Figure 5: Excerpt from a simple specification.

The procedure call interfaces are defined in terms of two concepts: data placement and the view change. Data placement defines where information should be placed/found and who is to place it there. The view change defines when and how the view of the machine's resources changes. We have found that these two concepts are enough to define most common calling conventions.

Figure 5 contains a data placement definition for the caller prologue. A data placement definition must contain a resource declaration and a placement operation. The **resources** statement defines the set of machine resources where values may be placed. We use the ordered set expression <**a**$^{1:4,}$ **M**[sp:∞]> in the **resources** statement to indicate that all of the machine resources must be allocated in order (**a**^{2} cannot be used after **a**^{3} has been allocated). The other statement in the data placement definition contains two notable operators, the universal quantifier (∀) and the placement operator (→). The universal quantifier iterates over the set <**ARG**$^{1:ARG\_TOTAL}$>, each time binding the variable *argument* to an element of the set. Here the set is ordered, ensuring that *argument* will take values of the set in order. The resource **ARG** is a special resource that is provided to convey information about the signature. The placement operator is invoked for each value *argument* is assigned. It takes a value (in this case an argument) and a list of classes. A *class* defines a list of machine resources that may be used as starting locations for placing values. Figure 5 contains four classes: machine registers, memory locations, four-byte aligned memory locations, and eight-byte aligned memory locations. The selection operator (⊥) is used to select one of the three class lists based on *argument*'s type field. The resulting classes are searched, in order, for an available resource to place the given value.

The example P-FSA implements the data placement that has been defined in Figure 5. It is not necessary to duplicate this definition in the callee prologue section of the CCL description.

[1] A more complete description of CCL is available in [BD93].

Since this data placement defines where the procedure arguments are to be placed, it also defines where they are to be found. It is precisely this kind of symmetry that we use to provide externally consistent descriptions.

The view change declaration indicates something has happened that causes 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 changes. We wish to indicate this change without causing the move to actually occur. The change of view declaration indicates how the names of locations have changed. This declaration is used more commonly when describing that a frame has been pushed on the stack. When a push occurs, all locations referenced by the stack pointer appear to shift.

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.

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 defines a slightly different calling convention than its successor ANSI C. One difference is that C always promotes arguments of type float to type double. ANSI C does not. These differences are part of the calling convention, and are, therefore, present in the resulting CCL descriptions.

5.2 The Interpreter

We have implemented an interpreter for the CCL specification language. The interpreter's source is approximately 2500 lines of Icon code [GG90]. 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 is 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 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 [JR]. Davidson and Whalley experimentally evaluated several different C

calling conventions [DW91]. 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. [JPA68] describes the use of FSA's to implement lexical analyzers. More recently, Proebsting and Fraser [PF94], and Muller [Mul93] have used finite state automata to model and detect structural hazards in pipelines for instruction scheduling.

7 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.

8 References

[BD93] Bailey, M.W. and Davidson, J.W. *A Formal Specification for Procedure Calling Conventions*. Technical Report CS-93-59. University of Virginia, 1993.

[DW91] 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.

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

[Fra93] Fraser, C.W. Personal Communication, November, 1993.

[GG90] Griswold, R.E. and Griswold, M.T. *The Icon Programming Language*, 2nd edition, Prentice-Hall, 1990.

[HU79] Hopcroft, J.E. and Ullman, J.D. *Introduction to Automata Theory, Languages, and Computation*. Addison-Wesley, 1979.

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

[JPA68] 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.

[KH92] Kane, G. and Heinrich, J. *MIPS RISC Architecture*. Prentice Hall, 1992.

[KR88] Kernighan, B.W. and Ritchie, D.M. *The C Programming Language*, 2nd edition. Prentice-Hall, 1988.

[Mea55] Mealy, G.H. A method for synthesizing sequential circuits, *Bell System Technical Journal*, 34(5):1045–1079, 1955.

[Mul93] Muller, T. Employing Finite Automata for Resource Scheduling. In *Proceedings of the 26th Annual International Symposium on Microarchitecture*, 1993, 12-20.

[PF94] 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.