The Static and Dynamic Semantics of C

James K. Huggins* Wuwei Shen‡

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, which is a new method for giving the semantics of a programming language. 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] and Oberon [19]. 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 semantics 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: the first part defining the static analysis and semantics of the language, the second part defining the dynamic semantics of the language. Each program of the specified language defines an initial state for the ASM, which contains the program parse tree. After we have given the semantics of a programming language $\mathcal{L}$ and a program written in $\mathcal{L}$, the corresponding ASM $M_{\mathcal{L}}$ begins by performing the static analysis for each node of the parse tree. The static analysis decorates the leaves of the parse tree with control and data flow information, building the sequence of tokens that are needed by the dynamic semantics. Next, $M_{\mathcal{L}}$ executes the dynamics semantics using the sequence of tokens generated by the first step. Thus, $M_{\mathcal{L}}$ can be seen as an interpreter for $\mathcal{L}$.

*Kettering University,Computer Science Program, 1700 W. Third Avenue, Flint, MI 48504-4898 USA.
†University of Michigan, EECS Department, 1301 Beal Avenue, Ann Arbor, MI 48109-2122 USA.
‡Partially supported by NSF grant CCR 95-04375.

In this paper we present the static and dynamic semantics of C by using Montages. As a continuation of [12], we also use the ANSI standard for C as described in [17]. We also concentrate on the following four topics respectively: statements, expressions, type and variable declarations, and function invocation and return.

The paper is organized as follows. In section 2, we give a brief introduction to ASMs and Montages. In section 3, 4, 5 and 6, we give the Montages for the above four topics respectively. In section 7 we discuss this work and compare it with other related works.

2 ASMs and Montages

2.1 Introduction to ASMs

In the following we describe an ASM model [9] which is sufficient to represent the semantics of C [17]. (ASMs have many features not presented here; see [9] for details.)

The *signature* of an ASM A is a finite collection of function names, each name having a fixed arity. A *state* of A is a set, the *superuniverse*, together with interpretations of the function names in the signature. These interpretations are called *basic functions* of the state. A basic of function of arity r is an $r-$ary operation on the superuniverse. When $r = 0$, such a basic function is called a *distinguished element*. The superuniverse does not change as A evolves; the basic functions may. The superuniverse contains some distinct elements *true, false* and *undef* which are used to describe relations and partial functions. They are *logical constants*, whose names do not appear in the signature. In addition, we use equality as a logical constant.

A *universe* U is an important concept in ASMs. It is a special type of basic function: a unary relation usually identified with the set $\{x : U(x)\}$. ASMs provide some built-in universes such as the logical constant $Boolean = \{true, false\}$. When we define a function f from a universe U to a universe V, and write $f : U \to V$, we mean that f is a unary operation on the superuniverse such that $f(a) \in V$ for all $a \in U$ and $f(a) = undef$ otherwise. We can extend this notation to notations such as $f : U_1 \times U_2 \to V$ and $f : V$, which means the distinguished element f belongs to V. In addition the expression $f(a)$ can be written in the form $a.f$. For the general case, the expression $f(a_1, \ldots a_n)$ can be written in the form $a_1.f(a_2, \ldots, a_n)$.

There are three kinds of functions in ASMs. A function f is *dynamic* if f can be changed as the ASM evolves. Functions which are not dynamic are called *static*. *External* functions are syntactically static, but have their values determined by an oracle (that is, the outside world).

In principle, a program of A is a finite collection of rules, which are defined inductively in the following:

- Update Rules:

  $$f(\overline{s}) := t$$

  is a rule with *head* f.

  Here $\overline{s}$ is a tuple $(s_1, \ldots, s_r)$ of terms where r is the arity of f and $r \geq 0$. If f is relational, then the term t must be Boolean. To fire such a rule, change the value of f at the value of term $\overline{s}$ to the value of t.

- Conditional Rules: if g is a Boolean term and R_1, R_2 are rules then

  $if \quad g \quad then \quad R_1$

  $else \quad R_2$

  $endif$

  is a rule. To fire this rule at a given state A, examine the guard g. If g's value is true at A, then fire R_1; otherwise, fire R_2.

- Block: If R_1, R_2 are rules then

  $do \quad in-parallel$

  R_1

  R_2

  $enddo$

  is a rule with *components* R_1, R_2. Do-in-parallel rules are called *blocks*. We often omit the words "do in-parallel" when the scope of the block is clear from context.

  Let r_1 and r_2 be update rules of the following forms:

  $$r_1 : \quad f_1(\overline{s_1}) := t_1 \qquad r_2 : \quad f_2(\overline{s_2}) := t_2$$

  r_1 and r_2 are said to be *mutually inconsistent* at a given state A if $f_1 = f_2$, and the values of $\overline{s_1}$ and $\overline{s_2}$ are equal but the values of t_1 and t_2 are not equal. Otherwise they are mutually consistent.

  To fire a block R at a given state A, determine first if the update rules which will be fired in R_1 and R_2 are mutually consistent. If yes, then fire them simultaneously. If not, do nothing; R is inconsistent at A.

- Do-forall Rules: If v is a variable, $g(v)$ is a Boolean term and $R_0(v)$ is a rule, then

  $do \quad forall\ v : \quad g(v)$

  $R_0(v)$

  $enddo$

  is a rule with *head variable* v, *guard* $g(v)$ and *body* R_0. A do-forall rule is similar to the do-in-parallel rule, except that the components are not listed explicitly. Suppose R is the do-forall rule above. At a state A which maps every variable in R to a value, the components of R are the rules $R_0(a)$ where a is any element in the state A satisfying $g(a) = true$. To fire R at A, fire simultaneously all these $R_0(a)$ unless they are mutually inconsistent. In the latter case, do nothing.

2.2 Introduction to Montages

Montages [18] are a semi-visual formalism that allows unified and coherent specification of syntax, static analysis and semantics, and dynamic semantics. Generally speaking, for every syntax rule there is one corresponding Montage. In every Montage there can be four parts. The first three parts define the *static aspects* of the language, which refers to the work which can be done at compile time (such as static analysis), and the fourth part defines the *dynamic semantics* of the language.

2.3 Structure of Montages

Montages are a formalism for the specification of programming languages. The aim of Montages is to document formally the decisions taken during the design process of realistic programming languages. Each Montage describes the properties of its instances, which are the nodes in the *parse tree* that is generated when a program is analyzed. Every Montage consists of four parts: the EBNF production rule, the (local) control and data flow graph, the static semantics, and the dynamic semantics of the construct. Symbols in the right-hand side of the EBNF rule are called (direct) *components* of the Montage, and symbols which can be reachable as components of components are called *indirect components*. In order to access descendants of a given node in the parse tree, we can use statically defined attributes called *selectors*.

In the local control and data flow graph of a Montage, we use the *Montage visual language* (MVL) to represent the control and data flow. There are two lexicons in MVL, *elements* and *edges* (or arrows).

Elements are labeled ovals and boxes. Boxes represent components of a Montage. Ovals represent the dynamic semantics actions associated with the Montage, and are labeled with the name of the action. If there is only one action then the generic label "self" is used. The dynamic semantic action will be executed when the oval "self" is reached. If a user gives a name for an action, then the dynamic semantic action associated with this name (if exists) will be executed when control reaches it.

Edges are used to connect elements in order to denote the control and data flow. There are two kinds of edges: solid and dotted arrows. Solid arrows denote data flow and dotted arrows denote control flow. I (initial) arrows and T (terminal) arrows are two special kinds of control flow arrows, denoting where the control flow initially enters and from where control finally exits the construct respectively.

In the static semantics of a Montage, we can give static semantics actions for the Montage. They are executed during the static analysis. In addition, some syntax rules have some restrictions which can be represented in the condition part of a Montage. More details about this restriction can be found in the following example.

2.4 An Example

Consider the following example for a while statement in a programming language shown in Figure 1. In this Montage, the topmost part is the production rule defining while statements.

The middle part defines data and control flow of the while statements. In particular, in the control and data flow part the unique action of a while-instance is denoted by the *self*-oval, whereas the components are represented with boxes labeled by selectors, e.g. the S-Expression box for the Expression component.

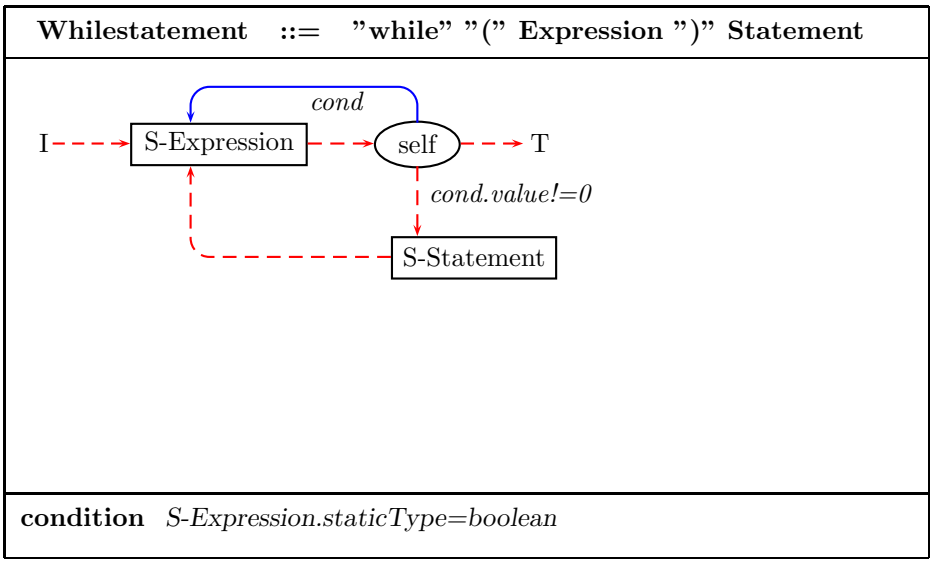


Figure 1: Semantics of while statements for a programming language.

Dotted control arrows link the exit point of their source with the entry point of their target: the exit of point of the Expression component is linked with the self action, which is linked with the entry point of the Statement, and the exit point of the Statement is linked with the entry point of Expression. In general, control follows this link unconditionally if no firing condition is given. However if a firing condition is given, like $cond.value! = 0$ in the above Montage, then control follows the arrow only if the condition is true. In general, the label of a control flow arrow is a boolean predicate, defining the firing condition; if the source of the link is active and the firing condition evaluates to true, control is passed to the target of the link. Otherwise control follows the dotted control arrow without any labeled names.

In the dynamical behavior of a construct, intermediate results are stored in attributes of the nodes. In the above example, we assume that each instance of Expression stores its value in an attribute *value*. Data flow links from one Montage instance to another are used to retrieve such data. A *data flow arrow* defines a data link from its source to its target. If the source is an action of some Montage instance, the link is attached to the instance. Thus the *cond*- labeled data flow arrow in the example links the current instance with the instance of Expression.

The third part of the above Montage contains the restriction for this construct. In the above example, the type for the condition must be *boolean*. In general, Montages have a fourth part which contains dynamic semantic actions to be performed when control reaches the self node.

2.5 Informal Meaning of Montages

2.5.1 Compact Derivation Tree

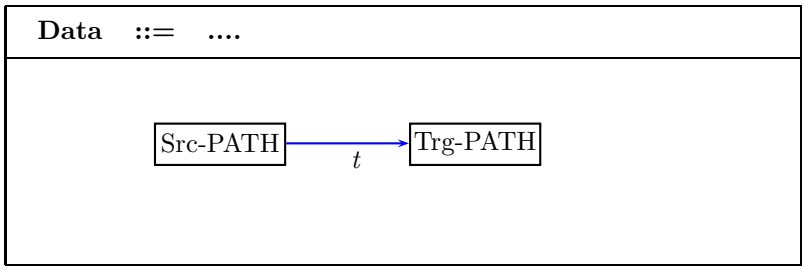
A Montage specification of a language $\mathcal{L}$ defines an ASM $M_{\mathcal{L}}$ that for a given program $\mathcal{P}$ of $\mathcal{L}$ analyzes and defines the control and data flow, checks the static semantics, and then executes the dynamic semantics. For each program $\mathcal{P}$, there is a different initial state $I_{\mathcal{P}}$ of the ASM which encodes the syntax of $\mathcal{P}$. $\mathcal{L}-$programs are strings generated by a context free grammar $G_{\mathcal{L}}$. Each program generated by $G_{\mathcal{L}}$ is represented in $I_{\mathcal{P}}$ as a *compact derivation tree*, which is derived from a parse tree by repeatedly collapsing each node n with a single child c to a single node with the labels of both n and c, and having only the children of c as children, until no such nodes remain. The initial state $I_{\mathcal{P}}$ associated with a program $\mathcal{P}$ is given by *universes* and *functions* representing the *nodes* and *branches* of the compact derivation tree. The functions are selector functions which are used to select nodes in some branch of the compact derivation tree. The initial state $I_{\mathcal{P}}$ contains the compact derivation tree of the program $\mathcal{P}$. Because it is possible for one node to have multiple disjoint labels such as n and c mentioned above, the universes are not disjoint and the selector function is used to select descendants from more than one category of nodes. Here the notation S- is used to distinguish the selector function names from the universe function names.

After obtaining the compact derivation tree, we introduce how to generate the control and data flow information which makes the ASM program executable during the dynamic semantics computation. The control and data flow information is provided as attributes of the tokens after the static analysis. These attributes are illustrated as data arrows and control arrows among the tokens. Montages provide a way to define control and data flow starting from the syntax of the program. When the parse tree for a program $\mathcal{P}$ is generated by a grammar $\mathcal{L}$, each syntax rule such as $n ::= E$ has multiple occurrences in the tree; in particular we can imagine the nonterminal n matching subtrees whose descendants are structured according to the right-hand side E of the syntax rule. The root of such a subtree is a node corresponding to the left-hand side n. Nonterminal and terminal symbols in E represent the direct descendants of that root. Linking together these symbols results in flow arrows among internal nodes of the derivation tree.

2.5.2 Control and Data Functions

After the static analysis phase, the dynamic semantics phase is executed. In the case of sequential languages, exactly one action is activated, then control is passed to the next node along the control links. In the following, we give the formal semantics of data and control arrows.

Assume that we have the following Montage for Data. A data flow arrow consists of two Montages *Src* and *Trg* and a label t which represents a data flow. The data flow connects these two Montages. In the compact derivation tree the arrow is instantiated for each instance of Data with links from the root node of the source Montage *Src* to the root node of the target Montage *Trg*. The semantics for this data flow is that we connect two nodes corresponding to the instances of these two Montages.



$\forall self \in Data$:
self.Src.t = Trg

With control arrows the situation is a little more complicated. If s is an instance of a Montage and s does not include any other Montages, then we have the following equations for functions *Initial* and *Terminal*:

$$s.Initial = s$$
$$s.Terminal = s$$

If s is an instance of a Montage which includes other Montages and there is an edge from I to a Montage denoted by path tgt, then

$$s.Initial = s.tgt.Initial$$

If there is an edge from a Montage denoted by path src to T, then

$$s.Terminal = s.src.Terminal$$

Using these definitions, the structured finite state machine can be flattened. The arrows of the flat finite state machine are given by the following equations defining the relation $ControlArrow$. For each instance n of a Montage N and each edge e associated with N,

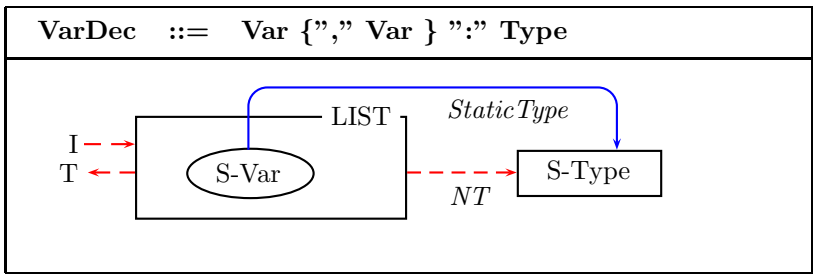
$$ControlArrow(e.src.Terminal, e.tgt.Initial)) = true$$

where src is the path of the source of e and tgt is the path of the target of e.

2.5.3 Lists

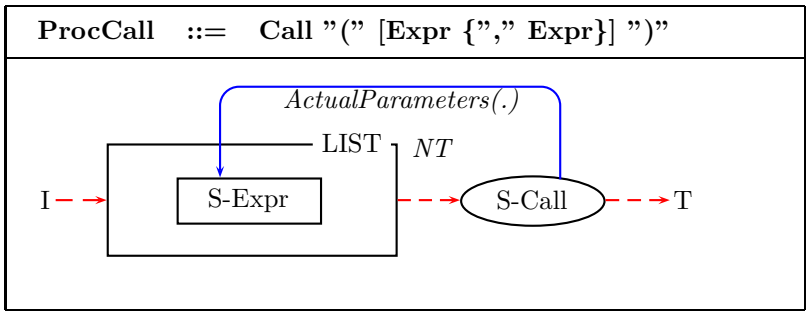
Montages also provide for the processing of lists with which most languages are concerned. If the right hand side of a syntax rule contains a symbol enclosed in { }, a list of descendants is generated. An additional node, called a *list node*, is generated as well. In order to access the elements and needed information about the list, Montages provide an attribute *ListLength* of the list node which is set to the length of the generated list and a binary infix function $\_[\_] : ListNode \times Integer \rightarrow Node$ can be used to retrieve the elements of the list. Moreover, a function $Position : Node \rightarrow Nat$ returns the physical position of an element within a list. The initial leaf of the list node is the initial leaf of the first element in the list, and the terminal leaf is the terminal leaf of the last element. For the control functions defined in the list rule, the corresponding terminal leaf of each list element is connected to the initial leaf of its successor in the list by them.

For data functions related to lists, there are two special cases. One case is an arrow defined from all the elements within a list to an outside node. In this case, we can regard the function as a group of arrows which connect all the elements within the list to the outside node respectively. As an example, the following Montage gives the static semantics for a list of variable declarations in some programming language. The list is graphically represented by a box, which is labeled in its upper right corner with the keyword `LIST`. The data flow arrows *StaticType* specifies a group of data flow arrows, one from each variable to the node *type*, all labeled *StaticType*. In addition, all variable objects are linked sequentially by a control function NT.



The other special case is a function defined from an outside node to the elements within the list. In this case the data function is regarded as a set of arrows which connect the outside node to all the elements within the list respectively by using an infix function. Let us look at another example. The following Montage gives the semantics for a function call in some programming language. The data function $ActualParameters$ in the following Montage defines a binary function $ActualParameters : Node \times Integer \rightarrow Node$ which maps a call node to all the actual parameters of this node. In order to implement the type check between the corresponding parameters of the function call and the function definition, we would need to get the positions of all the parameters in the function call parameter's list. To get an actual parameter position in the list, we can use the projection function which is defined as follows: $ActualParametersPosition : Token \rightarrow Integer$ which gives the position number for every $Expr$ referenced by the function $ActualParameters$ in the list of $Expr$. The semantics of the data function $ActualParameters$ is the following: $\forall x \in$ list of *Expr*: self.S-Call.ActualParameters(x.Position):=x.Terminal and x.ActualParametersPosition:=x.Position;

The projection function name of a data function, like ActualParametersPosition in the following Montage, is generated by the data function name (ActualParameters) followed by Position. The reason that we use the projection function is that the position of an item within a nesting of list boxes may be relative to the source of the arrow which points to it.



2.5.4 Some notation

In order to make it easier to write our Montages, we use the following abbreviation, which can be used to iterate over a list of nodes.

vary_over_ind(i,L) R(i) **endvary**

"L" is a list node, and "i" iterates over the indices of "L". The elements of "L" can be accessed using the notation "L[i]". The above rule is equivalent to the following ASM rule:

$$\begin{array}{ll} do & forall \quad i : 1 \leq i \leq ListLength(L) \\ & \quad R(i) \\ enddo & \end{array}$$

3 Montages for Statements in C

3.1 Some Basic Functions

Before giving the semantics of C, we introduce the following basic functions and notations. The constructs of C can be divided into statements, expressions, and types. In the Montages approach, Nodes are those elements which are chosen to execute the dynamic semantics for the Montages. $CValue$ represents the universe of data values that can be represented in a particular C program. A static function $Value : Nodes \rightarrow CValue$ indicates the value of a node. In addition we define a function $Name : Nodes \rightarrow String$ to indicate the name for the node.

3.2 Selection Statements

Selection statements include the `if` statement and `switch` statement. In the `if` statement, the guard expression must have an arithmetic or pointer type. This restriction is implemented by the function $IsArithmeticOrPointer : Nodes \rightarrow Boolean$. If the guard expression compares unequal to 0, the first substatement is executed. So in Figure 2, the control flow labeled "*guard.value!=0*" is followed when that condition expression is true. Otherwise the substatement following `else` is executed, which is shown in the default control flow. If the substatement

following `else` is omitted, then control will be passed to the next statement directly. Otherwise, in both cases, control will be passed to the next statement following the `if` statements.

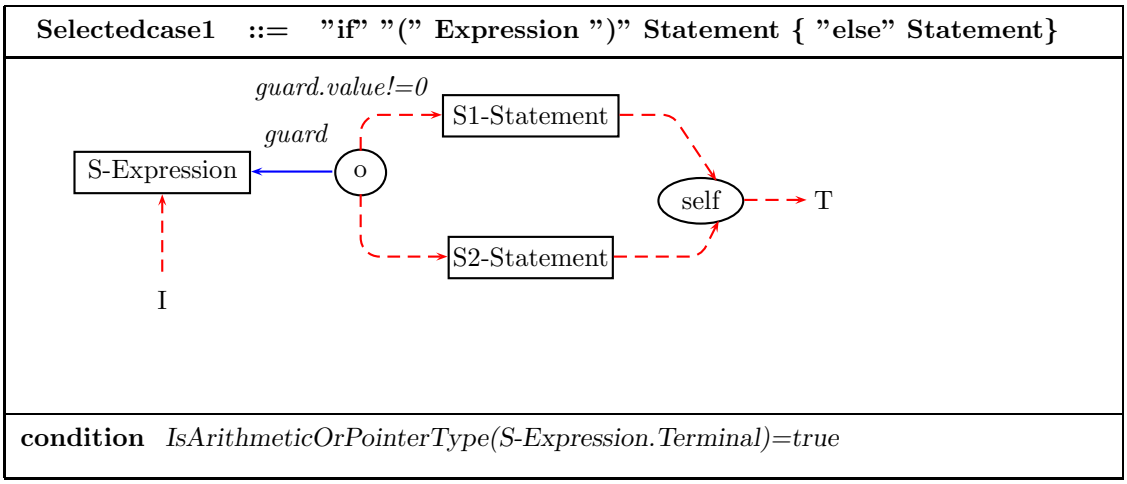


Figure 2: Montage for `if` statements.

The `switch` statement causes control to be transferred to one of several statements depending on the value of an expression, which must have integral type. When the `switch` statement is executed, its expression is computed and compared with each case constant. If one of the case constants is equal to the value of the expression, control passes to the statement of the matched `case` label. If no case constant matches the expression, and if there is a `default` label, control passes to the labeled statement. If no case matches, and if there is no `default`, then none of the substatements of the `switch` statement is executed.

The function $SwitchTable : Nodes \times CValue \rightarrow Nodes$ is used to indicate where control should be directed given a node and value. In the Montage for case expressions shown in Figure 4, we set the function $SwitchTable$ in the static part. Therefore we can direct control flow in the Montage's dynamic part in Figure 3 when a switch statement is met.

The semantics of the `switch` statement becomes complicated when there is a jump statement within it. 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 order to deal with smallest enclosing scope, we define functions $EnclosingCtrlSt, CurSwitchLevel : Nodes$ and they denote the current control level and switch table. Functions $lastCtrlStm, lastSwitchLevel : Nodes \rightarrow Nodes$ are used to denote the previous control level and switch table. These functions are updated before and after the corresponding Montage is visited. In the Montage notation, we can use "-" in the static analysis to denote the before and after actions. All the actions defined before "-" are executed before the corresponding Montage is analyzed and all the actions defined after "-" are executed after the Montage is reduced. If "-" is omitted, then all the actions are executed after the Montage is reduced. In Figure 3, before the `switch` statement is visited we set the

function *EnclosingCtrlSt* and *CurSwitchLevel* to reflect the current level. And after the `switch` statement is visited, we set these functions back by using functions *lastCtrlStm* and *lastSwitchLevel*. Functions $contPoint, breakPoint : Nodes \rightarrow Nodes$ are used to denote the two targets for a `continue` statement and a `break` statement respectively. Because we set (and reset) these functions as we process the loop's substatements, the Montages for those substatements will know where to direct control flow in those situations.

The Montages for the `switch` statement are shown in Figures 3 & 4.

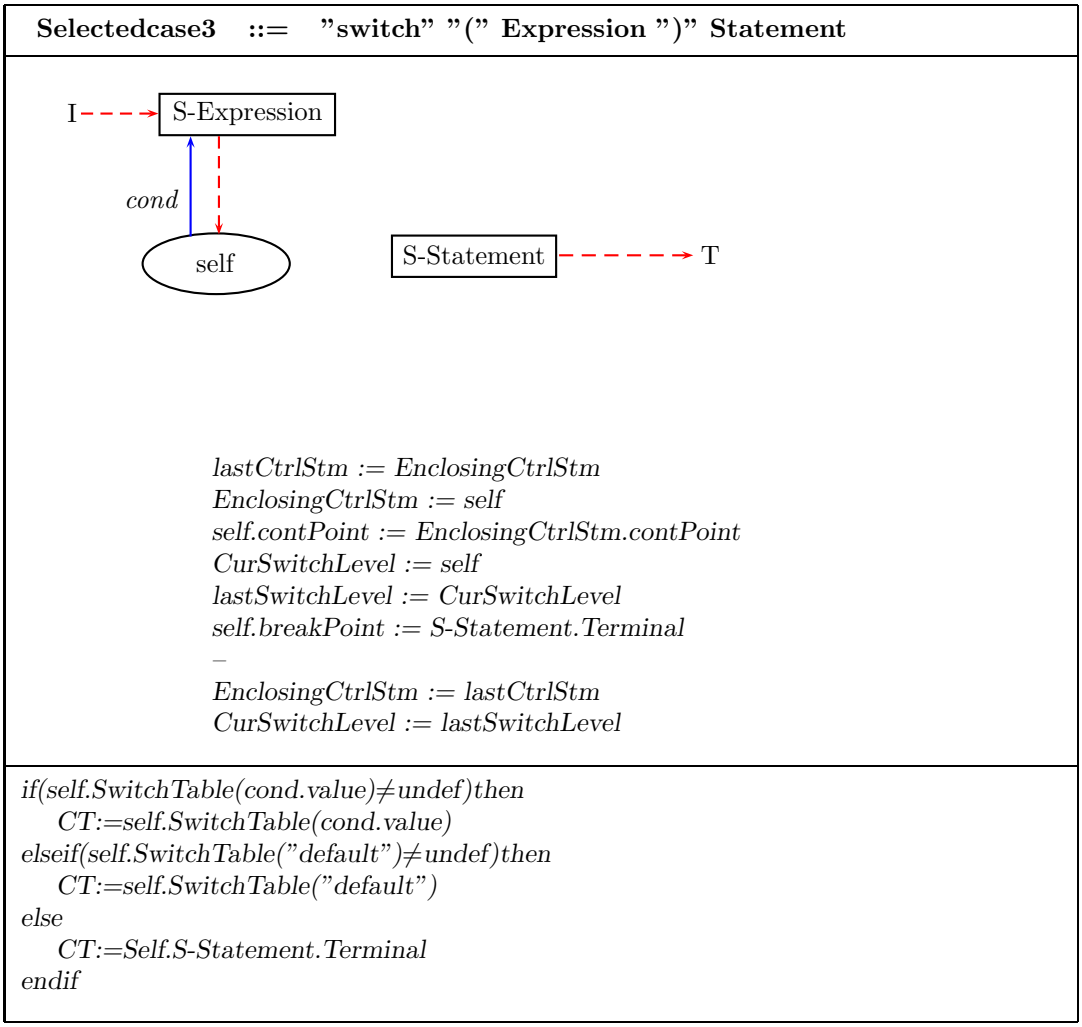


Figure 3: Montage for `switch` statements.

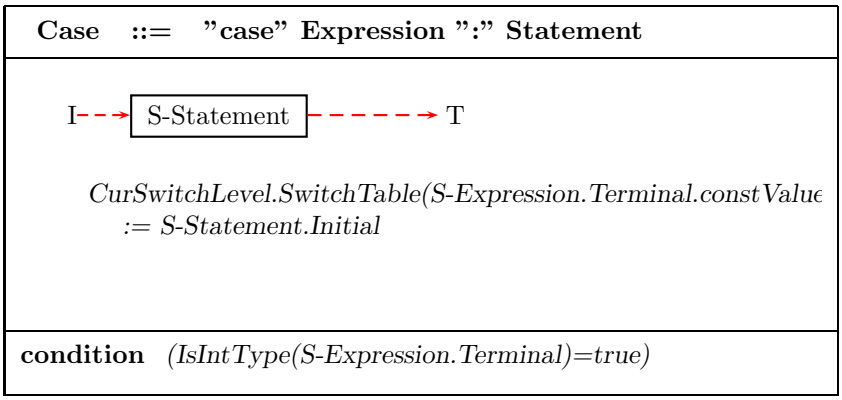


Figure 4: Montage for `case` statements within `switch` statements.

3.3 Iteration Statements

Iteration statements include the `while` statement, `do` statement, and `for` statement. In order to deal with `jump` statements, we set some functions associated with the instances of iteration statements which are *contPoint* and *breakPoint*. They are used for `continue` statements and `break` statements respectively. In addition, in order to implement the smallest enclosing scope, we set the function *EnclosingCtrlStm* to denote the instance of the current smallest enclosing iteration statements and the function $lastCtrlStm : Nodes \rightarrow Nodes$ to denote the previous smallest enclosing iteration statements. They are dealt with in the same way as the selection statement.

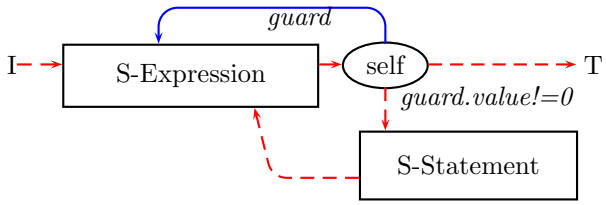
In the `while` statement, the substatement is executed repeatedly so long as the value of the expression remains unequal to 0. This behavior is represented by the two control flow arrows emerging from the node labeled "self" in the static portion of the Montage; the arrow labeled "$guard.value! = 0$" is followed when that guard expression is true (*i.e.*, when the loop should continue), while the other 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 by the function $IsArithmeticOrPointerType$ in the condition part of its Montage. We deal with jump statements in the same way as we do for `switch` statements. The Montage for the `while` statement is shown in the following Figure 5.

In the `do` statement, the substatement is executed repeatedly so long as the value of the expression remains unequal to 0; the expression must have have arithmetic type or pointer type. And the test occurs after each execution of the statement. The Montage for the `do` statement is shown in Figure 6.

In the `for` statement, the first expression is evaluated once, and thus specifies initialization for the loop. The second expression must have arithmetic or pointer type; it is evaluated before each iteration, and if it becomes equal to 0, the `for` statement is terminated. The third expression is evaluated after each iteration, and thus specifies a re-initialization for the loop. The Montage for the `for` statement is shown in Figure 7.

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

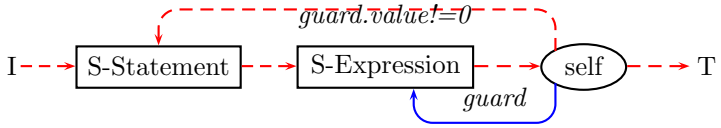


lastCtrlStm := EnclosingCtrlStm
EnclosingCtrlStm := self
self.breakPoint := self.Terminal
self.contPoint := S-Expression.Initial
–
EnclosingCtrlStm := lastCtrlStm

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

Figure 5: Montage for `while` statements.

Dostatement ::= "do" Statement "while" "(" Expression ")" ";"



lastCtrlStm := EnclosingCtrlStm
EnclosingCtrlStm := self
self.breakPoint := self.Terminal
self.contPoint := S-Expression.Initial
–
EnclosingCtrlStm := lastCtrlStm

condition *IsArithmeticOrPointerType(S-Expression)=true*

Figure 6: Montage for `do` statements.

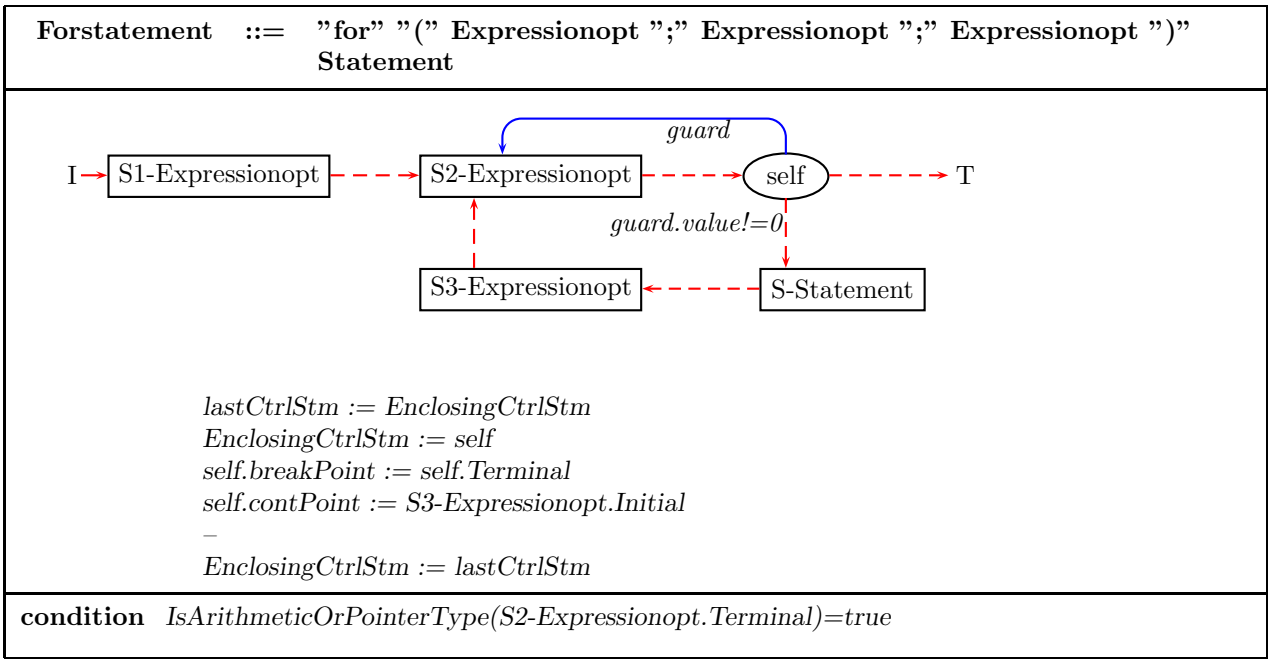


Figure 7: Montage for `for` statements.

3.4 Jump Statement

The `jump` statements include the following four forms: the `goto` statement, the `continue` statement, the `break` statement and the `return` statement.

A `continue` statement may appear only within an iteration statement. It causes control to pass to the loop-continuation portion of the smallest enclosing such statement. In the Montages for the iteration statement, we set the function *contTarget* to denote the node to which the `continue` statement jumps by using the function *contPoint*. Here the function EnclosingCtrlStm is used to denote the smallest enclosing iteration statement. In the dynamic semantics part of the Montage we direct control to the corresponding node by $CT := continueTarget$. The Montage for `continue` statements is shown in Figure 8.

A `break` statement may appear in an iteration statement or a `switch` statement, and terminates execution of the smallest enclosing such statement; control passes to the statement following the terminated statement. In the Montages for the iteration statement, we set the function *breakTarget* to denote the node to which the `break` statement jumps. Like the Montage for the `continue` statement, in the dynamic semantics part, we direct control to the corresponding node by $CT := breakTarget$. In addition, we set the guard value for the current smallest enclosing iteration statement to 0 so that we can exit from the current iteration statement. The Montage for `break` statements is shown in Figure 9.

In the `goto` statement, the identifier must be a label located in the current function. Control transfers to the labeled statement. We direct control to the appropriate node by using function

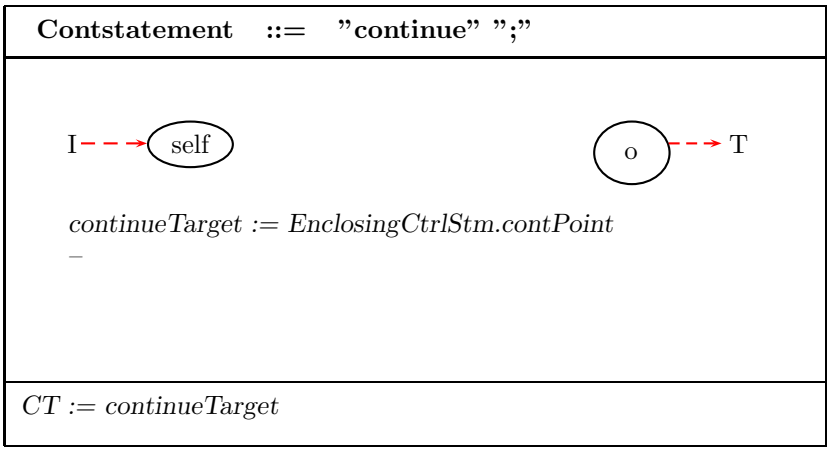


Figure 8: Montage for `continue` statements.

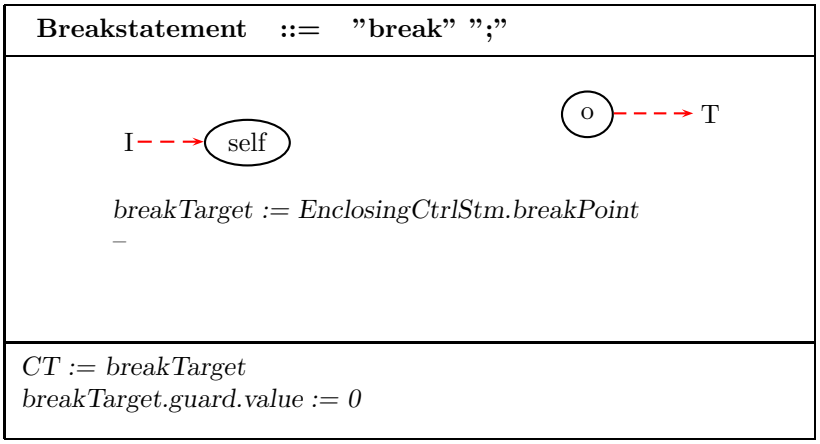


Figure 9: Montage for `break` statements.

$gotoTarget : Nodes \rightarrow Nodes$. The Montage for the `goto` statement is shown in Figure 10.

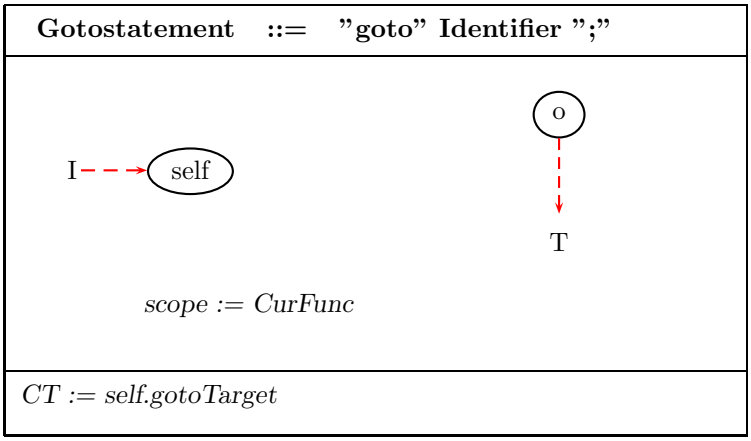


Figure 10: Montage for `goto` statements.

4 Expressions

4.1 Some New Functions and a Macro

The universe *address* comprises the positive integers, which is used to model memory. In order to access the memory address corresponding to a variable, we define a new partial function $var2memory : Nodes \rightarrow address$ which is used to set up the relation between a variable and its location in memory.

A static function $typelength : Nodes \rightarrow Integer$ indicates how many bytes are used by a particular value type in memory. A dynamic function $MemoryByteValue : address \rightarrow bytes$ indicates the values stored in memory at a given address. Since most values of interest are larger than one byte, we need a means for storing members of *CValue* as individual bytes.

A static partial function *ByteToResult:* $Nodes \times byte^n \rightarrow CValue$ converts the memory representation of a value of the specified basic type into its corresponding value in the *CValue* universe. Here n is the maximum number of bytes used by the memory representation of any particular basic type (and is implementation-dependent). For types whose memory representations are less than n bytes in length, we ignore any unused parameters.

A static partial function $ResultToByte : CValue \times Integer \times Nodes \rightarrow byte$ yields the specified byte of the memory representation of the specified value from the specified universe. This function is the inverse of *ByteToResult*.

In order to get the value of the specified type being stored in memory starting at the indicated address easily, we define an abbreviation *GetMemoryValue:address*$\times$ *Nodes*$\rightarrow$ *CValue. GetMemoryValue(addr, type)* abbreviates *ByteToResult(type,MemoryByteValue(addr), MemoryByteValue (addr+1),. . . , MemoryByteValue (addr+length(type)-1)).*

Because we use the *byte* to model memory, the rules for assignment to memory are a little complicated because a given assignment may require an arbitrary large number of updates to the $MemoryByteValue$ function. We use *do-forall* rules which perform a those arbitrary large number of updates in a systematic fashion. The number of updates is decided by the variable's type length, which is given by the function $staticType : Node \rightarrow Integer$. The transition rule for copying to memory is shown in the following:

AssignMemory(var,value)

do forall i: $0 \leq i \leq length(var.StaticType) - 1$
MemoryByteValue(var.var2memory+i):=
ResultToByte(value, i, var.staticType);
enddo

4.2 Comma Operator

A comma expression has the following form,

comma-expression $\rightarrow$ *expr1, expr2*

where $expr_1$ and $expr_2$ are expressions.

In a comma expression, the type and value of the result are the type and value of the right operand. The Montage for comma expressions is shown in Figure 11.

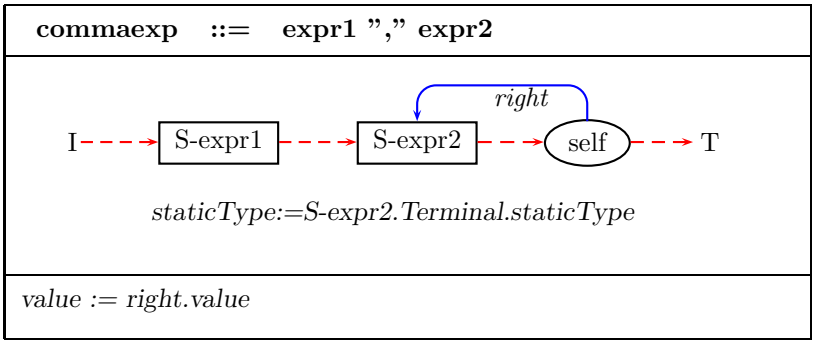


Figure 11: Montage for comma expressions.

4.3 Assignment Expression

A simple assignment has the following form:

assignment-expression → expr1 = expr2

where $expr_1$ and $expr_2$ are expressions. To evaluate a simple assignment expression, copy the value of $epxr_2$ into the memory location given by $expr_1$, returning that value as the value of the whole assignment expression.

In assignment expressions, the left operand must be an *lvalue*. It must not be an array, and it must not have an incomplete type, or be a function. The type of the result is that of its left operand. And the value is the value stored in the left operand after the assignment has taken place. This is implemented in the function $CompareType$, which is given in the Appendix.

In addition, the evaluation order of subexpressions is not decided in most C expressions. In order to implement this, we define a new function *leftfirst: Boolean*. If *leftfirst* is true then the left subexpression is evaluated first; otherwise the right subexpression is evaluated first.

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 $onlyArrayNameVisited : Node \rightarrow Boolean$ 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. More details about this can be found in primary expressions. The Montage for assignment expressions is shown in Figure 12.

Within C, there are other assignment operators ("+=", "*=", etc.) which perform a mathematic operation on the value of *expr2* and the value stored in the memory location given by *expr1*. The result is copied into the memory location given by *expr1*. Most these assignment operators have Montages like that shown in Figure 13.

The two exceptions are the additive and subtractive assignment operators, "+=" and "-=". Here we use "+=" to illustrate these two cases. One is to add an integer i to a pointer expression p, with the result being a pointer which is i units forward in memory from p. The other is to add two values. These two cases are handled by the macro $Add(op1, op2, opname)$, which is given in the Appendix. The additive assignment expression is shown in Figure 14.

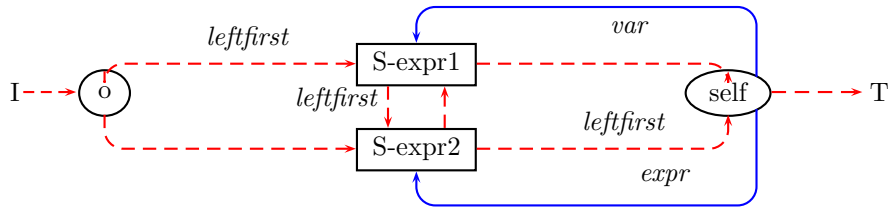
4.4 Conditional Expressions

A conditional expression has the following form:

conditional-expression → expr1 ? expr2 : expr3

To evaluate a conditional expression, evaluate $expr_1$. If the resulting value is non-zero, evaluate $expr_2$ and set its value to the value of the whole conditional expression, otherwise evaluate $expr_3$ and set its value to the value of the whole conditional expression. Only one of the second and third operands is evaluated. If the second and third operands are arithmetic, the usual arithmetic conversions are performed to bring them to a common type, and that is the type of the result. This conversion is given by the function $ConvertName : Nodes \times Nodes \rightarrow Nodes$. If both are `void`, or structure or unions of the same type, or pointers to objects of the same

assignmentexp **::=** **expr1 Assignop expr2**
Assignop = "="



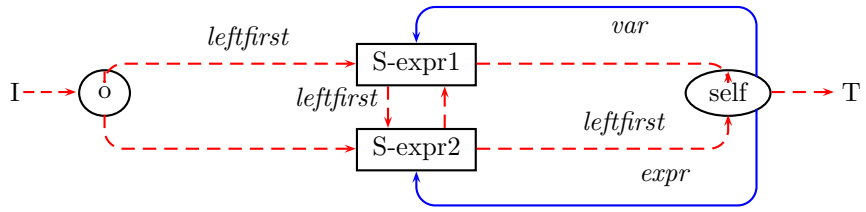
staticType := S-expr1.Terminal.staticType
if ((Primaryid(S-expr2)=true) and (S-expr2.Terminal.staticType.arrayType=true)) then
    S-expr2.onlyArrayNameVisited := true
else
    S-expr2.onlyArrayNameVisited := false
endif

condition *(CompareType(S-expr1.Terminal,S-expr2.Terminal,S-Assignop.Name)=true) and (S-expr1.Terminal.lvalue=true))*

value := self.expr.value
AssignMemory(self.var, self.expr.value)

Figure 12: Montage for assignment expressions.

multiassignexp **::=** **expr1 Assignop expr2**
Assignop = "*="



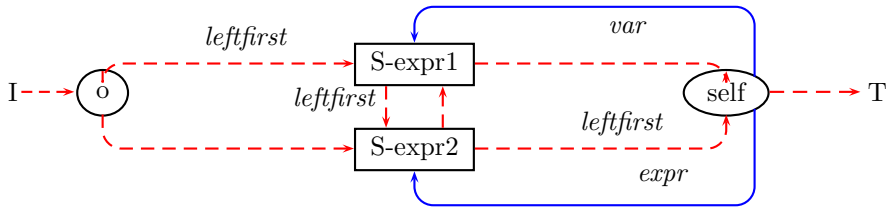
staticType := S-expr1.Terminal.staticType

condition *(CompareType(S-expr1.Terminal,S-expr2.Terminal,S-Assignop.Name)=true) and (S-expr1.Terminal.lvalue=true))*

*value := expr.value * GetMemoryValue(var.var2memory, var.staticType)*
*AssignMemory(var, expr.value * GetMemoryValue(var.var2memory, var.staticType))*

Figure 13: Montage for multiplicative assignment expressions.

addassignexp **::=** **expr1 Assignop expr2**
Assignop = "+="



staticType := S-expr1.Terminal.staticType

condition *(CompareType(S-expr1.Terminal,S-expr2.Terminal,S-Assignop.Name)=true) and (S-expr1.Terminal.lvalue=true))*

value := Add(var, expr, "+")
AssignMemory(var, Add(var, expr, "+"))

Figure 14: Montage for additive assignment expressions.

type, the result has the common type. If one is a pointer and the other the constant 0, the 0 is converted to the pointer type, and the result has that type. If one is a pointer to `void` and the other is another pointer, the other pointer is converted to a pointer to `void`, and that is the type of the result. All of this is performed by the macro *CheckConditions* defined in the Appendix. The Montage for the conditional expression is shown in Figure 15.

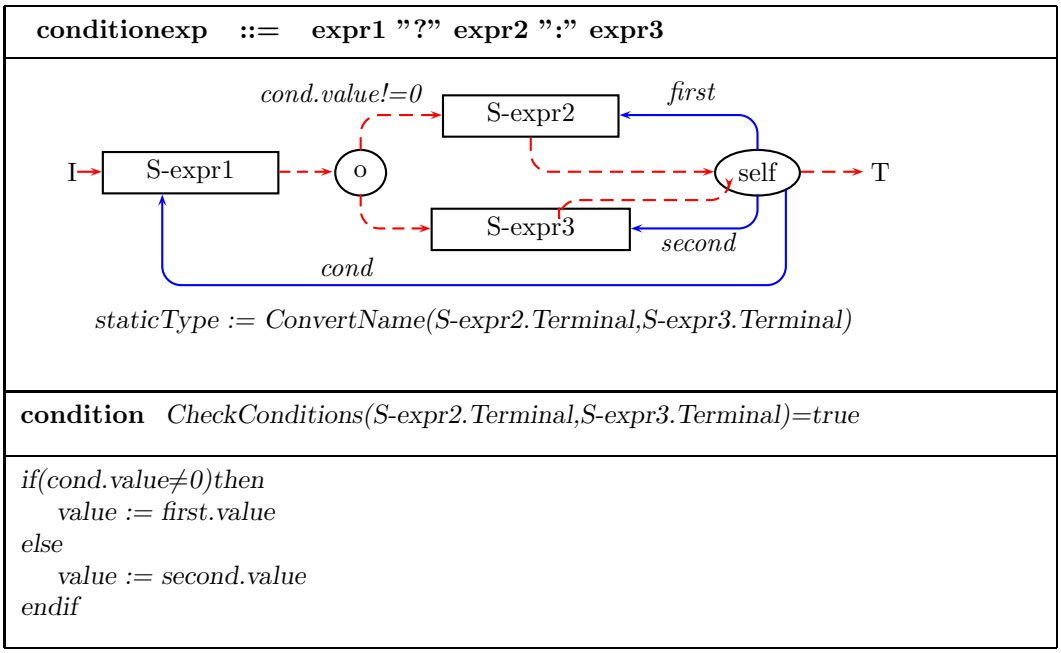


Figure 15: Montage for Condition Expressions.

4.5 Logical OR Expression

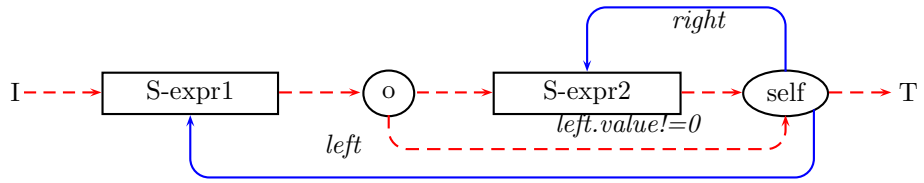
A logical OR expression has the following form:

logical-or-expression → *expr1* || *expr2*

where $expr_1$ and $expr_2$ are expressions. To evaluate a logical OR expression, start by evaluating $expr_1$. If the result is non-zero, the value of the logical OR expression is 1 and $expr_2$ is not evaluated. Otherwise, the value of the logical OR expression is the value of $expr_2$, with non-zero values coerced to 1. The operands need not have the same type, but each must have arithmetic type or be a pointer. The result type is `int`. The Montage for logical or expressions is shown in Figure 16.

The logical AND expression returns 1 if both operands compare unequal to zero, 0 otherwise. The first operand is evaluated, and if it is equal to 0, the value of the expression is 0. Otherwise, the right operand is evaluated, and the value of the result is given by the value of the right

orexp ::= expr1 "||" expr2



staticType := IntNode

condition *(IsArithmeticOrPointerType(S-expr1.Terminal)=true) and (IsArithmeticOrPointerType(S-expr2.Terminal)=true)*

```
if(left.value≠0) or (right.value≠0) then
    value := 1
else
    value := 0
endif
```

Figure 16: Montage for logic or expressions.

operand. Both operands' type must have arithmetic type or a pointer. The result type is `int`. The Montage for logic and expressions is shown in Figure 17.

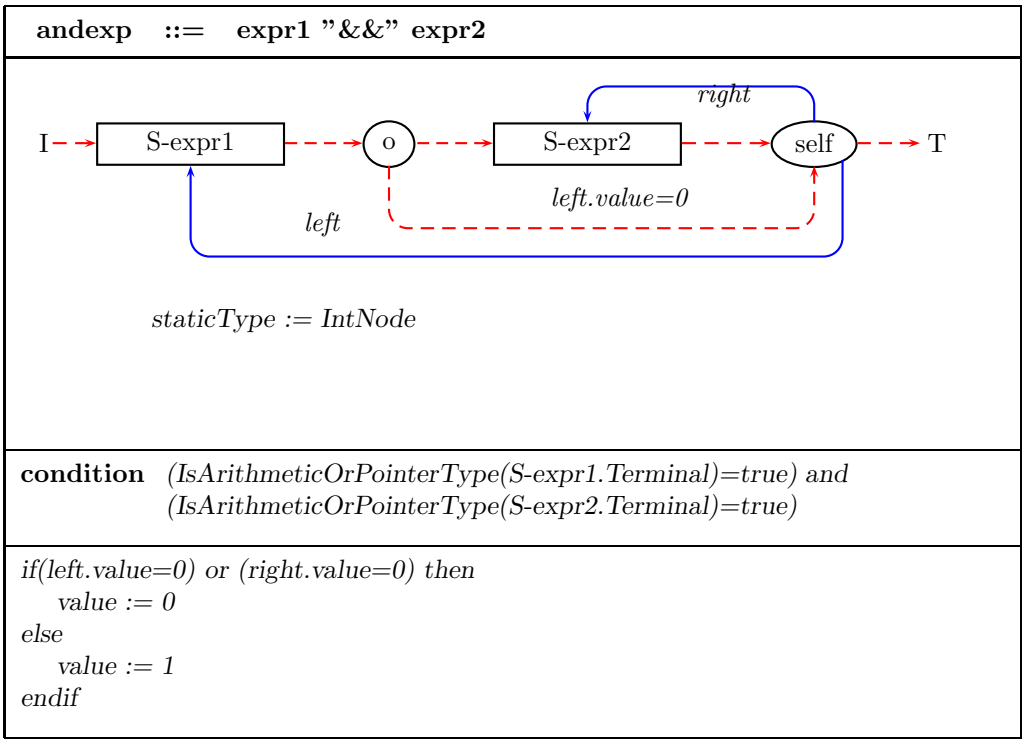


Figure 17: Montage for logic and expressions.

4.6 Equality Expressions

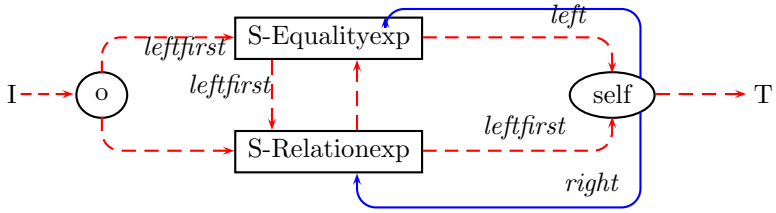
An equality expression has the following forms:

equality-expression == *relational-expression*
equality-expression ! = *relational-expression*

The equality expression is analogous to the relational expression except for their lower precedence. It follows the same rules as the relation expression, but permits additional possibilities: a pointer may be compared to a constant integral expression with value 0, or to a pointer to `void`.

The function $constValue : Nodes \rightarrow CValue$ is used to denote a constant value which is computed during the static analysis. The Montage for equality expressions is shown in Figure 18 and the Montage for non-equality expressions can be given in a similar way.

Equality ::= Equalityexp EqOp Relationexp
EqOp = "=="



staticType := IntNode
if (S-Equalityexp.Terminal.constValue=S-Relationexp.Terminal.constValue) then
constValue := 1
else
constValue := 0
endif

condition *((IsArithmeticType(S-Equalityexp.Terminal.staticType)=true) and (IsArithmeticType(S-Relationexp.Terminal.staticType)=true)) or (CheckPointStruct(S-Equalityexp.Terminal,S-Relationexp.Terminal)=1)*

if(left.value = right.value) then
value := 1
else
value := 0
endif

Figure 18: Montage for equality expressions.

4.7 General Mathematical Expressions

There are many mathematical expressions in C involving binary operators ("*","+","−", *etc.*) whose behaviors are similar. (We treat the bit-wise operators (e.g.,|,&) as ordinary mathematical operators.) To evaluate one of these expressions, evaluate both operand expressions and apply the appropriate function. We present the Montage for multiplication in the following as a representative of this category of expressions.

The operands of * and / must have arithmetic type; the operands of % must have integral type. The function $IsMultiOpType : String \times Nodes \times Nodes \rightarrow Boolean$ in the condition part is used to implement this restriction. The definition for this function is given in the Appendix. The result type of the expression is given by the usual arithmetic conversions on operands, which is given by the function ConvertName. The Montage for Multiplicative expressions is shown in Figure 19.

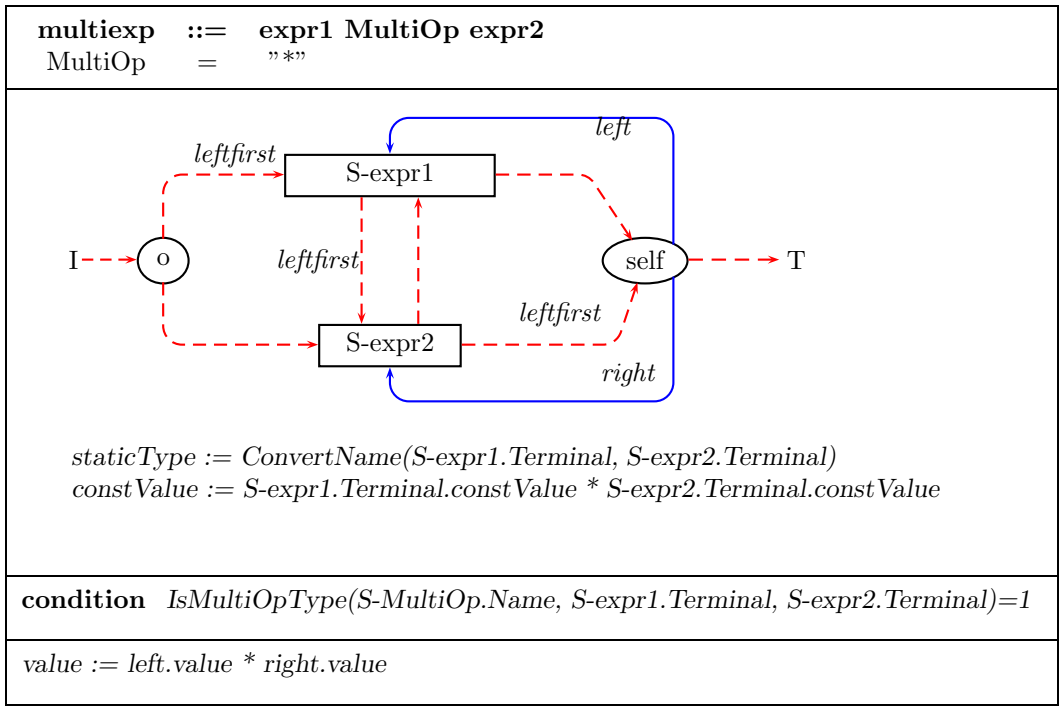


Figure 19: Montage for multiplication expressions.

Now we consider additive expressions. If the operands have arithmetic type, the usual arithmetic conversions are performed. The result type for the additive expression is the type for the arithmetic conversions. In addition, there are some additional type possibilities for each of the operands.

The result of the "+" operator is the sum of the operands. If a pointer to an object in an array and a value of any integral type are added, then the latter is converted to an address offset

by multiplying it by the size of the objects to which the pointer points. The result type for the result expression is still a pointer to an object in an array.

The result of the "-" operator is the difference of the operands. A value of any integral type may be subtracted from a pointer, and the same conversion and conditions as for addition apply. In addition, two pointers to objects of the same type may be subtracted; the result is an integral value representing the displacement between the pointed-to objects. The type of the result depends on the implementation, which is given by the function $Subimptype : Nodes$. The Montage for additive expressions is shown in Figure 20.

4.8 Mathematical Unary Operators

Unary operator expressions have one of the following forms:

unary-expression → *+ expression*
unary-expression → *− expression*
unary-expression → *˜ expression*
unary-expression → *! expression*

Evaluating these expressions takes a form similar to that for binary mathematical operators and all the unary operators' computations are similar. For logical negation expressions, the result type is *int*; otherwise the result type is decided by the type of subexpressions.

The operand of the unary minus operator must have arithmetic type, and the result is the negative value of the operand. We present the Montage for the negation operator in Figure 21; the other Montages can be given in a similar way.

4.9 Casting Expression

A unary expression preceded by the parenthesized name of a type causes conversion of the value of the expression to the named type:

cast-expression → *(type-name) expression*

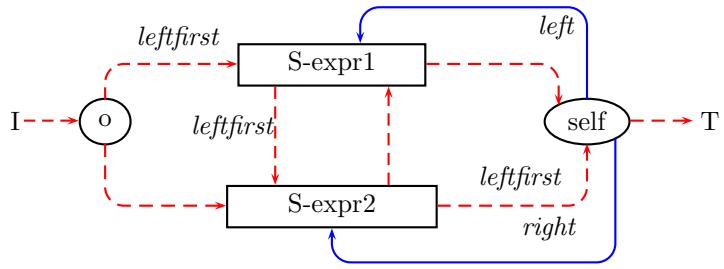
In order to implement the conversion from the old type to the new type, we define a new function $ConvertValue : CValue \times Node \rightarrow CValue$. So the Montage for the cast expression is shown in Figure 22.

4.10 Pre-Increment and Pre-Decrement expression

A pre-increment or pre-decrement expression has the following form:

pre-incr-expression → *++ expression*
pre-decr-expression → *−− expression*

addexp ::= expr1 AddOp expr2
AddOp = "+" |"-"



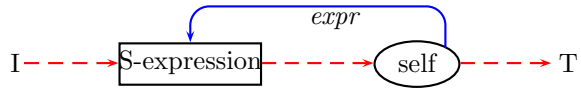
if(((S-expr1.Terminal.staticType.pointType=true) or
(S-expr1.Terminal.staticType.arrayType=true)) and
(S-expr2.Terminal.staticType.Name="int"))then
staticType:=S-expr1.Terminal.staticType
elseif(((S-expr2.Terminal.staticType.pointType=true) or
(S-expr2.Terminal.staticType.arrayType=true)) and
(S-expr1.Terminal.staticType.Name="int")) then
staticType:=S-expr2.Terminal.staticType
elseif((S-AddOp.Name="-") and ((S-expr1.Terminal.staticType.pointType=true) or
(S-expr1.Terminal.staticType.arrayType=true))
and ((S-expr2.Terminal.staticType.pointType=true) or
(S-expr2.Terminal.staticType.arrayType=true))) then
staticType := Subimptype(self)
else
staticType := ConvertName(S-expr1.Terminal,S-expr2.Terminal)
endif
constValue := S-expr1.Terminal.constValue + S-expr2.Terminal.constValue

condition *CompareArithType(S-expr1.Terminal,S-expr2.Terminal,S-AddOp.Name)=true*

value := Add(left, right, AddOp.Name)

Figure 20: Semantics for addition expressions.

unaryexp ::= UnaryOp expression
UnaryOp = "-"



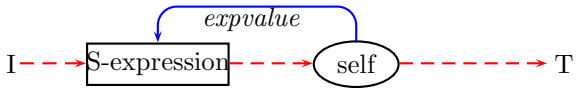
staticType := S-expression.Terminal.staticType
constValue := -(S-expression.Terminal.constValue)

condition *IsArithmeticOrPointerType(S-expression.Terminal)=true)*

value := -expr.value

Figure 21: Semantics of minus operator.

castexp ::= "(" Typename ")" expression



staticType := S-Typename.Terminal

value := ConvertValue(expvalue, S-Typename.Terminal)

Figure 22: Semantics for cast expressions.

To evaluate a pre-increment expression, increment the value stored at the indicated memory location by one and store the new value into that memory location; the incremented value is the value for the whole expression. The operand must be an lvalue and the result is not an lvalue. The Montage for the prefix incrementation expression is shown in Figure 23.

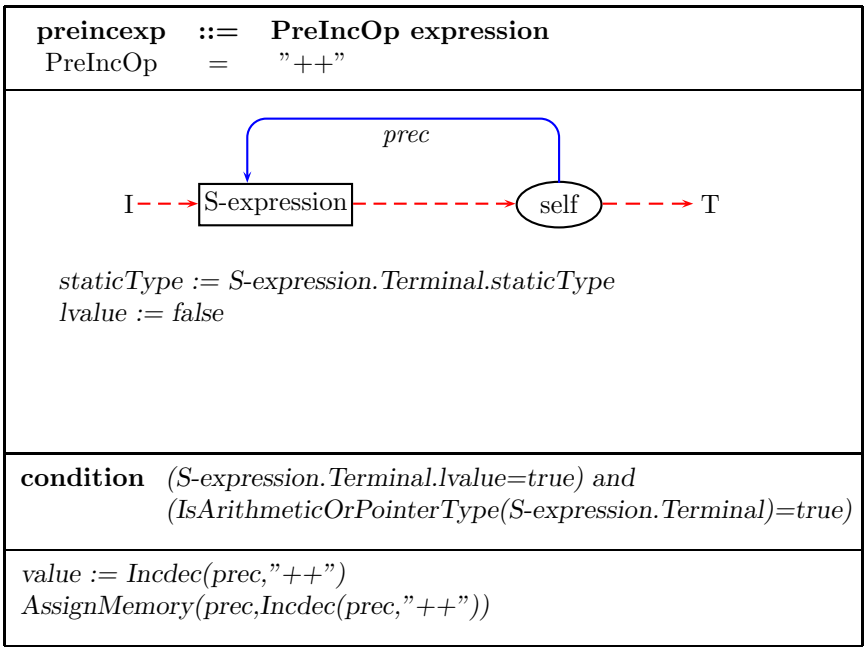


Figure 23: Semantics for prefix incrementation expressions.

4.11 Post-Increment and Post-Decrement Expressions

A post-increment or post-decrement expression has the following form:

post-incr-expression → *expression* ++
post-decr-expression → *expression* −−

Post-increment expressions are handled in the same manner as pre-increment expressions except that the sequence of operations is reversed: *i.e.,* the value of the whole expression is obtained before the incrementing takes place. So we get *value* by using the function $GetMemoryValue$ and increment the memory value by 1 using the macro *Add* in the dynamic part of the following Montage. The operand must be an lvalue. The result is not an lvalue. The Montage for postfix incrementation expressions is shown in Figure 24.

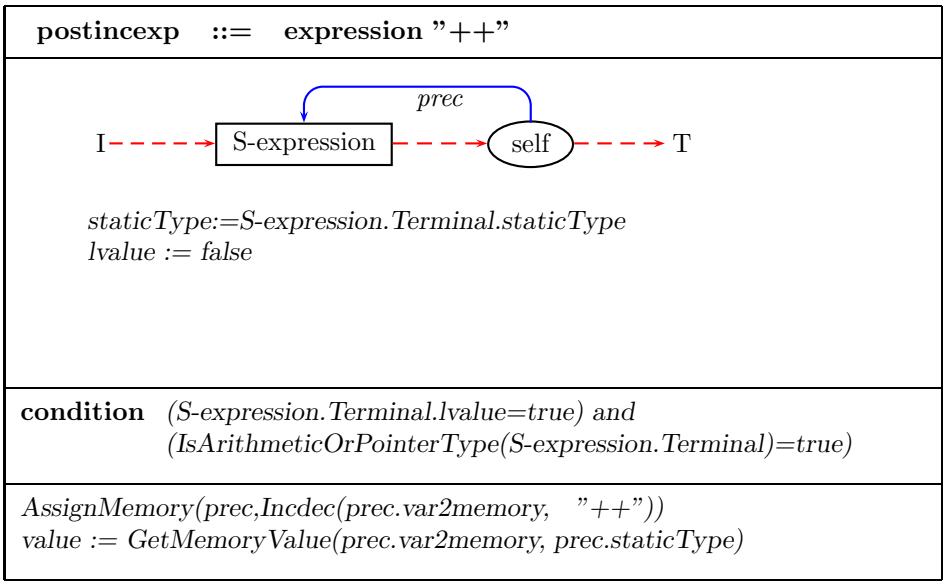


Figure 24: Semantics for postfix incrementation expressions.

4.12 Address

The address expression has the following form:

addr-exp : → `&` *expr1*

where *expr1* is an expression. The & operator in C passes back as its result the address of the memory location given by the argument expression.

The Montage for the address expression is shown in the following. The condition of the Montage guarantees that the expression should be an *lvalue* which means that it refers to a location in memory or a function type.

The unary & operator takes the address of its operand. The operand must be an lvalue or must be of function type. The result is a pointer to the object or function referred to by the lvalue. If the type of the operand is T, the type of the result is "pointer to T". The Montage for address expressions is shown in Figure 25.

4.13 Indirection Expression

The unary $\star$ operator denotes indirection, and returns the object or function to which its operand points. It is an lvalue if the operand is a pointer to an object of arithmetic, structure, union or pointer type. If the type of the expression is "pointer to T", the type of the result is T. In the dynamic part, for a function call we need to distinguish two cases: one is that in the corresponding function definition the return type for that function is a pointer; and the other

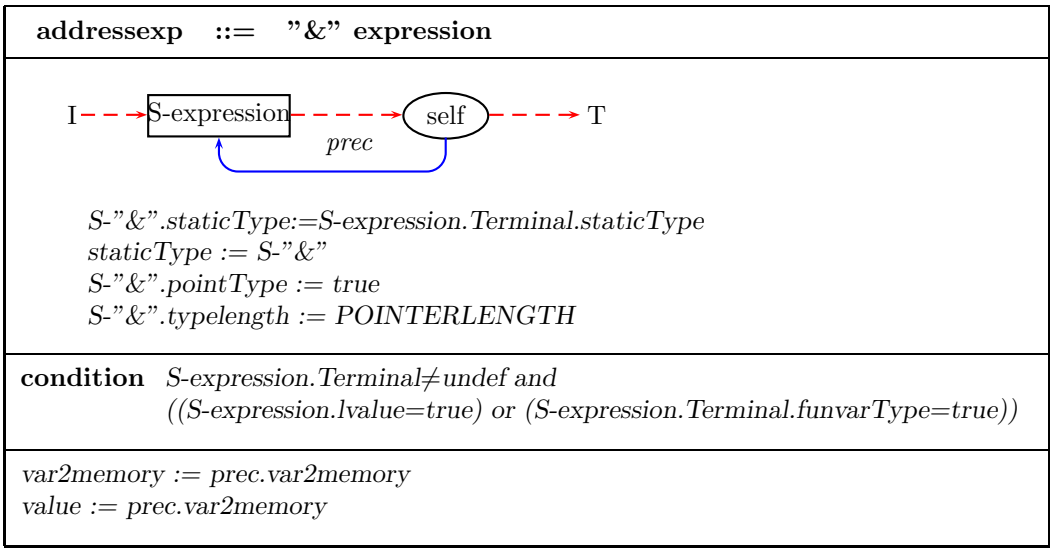


Figure 25: Montage for address expression.

is the return type, which is not a pointer. Because of the conversion rule for the functions in C, the indirection expressions can be used for the above two function definitions. Therefore, we deal with these two cases in the dynamic part in the Montage. The Montage for indirection expressions is shown in Figure 26.

4.14 Primary Expressions

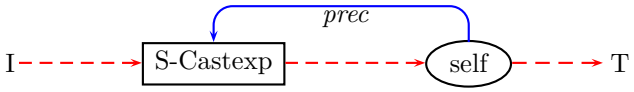
An identifier is a primary expression, if it has been declared as a variable before. Its type is specified by its declaration. An identifier is an *lvalue* if it refers to an object whose type is arithmetic, structure, union, or pointer. In the static part of the Montage, we use d to denote the object declared in the variable declaration.

In the dynamic part of the Montage, we need to distinguish several cases. If this identifier refers to an array name or a function name, we need to set the value and var2memory to the same corresponding variable address (var2memory) because an expression of type "function returning T" is converted to "pointer to function returning T". Otherwise, we treat the identifier as usual, setting the value and var2memory to the ones corresponding to the variable declaration. The Montage for primary variables is shown in Figure 27.

A postfix expression followed by an expression in square brackets is a postfix expression denoting a subscripted array reference. One of the two expressions must have type "pointer to T", where T is some type, and the other must have integral type; the type of the subscript expression is T. The Montage for array variables is shown in Figure 28.

A postfix expression followed by a dot followed by an identifier is a postfix expression. The first operand expression must be a structure or a union, and the identifier must name a member of the structure or union. The value is the named member of the structure or union, and its

Starexp ::= "*" Castexp



if(S-Castexp.Terminal.decl.funvarType=true)then
staticType:=S-Castexp.Terminal.staticType
else
staticType:=S-Castexp.Terminal.staticType.staticType
endif
if(S-Castexp.Terminal.staticType.staticType.funvarType≠true)then
lvalue := true
else
lvalue := false
endif

condition *(S-Castexp.Terminal.staticType.pointType=true)or(S-Castexp.Terminal.decl.funvarType=true)*

if(MemoryValue(prec.var2memory).AddrToFunc≠undef)then
*value := prec.value – for *funcname*
elseif IsPointerToFun(prec)=true then
var2memory:=prec.var2memory
value :=GetMemoryValue(prec.var2memory,prec.staticType)
else
var2memory := GetMemoryValue(prec.var2memory,prec.staticType)
value := GetMemoryValue(GetMemoryValue(prec.var2memory,prec.staticType),prec.staticType.staticType)
endif

Figure 26: Montage for indirection expressions.

Primaryid = Identifier

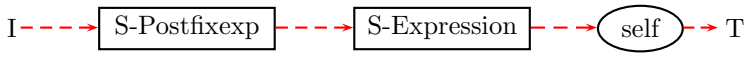
let d = LookUp(Name) in
staticType := d.staticType
decl := d
endlet
lvalue := true

condition *LookUp(Name) ≠ undef*

if(onlyArrayNameVisted=true) or (decl.funvarType = true) then
value := decl.var2memory
var2memory := decl.var2memory
else
if(GetMemoryValue(decl.var2memory,decl.staticType)≠undef)then
value := GetMemoryValue(decl.var2memory,staticType)
else
value := undef
endif
var2memory := decl.var2memory
endif

Figure 27: Montage for primary variables.

Postarray ::= Postfixexp "[" Expression "]"



if(S-Postfixexp.Terminal.staticType.arrayType=true) or
(S-Postfixexp.Terminal.staticType.pointType=true) then
staticType:=S-Postfixexp.Terminal.staticType.staticType
offset := S-Expression.Terminal
base := S-Postfixexp.Terminal
else
staticType := S-Expression.Terminal.staticType.staticType
offset := S-Postfixexp.Terminal
base := S-Expression.Terminal
endif
lvalue := true

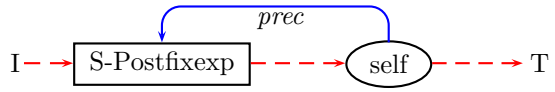
condition *CheckArray(S-Postfixexp,S-Expression)=true*

```
if(base.staticType.arrayType=true) then
  var2memory := base.var2memory + (staticType.typelength * offset.value)
  value:=
      GetMemoryValue((base.var2memory+(staticType.typelength * offset.value)),staticType)
elseif base.staticType.pointType=true then
  var2memory := GetMemoryValue(base.var2memory,base.staticType)
                + (staticType.typelength * offset.value)
  value := GetMemoryValue((GetMemoryValue(base.var2memory,base.staticType)
                + (staticType.typelength * offset.value)),staticType)
endif
```

Figure 28: Montage for array variables.

type is the type of the member. The expression is an lvalue if the first expression is an lvalue, and if the type of the second expression is not an array type. The Montage for this structure references is shown Figure 29.

Postdot ::= Postfixexp "." Identifier



staticType:= S-Postfixexp.Terminal.staticType.fieldPos(S-Identifier.Name).staticType
if(S-Postfixexp.Terminal.staticType.fieldPos(S-Identifier.Name).staticType.arrayType≠true)then
lvalue := true
endif

condition *S-Postfixexp.Terminal.staticType.fieldPos(S-Identifier.Name).staticType≠undef*

var2memory := prec.var2memory +
prec.staticType.fieldPos(S-Identifier.Name).offset
value:=GetMemoryValue(prec.var2memory+prec.staticType.fieldPos(S-Identifier.Name).offset,staticType)

Figure 29: Montage for structure variables.

A postfix expression followed by an arrow followed by an identifier is a postfix expression. The first operand expression must be a pointer to a structure or a union, and the identifier must name a member of the structure or union. The result refers to the named member of the structure or union to which the pointer expression points, and the type is the type of the member; the result is an lvalue if the type is not an array type. The Montage for structure variables with a pointer is shown in Figure 30.

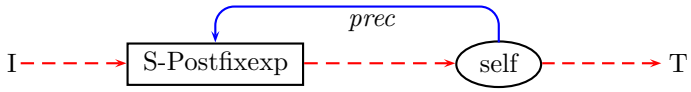
5 Type and Variable Declaration

Before we give the semantics for variables, we discuss types in C in the following.

5.1 Types

There are many kinds of types in C which include the ground types, structure or union type and others. Ground types contain all the basic types including `char, short, int, long, float, double singed,` and `unsinged.` The semantics for all these ground types are similar and they are shown in the following Montage. We set the length for every type in C by using the function $ComputeTypeLength : Nodes \rightarrow Integer$. The Montage for primitive types is shown in Figure 31.

Poststruct ::= Postfixexp "->" Identifier



staticType:=S-Postfixexp.Terminal.staticType.staticType.fieldPos(S-Identifier.Name).staticType
if(S-Postfixexp.Terminal.staticType.staticType.fieldPos(S-Identifier.Name).staticType.arrayType≠true)then
lvalue := true
endif

condition *S-Postfixexp.Terminal.staticType.staticType.fieldPos(S-Identifier.Name).staticType≠undef*

var2memory:=GetMemoryValue(prec.var2memory, prec.staticType)
+ prec.staticType.staticType.fieldPos(S-Identifier.Name).offset
value:=GetMemoryValue(GetMemoryValue(prec.var2memory, prec.staticType)
+ prec.staticType.staticType.fieldPos(S-Identifier.Name).offset, staticType)

Figure 30: Semantics of structure variables with a pointer.

Primitivetype = "int" |"char" |"short" |"void" |"long"
|"float" |"double" |"signed" |"unsigned"

typelength := ComputeTypeLength(self)
Initial := self
Terminal := self

Figure 31: Semantics of primitive types.

The other types of C are more complicated. The structure type consists of a sequence of named members of various types. For brevity, these members are called *fields* of the structure type. The names of members may not conflict with each other. The condition of the Montage for the `struct` type guarantees this restriction. In order to access the fields of the structure, we define a new function $field : Nodes \times String \rightarrow Nodes$. For every node associated with a structure and a field name, the function $field$ gives the corresponding field node.

In the static analysis part of the Montage for structure types, we define two new external functions. One is $AllocateField : token \rightarrow address$, which is used to indicate the field's relative offset in the structure. The other is $AllocateLength : Nodes \rightarrow Integer$ which is used to give the total length for this structure when a variable with this type is declared. By using these two functions, the static analysis sets the offset for every field for the structure type and the total length for this type. The new function *offset:* $Nodes \rightarrow Integer$ indicates the offset for every field in this structure type. The Montage for structure types is shown in Figure 32.

We define the function $incomplete : Nodes \rightarrow Boolean$ so that an *incomplete* type in `structure` type can be dealt with. When an instance for the Montage of incomplete structures is met, we first check whether this structure is defined before. If it is not, we set the function *incomplete* for the nodes denoting this structure and the outside structure to be true if exists. The Montage for "incomplete" structure types is shown in Figure 33.

The union type is similar to the structure type but it contains any one of several members of various types at different times. It may be thought of as a structure all of whose members begin in offset 0 and whose size is sufficient to contain any of its members. The static analysis of the Montage for union type sets the length for this `union` type by using the external function *AllocateLength*. Similarly, the condition of the Montage for union type guarantees that all the field names do not conflict with each other. The Montage for union types is shown in Figure 34.

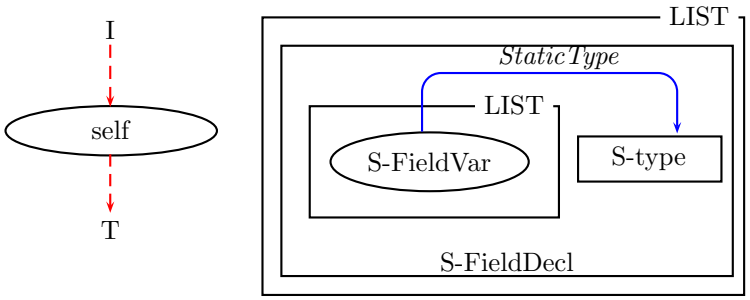
5.2 Variable Declaration and Initialization

Before we use a variable we need to declare it. C distinguishes between so-called *static variables* and other variables. The difference between static and non-static variables arises when control is passed to the declaration for a variable. If the variable is not static, new memory is always allocated to the variable and its initializing expression (if it exists) is evaluated with the value of the expression being assigned to the new memory location. If the variable is static, then the above allocation and initialization is performed only the first time that the declaration is executed; should the declaration become the focus of control again, the same memory segment is allotted to the variable.

5.2.1 Non-static Variable Declaration

The Montage for non-static variable declarations is shown in Figure 35. The condition of the Montage guarantees that the new variable name is not multiply defined. In order to set up the relation between the variable name and its corresponding node represented in the Montage, we define a new function $declTable : Nodes \times String \rightarrow Nodes$.

struct_type **::=** **"struct" "{" [FieldDecl { ";" FieldDecl }] "}"**
FieldDecl = type FieldVar {"," FieldVar }
FieldVar = Ident



CurStruct := self
lastStruct := CurStruct
–
CurStruct := lastStruct
vary_over_ind(i, self.S-FieldDecl)
vary_over_ind(j, self.S-FieldDecl[i].S-FieldVar)
self.S-FieldDecl[i].S-FieldVar[j].offset:=
AllocateField(self.S-FieldDecl[i].S-FieldVar[j]);
endvary
endvary
length:=AllocateLength(self);
vary(s, self.S-FieldDecl.S-FieldVar)
field(self,s.String):=s;
endvary

condition *vary(d1, self.S-FieldDecl.S-FieldVar)*
vary(d2, self.S-FieldDecl.S-FieldVar)
d1 ≠ d2 implies d1.Name ≠ d2.Name
endvary endvary

Figure 32: Semantics of structure types.

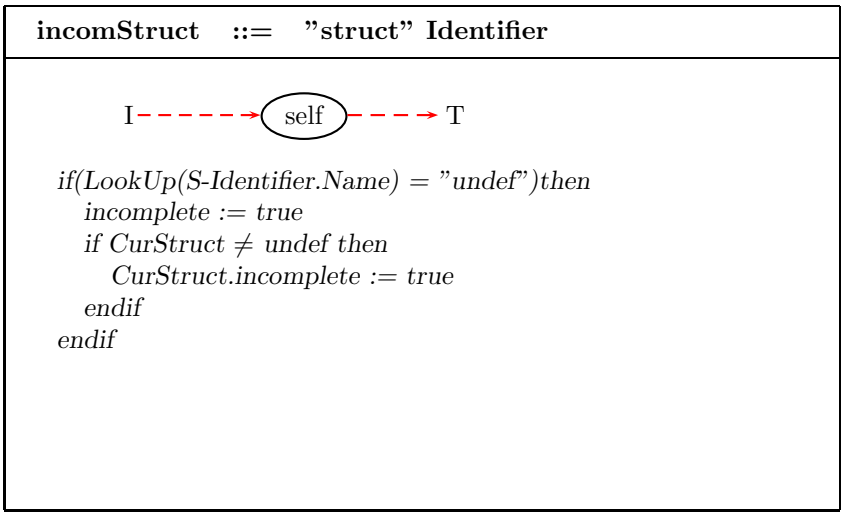


Figure 33: Semantics of "incomplete" structure types.

The dynamic semantics first checks whether an initializing expression exists. If it exists but is not evaluated, then the control is passed to the initializing expression to compute the value for the expression. Then the dynamic semantics allocates the memory for this variable by using the new external function $AllocateMem$ and assigns the initializing expression's value (if exists) to the corresponding memory. The Montage for non-static variable declarations is shown in Figure 35.

5.2.2 Static Variable Declaration

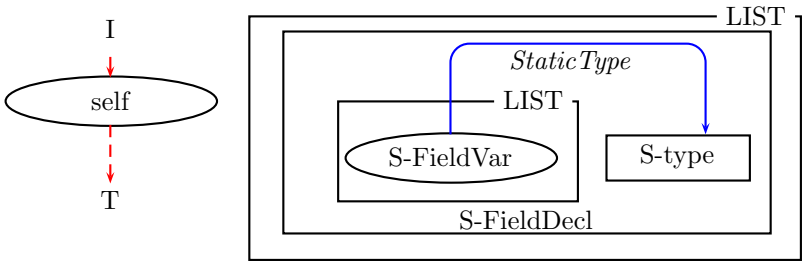
If the variable is static, the above allocation and initialization is performed only the first time that the declaration is executed; should the declaration become the focus of control once again, the same memory segment is allotted to the variable.

The condition of the Montage for static variable declarations is similar to that for non-static variable declaration. We define a new function $visited : Nodes \rightarrow Boolean$ to indicate whether this variable declaration has been visited before. The function *visited* is initially set to be false. If the variable has not been visited, it will allocate the new memory address for the new static variable by using the external function $AllocateMem$. Otherwise because the *var2memory* function for this node has contained the address for the `static` variable, nothing further needs to be done and the dynamic semantics transfers control to the next node. The Montage for static variable declarations is shown in Figure 36.

5.2.3 Array Declaration

Array variables are flexible in C, which can be treated as pointer variables. When a variable is declared, if the constant expression in the dimension in an array is missing, then the array is an

union_type **::=** **"union" "{" [FieldDecl { ";" FieldDecl }] "}**
FieldDecl = type FieldVar {"," FieldVar }
FieldVar = Ident

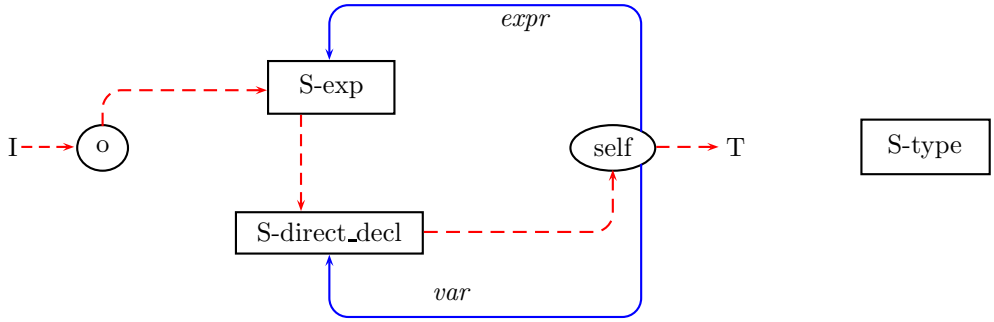


CurStruct :=self
lastStruct := CurStruct
-
CurStruct := lastStruct
vary_over_ind(i,self.S-FieldDecl)
vary_over_ind(j,self.S-FieldDecl[i].S-FieldVar)
self.S-FieldDecl[i].S-FieldVar[j].offset:=0;
endvary
endvary
length := AllocateLength(self);
vary(s, self.S-FieldVar)
field(self, s.String):=s;
endvary

condition *vary(d1, self.S-FieldDecl.S-FieldVar)*
vary(d2, self.S-FieldDecl.S-FieldVar)
d1 ≠ d2 implies d1.Name ≠ d2.Name
endvary *endvary*

Figure 34: Semantics for union types.

non_decl ::= type direct_decl ["=" exp]
direct_decl = Ident



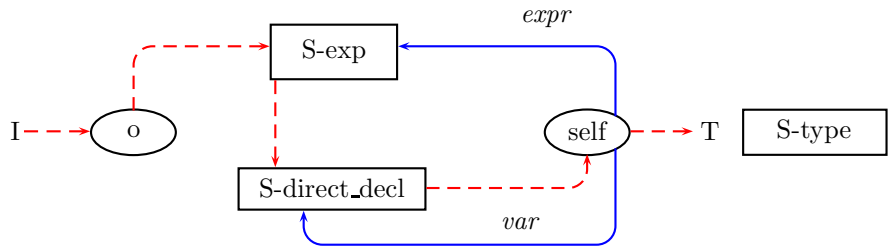
declTable(self.S-Ident.Name):=self.S-direct_decl;
self.S-direct_decl.staticType := S-Type.Terminal;

condition *declTable(self.S-Ident.Name)=undef*

var.var2memory := CurMemoryPos
CurMemoryPos := AllocateMem(CurMemoryPos,var.staticType.typelength)
var.value := expr.value
AssignMemory(var,expr.vlaue)

Figure 35: Semantics of non-static variable declarations.

static_decl ::= "static" type direct_decl ["=" exp]
direct_decl = Ident



declTable(self.S-Ident.Name):=self.S-direct_decl;
self.S-direct_decl.visited:=false;
S-direct_decl.staticType:= S-type.Terminal;

condition *declTable(self.S-Ident.name)=undef*

```
if(var.visited=false)then
   if(expr.value≠undef)then
      var.var2memory := CurMemoryPos;
      CurMemoryPos := AllocateMem(CurMemoryPos,var.staticType.typelength);
      AssignMemory(var, expr.value);
      var.value := expr.value;
   endif
   var.visited=true;
endif
```

Figure 36: Semantics of static variable declaration.

incomplete type. For a complete array variable, the size for the array variable is known.

Before we give the semantics for array variables, let us consider the following example first to see how we compute the length for a variable:

int d[3][5];

and we assume that an integer takes 4 bytes in the memory.

In order to name the nodes in Figure 37 easily, the above strings d, [3][5], [3] and [5] represent the nodes for the corresponding Montages. Now we show how to compute the length for every node. Here we adopt the inherited attribute to compute the function *typelength* for every node. Before the root node is analyzed, we set the typelength for the node [3][5] to be 4, which is shown in step 1. When a node [3][5] is analyzed, the function typelength for node [3] is set to 1 and the function typelength for node [5] is set to 4, which is shown in step 2 and 3. After the node for [3] is reduced, the function typelength for it is 3, shown in step 4; and after the node for [5] is reduced, the typelength for it 20, shown in step 5. And after the node [3][5] is analyzed, the typelength for it is $3\times\ 20 = 60$, shown in step 6. The whole process for this computation is shown in Figure 37.

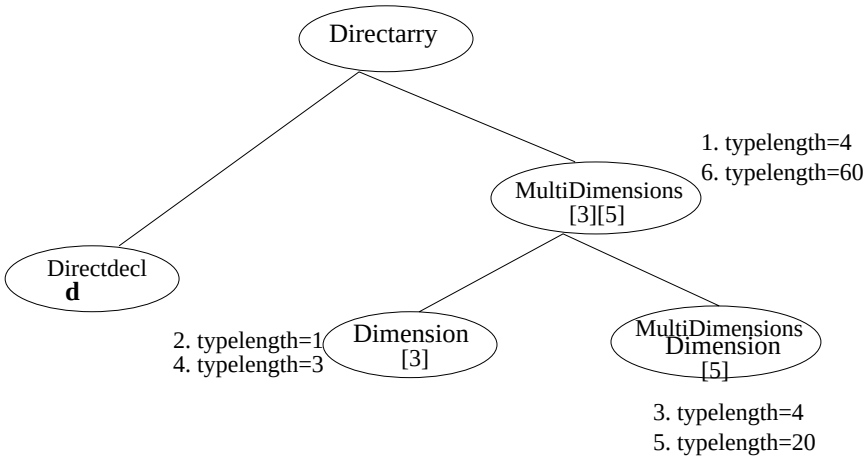


Figure 37: The Procedure for computing the function typelength.

In order to implement array variable declarations, we give the following Montages. In the next two Montages, we set the function *staticType* for the node associated with the array name and the node associated with the terminal node in MultiDimensions respectively. And if the array is incomplete in the node associated with MultiDimensions then we set *incomplete* for the node associated with array name. The Montage for array variable declarations is shown in Figure 38.

Now we consider how to deal with multiple dimensions. We treat the function *typelength* as an inherited attribute. Before an instance for the Montage of Dimensions is analyzed, we

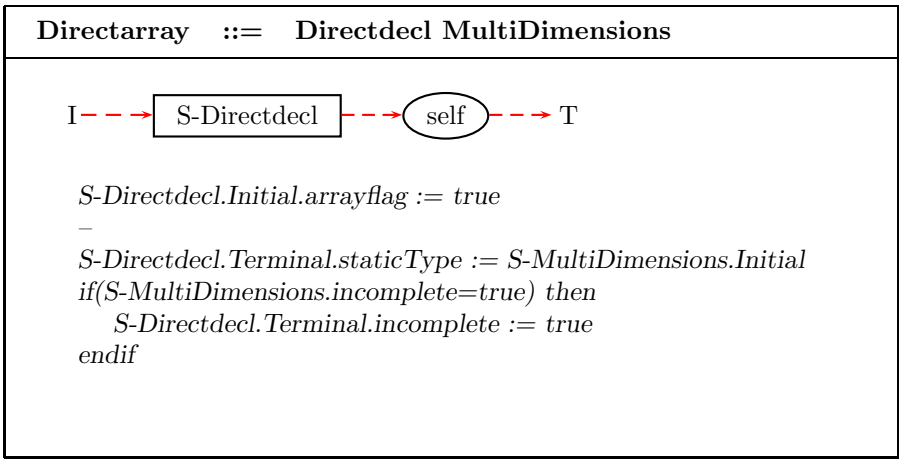


Figure 38: Semantics of array variable declarations.

set the length for the node associated with the instance for the Montage Dimension to 1. And because we already have the type for an array variable being analyzed, we pass the length to the nonterminal *MultiDimensions*. After all its components for the Montage Multi-Dimensions have been analyzed, we recompute the length for the nonterminal *Dimensions* to get the correct value.

In Figure 40, we show how to deal with every dimension in an array. If the value in the dimension is a constant value, then we can compute the current length for the array. If the value does not exist, we set the incomplete to be true, showing this array variable is an incomplete type.

6 Functions

A function is a very important and useful concept in C. Generally speaking, functions are block-structured; the variables which are declared in a C-function can only be referenced within that function. In order to implement this, we define two new functions *EnclosingProgramOrProcedureOrLocalScope: Nodes* and *lastProgramOrProcedureOrLocalScope: Nodes* $\rightarrow$ *Nodes*. The first one *EnclosingProgramOrProcedureOrLocalScope* is used to denote the current block and *lastProgramOrProcedureOrLocalScope* is used to denote the outside level block given by the current block structure. The function $declTable : Nodes \times String \rightarrow Nodes$ is used to denote a node given by the block structure and a name and the first *Nodes* is used to denote the block structure from which a name is looked up.

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 an ASM function for a given node.

The universe *Stack* comprises the positive integers. A dynamic distinguished element *StackTop:Integer* is used to indicate the current top of the stack. To store state-associated information

Dimensions ::= Dimension MultiDimensions

S-Dimension.Terminal.typelength := 1
S-MultiDimensions.Terminal.typelength := typelength
–
*typelength := S-Dimension.Terminal.typelength **
S-MultiDimensions.typelength
arrayType := true
staticType := S-MultiDimensions.Initial
if(S-Dimension.incomplete=true) then
incomplete := true
endif

Figure 39: Semantics of Multi-Dimensions.

Dimension ::= "[" ConditionexpOrNull "]"

arrayType := true
if(Conditionexp(S-ConditionexpOrNull)=true)then
*typelength := S-ConditionexpOrNull.constValue * typelength*
else
incomplete := true
endif

condition *(S-ConditionexpOrNull.constValue≠undef) or*
(Conditionexp(S-ConditionexpOrNull)≠true)

Figure 40: Semantics of One Dimension.

on the stack, we modify the functions Value, Memory and RecCaller to be binary functions from $Nodes \times Stack \rightarrow CValue$.

6.1 Declarations and Definitions for Functions

Now we consider the semantics for C-Functions. Here we assume that any C function is declared before it is defined. In the Montage for function declarations, we set the value of *declTable* at this function name to the root node of the declaration Montage. In the following Montage shown in Figure 41, we consider the two cases for function heads occurring in function definitions and function declarations. The function head refers to a function name and its parameter part.

When the following Montage in Figure 41 is used for a function declaration, we need to set *declTable* for the function name. When the following Montage is used for function definitions, we need to set the function $formalPar : Nodes \rightarrow Nodes$ for the node associated with the function declaration to the node corresponding to the parameter in this C-function. In addition, we need to check the type consistency between parameters given in the function definition and the function declaration by using the function $CheckParameter : Node \times Node \rightarrow Boolean$.

Directcase4 ::= Directdecl "(" Parametertypelist ")"
Directdecl = Ident


```
Name := S-Directdecl.Initial.Name
self.funvarType := true
typelength := FUNLENGTH
if(FuncDef =  undef) then
declTable(self.S-Ident.Name) := S-Directdecl;
else
let d=LookUpVarDcl(S-Directdecl.Initial.Name) in
  if S-Parametertypelist.S-Parameterdec ≠ undef then
    d.formalPar := S-Parametertypelist.S-Parameterdec
  endif
endlet
endif
```

condition *((FuncDef=true) and (CheckParameter(S-Directdecl,S-Parametertypelist,self)=true)) or (FuncDef=undef)*

Figure 41: Semantics of function declaration and definition.

Now we consider the semantics for parameters. There are two possible times at which the

copying of values required for parameter passing may occur. One is when an instance for the Montage of function call is reduced; the other is when an instance for the Montage of function parameters is reduced. Here we implement this assignment when the second situation occurs.

In the Montage shown in Figure 42, we give the dynamic semantics as follows. When an instance for the Montage of Parametertypelist is reduced, we assign all the arguments in the function call to the corresponding parameters. The actual arguments can be derived by the function RecCaller and actualParam, which are defined in the Montage of function calls.

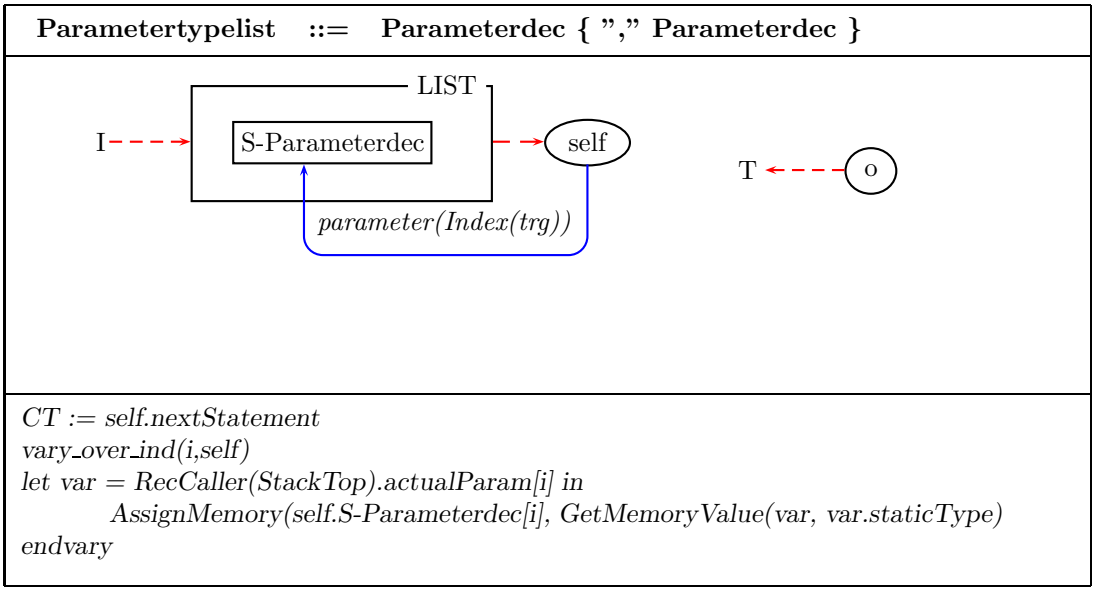


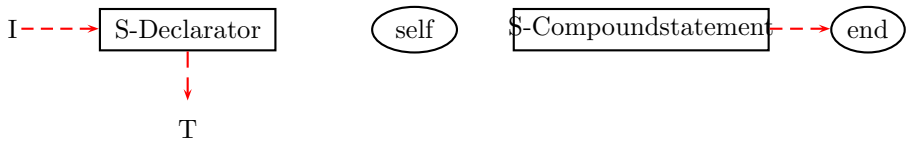
Figure 42: Semantics of Parameters.

6.2 Callee Part

Now we consider the function definition. In order to deal with the scope, we set the function $EnclosingScope$ for the function definition. Additionally, in the static analysis we set $gotoTarget$ for those nodes in the $LabelTable$. In addition, we regard a function name as a variable stored in the memory. Given an address which stores a function variable in the memory, the function $AddrToFunc : address \rightarrow Nodes$ is used to denote a node where control should be directed when that function is called.

The dynamic semantics decrements the value of $StackTop$ to the previous one because the C-function execution is over. Control passes to the node where this function is called by using the function $RecCaller : Integer \rightarrow Nodes$ which is used 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. This function will be used when return statements are met. The Montage for callee part of a function is shown in Figure 43.

Funcdefcase2 ::= Declspecifier Declarator Compoundstatement



CurFundef := true
CurFunc := self
lastScope := EnclosingScope
EnclosingScope := self
FuncDef := true
–
CurFundef := false
FuncDef := undef – used to distinguish fun def and decl
CurFunc := undef
EnclosingScope := 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

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

Figure 43: Semantics of Function Definitions.

6.3 Function Call

Although a function can be called by a function variable or a function pointer, we regard it as an expression whose value is a memory address from which we use the function $AddrToFunc$ to get the corresponding task for this C-function. The static analysis of the Montage for the function call sets the data and control functions as shown in Figure 44. Because the function call is an expression and in order to represent the type for this expression, we define a new node called "returnPoint" and set the staticType for it as the staticType for this expression. In C, a pointer to the function like $int(*p)()$ can be used in the following form $p()$ to call the corresponding function. So we need the two cases to deal with the staticType as shown in the following Montage. One is used to get the staticType for general function calls and the other is used for a pointer to a function.

A function call is a postfix expression, called the function designator, followed by parentheses containing a possibly empty, comma-separated list of assignment expressions, which constitute the arguments to the function. The postfix expression must be of type "pointer to function returning T", for some type T, and the value of the function call has type T. The dynamic semantics of the Montage increments the value of the stack. The dynamic semantics sets the value for the function $RecCaller$ in order to transfer control to the node next to this function call. The Montage for function calls is shown in Figure 44.

A function returns its caller by the `return` statement. When `return` is followed by an expression, the value is returned to the caller of the function. We direct control to the appropriate node by using function $RecCall : Nodes \times Integer \rightarrow Nodes$ which is set in the function call, shown in Figure 44. The Montage for the `return` statement is shown in Figure 45.

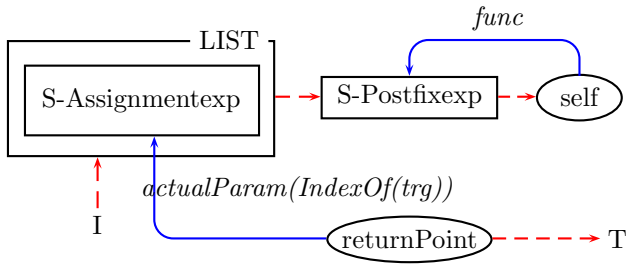
7 Discussion

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

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

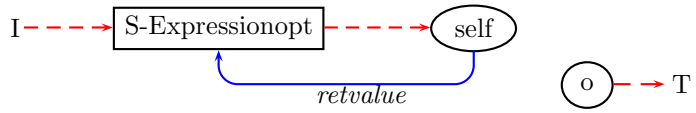


if((S-Postfixexp.Terminal.staticType.pointType=true) and
(S-Postfixexp.Terminal.staticType.staticType.funvarType=true))then
returnPoint.staticType:=S-Postfixexp.Terminal.staticType.staticType.staticTyp
else
returnPoint.staticType:=S-Postfixexp.Terminal.staticType.staticType
endif

CT := GetMemoryValue(func.value,func.staticType).AddrToFunc
StackTop := StackTop + 1
RecCaller(StackTop+1) := returnPoint

Figure 44: Semantics of function calls.

ReturnStatement ::= "return" Expressionopt ";"



RecCaller(StackTop).recValue(StackTop-1) := retvalue.value
StackTop := StackTop -1
CT := RecCaller(StackTop)

Figure 45: Montage for `return` statements.

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 extended abstract 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: Extended Abstract." Proceeding of ASM 2000 workshop, Monte Verit, Switzerland. Mar. 19-24, 2000.

[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.

8 Appendix: Some Macro Definitions

```
asm IsArithmeticOrPointerType(op) is
      if((IsArithmeticType(op.staticType)=true) or (op.staticType.pointType=true))then
         IsArithmeticOrPointerType_result := true
      else
         IsArithmeticOrPointerType_result := false
      endif
endasm


   asm CompareType(var,exp,name) is
         if(exp.staticType.funvarType=true) then
            CompareType_result := CompareBasicFunType(var.staticType,exp.staticType)
         elseif var.decl.funvarType= true then
            CompareType_result :=
               CompareBasicType(var.decl.staticType.staticType.staticType,exp.staticType)
         elseif (var.Terminal.lvalue!= true or (var.staticType.arrayType=true and
                     var.Terminal.incomplete = true)) then
            CompareType_result := false;
         else
            CompareType_result := CompareBasicType(var.staticType,exp.staticType)
         endif
```

```
    endif
endasm


asm ConvertName(token1,token2) is
    if((token1.staticType.funvarType=true) and (token2.staticType.Name="int"))then
        ConvertName_result:=token1.staticType
    elseif((token2.staticType.funvarType=true) and (token1.staticType.Name="int"))then
        ConvertName_result:=token2.staticType
    elseif token1.staticType.Name="double" then
        ConvertName_result:=token1.staticType
    elseif token2.staticType.Name="double" then
        ConvertName_result:=token2.staticType
    elseif token1.staticType.Name="float" then
        ConvertName_result:=token1.staticType
    elseif token2.staticType.Name="float" then
        ConvertName_result:=token2.staticType
    elseif token1.staticType.Name="int" then
        ConvertName_result:=token1.staticType
    elseif token2.staticType.Name="int" then
        ConvertName_result:=token2.staticType
    elseif(token1.staticType.funvarType=true and
        token2.staticType.funvarType=true and
            CompareBasicType(token1.staticType.staticType,
            token2.staticType.staticType)=true) then
        ConvertName_result:=token1.staticType
    else
        ConvertName_result:= "undef"
    endif
endasm


asm CheckConditions(exp1,exp2) is
    if((IsArithmeticType(exp1.staticType)=true) and (IsArithmeticType(exp2.staticType)=true))then
        CheckConditions_result:=true
    elseif((exp1.staticType.pointType=true) and (exp2.staticType.pointType=true)) then
        CheckConditions_result := CheckTypeName(exp1.staticType.staticType,
                    exp2.staticType.staticType);
    elseif(((exp1.staticType.pointType=true) and (exp2.staticType.Name="void"))or
            ((exp2.staticType.pointType=true) and (exp1.staticType.Name="void"))or
            ((exp1.staticType.Name="void") and (exp2.staticType.Name="void")))then
```

```
        CheckConditions_result := true
    elseif((Structorunion(exp1.staticType)=true) and
            (Structorunion(exp2.staticType)=true))then
        CheckConditions_result:=CheckStrucType(exp1.staticType,
                    exp2.staticType);
    elseif((exp1.staticType.funvarType=true) and (exp2.staticType.funvarType=true)) then
        CheckConditions_result:= CheckConditions(exp1.staticType, exp2.staticType)
    else
        CheckConditions_result:=false
    endif
endasm


asm CheckPointStruct(exp1,exp2) is
    if((exp1.staticType.pointType=true) and (exp2.staticType.pointType=true))then
        CheckPointStruct_result := CheckPointStruct(exp1.staticType.staticType,
                    exp2.staticType.staticType)
    endif
endasm


asm IsIntType(op) is
    if((op.staticType.Name="int") or (op.staticType.Name="char"))then
        IsIntType_result := true
    else
        IsIntType_result := false
    endif
endasm


asm IsArithmeticType(op) is
    if((op.Name="int") or (op.Name="char") or
        (op.Name="float") or (op.Name="double")) then
            IsArithmeticType_result := true
    else
            IsArithmeticType_result := false
    endif
endasm


asm IsMultiOpType(op,token1, token2) is
```

```
    if(op="%") then
        if((IsIntegerType(token1)=1) and (IsIntegerType(token2)=1))then
            IsMultiOpType_result:=1
        else
            IsMultiOpType_result:=0
        endif
    elseif((IsArithmeticType(token1.staticType)=true) and
            (IsArithmeticType(token2.staticType)=true))then
            IsMultiOpType_result:=1
        else
            IsMultiOpType_result:=0
        endif
    endif
endasm


asm CompareArithType(op1,op2,name) is
    if name = "+" then
        if (((op1.staticType.pointType=true) or (op1.staticType.arrayType=true))
                and (op2.staticType.Name="int")) or
        (((op2.staticType.pointType=true) or (op2.staticType.arrayType=true))
                and (op1.staticType.Name="int")) or
        ((op1.staticType.pointType!=true) and (op2.staticType.pointType!=true) and
        (op1.staticType.arrayType!=true) and (op2.staticType.arrayType!=true) and
        (op1.staticType.Name = op2.staticType.Name)) or
        ((op1.staticType.funvarType=true) and (IsIntType(op2)=true)) or
        ((op2.staticType.funvarType=true) and (IsIntType(op1)=true)) then
            CompareArithType_result := true
        else
            CompareArithType_result := false
        endif
    elseif name = "-" then
        if (((op1.staticType.pointType=true) or (op1.staticType.arrayType=true))
                    and (op2.staticType.Name="int")) or
        (((op2.staticType.pointType=true) or (op2.staticType.arrayType=true))
                    and (op1.staticType.Name="int")) or
        ((op1.staticType.pointType!=true) and (op2.staticType.pointType!=true) and
        (op1.staticType.arrayType!=true) and (op2.staticType.arrayType!=true) and
                    (op1.staticType.Name = op2.staticType.Name)) or
        (((op1.staticType.pointType=true) or (op1.staticType.arrayType=true)) and
        ((op2.staticType.pointType=true) or (op2.staticType.arrayType=true)) and
```

```
      (op1.staticType.staticType.Name = op2.staticType.staticType.Name))then
         CompareArithType_result := true
      elseif((op1.staticType.funvarType=true) and (IsIntType(op2)=true)) or
            ((op2.staticType.funvarType=true) and (IsIntType(op1)=true)) then
         CompareArithType_result := true
      else
         CompareArithType_result := false
      endif
   endif
endasm


asm Add(op1, op2, name) is
   if name = "+" then
      if(((op1.staticType.arrayType=true) or (op1.staticType.pointType=true)) and
            ((op2.staticType.arrayType!=true) and (op2.staticType.pointType!=true))) then
         Add_result := op1.value+(op2.value*op1.staticType.staticType.typelength)
      elseif (((op2.staticType.arrayType=true) or (op2.staticType.pointType=true)) and
            ((op1.staticType.arrayType!=true) and (op1.staticType.pointType!=true))) then
         Add_result := op2.value+(op1.value*op2.staticType.staticType.typelength)
      else
         Add_result := op2.value+op1.value
      endif
   elseif name = "-" then
      if(((op1.staticType.arrayType=true) or (op1.staticType.pointType=true)) and
            ((op2.staticType.arrayType=true) or (op2.staticType.pointType=true))) then
         Add_result := %divide(op1.value-op2.value,op1.staticType.staticType.typelength)
      elseif ((op1.staticType.arrayType=true) or (op1.staticType.pointType=true)) then
         Add_result := op1.value - (op2.value*op1.staticType.staticType.typelength)
      elseif ((op2.staticType.arrayType=true) or (op2.staticType.pointType=true)) then
         Add_result := op2.value-(op1.value*ope2.staticType.staticType.typelength)
      else
         Add_result := op1.value-op2.value
      endif
   endif
endasm


asm Unary(arg, op) is
   if arg = undef then
   elseif op = "++" then
```

```
        Unary_result := arg + 1
    elseif op = "-" then
        Unary_result := arg - 1
    elseif op = " " then
        Unary_result := ((-1)*arg) - 1
    elseif op = "!" then
        if(arg=0) then
            Unary_result := 1
        else
            Unary_result := 0
        endif
    endif
endasm


asm Incdec(op1,name) is – name is either "++" or "–"
    if name = "++" then
        if((op1.staticType.arrayType=true) or (op1.staticType.pointType=true)) then
            Incdec_result := GetMemoryValue(op1.var2memory,op1.staticType) +
                op1.staticType.staticType.typelength
        else
            Incdec_result := GetMemoryValue(op1.var2memory,op1.staticType) + 1
        endif
    else
        if((op1.staticType.arrayType=true) or (op1.staticType.pointType=true)) then
            Incdec_result := GetMemoryValue(op1.var2memory,op1.staticType) −
                op1.staticType.staticType.typelength
        else
            Incdec_result := GetMemoryValue(op1.var2memory,op1.staticType) - 1
        endif
    endif
endasm


asm CheckPara(var,pars) is
    let d = LookUpFuncDcl(var.Initial.Name,EnclosingScope.lastScope) in
        do forall i=0; i¡pars.S-Parameterdec.Listlength; i=i+1
            if(CheckperPara(d.formalPar[i], pars.S-Parameterdec[i])=false)then
                CHECK_PARA_ERROR := true
            endif
        enddo
```

```
    endlet
endasm


asm CheckParameter(var,pars,i) is
relation CHECK_PARA_ERROR
relation flag
    if Parameterdec(i.Parent) = true then
        CheckParameter_result := true – no check for fun par
    else
        if flag = false then – step one
            _CheckPara(var, pars)
            flag := true
        else – step two
            CheckParameter_result := not CHECK_PARA_ERROR
        endif
    endif
endasm
```