The Evolving Algebra Semantics of C
Preliminary Version

Yuri Gurevich and James K. Huggins

October 6, 1992

0 Introduction

0.1 Overview of the Method

Evolving algebras were first proposed in [Gu1] (and more recently discussed in [Gu3]) as an improvement upon (a stronger version of) Turing's thesis. One may use an evolving algebra to model any computation. In particular, one may describe an evolving algebra which models a particular computation in "lock-step"; that is, for every step taken by the modeled computation, the evolving algebra takes one step. In addition, one may describe an evolving algebra which models a particular computation at any desired level of abstraction. This is an improvement upon the traditional Turing machine model, where the abstraction level is fixed at a low level and may require many Turing machine steps to simulate one step of an algorithm.

An evolving algebra contains a description of a first-order logical *signature* which describes the states of an abstract machine, along with a collection of *transition rules* which describe the temporal relationships between states. Once combined with a description of the initial state (that is, a *structure* of the corresponding signature), a computation (or set of computations) is determined.

Evolving algebras may be used to provide operational semantics for a programming language. A programming language may be viewed as a kind of universal algorithm. It takes a program and data as input and runs the program on the data. An evolving algebra for a programming langauge describes this type of universal algorithm, thus giving an operational semantics for the programming language.

These types of semantic specifications may be provided on several abstraction levels for the same language. Having several such algebras is useful, for one can examine the semantics of a particular feature of a programming language at any desired level of abstraction, with unnecessary details omitted.

Evolving algebras have been used to provide operational semantics for Modula-2 [Mor], Occam [GMs], Prolog [Bo1, Bo2, Bo3, BR1, BR2], Prolog III [BS], and Smalltalk [Bl]. This technical report describes such a universal machine for the C programming language [KR].

0.2 Required Knowledge

We assume a basic familiarity with evolving algebras; no knowledge beyond that given in [Gu3] will be assumed. Knowledge of C is not necessarily for understanding (though it may be helpful), since we explain all relevant aspects of C as we proceed.

0.3 Separation of Concerns

Our primary concern here is with programming language semantics, not syntax. Consequently, we will assume that all syntactic information regarding a given program is available to us at the beginning of the computation through static functions of the algebra which contain that information.

We will also assume that our algebra will evolve without regard to resource bounds; while such an assumption does not reflect the resource constraints present during any computational activity, it allows us to focus more clearly on our interest in semantics. Resource management may be added to an evolving algebra without undue difficulty: see [Gu2] for further information and [Mor] for an example of resource management in an evolving algebra.

0.4 Abstraction Levels

In our report we will present a series of evolving algebras which model fragments of the C programming language. Each algebra will be presented as a refinement of the previous algebra. The final revised algebra will describe the C programming language in its entirety.

Our algebras will describe the C programming language at the following levels of abstraction.

1. Statements (*e.g.* `if`, `for`)
2. Expressions
3. Memory allocation and initialization
4. Function invocation and return
5. Memory structure

Note: This is a preliminary version of this report. We gratefully acknowledge the comments made on earlier drafts of this report by Raghu Mani, Arnd Poetzsch-Heffter, and Dean Rosenzweig.

1 Algebra One: C Statement Algebra

We now present our first evolving algebra: an algebra concerned with modeling control statements within C. In our presentation of this algebra (as with all succeeding algebras), we will occasionally present the context-free grammar rules from [KR] to show the syntactic form of the construct under consideration.

1.1 Initial Universes and Functions

There are certain common universes and functions which will be used in our algebras. We present most of those elements (whose usefulness hopefully is apparent to the reader) in the following sections.

1.1.1 Program Values

We define a universe *results* to be the universe of values which may appear as the "result" of a computation. We will specify more precisely the constituents of this universe in Algebra Two.

1.1.2 Program Representation and Execution

We define a universe *tasks* of elements representing tasks which must be accomplished by the program interpreter. The notion of "task" is a general one: a task may be the execution of a statement, initialization of a variable, or the evaluation of an expression. As new types of tasks are added to this universe, we will describe them.

At various times, we will need certain internal information to describe the nature of a given task or computational process. We accordingly will define a universe *internals*, whose elements will be used to represent this information. We will specify the elements of *internals* as we proceed.

We define dynamic zero-ary functions (hereafter *distinguished elements*) *CurTask: tasks* and *PrevTask: tasks* (that is, functions with null domain and range *tasks*) which indicate the current and previous task being executed, respectively.

To maintain the sequence of tasks to be accomplished, we define a function *NextTask: tasks* → *tasks* which indicates the next task (i.e. statement or expression) to be performed once the specified task has been completed. We will further constrain *NextTask* as we proceed.

We define a function *TestValue: tasks* → *values* which indicates the value of certain expression tasks. For purposes of this algebra, we will assume that the *TestValue* function is an external function, whose values are determined by an oracle external to the evolving algebra. This will allow us to show how computed values are used by control statements in C while delaying our discussion of how those values are computed.

1.2 Abbreviation: Moveto

As we present our algebras, we will occasionally define some transition rule abbreviations which will improve the readability of our transition rules. These notations are used solely for readability; it is assumed that the abbreviations may be replaced at any time by the appropriate transition rules.

An event which occurs frequently within our transition rules is the following: We wish to transfer control to a particular task, modifying the *CurTask* and *PrevTask* distinguished elements appropriately. The *Moveto(Task)* abbreviation will be used to accomplish this task. Its definition is given in Figure 1.

CurTask := Task
PrevTask := CurTask

Figure 1: Definition of the abbreviation *Moveto(Task)*.

1.3 Statement Classification in C

According to [KR], there are six categories of statements in C:

1. Expression statements, which call for the evaluation of the associated expression.
2. Compound statements, consisting of a (possibly empty) list of local variable declarations and a (possibly empty) list of statements.
3. Selection statements (`if` and `switch` statements).
4. Iteration statements (`for`, `while`, and `do-while` statements).
5. Jump statements (`goto`, `continue`, `break`, and `return` statements).
6. Labeled statements (`case` and `default` statements used within the scope of a `switch` statement, and targets of `goto` statements).

Each of these statement categories will be represented in our transition rules by a set of tasks, linked together by certain functions (such as *NextTask*). We define a static function *TaskType: tasks* $\rightarrow$ *internals* which indicates the action to be performed by the task. We will describe the range of the *TaskType* function as we proceed.

We now consider each of these statement categories in turn.

1.4 Expression Statements

The relevant context-free grammar rules are as follows:

expression-statement $\rightarrow$;
expression-statement $\rightarrow$ *expression* ;

To process an expression statement, we need to evaluate the attached expression (if any), although we will not use the result of the evaluation. While this may seem wasteful, note that the evaluation of an expression in C may generate desirable "side-effects" (such as assigning a value to a variable).

We have asserted that at this level of abstraction, the *TestValue* function will indicate the proper values of any expressions encountered in the program. Consequently, our algebra will simply proceed to the next task in the execution sequence without performing any additional tasks.

The transition rules for expression tasks are shown in Figure 2.

---

if *TaskType(CurTask) = expression* **then**
    Moveto(NextTask(CurTask))
endif

Figure 2: Transition rules for expression tasks.

1.5 Compound Statements

The most general form of the relevant context-free grammar rule is as follows:

compound-statement → { *declaration-list statement-list* }

(Note that the declaration list and/or the statement list may be empty.)

Since we have defined the *NextTask* function as our means of controlling the order in which tasks are processed, we have no direct need for rules concerning compound statements. We simply assume that each statement or declaration in a compound statement is linked to the others by means of the *NextTask* function, and allow our standard procedure of moving between tasks (*i.e.* *Moveto(NextTask(CurTask))*) to handle the problem of maintaing the proper task sequence for a compound statement.

1.6 Selection Statements

There are two major types of selection statements: `if` statements and `switch` statements. We consider each in turn.

1.6.1 The `if` Statement

The relevant context-free grammar rules are as follows:

selection-statement → `if` `(` *expression* `)` *statement*
selection-statement → `if` `(` *expression* `)` *statement* `else` *statement*

The semantics of the `if` statement are fairly simple; one begins execution of an `if` statement by evaluating the attached expression. If the value of the expression is non-zero (*i.e.* true), the statement immediately following the expression is executed. If the value of the expression is zero and an `else` clause is present, the statement following the `else` is executed. Otherwise, control passes to the statement following the `if` statement.

We define static functions *TrueTask: tasks* → *tasks* and *FalseTask: tasks* → *tasks* which indicate the task to be performed if the guard of the `if` statement evaluates to *true* (or *false*). (We will also use these functions in other contexts in our algebra).

We will represent the branching decision made in the `if` statement by an element of the *tasks* universe for which the *TaskType* function returns *test*.

We will represent an `if` statement in our algebra by the graph shown in Figure 3, where the ovals represent tasks, the arcs represent unary functions, and the boxes represent subgraphs.

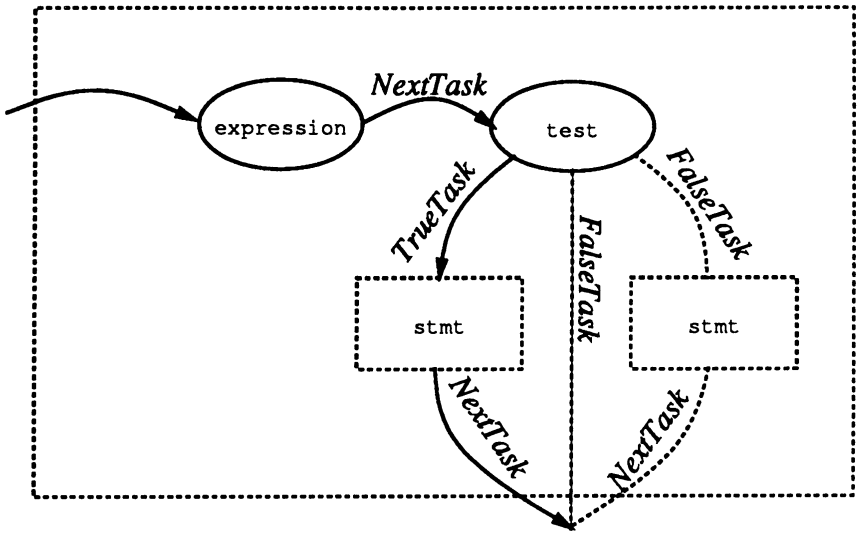

Figure 3: Pictorial description of the `if` statement.

If an `else` clause is not present in an `if` statement, the corresponding task graph omits the right-hand side of Figure 3, with the *FalseTask* function connecting the *test* node to the next task outside of the `if` statement.

Thus, all that remains is to describe the transition rules for tasks of type *test*. These transition rules are shown in Figure 4.

---

if *TaskType(CurTask)* = *test* **then**
  if *TestValue(CurTask)* $\neq$ *0* **then**
    Moveto(TrueTask(CurTask))
  endif
  if *TestValue(CurTask)* = *0* **then**
    Moveto(FalseTask(CurTask))
  endif
endif

Figure 4: Transition rules for *test* tasks.

1.6.2 The `switch` statement

The context-free grammar rule for a `switch` statement is as follows:

selection-statement → `switch` `(` *expression* `)` *statement*

The desired behavior of the `switch` statement is as follows: the expression is evaluated, and within the attached statement (usually a statement block), all statements are skipped (i.e. not executed) until a labeled statement is found whose label matches the value of the expression, or

until a statement labeled **default** is found. Once such a statement has been found, statement execution continues normally.

Since all labels on **case** statements are constant expressions, once the expression of a **switch** statement has been evaluated, there is exactly one statement within the scope of the **switch** to which control can pass. If there is a **case** statement within the **switch** which matches the value of the expression, control passes to the first such **case** statement, or to the first **default** statement if it occurs before the first matching **case**. If no **case** statements match the target expression, control passes to the first **default** statement (if one exists). If none of these apply, control proceeds to the first statement following the **switch**.

Consequently, we define a function *SwitchTask: tasks* $\times$ *results* $\rightarrow$ *tasks* which indicates the task to be executed next for the given **switch** statement and expression value.

We will represent a **switch** statement in our algebra by the graph shown in Figure 5.

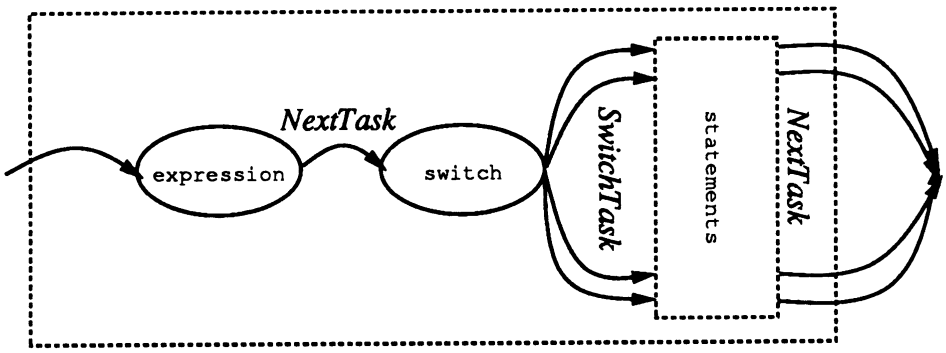

Figure 5: Pictorial description of the **switch** statement.

The rules for **switch** tasks are given in Figure 6.

---

if *TaskType(CurTask) = switch* **then**
    Moveto(SwitchTask(CurTask, TestValue(CurTask)))
endif

Figure 6: Transition rules for **switch** tasks.

1.7 The while Statement

There are three different types of iteration statements in C. We consider each of them in turn.

The relevant context-free grammar rule is as follows:

iteration-statement $\rightarrow$ **while** (*expression*) *statement*

To process a **while** statement, we repeatedly evaluate the attached expression until the value of the expression becomes zero. Each time that the evaluated expression has a non-zero value, we execute the attached statement.

We will represent a **while** statement in our algebra by the graph shown in Figure 7. Since we have re-used the *test* task first introduced in our consideration of **if** statements, we do not need to add any transition rules to our algebra at this time to handle **while** statements.

Note that it is possible to enter a **while** loop by means of a **goto** statement, thus circumventing the initial test of the expression at the beginning of the loop. [KR] do not give specific semantics for such behavior. Under our model, execution would continue as if the loop had been entered normally (*i.e.*, after completion of the attached statement, control returns to the expression to be re-evaluated). We believe this is a reasonable interpretation of such an event.

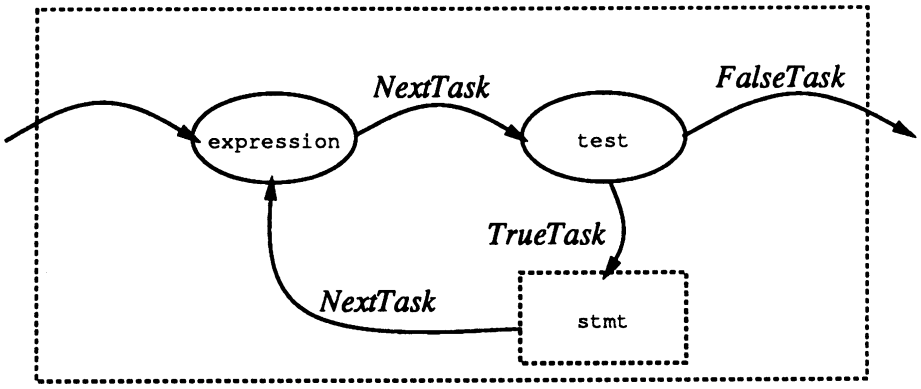

Figure 7: Pictorial description of the **while** statement.

1.8 The do-while Statement

The relevant context-free grammar rule is as follows:

iteration-statement → **do** *statement* **while** **(** *expression* **)** **;**

Processing a **do-while** statement is identical to processing a **while** statement except that the expression and attached statement are visited in the opposite order. We will thus represent a **do-while** statement in our algebra by the graph shown in Figure 8. Note the similarity between this structure and that of the **while** loop.

Again, since we have re-used elements of the algebra previously introduced, we will not need to add additional transition rules to correctly handle **do-while** statements.

1.9 The for Statement

The most general form of the relevant context-free grammar rule is as follows:

iteration-statement → **for** **(** *expression* **;** *expression* **;** *expression* **)** *statement*

For convenience in the following discussion, we will refer to the three expressions of the **for** loop as the initializing, testing, and updating expressions, respectively.

The behavior of a **for** loop may be described as follows: first, the initializing expression is evaluated. Afterwards, the testing expression is evaluated. If the value of the testing expression is non-zero (*i.e.*, true), the sub-statement and the updating expression are evaluated, in that order, and the testing expression is re-evaluated. If the value of the testing expression is zero (*i.e.*, false), control passes to the statement following the **for** loop.

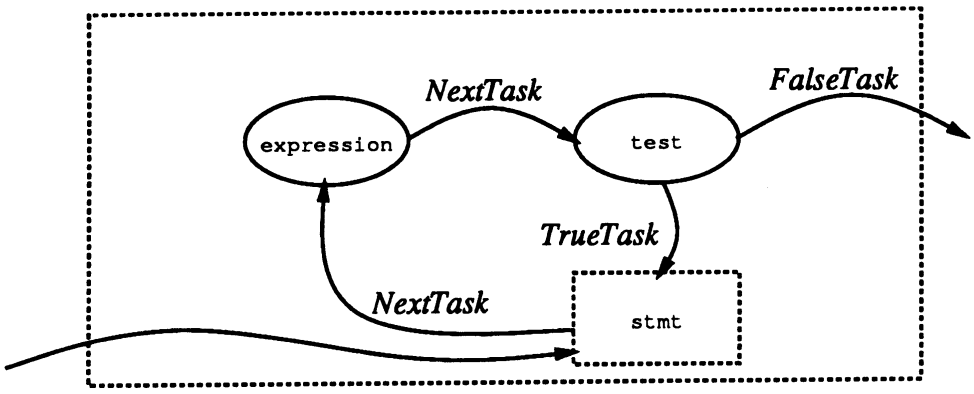

Figure 8: Pictorial description of the `do-while` statement.

We will represent the structure of a `for` statement in our algebra by the graph shown in Figure 9. Again, since we have re-used elements of the algebra previously presented, we will not need to present any further transition rules at this time.

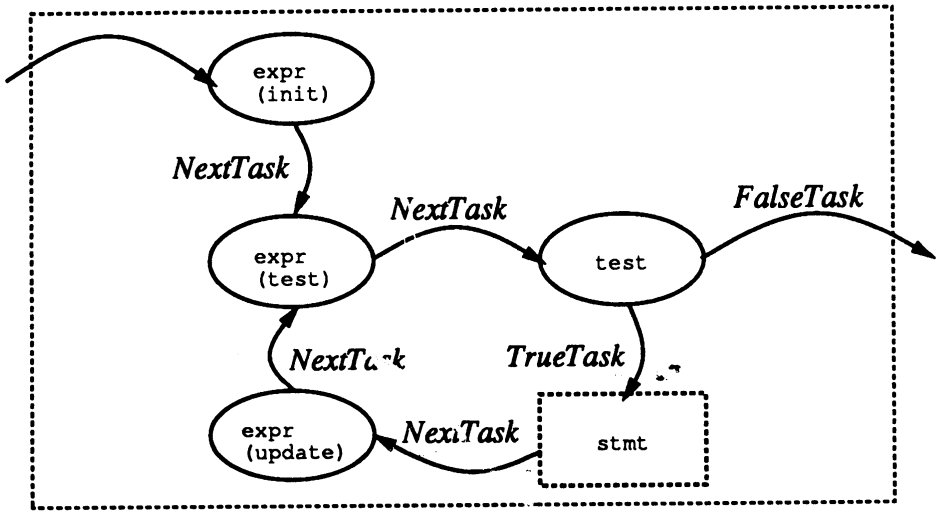

Figure 9: Pictorial description of the `for` statement.

Note that in C, any (or all) of the three expressions (initializing, testing, and updating) may be omitted. We assert that the algebra structures for `for` loops missing one or more of these expressions simply omit the corresponding task, with the *NextTask* function indicating the correct task to be executed next. In the event that the testing expression is omitted, we omit both the testing expression and the *test* branching task, creating an infinite loop (which may still be broken through the use of jump statements).

1.10 Jump Statements

The relevant context-free grammar rules are as follows:

jump-statement → `goto` *identifier* `;`
jump-statement → `continue ;`
jump-statement → `break ;`
jump-statement → `return ;`

jump-statement → **return** *expression* ;

We defer discussion of the **return** statement until our discussion of function invocation and return.

Each of these jump statements (with the exception of the **goto** statement) may be considered as a command indicating that another task in the task graph is to be considered as "completed". This statement may be uniquely identified through a syntactic analysis of the program:

- If the jump statement is a **continue** statement, the jump statement indicates that control should be passed to the closest iteration statement enclosing the jump statement, and the statement block associated with the iteration statement should be considered as completed (i.e. reporting).
- If the jump statement is a **break** statement, the jump statement indicates that the closest iteration statement should be considered as completed (i.e. reporting), and control should be passed to the parent task of that iteration statement.

The **goto** statement itself indicates directly the next statement which should receive computational control.

We assume that the *NextTask* function contains the above information for jump statement tasks. Thus, the transition rule for jump statements (shown in Figure 10) is trivial.

---

if *TaskType(CurTask) = jump* **then**
    Moveto(NextTask(CurTask))
endif

Figure 10: Transition rules for jump statements

1.11 Labeled Statements

The context-free grammar rules for labeled statements are as follows:

labeled-statement → *identifier* : *statement*
labeled-statement → **case** *constant-expression* : *statement*
labeled-statement → **default** : *statement*

Labeled statements provide a means for identifying the targets for control transfer in **goto** and **switch** statements, and are otherwise ignored by a running C program. We thus assume that the *NextTask* and *SwitchTask* functions will return the appropriate un-labeled statement (task) for a given argument, thus eliminating the need for transition rules to explicitly process labels on statements.

1.12 Initial State

It remains to describe the initial state of the relevant dynamic functions of the algebra. The only dynamic functions we have described to this point are *CurTask* and *PrevTask*.

We assert that initially, *CurTask* indicates the first subtask of the first statement of the program. *PrevTask* has the distinguished value $\perp$, indicating no previous task.

This concludes our presentation of an evolving algebra for statements.

2 Algebra Two: C Statements and Expressions

We now present a second evolving algebra, a refinement of the first, which handles the evaluation of expressions.

We will replace each occurrence of a task of type *expression* from the first algebra with a collection of tasks describing the structure of the expression. Our transition rules will describe how such expressions should be evaluated.

Consequently, we will now treat the *TestValue* function as an internal, dynamic function which we will update directly.

2.1 Abstraction Level of C

We will consider the evaluation of expressions in C at a fairly high level in this algebra. This high level of abstraction can be described by the following simplifying assumptions:

- Memory is organized into arbitrarily large units, each of which can hold any value which a C program might wish to store in memory.
- No function invocations are allowed.
- All local and global variables are automatically allocated memory (and initialized, if necessary).

As we refine our evolving algebra descriptions, we will gradually eliminate these abstractions, resulting ultimately in an evolving algebra for C in its entirety.

2.2 New Universes and Functions

2.2.1 Computational Results

The definition of C provides for several different types of fixed-point integer variables, such as `int`, `short int`, `unsigned int`, etc., whose possible values are determined by the specific implementation being modeled. We define the universe *integer* to be the set of integers (i.e. {...,-2,-1,0,1,2,...}). We define universes *int*, *short-int*, *unsigned-int*, etc. to be disjoint sets corresponding to those integers that may be stored in a variable of the corresponding type (`int`, `short int`, `unsigned int`, etc.).

Similarly, the definition of C provides for three different types of floating-point integer variables: `float`, `double`, and `long double`, whose possible values are determined by the specific implementation being modeled. We define three universes *float*, *double*, and *long-double*, disjoint sets corresponding to those numbers which may be stored in a variable of the corresponding type.

We define a universe *bytes* which consists of those values which may be stored in a variable of type `char`. (This universe is usually identical to {0,1,...,255}, but we prefer the more general definition.)

We define a universe *addresses* which consists of elements corresponding to those positive integers corresponding to valid memory locations in the computer system being modeled. This universe is also the universe of values which may be stored in a pointer-type variable.

All of the above universes come with the usual ordering function $<$ and arithmetic functions $+$, $-$, $\times$, and / (division), as well as the unary negation operation defined upon them. Integer sets also come with the additional functions & (bitwise AND), | (bitwise inclusive OR), ^ (bitwise exclusive OR), and ~ (bitwise one's complementation) defined upon them. (We will use conventional infix notation for these functions as they are used in our evolving algebra.)

We define universes *array*, *struct*, and *union* to contain all possible arrays, structures, and unions which may be represented in a program to be modeled by our abstract machine. Note that we will not directly describe the members of these universes; for example, an `array` need not be represented as an ordered set or a `struct` as a tuple.

We finally define the universe *results* to be the union of all previous universes in this section: that is, the universe of values which may appear as the "result" of a computation.

To represent the memory store of the system, we define a dynamic function *Memory: addresses* $\rightarrow$ *results* which indicates the values stored at different locations in memory. (Note that in this algebra, an entire value of interest may be stored in one memory cell. When we create further refinements of this algebra we will modify the definition of *Memory* to accomodate a byte-based view of memory.)

For tasks which involve the evaluation of an expression, we define a universe *typename* whose elements represent the different types of storable elements. We also define a static function *ValueType: tasks* $\rightarrow$ *typename* which indicates, for an expression task, the type of the resulting value when the expression has been evaluated.

2.2.2 Functions Relating To Tasks

We define a static function *ConstVal: tasks* $\rightarrow$ *results* which indicates the values of constants embedded within the program.

We define functions *LeftTask, RightTask: tasks* $\rightarrow$ *tasks* which indicate the left and right operands of binary operators whose order of evaluation is not defined within C.

We define functions *LeftValue, RightValue: tasks* $\rightarrow$ *results* which indicate the results of evaluating the left and right operands of binary operators with ambiguous evaluation order. We assert that *LeftValue* and *RightValue* are initially defined to be equal to $\perp$ everywhere.

Similarly, we define a function *OnlyValue: tasks* $\rightarrow$ *results* which indicates the result of evaluating the single operand of a unary operator.

We define a function *Parent: tasks* $\rightarrow$ *tasks* which indicates for a given task representing an expression which is part of another expression (*i.e.* a task representing a subexpression) the corresponding "parent" expression (if any). We assert that for expressions which are not contained in any other expressions, *Parent* returns the corresponding *test* task which uses the expression (if one exists) or $\perp$ (if none exists). We define a corresponding function *WhichChild: tasks* $\rightarrow$ {*left, right, only, test, none*} (where *left, etc.* are members of the *internals* universe) to indicate which sub-expression of the parent expression (if any) is being considered.

2.3 Abbreviation: Memory Assignments

With our current assertions regarding the size of individual memory units, the action of updating a memory location (for example, due to the affects of an assignment operator) can be modeled by a simple assignment statement involving the *Memory* function. It turns out to be convenient to define

an abbreviation *DoAssign (address, value, type)* to use in our transition rules. When we re-visit our assumptions regarding memory, we will need to re-define this abbreviation to account for the added complexity. For now, we may safely assume the definition of the *Memory* abbreviation given in Figure 11.

Memory(address) := value
CurTask := NextTask(CurTask)

Figure 11: Definition of the *DoAssign (address, value, type)* abbreviation

2.4 Abbreviation: SetValue

During the processing of tasks corresponding to expression evaluation in our algebra, we will often need to assign the value of an evaluated expression (say *CurTask*) to the appropriate storage function in the parent expression (either *LeftValue(Parent(CurTask))* or *RightValue(Parent(CurTask))*).

We will use the *SetValue* abbreviation to accomplish this task. Its definition is given in Figure 12.

---

if *WhichChild(CurTask) = left* **then**
    LeftValue(Parent(CurTask)) := value
endif
if *WhichChild(CurTask) = right* **then**
    RightValue(Parent(CurTask)) := value
endif
if *WhichChild(CurTask) = only* **then**
    OnlyValue(Parent(CurTask)) := value
endif
if *WhichChild(CurTask) = test* **then**
    TestValue(Parent(CurTask)) := value
endif

Figure 12: Definition of the *SetValue(value)* abbreviation.

2.5 Abbreviation and Definitions: EvaluateOperands

In C, many binary operators (including the assignment operator "`=`") do not have a defined order of evaluation: that is, either the left operand or the right operand may be evaluated first. While in many expressions this ambiguity in evaluation order is irrelevant, it is relevant in other situations where the operands may generate conflicting side-effects, such as the statement "`a[i] = i++;`", where the variable `i` is both accessed and updated.

Such ambiguities in operator evaluation order may be used by optimizing compilers, for example, to generate code which minimizes the total quantity of resources required to perform a particular computation [ASU]. Thus, [KR] deliberately does not specify an evaluation order for certain operands, leaving that decision to each particular implementation of a compiler for C.

Thus, it is perfectly legal within C to write expressions such as "`a[i] = i++;`", although the results of such an evaluation may vary from compiler to compiler (or even within a given compiler). Writing code that depends upon the order of evaluation of expressions such as these is therefore unwise; however, it is legal code, and thus our algebra must take this ambiguity into account in its descriptions.

In particular, since the order of evaluation of such operators is undefined, our algebraic description will need to be powerful enough to accomodate any possible means of deciding the evaluation order of an expression's operands. In particular, our algebra must be able to handle the situation where this decision is made dynamically during the course of the computation. While we believe most compilers make this decision at compile-time and not at run-time, [KR] do not specifically address this issue. We will thus provide a mechanism for making this decision dynamically. (If this decision is always made statically in a particular system, the algebra may be explicitly structured to incorporate those structures into the task graph).

We assert that expressions with binary operators of ambiguous evaluation order are represented in our algebra as shown in Figure 13.

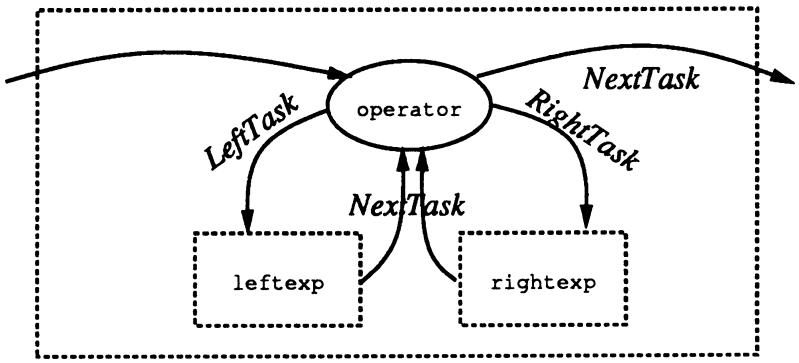

Figure 13: Pictorial representation of binary operators.

We define a function *Visited: tasks* → {*left,right, both,neither*} which will indicate which subexpressions have been evaluated at a given moment. Initially, *Visited* has the value *neither* for all tasks in the algebra.

Our algebra will operate on tasks corresponding to such operators as follows:

1. Upon first encountering the task, an external function *ChooseTask* will be consulted to determine which subtask to evaluate first.

2. When that subtask is about to complete its work and return control to the operator task, control will be passed directly to the appropriate other subtask.

3. When both subtasks have been evaluated, control will return to the operator task, which will perform the desired operation and proceed to the next task in the algebra.

To handle the part of this behavior focused upon the operator task, we define the *EVALUATE OPERANDS* abbreviation as shown in Figure 14.

To handle the part of this behavior focused upon movement within the subtasks, we redefine the *Moveto(Task)* abbreviation as shown in Figure 15.

```
EVALUATE OPERANDS WITH
      statements
END EVALUATE

if Visited(CurTask) = neither then
      if ChooseTask(CurTask) = LeftTask(CurTask) then
            Visited(CurTask) := left
      endif
      if ChooseTask(CurTask) = RightTask(CurTask) then
            Visited(CurTask) := right
      endif
      Moveto(ChooseTask(CurTask))
endif
if Visited(CurTask) = both then
      Visited(CurTask) := neither
            statements
endif
```

Figure 14: Definition of the *EVALUATE OPERANDS* abbreviation.

```
PrevTask := CurTask
if Visited(Task) = neither then
      CurTask := Task
endif
if Visited(Task) = both then
      CurTask := NextTask(Task)
endif
if Visited(Task) = left then
      CurTask := RightTask(Task)
endif
if Visited(Task) = right then
      CurTask := LeftTask(Task)
endif
```

Figure 15: Revised definition of the *Moveto(Task)* abbreviation.

2.6 Comma Operators

The relevant context-free grammar rule is as follows:

expression → *expression* , *assignment-expression*

In a comma-delimited expression, both expressions are evaluated, left to right, and the value of the second expression becomes the value of the parent expression. (Though it may seem wasteful to evaluate the first expression and discard its value, recall that expressions in C may generate side-effects, such as assignment, which can be desirable.)

We will thus represent comma expressions simply as a sequence of two expressions, linked by the *NextTask* function. Thus, no specific additional transition rules are needed to process a comma operator.

2.7 Assignment Expressions

There are several types of assignment operators in C. These operators may be placed into two general categories: those that perform a mathematical operation as well as an assignment, and those that only perform an assignment. We call the later a "simple assignment" operator.

The context-free grammar rule for a simple assignment is as follows:

assignment-expression → *unary-expression* = *assignment-expression*

In a simple assignment expression, the value of the right expression is copied directly into the specified memory location (the value of the left expression). The value of the expression is the copied value.

The transition rules for assignment operators are given in Figure 16.

---

if *TaskType(CurTask) = simple-assignment* **then**
    EVALUATE OPERANDS WITH
        SetValue(RightValue(CurTask))
        DoAssign(LeftValue(CurTask),RightValue(CurTask),ValueType(CurTask))
    END EVALUATE
endif

Figure 16: Transition rules for simple assignments.

There are several other assignment operators ("`*=`", "`/=`", "`+=`", "`-=`", and so on) which apply the specified mathematic operation to the value of the right expression and the value stored in memory at the location specified by the left expression. The resulting value is then copied into memory at the location specified by the left expression. (Thus, "`i *= 2;`" has the same effect as "`i = i * 2;`".)

We present the transition rules for the multiplicative assignment operator "`*=`" in Figure 17 as a representative example for this group of operators.

One should note that in C, one may add to and subtract from pointers using the "`+=`" and "`-=`" operators, with the result incrementing (or decrementing) the pointer variable the specified number

```
if TaskType(CurTask) = multiplicative-assignment then
    EVALUATE OPERANDS WITH
        SetValue(RightValue(CurTask) * Memory(LeftValue(CurTask)))
        DoAssign(LeftValue(CurTask)),
            RightValue(CurTask) * Memory(LeftValue(CurTask)),
            ValueType(CurTask))
    END EVALUATE
endif
```

Figure 17: Transition rules for multiplicative assignments.

of positions (dependent upon the size of the object to which the pointer points). Since our current abstraction of C asserts that all objects may reside in a single memory location, this behavior may be simulated by normal addition and subtraction. We will need to refine this rule when we refine our model of memory.

2.8 Conditional Expressions

The context-free grammar rule for conditional expressions is as follows:

conditional-expression → *logical-OR-expression* ? *expression* : *conditional-expression*

To evaluate a conditional expression, one begins by evaluating the left expression. If the resulting value is non-zero (i.e. true), the center expression is evaluated and the resulting value becomes the value of the conditional expression. Otherwise, the right expression is evaluated and the resulting value becomes the value of the conditional expression. Exactly one of the expressions following the question mark is evaluated.

We will represent conditional expression in our algebra in a manner similar to that in which we represent conditional statements, as shown in Figure 18.

We assert that the tasks corresponding to the second and third sub-expressions will update the appropriate *Value* function for the parent expression upon completion of the evaluation of the subexpression.

Since we have used constructs previously introduced in our algebra, we do not need to present any new transition rules at this time.

2.9 Logical OR Expressions

The context-free grammar rule for logical OR expressions is as follows:

logical-OR-expression → *logical-OR-expression* || *logical-AND-expression*

To evaluate a logical OR expression, we begin by evaluating the leftmost expression. If the resulting value is non-zero, the value of the expression is one (1) and the second expression is not evaluated. If the resulting value is zero, the value of the logical-OR-expression is the value returned

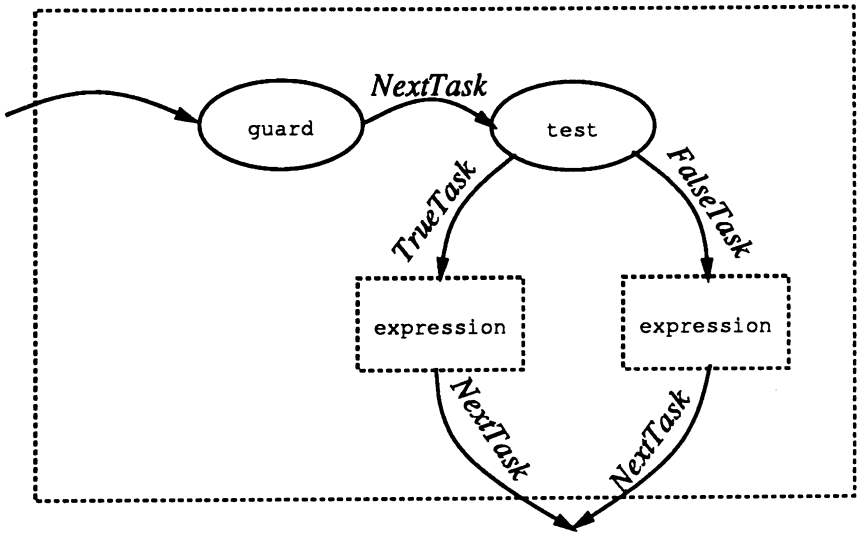

Figure 18: Pictorial description of a conditional expression.

by the right expression, coerced to the logical values zero (for false) or one (for true). Any non-zero value is coerced to one.

To represent a logical OR expression, we will introduce two new task types, *OR* and *makeBool*, whose behavior will be described in a moment. We will represent a logical OR expression as shown in Figure 19.

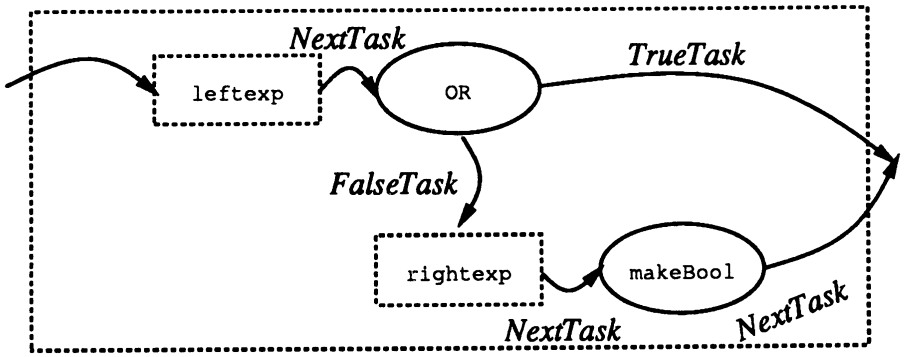

Figure 19: Pictorial description of a logical OR expression.

It remains to describe the behavior of the *OR* and *makeBool* tasks. The *OR* task will examine the value of the left expression. If the value is non-zero, the *OR* task will set the value of the expression to 1 and end processing of the expression. If the value is zero, the *OR* task will pass control to the tasks which evaluate the right expression.

The *makeBool* task will examine the value of the right expression and convert it into a Boolean value. If the value of the right expression is non-zero, *makeBool* will yield the value 1. Otherwise, *makeBool* will yield the value 0.

The transition rules for *OR* and *makeBool* tasks are given in Figures 20 and 21.

2.10 Logical AND expressions

The context-free grammar rule for logical AND expressions is as follows:

```
if TaskType(CurTask) = OR then
    if OnlyValue(CurTask) ≠ 0 then
        SetValue(1)
        Moveto(TrueTask(CurTask))
    endif
    if OnlyValue(CurTask) = 0 then
        Moveto(FalseTask(CurTask))
    endif
endif
```

Figure 20: Transition rules for OR tasks.

```
if TaskType(CurTask) = makeBool then
    if OnlyValue(CurTask) ≠ 0 then
        SetValue(1)
    endif
    if OnlyValue(CurTask) = 0 then
        SetValue(0)
    endif
    Moveto(NextTask(CurTask))
endif
```

Figure 21: Transition rules for makeBool tasks.

logical-AND-expression → *logical-AND-expression* && *inclusive-OR-expression*

Processing here is similar to the case for logical OR expressions. The left expression is evaluated, and if the resulting value is zero, the value of the parent expression becomes zero. Otherwise, the value of the right expression becomes the value of the logical AND expression.

We will represent a logical AND expression in an identical manner to a logical OR expression, with a task of a new type *AND* replacing the *OR* task, and with the reversal of the values of the *TrueTask* and *FalseTask* functions upon that new task. A pictorial representation of this structure is given in Figure 22.

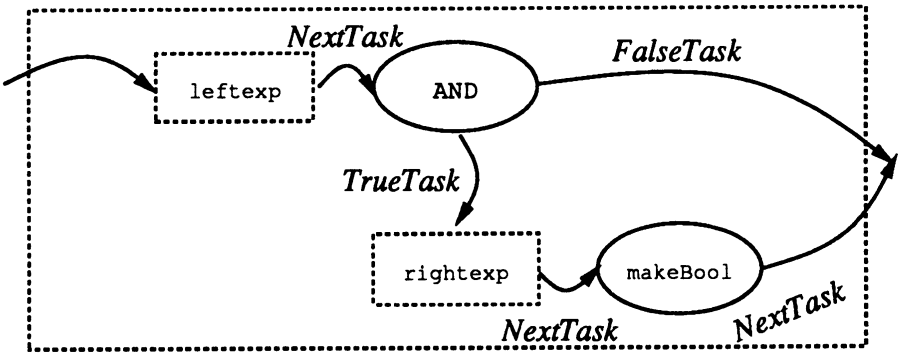

Figure 22: Pictorial description of a logical AND expression.

The transition rules for *AND* tasks are given in Figure 23.

---

if *TaskType(CurTask) = AND* **then**
  if *OnlyValue(CurTask) = 0* **then**
    SetValue(0)
    Moveto(FalseTask(CurTask))
  endif
  if *OnlyValue(CurTask)* $\neq$ *0* **then**
    Moveto(TrueTask(CurTask))
  endif
endif

Figure 23: Transition rules for AND tasks.

2.11 General Mathematic Expressions

There are a large number of mathematic expressions in C involving binary operators ("*", "/", "+", "-", etc.) whose semantics are virtually identical.

We assume that these operators are available to us as infix functions within our evolving algebras. Thus, evaluation of these expressions is handled by evaluating both of the operands and applying the appropriate operator. We present the transition rules for multiplication in Figure 24

as a representative of these types of expressions. (We omit the transition rules for the other binary operators of this form to save space.)

---

if *TaskType(CurTask) = multiplication* **then**
    EVALUATE OPERANDS WITH
        *SetValue(LeftValue(CurTask) * RightValue(CurTask))*
        Moveto(NextTask(CurTask))
    END EVALUATE
endif

Figure 24: Transition rules for multiplication expressions.

2.12 Casting Expressions

The context-free grammar rule for casting expressions is as follows:

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

In order to cast an expression from one type into another, we need a static function *Convert: typename* × *typename* × *values* → *values* which converts elements from one universe into the corresponding elements of another universe. For example, *Convert(float,int,X)* would return the "closest" integer to X (assuming X is a floating-point value). Note that the meaning of "closest" is implementation-defined.

Thus, to perform a cast, we evaluate the expression to be cast and use the *Convert* function to generate the proper return value. We assume our task sequence presents the expression to be cast before the task which performs the casting; thus, we may assume the expression to be evaluated already and its value accessible to us.

The transition rules for casting expressions are shown in Figure 25.

---

if *TaskType(CurTask) = cast* **then**
    SetValue(Convert(ValueType(PrevTask),
        ValueType(CurTask),
        OnlyValue(CurTask)))
    Moveto(NextTask(CurTask))
endif

Figure 25: Transition rules for casting expressions.

2.13 Pre-Increment and Pre-Decrement

The transition rules for pre-increment and pre-decrement expressions are as follows:

unary-expression → `++` *postfix-expression*
unary-expression → `--` *postfix-expression*

In a pre-increment (resp. pre-decrement) expression, the value stored at the memory location indicated by the postfix expression is incremented (decremented) by one and stored into that memory location; the incremented (decremented) value then becomes the value of the unary expression. (Note again that these rules will need to be modified slightly when we revise our model of memory.)

The transition rules for pre-increment expressions are shown in Figure 26. The transition rules for pre-decrement expressions are similar to those presented here, and thus omitted.

---

if *TaskType(CurTask) = pre-increment* **then**
    SetValue(Memory(OnlyValue(CurTask))+1)
    DoAssign(OnlyValue(CurTask),
        Memory(OnlyValue(CurTask)) + 1,
        ValueType(CurTask))
endif

Figure 26: Transition rules for the pre-increment operator.

2.14 Addresses

The context-free grammar rule for the address operator is as follows:

unary-expression → `&` *cast-expression*

The `&` operator passes back as its result the address of the memory location indicated by the cast expression.

As we evaluate different expressions, it becomes important to determine for expressions referring to objects in memory (that is, *lvalues*), whether we desire to use the memory reference or the object being referenced in our calculations. (For example, in the assignment statement "`a = b;`", we need the memory reference or *lvalue* of variable `a`, but the object being referenced or *rvalue* of variable `b`.)

We thus add two new elements *lvalue* and *rvalue* to the *internals* universe and define a static function *LRValueType: tasks* → {*lvalue, rvalue*} which indicates which of the two pieces of information should be computed for a given task.

We will assert that the static *LRValueType* function has the value *lvalue* throughout the cast expression's subtasks; thus, the value returned through evaluation of the cast expression is the address (and not the value) of the cast expression in memory. We simply pass this address up the task graph. The resulting simple transition rule for the addressing operator is shown in Figure 27.

```
if TaskType(CurTask) = address then
    SetValue(OnlyValue(CurTask))
    Moveto(NextTask(CurTask))
endif
```

Figure 27: Transition rules for the addressing operator.

2.15 De-Referencing

The context-free grammar rule for de-referencing expressions is as follows:

unary-expression → * *cast-expression*

To de-reference an expression, if the unary-expression is an rvalue, we evaluate the expression and then use the *Memory* function to return the value stored in memory at the indicated location. Otherwise, we simply return the address indicated by the cast expression (since the expression is an lvalue and requires that a pointer be returned to the parent expression). The transition rules for de-referencing are presented in Figure 28.

```
if TaskType(CurTask) = de-referencing then
    if LRValueType(CurTask) = rvalue then
        SetValue(Memory(OnlyValue(CurTask)))
    endif
    if LRValueType(CurTask) = lvalue then
        SetValue(OnlyValue(CurTask))
    endif
    Moveto(NextTask(CurTask))
endif
```

Figure 28: Transition rules for de-referencing expressions.

2.16 Mathematical Unary Operators

The context-free grammar rules for mathematical unary operators are as follows:

unary-expression → + *cast-expression*
unary-expression → - *cast-expression*
unary-expression → ~ *cast-expression*
unary-expression → ! *cast-expression*

We assume that these four unary operators (identity, negation, bitwise negation, and Boolean negation) are available in our algebra. Thus, processing these expressions takes a form similar to that for binary mathematical operators: we evaluate the attached expression and apply the appropriate operator to the resulting value. We present the transition rules for negation in Figure 29 as a representative example of the transition rules for this group.

---

if *TaskType(CurTask) = negation* **then**
    SetValue(- OnlyValue(CurTask))
    Moveto(NextTask(CurTask))
endif

Figure 29: Transition rules for the negation unary operator.

2.17 The `sizeof` Operator

The relevant context-free grammar rules are as follows:

unary-expression → `sizeof` *unary-expression*
unary-expression → `sizeof` `(` *type-name* `)`

Under our current level of abstraction, since all memory locations are large enough to hold any single value, we will assume that the `sizeof` operator, which indicates how many bytes are needed to represent the specified expression or type, always returns the integer one. The relevant transition rules (which will need to be revised when we reconsider the nature of memory locations) are shown in Figure 30.

---

if *TaskType(CurTask) = sizeof* **then**
    SetValue(1)
    Moveto(NextTask(CurTask))
endif

Figure 30: Transition rules for the `sizeof` operator.

2.18 Array References

The context-free grammar rule for array references is as follows:

postfix-expression → *postfix-expression* `[` *expression* `]`

According to [KR], an array reference of the form `a[b]` is identical, by definition, to the expression `*((a)+(b))`.[1] This definition is valid because the name of an array in C may be used as a pointer to the first element of the array.

The transition rules for array references, shown in Figure 31, perform all the work of addition and dereferencing accomplished by the + and * operators. We present the transition rules without discussion, referring the reader to previous sections regarding these operators.

---

if *TaskType(CurTask) = array-reference* **then**
    EVALUATE OPERANDS WITH
        if *LRValueType(CurTask) = lvalue* **then**
            SetValue(LeftValue(CurTask) + RightValue(CurTask))
        endif
        if *LRValueType(CurTask) = rvalue* **then**
            SetValue(Memory(LeftValue(CurTask) + RightValue(CurTask)))
        endif
        Moveto(NextTask(CurTask))
    END EVALUATE
endif

Figure 31: Transition rules for array references.

2.19 Function Invocations

The context-free grammar rule for function invocations is as follows:

postfix-expression → *postfix-expression* (*argument-expression-list*)

Since we have disallowed function invocations for the moment, we will assume that the appropriate *Value* function applied to this node returns an appropriate value for the C function abstraction being modeled; that is, the appropriate *Value* function acts as an external function for this node at this time. Thus, we do not need to perform any activity here at this time and we return control to the parent node. The appropriate transition rules are presented in Figure 32.

2.20 Struct or Union References

The context-free grammar rules for `struct` references are as follows:

postfix-expression → *postfix-expression* . *identifier*
postfix-expression → *postfix-expression* -> *identifier*

The former rule is used when the postfix expression refers to a `struct` or `union` directly; the latter rule is used when the postfix expression refers to a pointer to a `struct` or `union`. Thus, "`a->b`" is equivalent to "`(*a).b`".

[1] Note that this means that `a[b]` and `b[a]` evaluate to the same value.

---

if *TaskType(CurTask) = function-invoke* **then**
    Moveto(NextTask(CurTask))
endif

Figure 32: Transition rules for function invocations.

We assert that the *ConstVal* function applied to the `struct` reference task returns the offset from the beginning of the specified structure to be used in obtaining the address or value of the specified field of the structure. We use this information to return the corresponding memory address or memory value, as specified by the *LRValueType* function. The transition rules for struct references are presented in Figures 33 and 34.

---

if *TaskType(CurTask) = struct-plain-reference* **then**
    if *LRValueType(CurTask) = lvalue* **then**
        SetValue(OnlyValue(CurTask) + ConstVal(CurTask))
    endif
    if *LRValueType(CurTask) = rvalue* **then**
        SetValue(Memory(OnlyValue(CurTask) + ConstVal(CurTask)))
    endif
    Moveto(NextTask(CurTask))
endif

Figure 33: Transition rules for plain structure references.

2.21 Post-Increment and Post-Decrement

The context-free grammar rules for post-increment and post-decrement are as follows:

postfix-expression → *postfix-expression* **++**
postfix-expression → *postfix-expression* **--**

Post-increment (resp. post-decrement) operators are handled in the same manner as pre-increment (pre-decrement) operators except that the sequence of operations is reversed: that is, the value of the parent expression is established before the incrementing (decrementing) takes place.

The transition rules for the post-increment operator are shown in Figure 35. (As before, the transition rules for the post-decrement operator are similar and thus omitted.)

2.22 Identifiers

In order to handle expressions consisting solely of identifiers, we need to have a means of mapping identifiers to their corresponding memory locations. However, in our presentation we have not yet

```
if TaskType(CurTask) = struct-pointer-reference then
    if LRValueType(CurTask) = lvalue then
        SetValue(Memory(OnlyValue(CurTask) + ConstVal(CurTask)))
    endif
    if LRValueType(CurTask) = rvalue then
        SetValue(Memory(Memory(OnlyValue(CurTask)) + ConstVal(CurTask)))
    endif
    Moveto(NextTask(CurTask))
endif
```

Figure 34: Transition rules for pointer structure references.

```
if TaskType(CurTask) = post-increment then
    SetValue(Memory(OnlyValue(CurTask)))
    DoAssign(OnlyValue(CurTask),
        Memory(OnlyValue(CurTask))+1,
        ValueType(CurTask))
endif
```

Figure 35: Transition rules for the post-increment operator.

discussed how memory is allocated to variables.

For now, we shall make use of an external function *FindID: tasks* $\rightarrow$ *addresses* which yield this information. In later algebras we shall eliminate the use of this function.

Having now accounted for addresses in memory, the task of handling an identifier expression is reduced to returning the appropriate address or value, as specified by the *LRValueType* function. The transition rules for identifiers are shown in Figure 36.

---

if *TaskType(CurTask) = identifier* **then**
  if *LRValueType(CurTask) = lvalue* **then**
    SetValue(FindID(CurTask))
  endif
  if *LRValueType(CurTask) = rvalue* **then**
    SetValue(Memory(FindID(CurTask)))
  endif
  Moveto(NextTask(CurTask))
endif

Figure 36: Transition rules for identifiers.

2.23 Constants and Strings

In the case of either a constant or string expression, we assume that the *ConstVal* function, when applied to the string or constant task, returns the appropriate value. The transition rules for constants and strings are given in Figure 37.

---

if *TaskType(CurTask) = constant* **then**
  SetValue(ConstVal(CurTask))
  Moveto(NextTask(CurTask))
endif

if *TaskType(CurTask) = string* **then**
  SetValue(ConstVal(CurTask))
  Moveto(NextTask(CurTask))
endif

Figure 37: Transition rules for constants and strings.

This concludes our second evolving algebra dealing with expressions.

3 Algebra Three: Memory Allocation and Initialization

We now present a third algebra, a refinement of the second algebra, which include the tasks of C dealing with memory allocation and initialization of local variables (called *automatic variables* in C parlance).

3.1 Declarations

As with statements and expressions, we will represent declarations in C as elements of the *tasks* universe, along with several "decoration" functions which describe the nature of the declarations to be performed. We assume that these declarations are linked with their surrounding statement tasks by the *NextTask* function, occuring in their proper sequence at the beginning of compound statements, as well as at the uppermost (i.e. global) level.

Under normal situations, when processing a task corresponding to a declaration, we would like to allocate an appropriate portion of memory and, if an initializing value is present, assign that value to the memory location. We thus define a static function *Initializer: tasks* $\rightarrow$ *tasks* indicating the initializing expression (if any) present. We assert that if no such initializer is present, *Initializer* takes the value $\perp$.

One added complication arises at this time: the presence of static variables (which are allocated and initialized only once). Consequently, we add two new values *static* and *normal* to the *internals* universe and define a static function *DecType: tasks* $\rightarrow$ {*static, normal*} which contains this information. (Note that there are declarations in C which do not reserve memory but serve as syntactic linkage between variables. We omit consideration of such declarations, since their function is wholly syntactic in nature.)

We will need some place to store the current address (if any) that has been assigned for a static variable; consequently, we define a dynamic function *StaticAddr: tasks* $\rightarrow$ *addresses* which will indicate the currently allocated address for the static variable. ($\perp$ indicates no allocation has been performed.)

We introduce an external function *NewMemory: tasks* $\rightarrow$ *addresses* which returns an address in memory to be used for the given declarator node. We use this function, along with the information conveyed by the *DecType* function, to perform the necessary memory allocation.

Finally, we can discuss the transition rules for declarations.

- If the variable is a static variable which has not been established before (indicated by *StaticAddr* having the value $\perp$), we allocate new memory for the variable and (if necessary) evaluate the initializer and assign its value to that memory location.
- If the variable is a static variable which has been established before (indicated by *StaticAddr* having a value other than $\perp$), we use the previous address assigned to the static variable as the new address for the variable, and ignore any initializer present.
- Otherwise, we allocate new memory and (if necessary) evaluate the initializer and assign its value to that memory location.

The transition rules for declarations are presented in Figures 38 and 39.

3.2 Initializers

Initializers in C come in two forms: simple expressions (for variables of the basic types) and lists of expressions (for variables representing arrays and structures). We must dynamically evaluate these

```
if TaskType(CurTask) = declaration then
    if DecType(CurTask) = static then
        if StaticAddr(CurTask) ≠ ⊥ then
            OnlyValue(CurTask) := StaticAddr(CurTask)
            Moveto(NextTask(CurTask))
        endif
        if StaticAddr(CurTask) = ⊥ then
            if Initializer(CurTask) ≠ ⊥ and
                OnlyValue(CurTask) = ⊥ then
                Moveto(Initializer(CurTask))
            else
                OnlyValue(CurTask) := NewMemory(CurTask)
                StaticAddr(CurTask) := NewMemory(CurTask)
                if Initializer(CurTask) ≠ ⊥ then
                    DoAssign(NewMemory(CurTask),
                        OnlyValue(CurTask),
                        ValueType(CurTask))
                else
                    Moveto(NextTask(CurTask))
                endif
            endif
        endif
    endif
endif
```

Figure 38: Transition rules for static declarations.

```
if TaskType(CurTask) = declaration then
    if DecType(CurTask) = normal then
        if Initializer(CurTask) ≠ ⊥ and
            OnlyValue(CurTask) = ⊥ then
            Moveto(Initializer(CurTask))
        else
            OnlyValue(CurTask) := NewMemory(CurTask)
            if Initializer(CurTask) ≠ ⊥ then
                DoAssign(NewMemory(CurTask),
                    OnlyValue(CurTask),
                    ValueType(CurTask))
            else
                Moveto(NextTask(CurTask))
            endif
        endif
    endif
endif
```

Figure 39: Transition rules for non-static declarations.

initializing expression(s) each time the variable is created, since such expressions are not restricted to constants.

Our previous rules for evaluating expressions will suffice to handle initializers for simple expressions. To assist in handling aggregate expressions, we will define a function *AddTo: typename* × *results* × *results* → *results*, which appends a value onto the end of an aggregate structure of the specified type. (For example, if [1] is an array containing the integer 1, then $AddTo(array, [1], 2) = [1, 2]$.)

We add a new element *aggregate* to our signals universe and specify that the *WhichChild* function will return *aggregate* when the expression being evaluated is a component of an aggregate initializer. We extend our *SetValue* abbreviation as shown in Figure 40 to correctly compose the desired aggregate expression.

3.3 Revision: Identifiers

Having now described how identifiers have memory allocated to them, we can revise our previous rules for evaluating identifier expressions.

We define the static function *Decl: tasks* → *tasks* which maps tasks corresponding to occurences of an identifier to the task corresponding to the declaration task for that variable.

The revised rules are shown in Figure 41.

4 Algebra Four: Functions

Our fourth algebra re-introduces function abstractions into the C programming language. With this information, we will also now (formally) present rules for beginning a C program, since the

```
if WhichChild(CurTask) = aggregate then
    OnlyValue(Parent(CurTask)) :=
        AddTo(ValueType(Parent(CurTask)),
            Value(Parent(CurTask)),
            value)
endif
```

Figure 40: Extension of the *SetValue(value)* abbreviation.

```
if TaskType(CurTask) = identifier then
    if LRValueType(CurTask) = lvalue then
        SetValue(OnlyValue(Decl(CurTask)))
    endif
    if LRValueType(CurTask) = rvalue then
        SetValue(Memory(OnlyValue(Decl(CurTask))))
    endif
endif
```

Figure 41: Revised transition rules for identifier expressions.

distinguished starting function `main()` may be viewed as simply another C function (with special parameters).

4.1 Modeling The Stack

Since functions in C may be recursive, it becomes necessary to revise our model for storing computational values (both internal and external). Clearly, we must have some means for storing multiple values at a given task, if the task falls within a function which may have several active instantiations at a given moment.

We accomplish this by simulating a stack in our algebra. We thus add a new universe *stack*, equinumerous with the positive integers, with a distinguished element *StackRoot: stack* which corresponds to the integer one. We also add static functions *StackPrev: stack* $\rightarrow$ *stack* and *StackNext: stack* $\rightarrow$ *stack* which correspond to the predecessor and successor functions for the positive integers. We will use a distinguished element *StackTop* to indicate the current element at the top of the stack.

To store state-associated information on the stack, we modify the various *Value* functions to be binary functions of the form *LeftValue, RightValue, OnlyValue, TestValue: tasks* $\times$ *stack* $\rightarrow$ *results.* This requires us to rewrite almost every rule that has appeared up until this point; we will simplify matters by stating that every reference to *LeftValue(X), RightValue(X), OnlyValue(X),* or *TestValue(X)* up until this point should be replaced by *LeftValue(X,StackTop), RightValue(X,StackTop), OnlyValue(X,StackTop),* or *TestValue(X,StackTop),* respectively.

4.2 Function Invocation and Return: The Caller

We are now ready to discuss function invocation and return from the perspective of the caller. The context-free grammar rules governing function invocations are as follows:

postfix-expression $\rightarrow$ *postfix-expression* ( )
postfix-expression $\rightarrow$ *postfix-expression* (*argument-expression-list*)

Note that the "name" of the function is actually a postfix expression, referring to the "address" of the function. (The significance of the "address" of a function is implementation-dependent.) We will thus need the use of a static function *AddrToFunc: addresses* $\rightarrow$ *tasks*, which maps function addresses to the root task of the function definition.

As we proceed, we will want to copy the value [2] of each parameter to an appropriate place in the task space for the callee to process. We thus assert that the *Parent* function (utilized by our *SetValue* abbreviation) maps argument expressions to tasks corresponding to the appropriate function parameters. We add an element *param* to the *internals* universe and add a function *ParamValue: tasks* $\rightarrow$ *results* to store the values of parameters being passed. Finally, we extend the definition of the *SetValue* abbreviation as shown in Figure 42.

We will also want to store on the stack our current location within the task graph, so that we will be able to resume execution at this point after the called function is completed. We thus define a function *StackTask: stack* $\rightarrow$ *tasks* which indicates the task which should become the value of *CurTask* when the current function terminates.

To process a function invocation, we evaluate the postfix expression corresponding to the pointer to the function along with all of the arguments in the expression list, and then transfer control to the specified function, placing a new element upon the function invocation stack. When control returns from the function, all values will be reset and the function's return value will be returned to the parent expression.

[2] In C, all function parameters are call-by-value.

```
if WhichChild(CurTask) = param then
    ParamValue(Parent(CurTask),StackTop) := value
endif
```

Figure 42: Extension of the *SetValue(value)* abbreviation.

As with the operands to most arithmetic operators, [KR] do not specify the order in which arguments to a function are evaluated. We thus must present specialized rules for evaluating the expressions associated with a function invocation.

We assert that the *ChooseTask* external function will indicate at each moment which expression associated with a function invocation should be evaluated next. Thus, our transition rules will simply make repeated calls to *ChooseTask* until all expressions have been evaluated, which we assert will occur when *ChooseTask* returns $\perp$.

The transition rules for function invocation are presented in Figure 43.

```
if TaskType(CurTask) = function-invocation then
    if ChooseTask(CurTask) ≠ ⊥ then
        Moveto(ChooseTask(CurTask))
    endif
    if ChooseTask(CurTask) = ⊥ then
        if OnlyValue(CurTask,StackTop) = ⊥ then
            StackTop := StackNext(StackTop)
            StackTask(StackNext(StackTop)) := CurTask
            Moveto(AddrToTask(LeftValue(CurTask,StackTop)))
        endif
        if OnlyValue(CurTask,StackTop) ≠ ⊥ then
            SetValue(OnlyValue(CurTask,StackTop))
            Moveto(NextTask(CurTask))
        endif
    endif
endif
```

Figure 43: Transition rules for function invocations.

4.3 Function Invocation and Return: The Callee

A function definition in C consists of a list of parameters and a compound statement (which contains local variable declarations and a sequence of statements to be executed).

For parameter declaration tasks, we must allocate new memory for each parameter and assign the appropriate value (stored here by the function invocation transition rules) to that new memory location. The transition rules for parameter declarations are shown in Figure 44.

---

if *TaskType(CurTask) = parameter-declaration* **then**
    SetValue(NewMemory(CurTask))
    DoAssign(NewMemory(CurTask),
        ParamValue(CurTask,StackPrev(StackTop)),
        ValueType(CurTask))
endif

Figure 44: Transition rules for parameter declarations.

4.4 The `return` Statement

We can now consider the transition rules for the **return** statement, whose context-free grammar rules appears below:

jump-statement → **return** ;
jump-statement → **return** *expression* ;

If an expression is present, our transition rules will (implicitly) evaluate the attached expression, and copy that value to the appropriate *function* task, which we assert is returned by the *NextTask* function. The transition rules for **return** statements are given in Figure 45.

---

if *TaskType(CurTask) = return* **then**
    OnlyValue(StackTask(StackTop),StackPrev(StackTop)) :=
        OnlyValue(CurTask,StackTop)
    StackTop := StackPrev(StackTop)
    PrevTask := CurTask
    CurTask := StackTask(StackTop)
endif

Figure 45: Transition rules for return statements.

If a **return** statement is not explicitly present at the end of a function, our algebra will contain still contain a *return* task as the last task of the function, as if a **return** statement were implicitly present as the last statement of the function.

4.5 Global Variables

With the introduction of the function stack, we now must consider how references to global variables within C will be modeled by our evolving algebra, and revise our rules for handling identifiers in expressions.

Lexical scoping within C is relatively simple, since function definitions may not contain other function definitions. Thus, a given variable identifier refers either to a variable local to the current function or to a variable global to the entire program. We will thus use a static function *GlobalVar: tasks* $\rightarrow$ {*true, false*} to indicate whether or not a given identifier refers to a global variable. We present the modified transition rules for identifiers in Figure 46.

```
if TaskType(CurTask) = identifier then
    if LRValueType(CurTask) = lvalue then
        if GlobalVar(CurTask) = true then
            SetValue(OnlyValue(Decl(Current),StackRoot))
        endif
        if GlobalVar(CurTask) = false then
            SetValue(OnlyValue(Decl(Current),StackTop))
        endif
    endif
    if LRValueType(CurTask) = rvalue then
        if GlobalVar(CurTask) = true then
            SetValue(MemoryValue(OnlyValue(Decl(CurTask),StackRoot),
                ValueType(CurTask)))
        endif
        if GlobalVar(CurTask) = false then
            SetValue(MemoryValue(OnlyValue(Decl(CurTask),StackTop),
                ValueType(CurTask)))
        endif
    endif
    Moveto(NextTask(CurTask))
endif
```

Figure 46: Revised transition rules for identifiers.

5 Algebra Five: Memory Structure

We now present our fifth and final algebra, which will revise our assumptions regarding the structure of memory.

One of C's more powerful features is the ability to "cast" memory elements from one type into another. For example, one may cast a pointer to a structure into a pointer to an array of characters, as long as the structure and the array are identical in size (with respect to their individual memory representations). Thus, one can access the individual bytes of most "values" which might exist

during the execution of the program.[3]

Clearly, we will need to have a byte-based model of memory. Thus, we now re-define the memory function as *Memory: addresses* → *bytes*. This now creates a number of difficulties, since most memory elements will not fit into a single byte.

Accordingly, we define an n+1-ary function *ByteToResult: typename* × *byte*^{n} → *results* which converts the memory representation of a value of the specified type into its corresponding value in the *results* universe. n is the maximum number of bytes used by the memory representation of any particular type. For types whose memory representations are less than n bytes in length, we may fill any unused parameters with don't care terms (*e.g.*, 0). We also define a partial function *ResultToByte: results* × *integer* × *typename* → *byte* which yields the specified byte of the memory representation of the specified value from the specified universe. This function can be thought of as the inverse of *ByteToResult*.

In addition, we define a static function *Size: typename* → *integer* which indicates how many bytes are used by a particular value type in memory.

5.1 References to Memory

Retrieving elements of the *values* universe from memory is now slightly more complicated than before. We define an abbreviation *MemoryValue: address* × *typename* → *results*, which indicate the value of the specified type being stored in memory beginning at the indicated address. *MemoryValue (addr,type)* abbreviates *ByteToResult (type, Memory(addr), Memory(addr+1), ..., Memory(addr + Size(type) - 1))*.

Having done this, it now becomes necessary to revise all previous references to the *Memory* function. We need to replace each occurrence of *Memory(address)* in previous transition rules with an occurrence of *MemoryValue(address, ValueType(X))*, where X is an "appropriate" node of the task graph for the conversion. The exact replacement for X should be clear from context in each case, and thus we omit the details.

5.2 Assignment to Memory

Rules for assignment to memory now become more complicated, since a given assignment may require an arbitrarily large number of updates to the *Memory* function. We will need to have rules which perform a loop to make those arbitrarily large number of updates in a systematic fashion.[4]

To facilitate this loop, we define the following distinguished elements:

- *CopyValue: results* denotes the value to be copied.
- *CopyType: typename* denotes the type of value to be copied.
- *CopyLocation: address* denotes the location to which the value is to be copied.
- *CopyByte: integer* denotes which byte of the representation of *CopyValue* is being copied into memory.
- *OldTask: tasks* denotes the task which invoked the memory copying procedure.

[3]The distinguished value void is an example of a value in a C program which cannot be accessed in this manner.

[4]Most computer systems provide a means for memory assignments in units larger than a byte, but the particular sizes available for assignment are implementation-dependent. We thus present rules using the lowest-common denominator, the byte.

We also add a distinguished element *CopyTask* to the *tasks* universe, to indicate that a memory copying procedure is in progress.

We will invoke the copying procedure using the *DoAssign(address, value, type)* macro, re-defined here in Figure 47.

DoAssign(address, value, type)

CopyValue := value
CopyType := type
CopyLocation := address
CopyByte := 0
OldTask := CurTask
CurTask := CopyTask

Figure 47: Revisied definition of the DoAssign macro.

The copying process is relatively straight forward. We utilize the distinguished element *CopyByte* to denote which byte of the memory representation of *CopyValue* we are copying into memory at a given moment in time. We copy bytes singly, incrementing the value of *CopyByte* after each assignment to memory, halting when all bytes have been copied. The transition rules for copying to memory are presented in Figure 48.

---

if *CurTask = CopyTask* **then**
  if *CopyByte < Size(CopyType)* **then**
    Memory(CopyLocation + CopyByte) :=
      ResultToByte(CopyValue, CopyByte, CopyType)
    CopyByte := CopyByte + 1
  endif
  if *CopyByte = Size(CopyType)* **then**
    PrevTask := OldTask
    CurTask := NextTask(OldTask)
  endif
endif

Figure 48: Transition rules for copying values to memory.

5.3 Pointer Arithmetic: Addition and Subtraction

We must also re-visit the transition rules dealing with pointer arithmetic at this time. In C, one may add an integer i to a pointer variable p with the following effect: the result is a pointer which

is i units forward in memory from p. (For example, since the name of an array is equivalent to a pointer to its first element, the expressions "`p[i]`" and "`p + i`" are equivalent.)

One may also subtract an integer i from a pointer variable p, with the result being a pointer i units in memory preceding p. In addition, one may subtract two pointers, resulting in the number of units of memory lying between the two pointers. The size of a "unit" of memory is determined by the size of the object type to which the pointer points.

This requires revisions to our rules for both the additive and subtractive assignment operators ("`+=`" and "`-=`") as well as the addition and subtraction operators. As we process each of these operators, it now becomes necessary to know whether or not a given variable is a pointer. We thus define new static functions *PointerType: tasks* $\rightarrow$ {*true, false*} which convey this information. For tasks for which *PointerType* returns *true*, we define a static function *PointsToType : tasks* $\rightarrow$ *typename* which indicates the object type to which the pointer points.

We give modified transition rules for the additive assignment, the simple addition operator, and the simple subtraction operator in Figures 49, 50, and 51, respectively. (The rules for subtractive assignment are similar to those for additive assignment and are thus omitted.)

---

if *TaskType(CurTask) = additive-assignment* **then**
  if *PointerType(LeftOperand(CurTask)) = true* **then**
    SetValue(MemoryValue(LeftValue(CurTask,StackTop), ValueType(CurTask))
      *+ (Size(PointsToType(CurTask)) * RightValue(CurTask,StackTop)))*
    DoAssign(LeftValue(CurTask,StackTop),
      MemoryValue(LeftValue(CurTask,StackTop), ValueType(CurTask))
        *+ (Size(PointsToType(CurTask)) * RightValue(CurTask,StackTop)),*
      ValueType(CurTask))
  endif
  if *PointerType(LeftOperand(CurTask)) = false* **then**
    SetValue(MemoryValue(LeftValue(CurTask,StackTop), ValueType(CurTask))
      + RightValue(CurTask,StackTop))
    DoAssign(LeftValue(CurTask,StackTop),
      MemoryValue(LeftValue(CurTask,StackTop), ValueType(CurTask))
        + RightValue(CurTask,StackTop),
      ValueType(CurTask))
  endif
endif

Figure 49: Revised transition rules for additive assignments.

5.4 Pointer Arithmetic: Array References

Since array references are defined in terms of pointer arithmetic, we must revise our rules for array references as well.

For convenience, we expand our previous tasks of type *array-reference* into two tasks of type *left-array-reference* and *right-array-reference*, which are distinguished by the presence of the array

```
if TaskType(CurTask) = addition then
    if PointerType(LeftOperand(CurTask)) = true then
        SetValue(LeftValue(CurTask,StackTop) + (Size(PointsToType(CurTask))
            * RightValue(CurTask,StackTop)))
    endif
    if PointerType(RightOperand(CurTask)) = true then
        SetValue(RightValue(CurTask,StackTop) + (Size(PointsToType(CurTask))
            * LeftValue(CurTask,StackTop)))
    endif
    if PointerType(LeftOperand(CurTask)) = false and
        PointerType(RightOperand(CurTask)) = false then
        SetValue(LeftValue(CurTask,StackTop) + RightValue(CurTask,StackTop))
    endif
    Moveto(NextTask(CurTask))
endif
```

Figure 50: Revised transition rules for addition expressions.

```
if TaskType(CurTask) = subtraction then
    if PointerType(LeftOperand(CurTask)) = true and
        PointerType(RightOperand(CurTask)) = true then
        SetValue((MemoryValue(LeftValue(CurTask,StackTop), ValueType(CurTask))
            - MemoryValue(RightValue(CurTask,StackTop), ValueType(CurTask)))
            / Size(PointsToType(CurTask)))
    endif
    if PointerType(LeftOperand(CurTask)) = true and
        PointerType(RightOperand(CurTask)) = false then
        SetValue(LeftValue(CurTask,StackTop) - (Size(PointsToType(CurTask))
            * RightValue(CurTask,StackTop)))
    endif
    if PointerType(LeftOperand(CurTask)) = false and
        PointerType(RightOperand(CurTask)) = false then
        SetValue(LeftValue(CurTask,StackTop) - RightValue(CurTask,StackTop))
    endif
    Moveto(NextTask(CurTask))
endif
```

Figure 51: Revised transition rules for subtraction expressions.

(*i.e.* pointer) operand to the left (resp., right) of the other operand.

The revised transition rules for array references are given in Figure 52.

```
if TaskType(CurTask) = left-array-reference then
    if LRValueType(CurTask) = lvalue then
        SetValue(LeftValue(CurTask,StackTop) + (Size(PointsToType(CurTask))
            * RightValue(CurTask,StackTop)))
    endif
    if LRValueType(CurTask) = rvalue then
        SetValue(MemoryValue(LeftValue(CurTask,StackTop)
            + (Size(PointsToType(CurTask))* RightValue(CurTask,StackTop)),
            ValueType(CurTask)))
    endif
    Moveto(NextTask(CurTask))
endif

if TaskType(CurTask) = right-array-reference then
    if LRValueType(CurTask) = lvalue then
        SetValue(RightValue(CurTask,StackTop) + (Size(PointsToType(CurTask))
            * LeftValue(CurTask,StackTop)))
    endif
    if LRValueType(CurTask) = rvalue then
        SetValue(MemoryValue(RightValue(CurTask,StackTop)
        + (Size(PointsToType(CurTask))* LeftValue(CurTask,StackTop)),
            ValueType(CurTask)))
    endif
    Moveto(NextTask(CurTask))
endif
```

Figure 52: Revised transition rules for array references.

5.5 The `sizeof` Operator

Finally, we must revise our rules dealing with the **`sizeof`** operator. We repeat the relevant context-free grammar rules:

unary-expression → **`sizeof`** *unary-expression*
unary-expression → **`sizeof`** **`(`** *type-name* **`)`**

The *ValueType* function allows us to convert either expansion into a value in the *typename* universe. With that information, we may use the *Size* function to determine the size, in bytes, of an element of that particular type and return that number as the value of the unary expression. The transition rules for size operators are shown in Figure 53.

This concludes our presentation of an evolving algebra for C.

```
if TaskType(CurTask) = sizeof then
    SetValue(Size(ValueType(CurTask)))
    Moveto(NextTask(CurTask))
endif
```

Figure 53: Transition rules for size operators.

References

[ASU] Alfred V. Aho, Ravi Sethi, and Jeffrey D. Ullman, "Compilers: Principles, Techniques, and Tools", Addison-Wesley, 1988.

[Bl] George Robert Blakley, "A Smalltalk Evolving Algebra And Its Uses", Ph.D. Thesis, The University of Michigan, 1992.

[Bo1] Egon Börger, "A Logical Operational Semantics for Full Prolog. Part I: Selection Core and Control" in Proc. of CSL'89, 3rd Workshop on Computer Science Login (eds. E. Börger, H. Kleine Büning and M. Richter), Springer LNCS 440 (1990), 36–64.

[Bo2] Egon Börger, "A Logical Operational Semantics for Full Prolog. Part II: Built-in Predicates for Database Manipulations", in Proc. of Mathematical Foundations of Computer Science 1990 (ed. B. Rovan), Springer LNCS 452, 1–14.

[Bo3] Egon Börger, "A Logical Operational Semantics for Full Prolog. Part III: Built-in Predicates for Files, Terms, Arithmetic and Input-Output", IWBS Report no. 117, IBM Germany, Institut für Wissensbasierte Systeme, April 1990, pp. 25; to appear in Proc. of Workshop on Logic from Computer Science (Berkeley 1989), MSRI Proceedings Series, ed. Y. Moschovakis, Springer Verlag.

[BR1] Egon Börger and Dean Rosenzweig, "A Formal Analysis of Prolog Database Views and their Uniform Implementation", University of Michigan Technical Report CSE-TR-89-91, 1991.

[BR2] Egon Börger and Dean Rosenzweig, "A Formal Specification of Prolog by Tree Algebras", *Proceedings of ITI*, Cavtat, 1991.

[BS] Egon Börger and Peter Schmitt, "A formal operational semantics for languages of type Prolog III", in Proc. of CSL'90, 4th Workshop on Computer Science Logic (Eds. E. Börger, H. Kleine Büning and M. Richter), Springer LNCS, 1991.

[Gu1] Yuri Gurevich, "Logic and the Challenge of Computer Science", in *Current Trends in Theoretical Computer Science* (ed. E. Börger), Computer Science Press, 1987, 1–57.

[Gu2] "Algorithms in the world of bounded resources", in "The Universal Turing Machine: A Half-Century Story" (ed. R. Herken), Oxford University Press, 1988, 407–416.

[Gu3] Yuri Gurevich, "Evolving Algebras: An Introductory Tutorial", Bulletin of European Assocation for Theoretical Computer Science, February 1991.

[Gu4] Yuri Gurevich and James Morris, "Algebraic Operational Semantics and Modula 2", Lecture Notes in Computer Science, Proceedings of Logik und Informatik (Karlsruhe, October 1987), Springer-Verlag, Berlin.

[GMs] Yuri Gurevich and Larry Moss, "Algebraic Operational Semantics and Occam", Springer LNCS 440, 176–192.

[KR] Brian W. Kernighan and Dennis M. Ritchie, "The C Programming Language", Prentice Hall, 2nd edition, 1988, Englewood Cliffs, NJ.

[Mor] James Morris, "Algebraic Operational Semantics for Modula 2", Ph.D. thesis, The University of Michigan, 1988.