The Static and Dynamic Semantics of C (Extended Abstract)

James K. Huggins[1] and Wuwei Shen[2]*

[1] Kettering University
Computer Science Program
1700 W. Third Avenue
Flint, MI 48504-4898 USA
jhuggins@kettering.edu
[2] University of Michigan
EECS Department
1301 Beal Avenue
Ann Arbor, MI 48109-2122 USA
wwshen@eecs.umich.edu

Abstract. Montages are a semi-visual formalism for defining the static and dynamic semantics of a programming language using Gurevich's Abstract State Machines (ASMs). We describe an application of Montages to describe the static and dynamic semantics of the C programming language.

1 Introduction

In this paper we present the semantics of the C programming language by Montages. In general the only comprehensive description of a programming language is likely its reference manual, which can be informal and open to misinterpretation. Formal approaches are therefore sought. As an attempt in this field, Gurevich's Abstract State Machines (or ASMs) have been successfully used to model the dynamic semantics of programming languages such as Prolog [6], Occam [3, 4], C [12], C++ [22], Java [7, 23] and Oberon [19] (see [5, 14] for many other examples). But how to represent the static semantics of a programming language remains a problem when we want to give the complete semantics for a programming language.

Montages [18] provide a way to describe the static and dynamic semantics of a programming language. A language specification (*i.e.*, the description of its syntactical and semantical aspects) is given as a collection of Montages, each of which is associated with a syntax rule. The semantics of such a collection is given by an ASM. Such an ASM is composed of two parts, which give the static and dynamic semantics of the language, respectively.

In this paper we report on an application of the Montages technique to provide static and dynamic semantics for C. As a continuation of [12], we use

* Partially supported by NSF grant CCR 95-04375.

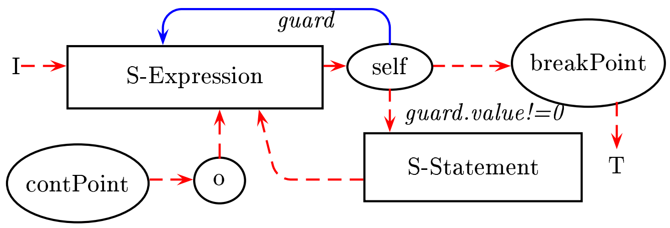
the ANSI standard for C as described in [17] as our definition of C. In the following sections, we present representative Montages from the full semantics, showing the capabilities of the Montage technique. The full semantics may be found in [16].

ASMs have been introduced and defined in many places, most recently in [9, 10]. Montages were recently introduced in [18] and new features were introduced in [2]. We assume basic familiarity with both ASMs and Montages in the following sections.

2 While Statements

We choose the `while` statement in the following to show how we use Montages to represent the semantics of statements in C. The Montage is shown in Figure 1; here we give an explanation of key points of the Montage.

Whilestatement ::= "while" "(" Expression ")" Statement



self. lastCtrlStm := EnclosingCtrlStm
EnclosingCtrlStm := self
–
EnclosingCtrlStm := self.lastCtrlStm

condition *IsArithmeticOrPointerType(self.S-Expression.Terminal)=true*

Fig. 1. Montage for `while` Statements.

In a `while` statement, the substatements in it are executed repeatedly so long as the value of the guard expression remains unequal to 0. This behavior is represented by the two control flow arrows emerging from the node labeled "self" in the static portion of the Montage; the arrow labeled "$guard.value! = 0$" is followed when that guard expression is true (*i.e.*, when the loop should continue), while the unlabeled (default) arrow is followed when the guard is false.

The guard expression in the `while` statement must be of arithmetic or pointer type. This restriction is given in the condition part of its Montage; the function $IsArithmeticOrPointerType$ has a detailed definition which we omit for brevity.

The semantics of the `while` statement becomes complicated when there is a jump statement within it. For example, a `break` statement terminates execution of the smallest enclosing loop or `switch` statement; a `continue` statement causes control to pass to the loop-continuation portion of the smallest enclosing loop statement. In the static part of the Montage, we use two ovals named "contPoint" and "breakPoint" to denote the two targets for a `continue` statement and a `break` statement respectively. Function $EnclosingCtrlStm : Node$ and $lastCtrlStm : Node \to Node$ are used to represent the smallest enclosing loop statement; we set (and reset) those functions as we process the loop's substatements, so that the Montages for those substatements will know where to direct control flow in those situations.

3 Assignment Expressions

We consider here the simplest form of the assignment statement in C, shown in Figure 2; again, we give an explanation of key points of the Montage.

The value of an assignment expression is the value stored in the left operand after the assignment has taken place. This is given in the dynamic part of the following Montage. The order in which the two operands are evaluated is ambiguous (*i.e.* under-specified by the definition of C); a function $leftfirst : Boolean$ is used to denote whether the left subexpression should be evaluated first or not.

The condition portion of the Montage makes use of a function $CompareType$ which ensures that both operands are of types which are compatible for assignment (*e.g.*, both are arithmetic values). The definition is lengthy but straightforward; we omit it for brevity.

A special case in handling assignment statements occurs when the right-hand expression in the statement represents the name of an array rather than a variable name; in such situations, we need a different value to be computed by the subexpressions in order to complete the operation. The function $onlyArrayNameVisited : Node \to Boolean$ is used to denote whether this case has occurred; the value of this function is used by other Montages while evaluating the subexpression in order to generate the correct value.

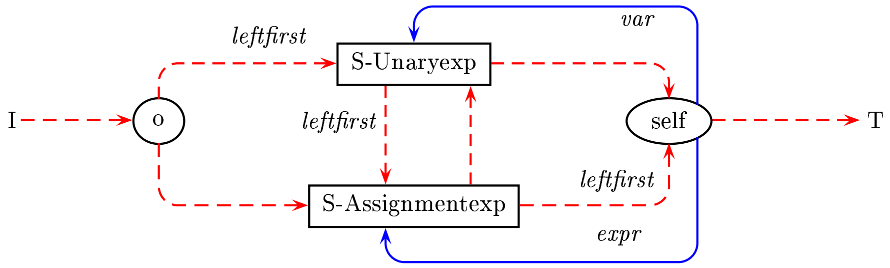
The dynamic portion of the Montage assigns the expression value to the variable. To model the low level implementation, we use the function $AssignOp$ to implement the assignment byte by byte. Again we omit the definition here. The difference between this function and the function Apply, which is defined in the later section, is that the latter takes two constant value and one arithmetic operator and computes the expression value during the static analysis.

4 Additive Expressions

There are many mathematical expressions in C involving binary operators ("*", "+", "−", *etc.*) whose behaviors are similar. (We treat the bit-wise operators

Assignment ::= Unaryexp Assignop Assignmentexp

Assignop = ”=” |”*=” |”/=” |”%=” |”+=” |”-=” |”<<=” |”>>=” |”&=” |”↑=” |”—=”



staticType := S-Unaryexp.Terminal.staticType
if ((Primarycase1(S-Assignmentexp)=true) and (S-Assignmentexp.Terminal.staticType.arrayType=true)) then
S-Assignmentexp.onlyArrayNameVisted := true
else
S-Assignmentexp.onlyArrayNameVisted := false
endif

condition *(CompareType(S-Unaryexp.Terminal,S-Assignmentexp.Terminal,S-Assignop.Name)=true) and (S-Unaryexp.Terminal.lvalue=true)*

value := AssignOp(var, expr, S-Assignop.Name)

Fig. 2. Montage for assignment expressions.

(e.g.,|,&) as ordinary mathematical operators.) The Montage for addition expressions is shown in Figure 3; again, we comment on relevant features of the Montage.

First, we use the function *Apply* to compute a constant value during the static analysis. One of the reasons for doing this is that the value for an expression is required in several contexts: after `case` etc. The difference between this function and the function *AssignOp*, which is defined in the previous section, is that this one takes two constant values and one arithmetic operator and computes the expression value during the static analysis.

To evaluate an addition expression, one evaluates both subexpressions (in some order) and combines the results appropriately. The control-flow for this Montage is thus similar to that for the assignment operator previously considered.

The combining operation is usually conventional addition; however, C overloads the addition operator by allowing addition of an integer i and a pointer p, with the result being a pointer which is i units forward in memory from p. The dynamic rules for this Montage implement this overloading.

For one variable with a type T, the function *staticType* associated with this variable is set to the type T; however, for a variable with a pointer pointing to a type T, the function *staticType* associated with this variable is set to a node, representing a pointer type, whose function *staticType* is set to the type T. Therefore we use the function *staticType* twice to get a pointer variable's type.

Determining the static type of this expression presents a couple of complications. One is the operator overloading just discussed; most of the rules in the static section distinguish between various integer/pointer operand combinations. The other complication comes with arithmetic conversions between arithmetic types; the function $ConvertName : Node \times Node \to Node$ encodes the complicated rules of [17] for this situation (again, we omit the details).

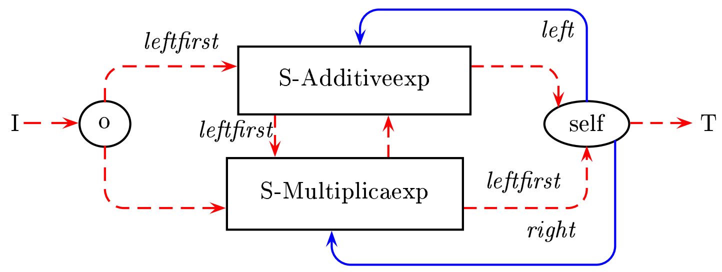
5 Function Definition

We consider the Montage for user-defined function definitions (as opposed to ASM functions) in Figure 4; we highlight relevant portions of the Montage below.

As one might expect, function definitions present a great deal of static information which must be analyzed; consequently, the static portion of the Montage is large. Here we discuss various portions of the static analysis.

User-defined functions (as opposed to ASM functions) are block-structured in C; the variables which are declared in a C-function can only be referenced within that function. Thus, when a variable is referenced, the corresponding declaration in the smallest enclosing block structure is found. The function $EnclosingScope : Node$ is used to denote the current block structure; the function $lastScope : Node \to Node$ is used to denote the outer block structure containing the current block structure. They are assigned before and after a C-function is called, thus

Addexp ::= Additiveexp AddOp Multiplicaexp
AddOp = "+"



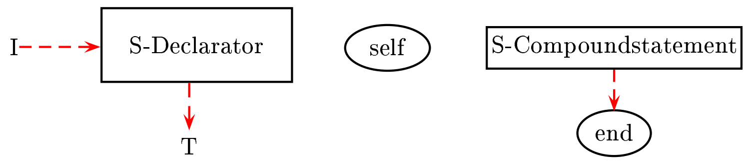
if(((S-Additiveexp.Terminal.staticType.pointType=true) or
(S-Additiveexp.Terminal.staticType.arrayType=true)) and
(S-Multiplicaexp.Terminal.staticType.Name="int"))then
staticType:=S-Additiveexp.Terminal.staticType
elseif(((S-Multiplicaexp.Terminal.staticType.pointType=true) or
(S-Multiplicaexp.Terminal.staticType.arrayType=true)) and
(S-Additiveexp.Terminal.staticType.Name="int")) then
staticType:=S-Multiplicaexp.Terminal.staticType
else
staticType := ConvertName(S-Additiveexp.Terminal,S-Multiplicaexp.Terminal)
endif
constValue := Apply(S-Additiveexp.Terminal.constValue,
S-Multiplicaexp.Terminal.constValue,S-AddOp.Name)

condition *CompareArithType(S-Additiveexp.Terminal,S-Multiplicaexp.Terminal,*
S-AddOp.Name)=true

if((left.staticType.arrayType=true) or
(left.staticType.pointType=true)) then
*value := left.value + (right.value * left.staticType.staticType.typelength)*
elseif((right.staticType.arrayType=true) or (right.staticType.pointType=true))then
*value := right.value + (left.value * right.staticType.staticType.typelength)*
else
value := left.value + right.value
endif

Fig. 3. Montage for addition.

Funcdefcase2 ::= Declspecifier Declarator Compoundstatement




```
CurFundef := true
CurFunc := self
self.lastScope := EnclosingScope
EnclosingScope := self
FuncDef := true
–
CurFundef := false
FuncDef := undef – used to distinguish fun def and decl
CurFunc := undef
EnclosingScope := self.lastScope
do forall g in Gotostatement
   if Within(g, self)=true then
    g.gotoTarget := RecLabelLookup(g.S-Identifier.Name,g.scope)
   endif
enddo
let d=LookUpFuncDcl(S-Declarator.Initial.Name,self.lastScope) in
    if(S-Declarator.S-Parametertypelist≠undef)then
         d.AddrToFunc := S-Declarator.S-Parametertypelist.Initial
         S-Declarator.S-Parametertypelist.nextStatement:=
                                 S-Compoundstatement.Initial
    else
         d.AddrToFunc := S-Compoundstatement.Initial
    endif
    d.funvarType := true
endlet
```

condition *do forall g in GotoStatement*

```
    if Within(g, self)=true then
         RecLabelLookup(g.S-Identifier.Name,g.scope)≠undef endif
enddo
```

```
@end:
StackTop := StackTop - 1
CT := RecCaller(StackTop)
```

Fig. 4. Montage for function declarations.

maintaining a chain of enclosing scopes while permitting other Montages to immediately know their enclosing scope.

C permits the infamous `goto` statement within a function block. While processing that block, we build a function $RecLabelLookup : String \times Node \rightarrow Node$ to indicate the node within the specified scope labeled with the specified string. At the point that static processing returns to this node, we will have seen all targets for `goto` statements; we thus can patch those `goto` statements with the locations of their targets. *g.scope* denotes the scope where we look for the target for `goto` statements. The condition part of the Montage ensures that all `goto` statements have defined targets.

In C, user-defined functions are stored in memory; the names of functions are treated as variable names, just like other variables. The value that is stored in a C-function variable is an implementation-dependent address which is used to transfer control to that function. In order to create an executable Montage for C, we will have to give meaning to this implementation-dependent value; we choose to store in this variable the Montage node corresponding to the declaration for this function. We use an additional function $AddrToFunc : Node \rightarrow Node$ to map the node where a function is declared to its initial definition node. In the static analysis, we set $AddrToFunc$ to point to the first executable node for this function: either the list of parameters to be initialized (in the function declarator) if parameters exist, or the first statement of the function block otherwise.

At the same time, C functions may have several active incarnations at a given time during their execution. Thus we must have some means for storing multiple values of some ASM functions (*e.g.* "*value*") for a given token. We use a universe *Stack* comprising the positive integers for this purpose; a dynamic distinguished element *StackTop:Integer* is used to indicate the current top of the stack.

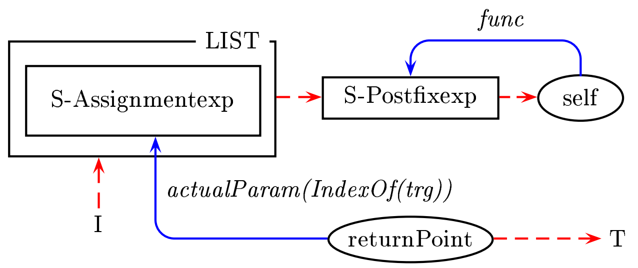
The dynamic semantics of the Montage decrements the value of $StackTop$ because the C-function execution has finished. In order to make control transfer to the corresponding token after a C-function has been executed, we define a new function $RecCaller : Stack \rightarrow Node$ to store the node that calls this C-function at a given recursion level. We set the value of $RecCaller$ when a C-function is called, which is shown in the Montage for C-function calls. Then the dynamic semantics transfers control to the node following the function call by using the function $RecCaller$. This same mechanism is also used by our Montage for `return` statements to handle function returns.

6 Function Call

The Montage for function calls is shown in Figure 5; again, we highlight the relevant features.

In C, a function call is an expression whose type is given by the return type of the function. An expression of type "function returning T" is usually converted to "pointer to function returning T". So we need to distinguish between the two cases when computing the type of a function.

Postfunpar ::= Postfixexp "(" Assignmentexp {"," Assignmentexp } ")"



if((S-Postfixexp.Terminal.staticType.pointType=true) and
(S-Postfixexp.Terminal.staticType.staticType.funvarType=true))then
returnPoint.staticType:=S-Postfixexp.Terminal.staticType.staticType.staticType
else
returnPoint.staticType:=S-Postfixexp.Terminal.staticType.staticType
endif

CT := MemoryValue(func.value).AddrToFunc
StackTop := StackTop + 1
RecCaller(StackTop+1) := returnPoint

Fig. 5. Montage for function calls.

We showed in the previous section how the $AddrToFunc$ function is used to establish the correspondence between addresses where (user-defined) functions are stored and the first node in their Montage representations. To perform a function call, we use the $AddrToFunc$ function to find the corresponding next node, increment the value of the stack, set the return node in the $RecCaller$ function, and perform the branch. (Note that the arguments to the function call will have already been evaluated; those values will be copied into the callee when the callee's parameter nodes are visited.)

7 Discussion

This work is a continuation of the original ASM semantics for C [12], and draws much of its inspiration from the many applications of ASMs to provide programming language semantics ([5, 14]). The previous work, pre-dating the invention of Montages, focused solely on the dynamic semantics of C, assuming that static analysis had been previously performed and was available through various static ASM functions. (In fact, it was the omission of explicit derivation of these static functions, along with the observation that ASMs could be used to provide definitions of these as well, which led to the development of Montages [2].)

This work shows that in fact the full static and dynamic semantics of C can be given using ASMs. This work affirms the basic correctness of the original work; in designing and testing the Montages for C, no errors were found in the original specification. In most places, the dynamic rules used in [12] appear unaltered in the dynamic portion of the Montages (up to renaming).

At the same time a tool called Gem-Mex [1], which supports Montages, has been used to translate our Montages into an executable form for simulation and testing. Since the semantics of Montages are given in terms of ASMs [18], Gem-Mex can translate the Montages into ASMs, which are then executed by an ASM interpreter. This gives a truly executable semantics for C; one can observe directly the particular effect of a given language construct by coding a C program using that construct and observing its execution. Thus, traditional testing and debugging techniques can be used to ensure the correctness of a given program, or of the semantics themselves. The tool itself is raw in some places, and occasionally forces us into certain modes of specifying semantics which are less natural than others; still, the benefit of executable testing more than makes up for the occasional awkwardness.

Our first set of Montages for C [15] used an older version of the Montage tool; our current work [16] uses the latest version of the tool. The most significant difference (for us) between the two versions of the two is the inclusion of conditional control flow arrows in the control flow diagram of each Montage. A significant portion of the original C dynamic semantics [12] was devoted to control flow issues; with control flow arrows in Montages, most of these rules disappear. This provides a separation between control flow and data value generation, which provides a cleaner look to the semantics.

Montages have been used to provide semantics for other programming languages, notably Oberon [19], the first language to be given semantics using Montages, and Java [23]. Oberon and Java are both imperative languages similar in style to C, but with different features. Of particular interest is the fact that C is much more low-level than either Oberon or Java, leading to additional complications in the semantics. Additionally, C admits certain ambiguity in the evaluation order of expressions which is not present in Oberon or Java. Our work shows that the Montages technique is capable of expressing the semantics languages with lower-level details than Oberon or Java.

Specifying semantics of programming languages is a popular formal methods topic; practitioners know there are innumerable formal methods proposed for this purpose. The C programming language has been a popular topic for treatment by various techniques, in part due to its ubiquituous nature in modern software systems. A recent Ph.D. thesis by Norrish [20] contains a comprehensive summary of recent semantic treatments of C. Most of the works cited differ in their desired end point (*e.g.*, mechanical theorem proving, refinement formalisms) and thus tailor their semantics to more easily achieve that end. In particular, Norrish's work, which expresses the semantics of C in HOL, has the explicit goal of "demonstrating practical utility" for his HOL semantics by proving various properties regarding programs.

Our work does not explicitly treat the problem of program correctness, instead focusing on providing a semantics which is both correct and readable by people rather than one designed for further automated processing by theorem provers or other devices. We do not deny the usefulness of being able to use such semantics for validation and verification. Indeed, one of the benefits of having an executable semantics such as ASMs/Montages is the ability to test the semantics on actual programs and observe the correct (or incorrect) behavior of the system, as discussed above. Other ASM works [8, 21, 24] have shown the utility of interfacing ASM semantics with automated correctness tools. But often, in order to produce output suitable for automated verification, one sacrifices expressiveness or readability. We have chosen to focus on the latter.

Norrish (among others) rightly points out certain weaknesses in our semantics, most notably in our handling of the evaluation of expressions. The true semantics of C permits expressions to be evaluated in an interleaving fashion. For example, when evaluating an expression involving an operator such as "$(a+b)$", the evaluation of the subexpressions a and b can occur in an interleaved fashion (*e.g.* evaluate a for a few steps, then b for a few steps, then a again, and so on). Our semantics forces one subexpression to evaluate completely before evaluation of the other can begin; since expression evaluation can generate side effects, our semantics can miss certain possible results.

One solution to this problem involves using a multi-agent ASM [9] to execute the dynamic portion of the semantics. When reaching an expression which permits an interleaving semantics, the multi-agent ASM would create two child agents which would simultaneously traverse the two subexpressions. Once the two child agents finish, the parent agent would kill the child agents and continue on as before. Of course, child agents could create child agents of their own and so forth. Currently, neither Montages nor Gem-Mex support multi-agent ASMs.

With respect to multi-agent ASMs and language semantics, Gurevich [11] has observed that multiple agents may provide a useful semantic notion for expressing the semantics of subprogram (*i.e.*, procedure and function) calls in imperative languages. Traditionally, the semantics of subprogram calls are explained in terms of a stack of values, corresponding to the traditional implementation techniques for such languages. Alternatively, one could view the execution of a program in terms of a set of agents, each of which is responsible for the execution of a given subprogram. When a subprogram call is reached, the agent currently executing would create a new agent, give that agent the parameters needed to execute the subprogram, and suspend its own behavior until completion of the child agent. (A similar technique was used in [13] to give semantics to recursive ASMs.)

In a sense, this solution is equivalent to previous solutions as we have essentially a stack of agents rather than stacks of values. On the other hand, the traditional stack-based view of subprogram calls ties one to a particular implementation technique; an agent-based view would admit alternative implementations.

Acknowledgements. An early version of this work appeared as [15], and was announced at the FM'99 World Congress on Formal Methods; the current version appears as [16]. Yuri Gurevich suggested this project to us and has supported its development. Matthias Anlauff and Philipp Kutter provided frequent assistance with the Montages tool. Egon Börger, Chuck Wallace, and several anonymous referees made comments on various drafts of this work. We thank all of them for their contributions.

References

1. M. Anlauff, P. Kutter, and A. Pierantonio, "Formal Aspects of and Development Environments for Montages." 2nd International Workshop on the Theory and Practice of Algebraic Specifications, Amsterdam 1997.
2. M. Anlauff, P. Kutter, and A. Pierantonio, "Enhanced Control Flow Graphs in Montages." Proceedings of Perspectives of System Informatics (PSI'99), vol. 1755, pp. 40-53, Springer LNCS, 1999.
3. E. Börger and I. Durdanovic, "Correctness of compiling Occam to Transputer code." Computer Journal, 39(1):52-92, 1996.
4. E. Börger, I. Durdanovic, and D.Rosenzweig, "Occam: Specification and Compiler Correctness. Part I: Simple Mathematical Interpreters." In U. Montanari and E.R. Olderog, editors, Proc. PROCOMET'94 (IFIP Working Conference on Programming Concepts, Methods and Calculi), pages 489-508. North-Holland, 1994.
5. E. Börger and J. Huggins, "Abstract State Machines 1988-1998: Commented ASM Bibliography." Formal Specification Column (H. Ehrig, ed.), EATCS Bulletin 64, February 1998, 105-127.
6. E. Börger and D. Rosenzweig, "A Mathematical Definition of Full Prolog." In Science of Computer Programming, volume 24, pages 249-286. North-Holland, 1994.

7. E. Börger and W. Schulte, "Programmer Friendly Modular Definition of the Semantics of Java." In J. Alves-Foss, ed., *Formal Syntax and Semantics of Java.* Springer LNCS, 1998.
8. G. Del Castillo and K. Winter, "Model Checking Support for the ASM High-Level Language". Technical Report TR-RI-99-209, Universitat-GH Paderborn, June 1999.
9. Y. Gurevich, "Evolving Algebras 1993: Lipari Guide." In E. Börger, editor, Specification and Validation Methods, pages 9-36. Oxford University Press, 1995.
10. Y. Gurevich, "May 1997 Draft of the ASM Guide", University of Michigan EECS Department Technical Report CSE-TR-336-97.
11. Y. Gurevich. Personal communication, September 1999.
12. Y. Gurevich and J.Huggins. "The Semantics of the C Programming Language." In E. Börger, H. Kleine Büning, G. Jäger, S.Martini, and M.M. Richter, editors, Computer Science Logic, volume 702 of LNCS, pages 274-309. Springer, 1993.
13. Y. Gurevich and M. Spielmann, "Recursive Abstract State Machines." Journal of Universal Computer Science, 3(4):233-246, 1997.
14. J. Huggins, ed., Abstract State Machines Home Page, `http://www.eecs.umich.edu/gasm/`.
15. J. Huggins and W. Shen, "The Static and Dynamic Semantics of C: Preliminary Version." Technical Report CPSC-1999-1, Computer Science Program, Kettering University, February 1999.
16. J. Huggins and W. Shen, "The Static and Dynamic Semantics of C." Technical Report, Computer Science Program, Kettering University, March 2000, to appear.
17. B. Kernighan and D. Ritchie, *The C programming Language*, 2nd ed.. Prentice Hall, 1988.
18. P. Kutter and A. Pierantonio, "Montages: Specifications of Realistic Programming Languages." Journal of Universal Computer Science, 3(5):416-442, 1997.
19. P. Kutter and A. Pierantonio, "The Formal Specification of Oberon." Journal of Universal Computer Science, 3(5):443-503, 1997.
20. M. Norrish, "C formalized in HOL." Ph.D. Thesis, University of Cambridge, 1998.
21. G. Schellhorn and W. Ahrendt, "Reasoning about Abstract State Machines: The WAM Case Study", Journal of Universal Computer Science, vol. 3, no. 4 (1997), 377–413.
22. C. Wallace, "The Semantics of the C++ Programming Language." In E. Börger, editor, Specification and Validation Methods, pages 131-164. Oxford University Press, 1995.
23. C. Wallace, "The Semantics of the Java Programming Language: Preliminary Version." Technical Report CSE-TR-355-97, EECS Dept., University of Michigan, December 1997.
24. W. Zimmerman and T. Gaul, "On the Construction of Correct Compiler Back-Ends: An ASM Approach", Journal of Universal Computer Science, vol. 3, no. 5 (1997), 504–567.