A Formal Specification for Procedure Calling Conventions

Mark W. Bailey
Jack W. Davidson

Computer Science Report No. CS-93-59
November 5, 1993

A Formal Specification for 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 level require an accurate description of the calling convention in use. Until now, the portion of such an application that concerns itself with the procedure call interface was implemented in an ad-hoc manner. The resulting code is complicated with details, difficult to maintain, and often incorrect. In this paper, we present a model and language, called CCL, for specifying procedure calling conventions. The language can be used to automatically generate the calling-convention specific portions of applications. It also serves as accurate documentation for a given calling convention. CCL provides a concise and natural method of specifying calling conventions that are easy to read and modify. By using a convention specification to automatically generate code, one can enhance the retargetablility of applications that make use of procedure calling conventions. Additionally, different calling conventions can easily be implemented, thereby simplifying experimentation with different call conventions to increase the efficiency of the procedure call.

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*[1] and *callee*[2] is known as the *procedure calling convention*. Because of the machine-specific nature of the calling convention, it varies widely from machine-to-machine, programming-language-to-programming-language, and in fact, language-implementation-to-language-implementation.

1.1 Why a 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 [KH92] the

1. The calling procedure is known as the *caller.*
2. The called procedure is known as the *callee.*

English description is so difficult to understand that the authors provide fifteen examples, several of which are contradictory. However, this is understandable since the convention, once understood, is so difficult to implement that 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.

1.2 Applications

Any application that must process or generate assembly language code is likely to need to know about a procedure calling convention. Examples of such applications 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 an accurate 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 based on desired 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 set forth the issues in providing an efficient calling sequence having already defined a calling convention [JR]. 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 [DW91]. 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.

We divide the remainder of this paper into three sections. Section 2 presents the model we use as a framework for defining calling conventions. Section 3 describes the language we have developed to specify calling conventions and presents a couple of examples. The paper concludes with Section 4.

2 The Model

In this section we describe the underlying model for the convention descriptions. Many features of the description language have their foundation in the underlying model.

1. This document also includes information on exception handling and information pertinent to multi-threaded execution environments.

2.1 Convention vs. Sequence

When one first tries to model the procedure call interface, one undoubtedly would 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 investigation, the subtle differences between the convention and the sequence become more 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. The calling sequence can be thought of as a definition of *how* to realize the convention.

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* that 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. Figure 1 shows the two interfaces and the four corresponding agents. For the procedure call

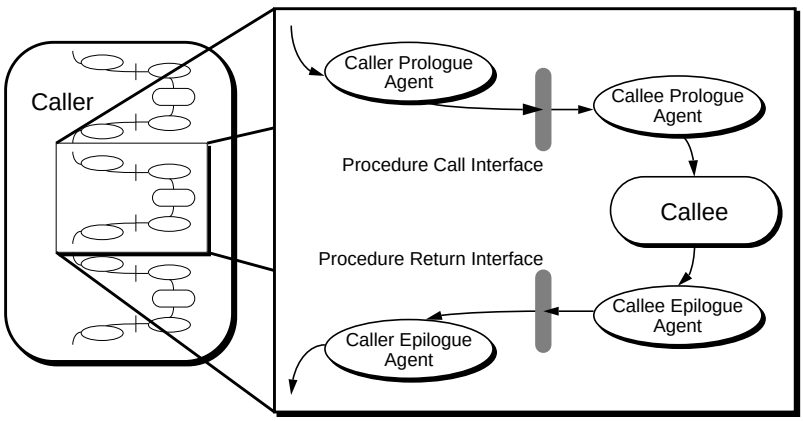

Figure 1: The Role of Agents in Procedure Call and Return Interfaces.

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 correctly constructing a calling sequence difficult.

2.3 Defining the Interface

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. These two concepts are enough to define most common calling conventions.

2.3.1 The Data Placement Definition

Data placement is used to define what information must be located where. 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 (values of registers). In the former, this is an understanding 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 applica-

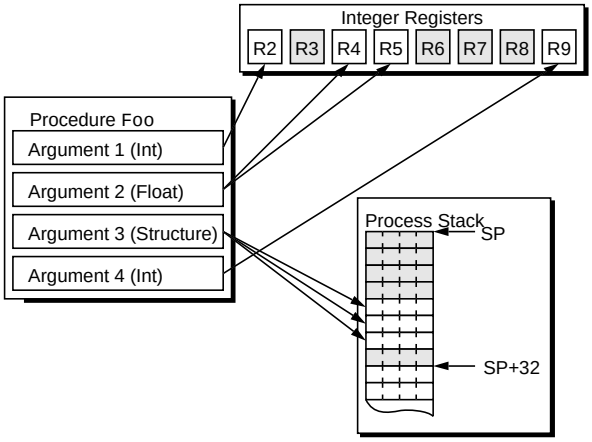

Figure 2: Mapping from arguments to machine resources.

tion 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 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.

2.3.2 The View Change Declaration

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 con-

tents 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 declaration indicates how the names of locations have changed. This declaration 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.

In summary, we model the procedure call interface using not one, but two interfaces, each with a pair of agents. The responsibilities of these agents are described using two concepts: the data placement definition and the view change declaration. These abstractions are all that one needs to accurately define procedure calling conventions.

3 The Language

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

3.1 Typographical Extensions

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 are able to 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 of the expressions used in CCL:

- Sets: {2:9} ≡ {2,3,4,5,6,7,8,9}
- Ordered sets: <2,8,3,9,4,10>, <0:∞>
- Labelled 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*

An advantage of using typographical extensions is that a simple, concise convention indicates the portions of the description 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 slight disadvantages of typographical extensions. One is that descriptions cannot be edited with existing text editors (e.g., vi, emacs, 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.

3.2 External Environment

CCL is one 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.

A CCL description is typically language depended as well. This is, in part, because the language definition influences the calling convention. For example, the **C** language [KR78] defines a slightly different calling convention than its successor ANSI **C** [KR88]. 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. All of the examples in this paper assume the traditional **C** language calling convention.

3.3 A Simple Example

To introduce the language concepts and notation, we first look at a simple example. In this example, procedure arguments are passed on the stack and procedure return values are passed in registers. All return values and arguments are passed by value, except structure return values which are passed by reference (i.e., a pointer to the structure is placed a register). Additionally, in this convention, some registers values must be preserved across the procedure call. Consequently, the callee must save the values of these registers if it uses them. This convention is almost identical to Digital's VAX **C** language calling convention [DEC78].

3.3.1 Placement of Procedure Arguments

First, we examine the placement of procedure arguments. In this scheme, all the arguments, regardless of type, are passed by value on the stack. Further, there are no alignment constraints placed on the stack pointer, or the locations of argument values. Figure 3 contains the brief specification for this scheme. We use the **alias** statement to

```
1   alias mindex ≡ <r14+4:∞>
2   resources {M[mindex]}
3   class mem ← <<M[addr]> | addr ∈ mindex>
4   ∀ argument ∈ <ARG1:ARG_TOTAL> map argument → <mem>
```

Figure 3: A simple stack-based argument passing scheme.

introduce the name 'mindex' as a set of stack addresses ($\mathbf{r}^{14}$ is the stack pointer). The **resources** declaration defines the set of possible destinations for data placement, which we will simply call the resources. The **class** statement defines a subset of these resources where placements may start. In this simple example, since there are no restrictions on where procedure arguments are placed, this class contains all the resources. 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, we consider resources 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 final line contains two notable operators, the universal quantifier ($\forall$) and the placement operator ($\rightarrow$). The universal quantifier 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 external environment. It contains information such as the type and size of the arguments for the call. The placement operator is invoked for each value *argument* is assigned. The placement operator takes a value (in this case an argument) and 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.

3.3.2 Placement of Procedure Return Values

Specifying the locations of procedure return values is similar to procedure arguments. Figure 4 shows an example. We see the other special, externally defined resource—**RVAL**—which refers to the list of return values (in most languages, there is only one). The resources used for returning values are the registers $\mathbf{r}^0$ and $\mathbf{r}^1$. We see the use of the selection operator ($\perp$) on labeled sets. This is simply a case expression. Based on the value of $\mathbf{RVAL}^1$'s type attribute, an expression in the set is selected. Thus, for integer and floating point values, the return values are passed by value using class 'reg'. For a structure, which might not fit in $\mathbf{r}^0$ and $\mathbf{r}^1$, the value is returned by reference, also using class 'reg'. This is indicated by the indirection operator ($\uparrow$) on the class. Let us assume that the register $\mathbf{r}^0$ and $\mathbf{r}^1$ have size 4 bytes, integers are 4 byte quantities and floating-points are 8 byte quantities. Then, this specification indicates only $\mathbf{r}^0$ will be used for integers, but $\mathbf{r}^0$ and $\mathbf{r}^1$ will be used for floating-point 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.

```
1    resources {r^0:1}
2    class reg <<<r^0>>
3    map RVAL^1 → RVAL^1.type ⊥ {
4        byte, word, longword, float, double: <reg>,
5        struct: <↑(reg)>
6    }
```

Figure 4: Specifying the layout of procedure return values

3.3.3 Placement of Register Values

The third, and final type of data placement that is typically part of a procedure calling convention is the saving of register values. We show a typical specification in Figure 5. The declared resources are just some available

```
1    alias mindex ≡ <sp + SPILL_SIZE + LOCALS_SIZE:∞>
2    resources {<M[mindex]>}
3    ∀ register ∈ <r^6:15> register → <<<M[mindex]>>>
```

Figure 5: Specifying the locations to save register values.

stack space. 'SPILL_SIZE' and 'LOCALS_SIZE' are external values used as offsets into the stack area. We use $\forall$ to sequentially place each of registers $\mathbf{r}^6$ thru $\mathbf{r}^{15}$ onto the stack.

3.3.4 Putting it All Together

So far, we have examined the specification of each aspect of our simple stack-based convention in isolation. We now gather them together to form the complete description shown in Figure 6. 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 placed each of the three data placement schemes discussed above in their corresponding locations in the description.

First, lets examine the **call prologue** section. This section indicates the responsibilities of the caller prologue agent. The **data transfer** statement essentially contains the information from Figure 3 specifying the placement of arguments. Line 11, however, is new. It computes the amount of space that was assigned by the placement operator.

```
1   external SPILL_SIZE, LOCALS_SIZE
2   alias sp ≡ r^14
3   caller prologue
4       view change
5           M[sp] becomes M[sp + ARGS_SIZE]
6       end view change
7       data transfer (asymmetric)
8           alias mindex ≡ <sp+4:∞>
9           resources {M[mindex]}
10          class mem ← <<M[addr]> | addr ∈ mindex>
11          internal ARGS_SIZE ← ∑({M[addr].size | addr ∈ mindex ∧ M[addr].assigned})
12          ∀ argument ∈ <ARG^1:ARG_TOTAL> map argument → <mem>
13      end data transfer
14  end caller prologue
15  callee prologue
16      view change
17          M[sp] becomes M[sp + SPILL_SIZE + LOCALS_SIZE + NVSIZE]
18      end view change
19      data transfer
20          alias mindex ≡ <sp + SPILL_SIZE + LOCALS_SIZE:∞>
21          resources {<M[mindex]>}
22          internal NVSIZE ← ∑({M[addr].size | addr ∈ mindex ∧ M[addr].assigned})
23          ∀ register ∈ <r^6:15> register → <<<M[mindex]>>>
24      end data transfer
25  end callee prologue
26  callee epilogue
27      data transfer (asymmetric)
28          resources {r^0:1}
29          class reg <<<r^0>>
30          map RVAL^1 → RVAL^1.type ⊥ {
31              byte, word, longword, float, double: <reg>,
32              struct: <↑(reg)>
33          }
34  end callee epilogue
```

Figure 6: A Complete Simple Example

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 call 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 call prologue section. In this way, describing the change of view makes it unnecessary to restate where the procedure arguments are located when the view changes.

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.

A final note about this description. Two of the **data transfer** statements (for passing arguments and return values) are tagged with the keyword **(asymmetric)** while the third is not. 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 the third data transfer statement, as well as for all of the view changes, the lack of the **(asymmetric)** keyword indicates that a symmetric action takes place in the symmetric agent. Without the concept of symmetry, the description in Figure 6 would be considerably more involved.

3.4 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 6 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 10) now includes the integer and floating-point registers. Each resource set is ordered to indicate that the resources within an ordering must be assigned in order. This prevents the subsequent placement operator from using element *n* after element *n* + 1 has been assigned. Second we 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 (Line 24) 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, the floating-point registers are used. For all other types, 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 32 bytes of the stack, which correspond to the argument registers, must be reserved so the callee can save the register arguments if necessary. This is specified on Line 11 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 Line 19 to Line 22, using the existential quantifier (∃) which is simply a conditional expression. These restrictions complete the calling convention for the MIPS. The remaining details are similar to the stack example presented earlier.

3.5 Remaining Details

3.5.1 Variable Length Argument Lists

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

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

Figure 7: The MIPS R3000 Specification.

spend considerable time explaining the ramifications that *varargs* has on the calling sequence [JR]. 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.

3.6 Caller vs. Callee Saving of Register Values

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 convention as caller or callee save, we simply define who is responsible for the save (as shown in Section 3.3.3). In both our examples, there is a set of registers the callee must preserve if it uses of them. However, the responsibility for saving any registers that are not explicitly saved by the callee must be saved by the caller.

4 An Implementation

We are currently developing an implementation that uses CCL within a **C** compiler framework. The tools needed to extract and interpret the extended font information are complete. In addition, we have a working front-end for the language that parses and builds the data structures, such as sets, used in CCL. The back-end of the CCL translator that generates the code used in the **C** compiler represents the work yet to be completed

> *At time of submission, this implementation is incomplete. We anticipate its completion in the following months. At that time, we will be able to expand the details in this section, and verify that the descriptions in this paper are correct.*

5 Conclusion

Applications, such as compilers and debuggers, require an accurate description of the calling convention. The portions of such an application that concerns itself with the calling convention are complicated with details that are difficult to maintain, and are often incorrect. We have developed a model and language, called CCL, for specifying procedure calling conventions that can be used to automatically generate the convention-specific portions of these applications. CCL concisely captures the elements of common calling conventions and provides a natural notation for describing them. Rather than describing the calling convention algorithmically, our language allows the designer to define the rules of the procedure call interface that a procedure must adhere to.

We have developed complete descriptions for five machines: Digital's VAX, the Motorola 88100, the MIPS, the Motorola 68020, and the SPARC. We are developing tools that read the descriptions and produce the relevant compiler code for generating calling sequences in a compiler back end. By automatically generating this portion of the compiler, we make the compiler more retargetable. Additionally, different calling conventions can easily be implemented, thereby facilitating experimentation with different call conventions to increase the efficiency of the procedure call.

6 References

[Abr93] Abrahams, P.W. Typographical Extensions for Programming Languages: Breaking out of the ASCII Straitjacket. *SIGPLAN Notices* 28(2):61–68 February 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.

[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.

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

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

[KR78] Kernighan, B.W. and Ritchie, D.M. *The C Programming Language*. Prentice-Hall, 1978.

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