The Semantics of the C Programming Language

Yuri Gurevich*and James K. Huggins*
EECS Department, University of Michigan, Ann Arbor, MI 48109-2122, USA

February 19, 1993

This paper first appeared in [GH2], and incorporates the corrections indicated in [GH3].

0 Introduction

We present formal operational semantics for the C programming language. Our starting point is the ANSI standard for C as described in [KR]. Knowledge of C is not necessary (though it may be helpful) for comprehension, since we explain all relevant aspects of C as we proceed.

Our operational semantics is based on evolving algebras. An exposition on evolving algebras can be found in the tutorial [Gu]. In order to make this paper self-contained, we recall the notion of a (sequential) evolving algebra in Sect. 0.1.

Our primary concern here is with semantics, not syntax. Consequently, we assume that all syntactic information regarding a given program is available to us at the beginning of the computation (via static functions). We intended to cover all constructs of the C programming language, but not the C standard library functions (*e.g.* `fprintf()`, `fscanf()`). It is not difficult to extend our description of C to include any desired library function or functions.

Evolving algebra 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 the desired level of abstraction, with unnecessary details omitted. It also makes comprehension easier. We present a series of four evolving algebras, each a refinement of the previous one. The final algebra describes the C programming language in full detail.

Our four algebras focus on the following topics respectively:

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

What about possible errors, *i.e.*, division by zero or de-referencing a pointer to an invalid address? These issues are very implementation-dependent. Even what constitutes an error is implementation-dependent. If an external function does not produce any value in a state where a value is expected, the evolving algebra will be stalled in that state forever. It is natural to suppose that if an external function does produce a value, it is of the appropriate type. One may want to augment the guards of transition rules to check for errors; in this way, the evolving Nalgebra will halt on error conditions (and may even output an error message if desired). There are more subtle ways to handle errors. We ignore the issue here.

To reflect the possibility of different implementations, our evolving algebras contain implementation-dependent parameters. For example, the set of values "storable" in a pointer variable is implementation-dependent. Thus, each of our four evolving algebras gives rise to a family of different evolving algebras.

*Partially supported by ONR and NSF.

0.1 Evolving Algebras

An evolving algebra $\mathcal{A}$ is an abstract machine. Here we restrict attention to sequential evolving algebras. The *signature* of $\mathcal{A}$ is a (finite) collection of function names, each name having a fixed arity. A state of $\mathcal{A}$ is a set, the *superuniverse*, together with interpretations of the function names in the signature. These interpretations are called *basic functions* of the state. The superuniverse does not change as $\mathcal{A}$ evolves; the basic functions may.

Formally, a basic function of arity r (*i.e.* the interpretation of a function name of arity r) is an r-ary operation on the superuniverse. (We often use basic functions with $r = 0$; such basic functions will be called *distinguished elements*.) But functions naturally arising in applications may be defined only on a part of the superuniverse. Such partial functions are represented by total functions in the following manner.

The superuniverse contains distinct elements *true, false, undef* which allow us to deal with relations (viewed as binary functions with values *true* or *false*) and partial functions (where $f(\overline{a}) = undef$ means f is undefined at the tuple $\overline{a}$). These three elements are *logical constants*. Their names do not appear in the signature; this is similar to the situation in first-order logic with equality where equality is a logical constant and the sign of equality does not appear in the signature. In fact, we use equality as a logical constant as well.

Further, a *universe* U is a special type of basic function: a unary relation usually identified with the set $\{x : U(x)\}$. The universe $Bool = \{true, false\}$ is another logical constant. When we speak about, say, a function f from a universe U to a universe V, we mean that formally 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 use self-explanatory notations like $f : U \rightarrow V$, $f : U_1 \times U_2 \rightarrow V$, and $f : V$. The last means that the distinguished element f belongs to V.

In principle, a program of $\mathcal{A}$ is a finite collection of transition rules of the form

$$if \quad t_0 \, then \quad f(t_1, \ldots, t_r) \; := \; t_{r+1} \; endif \tag{1}$$

where t_0, $f(t_1, \ldots, t_r)$, and t_{r+1} are closed terms (*i.e.* terms containing no free variables) in the signature of $\mathcal{A}$. An example of such a term is $g(h_1, h_2)$ where g is binary and h_1 and h_2 are zero-ary. The meaning of the rule shown above is this: Evaluate all the terms t_i in the given state; if t_0 evaluates to *true* then change the value of the basic function f at the value of the tuple $(t_1, .., t_r)$ to the value of t_{r+1}, otherwise do nothing.

In fact, rules are defined in a slightly more liberal way; if k is a natural number, $b_0, \ldots, b_k$ are terms and $C_0, \ldots, C_{k+1}$ are sets of rules then both of the following are rules:

```
if $b_0$ then $C_0$
elseif $b_1$ then $C_1$
      ⋮
elseif $b_k$ then $C_k$
else $C_{k+1}$
endif
```

```
if $b_0$ then $C_0$
elseif $b_1$ then $C_1$
      ⋮
elseif $b_k$ then $C_k$
endif
```

Since the C_i are sets of rules, nested transition rules are allowed (and occur frequently). To save space, we abbreviate the series of **endif**'s at the tail of a transition rule by **ENDIF**.

A program is a set of rules. It is easy to transform a program to an equivalent program comprising only rules of the stricter form (1). We use rules of the more liberal form, as well as macros (textual abbreviations), for brevity.

How does $\mathcal{A}$ evolve from one state to another? In a given state, the demon (or interpreter) evaluates all the relevant terms and then makes all the necessary updates. If several updates contradict each other (trying to assign different values to the same basic function at the same place), then the demon chooses nondeterministically one of those updates to execute.

We call a function (name) f *dynamic* if the demon (interpreter) may change f as the algebra evolves; *i.e.* if an assignment of the form $f(t_1, \ldots, t_r) := t_0$ appears anywhere in the transition rules. Functions which are not dynamic are called *static*. To allow our algebras to interact conveniently with the outside world, we also make use of *external* functions within our algebra. External functions are syntactically static (that is, never changed by rules), but have their values determined by an oracle. Thus, an external function may have different values for the same arguments as the algebra evolves.

0.2 Acknowledgements

An earlier version of this paper appeared as a technical report [GH1]. We gratefully acknowledge comments made by on the original report by Egon Börger, Andre Burago, Martin J. Dürst, Stefano Guerrini, Raghu Mani, Arnd Poetzsch-Heffter, Dean Rosenzweig, and Marcus Vale, as well as comments in the errata made by Lars Ole Andersen, L. Douglas Baker, Arnd Poetzsch-Heffter, Thomas Tsukada, and Chuck Wallace.

1 Algebra One: Handling C Statements

Our first evolving algebra models the control structures of C.

1.1 Some Basic Functions

A universe *tasks* consists of elements representing tasks to be accomplished by the program interpreter. The notion of task is a general one: *e.g.*, a task may be the execution of a statement, initialization of a variable, or the evaluation of an expression. The elements of this universe are dependent on the particular C program being executed by the abstract machine. It is often useful to mark a given task with tags indicating its nature. This gives rise to a universe of *tags*.

A distinguished element *CurTask: tasks* indicates the current task. In order to execute tasks in the proper order, a static function *NextTask: tasks → tasks* indicates the next task to be performed once the current task has been completed. A static function *TaskType: tasks → tags* indicates the action to be performed by the task.

A universe *results* contains values which may appear as the result of a computation.

1.2 Macro: Moveto

Often, we transfer control to a particular task, modifying *CurTask* to indicate the transfer of control. In Algebra Two, the rules for modifying *CurTask* will change somewhat; in order to facilitate this change, we will use the *Moveto(Task)* macro each time that we wish to transfer control. For now, the definition of *Moveto(Task)* is shown in Fig. 1.

Moveto(Task)

CurTask := Task

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

1.3 Statement Classification in C

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

1. Expression statements, which evaluate the associated expression.

2. Selection statements (`if` and `switch`).

3. Iteration statements (`for`, `while`, and `do-while`).

4. Jump statements (`goto`, `continue`, `break`, and `return`).

5. Labeled statements (`case` and `default` statements used within the scope of a `switch` statement, and targets of `goto` statements).

6. Compound statements, consisting of a (possibly empty) list of local variable declarations and a (possibly empty) list of statements.

.

1.4 Expression Statements

An expression statement has one of the following forms:

expression-statement → *;*
expression-statement → *expression ;*

To execute an expression statement, evaluate the attached expression (if any), even though the resulting value will not be used. While this may seem unnecessary, note that the evaluation of an expression in C may generate side-effects (such as assigning a value to a variable). Note also that the evaluation of an expression may not halt. In this algebra, the evaluation of expressions is handled by an external function *TestValue: tasks* → *results*.

Since expression statements perform no additional work, the algebra simply proceeds to the next task. The transition rule for expression tasks is shown in Fig. 2.

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

Figure 2: Transition rule for expression tasks.

1.5 `if` Statements

There are two types of selection statements in C: `if` statements and `switch` statements. An `if` statement has one of the following forms:

if-statement → `if` *(expression) statement1*
if-statement → `if` *(expression) statement1* `else` *statement2*

where *statement1* and *statement2* are statements.

To execute an `if` statement, begin by evaluating the guard expression. If the resulting value is non-zero, execute *statement1*. If the resulting value is zero and an `else` clause is present, execute *statement2*. otherwise, execute the statement following the `if` statement. Static partial functions *TrueTask: tasks* → *tasks* and *FalseTask: tasks* → *tasks* indicate the task to be performed if the guard of the `if` statement evaluates to a non-zero value or zero, respectively.

The branching decision made in the `if` statement is represented by an element of the *tasks* universe for which the *TaskType* function returns *branch*. We illustrate a typical `if` statement with the graph in Fig. 3, where ovals represent tasks, labeled arcs represent the corresponding unary functions, and boxes represent subgraphs. If an `else` clause is not present in an `if` statement, the corresponding task graph omits the lower portion of Fig. 3, with the *FalseTask* function connecting the *branch* task to the task following the `if` statement. The transition rule for *branch* tasks is shown in Fig. 4.

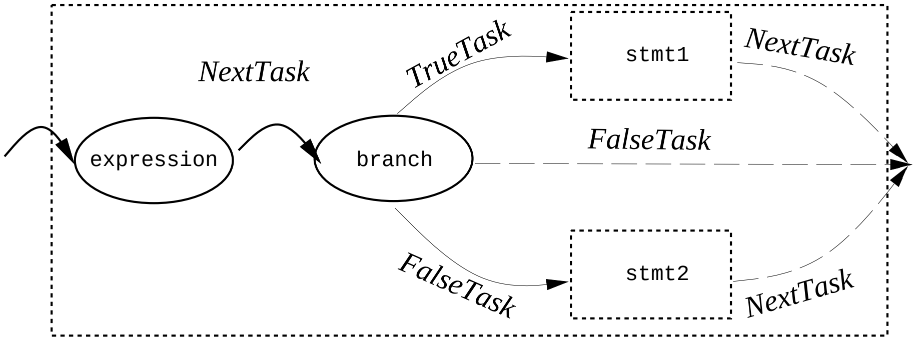

Figure 3: A typical `if` statement.

```
if TaskType(CurTask) = branch then
    if TestValue(CurTask) $\neq$ 0 then
        Moveto(TrueTask(CurTask))
    elseif TestValue(CurTask) = 0 then
        Moveto(FalseTask(CurTask))
ENDIF
```

Figure 4: Transition rule for *branch* tasks.

Remark. Fig. 3 shows a typical `if` statement, but it is not representative of all `if` statements. The presence of a jump statement in *statement1* or *statement2* may cause *NextTask* to point to a different task than the one which immediately follows the `if` statement.

1.6 `switch` Statements

A `switch` statement has the following form:

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

where *body* is a statement, usually compound.

Within the body of a `switch` statement there are (usually) labeled `case` and `default` statements. Each `case` or `default` is associated with the smallest enclosing `switch` statement.

To execute a `switch` statement, evaluate the guard expression, and within the body of the switch, transfer control to the `case` statement for the `switch` whose labeled value matches the value of the expression, or to the `default` statement for the `switch`, whichever comes first. If no such statement is found, transfer control to the statement following the `switch` statement.

Labels on `case` statements are required to be unique within a `switch`, and a `switch` may not have more than one `default` statement. Thus, for a given expression value, there is exactly one statement to which control should be passed. A static partial function *SwitchTask: tasks* × *results* → *tasks* indicates the next task to be executed for a given expression value. We illustrate a typical `switch` statement with the graph in Fig. 5; with regard to the possible effects of embedded jump statements, see the remark in Sect. 1.5. The rule for `switch` tasks is shown in Fig. 6.

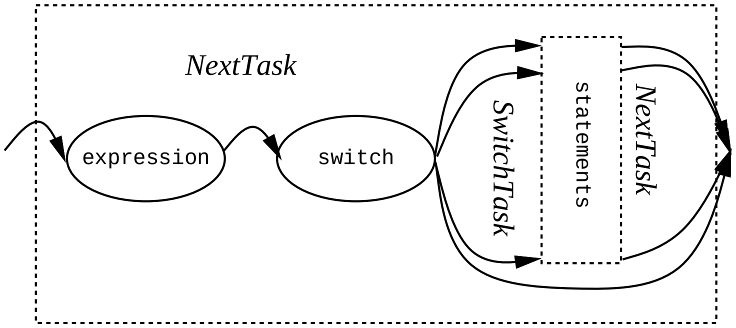

Figure 5: A typical `switch` statement.

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

Figure 6: Transition rules for *switch* tasks.

1.7 `while` Statements

A `while` statement has the following form:

while-statement → `while` (*expression*) *body*

where *body* is a statement.

To execute a `while` statement, keep evaluating the guard expression until the value of the expression becomes zero. Each time that the value of the guard expression is not zero, execute *body*.

We illustrate a typical `while` statement with the graph in Fig. 7; with regard to the possible effects of embedded jump statements, see the remark in Sect. 1.5. Since the only types of tasks used to represent `while` statements are the *expression* and *branch* tasks, our previously-presented transition rules are sufficient to model the behavior of `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. The ANSI standard [KR] does not give specific semantics for such behavior. In such a situation, our abstract machine would continue as if the loop had been entered normally (*i.e.*, after completion of the statement body, control returns to the guard expression to be evaluated). We nbelieve this is a reasonable interpretation of such an event.

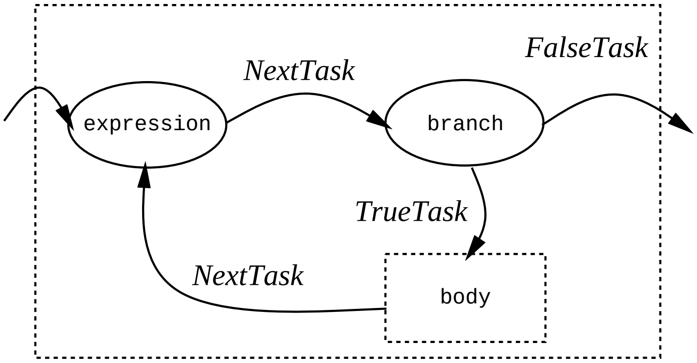

Figure 7: A typical `while` statement.

1.8 `do-while` Statements

A `do-while` statement has the following form:

> *do-while-statement* → `do` *body* `while` `(` *expression* `)` `;`

where *body* is a statement.

`do-while` statements are identical to `while` statements except that the guard expression and statement body are visited in the opposite order. We illustrate a typical `do-while` with the graph shown in Fig. 8; with regard to the possible effects of embedded jump statements, see the remark in Sect. 1.5. (Note the similarity between this graph and that of the `while` loop.) As with `while` loops, no new transition rules are required to model the behavior of `do-while` statements.

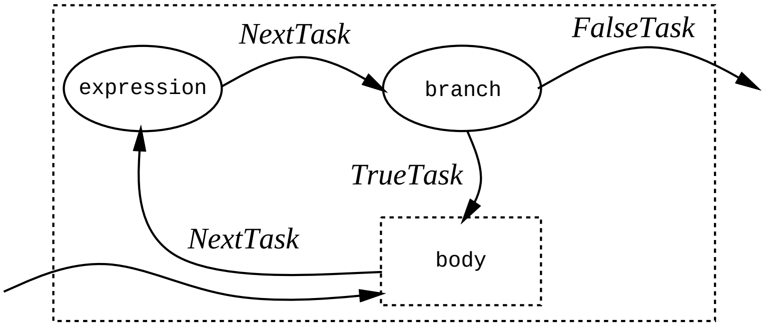

Figure 8: A typical `do-while` statement.

1.9 `for` Statements

The most complete form of the `for` statement is:

> *for-statement* → `for` `(` *initializer* `;` *test* `;` *update* `)` *body*

where *initializer*, *test*, and *update* are expressions, any of which may be omitted, and *body* is a statement, usually compound. We begin by describing the behavior and representation of a `for` statement when all expressions are present.

In executing a `for` statement, begin by evaluating the initializer. Evaluate the test next; if the result is non-zero, execute the body and evaluate the update (in that order) and re-evaluate the test. If the value of the test is zero, transfer control to the statement following the `for` loop.

We illustrate a typical `for` statement with the graph in Fig. 9; with regard to the possible effects of embedded jump statements, see the remark in Sect. 1.5. Again, no new transition rules are required to model the behavior of `for` statements.

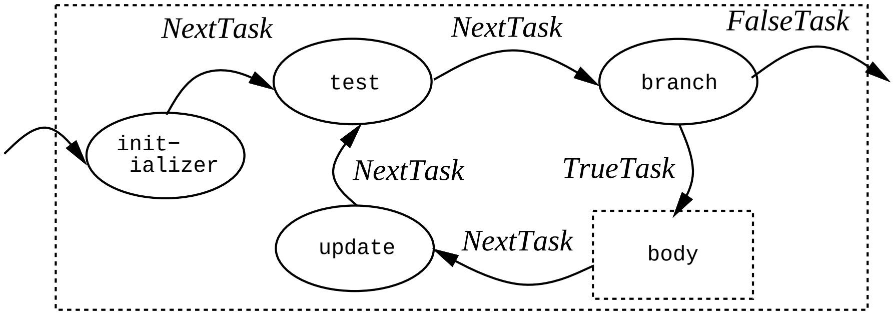

Figure 9: A typical `for` statement.

The graphs for `for` loops missing one or more of the three expressions (initializer, test, and update) omit the corresponding tasks, with *NextTask* pointing to the next task in the graph sequence. If the test is omitted, both the test and the *branch* task are omitted, which creates an infinite loop (which may still be broken through the use of jump statements).

1.10 Jump Statements

A jump statement has one of the following forms:

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

Each of these jump statements is a command indicating that control should be unconditionally transferred to another task in the task graph:

- `goto` statements indicate directly the task to which control passes.
- `continue` statements may only occur within the body of an iteration statement. For a given `continue` statement C, let S be the smallest iteration statement which includes C. Executing C transfers control to the task within S following the statement body of S: *e.g.*, for `for` statements, control passes to the update expression, while for `while` statements, control passes to the guard expression.
- `break` statements may occur within the body of an iteration or `switch` statement. For a given `break` statement B, let S be the smallest iteration or `switch` statement which includes B. Executing B transfers control to the first task following S.
- `return` statements occur within the body of function abstractions, indicating that the current function execution should be terminated. A more complete discussion of `return` statements will be presented in Algebra Four, where function abstractions are presented. For now, we assert that executing a `return` statement should set *CurTask* to *undef*, which will bring a halt to the algebra, since we only have one function (`main`) being executed.

The *NextTask* function contains the above (static) information for jump statement tasks. Thus, the transition rule for jump statements (shown in Fig. 10) is trivial.

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

Figure 10: Transition rule for *jump* tasks.

1.11 Labeled Statements

A labeled statement has one of the following forms:

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

Statement labels identify the targets for control transfer in `goto` and `switch` statements. *NextTask* and *SwitchTask* return the appropriate tasks in each case; no further transition rules are needed.

1.12 Compound Statements

A compound statement has the following form:

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

where the declaration and/or statement lists may be empty.)

Since *NextTask* indicates the order in which tasks are processed, we have no need for rules concerning compound statements. Each statement or declaration in a compound statement is linked to its successor via *NextTask*. (Declarations are not treated until Algebra Three; nonetheless, the same principle holds for declaration tasks.)

1.13 Initial and Final States

We assert that initially, *CurTask* indicates the first task of the first statement of the program.

A final state in our algebra is any state in which *CurTask* = *undef*. In this state, no rules will be executed, since *TaskType(undef)* = *undef*.

2 Algebra Two: Evaluating Expressions

Our second evolving algebra refines the first and focuses on the evaluation of expressions.

We replace each occurrence of a task of type *expression* from the first algebra with numerous tasks reflecting the structure of the expression. Also, *TestValue* is now an internal, dynamic function.

In Algebra Two, we treat the evaluation of expressions at a relatively high level of abstraction. We map variable identifiers to memory locations through external functions. We also treat function invocations as expressions whose values are provided by external functions. In Algebras Three and Four we will eliminate these abstractions.

2.1 New Basic Functions Related To Memory Management

In C, one may (re)cast types. For example, one may cast a pointer to a structure into a pointer to an array of characters. Thus, one can access the individual bytes of most values which might exist during the execution of the program.[1]

A static function *Size: typename* → *integer* indicates how many bytes are used by a particular value type in memory. A dynamic function *Memory: addresses* → *bytes* indicates the values stored in memory at a given byte. Since most values of interest are larger than a byte, we need a means for storing members of *results* as individual bytes. For example, assume that the `int` value 258 is represented in the memory of a particular system by the four (eight-bit) bytes 0, 0, 1, and 2, stored consecutively. We need a way to go from a value in *results* (*e.g.*, 258) to its component bytes (0, 0, 1, 2) and vice versa.

A static partial (n+1)-ary function *ByteToResult: typename* × $byte^n$ → *results* converts the memory representation of a value of the specified basic type into its corresponding value in the *results* 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. In our example above, *ByteToResult(int,0,0,1,2) = 258*.

A static partial function *ResultToByte: results* × *integer* × *typename* → *byte* 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 our example above, *ResultToByte(258,3,int) = 2*. (We assume tacitly that the arguments of *ResultToByte* uniquely define the value of the function, which is the case in all the implementations that we know.)

We define an abbreviation *MemoryValue: address* × *typename* → *results*, which indicates 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))*.

2.2 Other New Basic Functions

Two sub-universes of *results*, the universe of computational results, are of particular interest in Algebra Two. A universe *bytes* contains those values which may be “stored” in a `char` variable. (This universe is usually identical to {0,1,...,255}, but we prefer the more general definition.) A universe *addresses* contains positive integers corresponding to valid memory locations. This is also the universe of values which may be stored in a pointer-type variable. (Of course, these two universes are implementation-dependent.)

A universe *typename* contains elements representing the different types of storable values. A static partial function *ValueType: tasks* → *typename* indicates the type of the resulting value when an expression has been evaluated.

Static partial functions *LeftTask, RightTask: tasks* → *tasks* indicate the left and right operands of binary operators whose order of evaluation is not defined within C (*e.g.*, `+`). A static partial function *Parent: tasks* → *tasks* indicates the parent (*i.e.*, closest enclosing) expression for a given expression. For expressions which are not contained in any other expressions, *Parent* returns the corresponding *branch* task which uses the expression (if one exists) or *undef* (if none exists). A static partial function *WhichChild: tasks* → {*left, right, only, test, ...*} (where *left*, *etc.* are members of the *tags* universe) indicates the relationship between a task and its parent.

Dynamic partial functions *LeftValue, RightValue: tasks* → *results* indicate the results of evaluating the left and right operands of binary operators with ambiguous evaluation order. Similarly, a dynamic partial function *OnlyValue: tasks* → *results* indicates the result of evaluating the single operand of a unary operator. A static partial function *ConstVal: tasks* → *results* indicates the values of program constants.

[1] The distinguished value `void` is an example of a value which cannot be accessed in this manner.

2.3 Macro: DoAssign

Our rules for assignment to memory are a little complicated, since a given assignment may require an arbitrarily large number of updates to the *Memory* function. We need rules which perform a loop to make those arbitrarily large number of updates in a systematic fashion.[2]

To facilitate this loop, we use several 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. *CopyTask: tasks* is a static distinguished element used to indicate that the copying procedure should begin.

We will invoke the copying procedure using the *DoAssign(address, value, type)* macro, defined in Fig. 11. The copying process itself is relatively straightforward. 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 rule for copying to memory is shown in Fig. 12.

```
DoAssign(address, value, type)

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

Figure 11: Definition of the *DoAssign* macro.

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

Figure 12: Transition rule for copying to memory.

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

2.4 Macro: ReportValue

When we process tasks corresponding to expression evaluation, we assign the value of an evaluated expression to the appropriate storage function in the parent expression (*e.g. LeftValue(Parent(CurTask))*). We use the *ReportValue* macro (defined in Fig. 13) to accomplish this.

```
ReportValue(value)

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

Figure 13: Definition of the *ReportValue* macro.

2.5 Macros: EvaluateOperands and Moveto

In C, as described by [KR], many binary operators (such as the assignment operator "`=`") do not have a defined order of evaluation: either operand may be evaluated first. When one or more operands of such an operator generate side effects (as in "`a[i] = i++`"), the value or the side-effects generated by the expression may depend upon the order of evaluation. Writing such code is usually unwise, since such code may not be portable; however, an optimizing compiler may take advantage of this ambiguity to generate code which minimizes the resources required to perform a particular computation [ASU].

For expressions involving such operators, our algebra must be flexible enough to reflect any possible evaluation order of an expression's operands, even if this decision is made at run-time. While we believe most compilers make this decision at compile-time, we must still 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 static decisions into the task graph). We will use an external function *ChooseTask* to represent this decision.

We illustrate how expressions with such operators are represented in our algebra by the graph in Fig. 14. A dynamic function *Visited: tasks* → {*left, right, both, neither*} indicates which subexpressions have been evaluated at a given moment. Initially, *Visited* has the value *neither* for all tasks.

To evaluate expressions of this type, begin by evaluating the sub-expression indicated by *ChooseTask*. When that sub-expression has been evaluated, evaluate the other sub-expression. Finally, when both expressions have been evaluated, perform the desired operation.

To handle the portion of this behavior dealing with the operator task, we use the *EVALUATE OPERANDS* macro. Informally, the macro means "Evaluate both operands in the order given by *ChooseTask*; when both operands are evaluated, then do ...". To handle the portion of this behavior dealing with movement between subtasks, we redefine the *Moveto(Task)* macro to jump directly between the subtasks of an operator of this type. The definitions for *EVALUATE OPERANDS* and *Moveto* are shown in Fig 15 and Fig. 16.

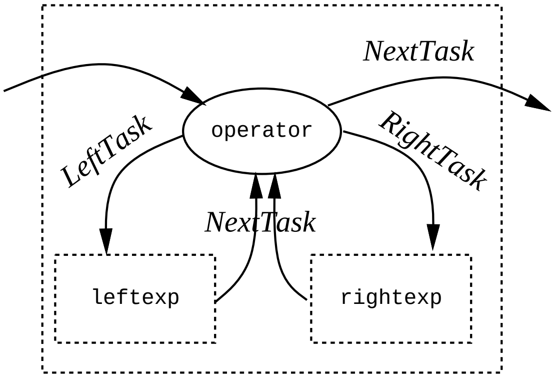

Figure 14: Binary operators.

```
EVALUATE OPERANDS WITH
    statements
END EVALUATE

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

Figure 15: Definition of the *EVALUATE OPERANDS* macro.

```
Moveto(Task)

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

Figure 16: Revised definition of the *Moveto(Task)* macro.

2.6 Comma Operators

A comma expression has the following form:

comma-expression → *expr1* **,** *expr2*

where *expr1* and *expr2* are expressions.

To evaluate a comma expression, evaluate *expr1* and *expr2*, left to right, returning the value of *expr2* as the value of the parent expression. (Though it may seem unnecessary to evaluate the first expression since we ignore its value, recall that expressions in C may generate side-effects.) We represent comma expressions as a sequence of two expressions linked by the *NextTask* function. Thus, no additional transition rules are needed to process comma operators.

2.7 Conditional Expressions

A conditional expression has the following form:

conditional-expression → *expr1* **?** *expr2* **:** *expr3*

where *expr1*, *expr2*, and *expr3* are expressions.

To evaluate a conditional expression, evaluate *expr1*. If the resulting value is non-zero, evaluate *expr2* and return its value as the value of the parent expression; otherwise, evaluate *expr3* and return its value as the value of the parent expression.

We will represent conditional expressions in our algebra in a manner similar to that in which we represent conditional statements, as illustrated in Fig. 17. The tasks corresponding to the center and right subexpressions will update the appropriate *Value* function for the parent expression upon completion of the evaluation of the subexpression. No new transition rules are needed to handle conditional expressions.

2.8 Logical OR Expressions

A logical OR expression has the following form:

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

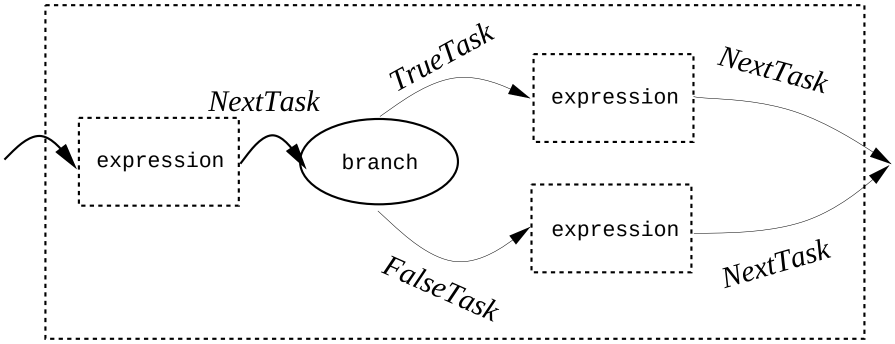

Figure 17: Conditional expression.

where *expr1* and *expr2* are expressions.

To evaluate a logical OR expression, start by evaluating *expr1*. If the result is non-zero, the value of the parent expression is 1 and *expr2* is not evaluated. Otherwise, the value of the parent expression is the value of *expr2*, with non-zero values coerced to 1.

To represent a logical OR expression, we will introduce two new task types, *OR* and *makeBool*. We illustrate how logical OR expressions are represented in our algebra by the graph in Fig. 18.

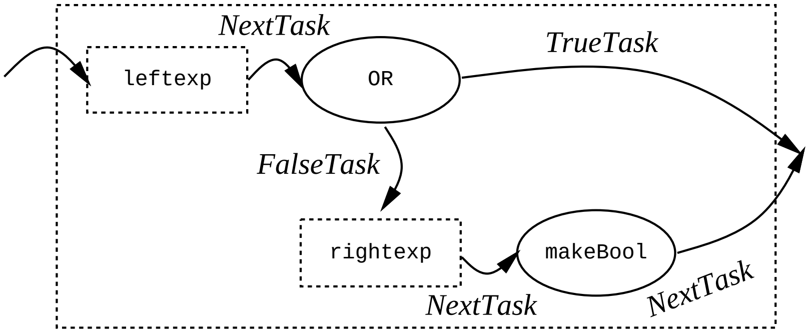

Figure 18: Logical OR expression.

In processing the *OR* task, the value of *expr1* will be examined. If the value is not zero, the rules for *OR* tasks will set the value of the expression to 1 and end processing of the parent expression. Otherwise, the rules will pass control to the tasks which evaluate *expr2*. The rules for *makeBool* tasks will examine the value of *expr2* and coerce it to 0 or 1. The transition rules for *OR* and *makeBool* tasks are shown in Fig. 19 and Fig. 20.

2.9 Logical AND expressions

A logical AND expression has the following form:

logical-AND-expression → *expr1* && *expr2*

where *expr1* and *expr2* are expressions.

To evaluate an AND expression, begin by evaluating *expr1*. If the resulting value is 0, the value of the parent expression is 0. Otherwise, the value of *expr2* (coerced to 0 or 1) is the value of the parent expression.

```
if TaskType(CurTask) = OR then
    if OnlyValue(CurTask) $\neq$ 0 then
        ReportValue(1)
        Moveto(TrueTask(CurTask))
    elseif OnlyValue(CurTask) = 0 then
        Moveto(FalseTask(CurTask))
ENDIF
```

Figure 19: Transition rule for *OR* tasks.

```
if TaskType(CurTask) = makeBool then
    if OnlyValue(CurTask) $\neq$ 0 then
        ReportValue(1)
    elseif OnlyValue(CurTask) = 0 then
        ReportValue(0)
    endif
    Moveto(NextTask(CurTask))
endif
```

Figure 20: Transition rule for *makeBool* tasks.

The representation of logical AND expressions is similar to that of logical OR expressions, as illustrated in Fig. 21. The transition rule for *AND* tasks (used in such representations) is shown in Fig. 22.

2.10 Assignment Expressions

A simple assignment has the following form:

assignment-expression → *expr1* = *expr2*

where *expr1* and *expr2* are expressions.

To evaluate a simple assignment expression, copy the value of *expr2* into the memory location given by *expr1*, returning that value as the value of the parent expression.

This is the first occurrence of an operator in Algebra Two with an ambiguous evaluation order, as discussed in Sect. 2.5. Thus, the expression is represented in our algebra as in Fig. 14. The transition rule for assignment operators is shown in Fig. 23.

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*. For example, “`i *= 2`” has the same value and effect as “`i = i * 2`”. However, it is not true in general that “`a` *op*`= b`” can be seen as an abbreviation for “`a = a` *op* `b`”, since the expression `a` is evaluated only once in the former expression, but twice in the latter. Since evaluating expressions in C may cause side effects, the distinction is important. Thus, we must present additional transition rules for such operators.

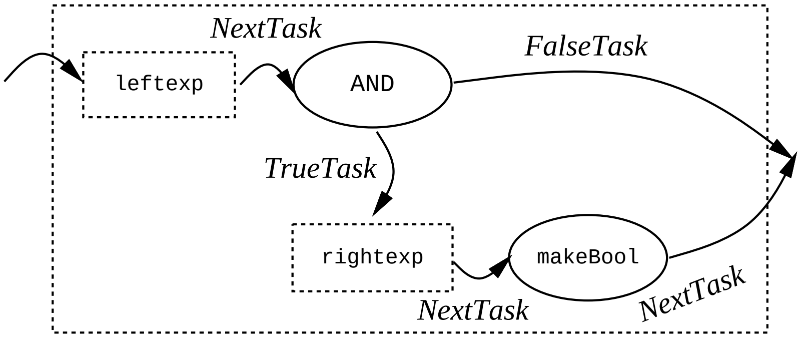

Figure 21: Logical AND expression.

```
if TaskType(CurTask) = AND then
    if OnlyValue(CurTask) = 0 then
        ReportValue(0)
        Moveto(FalseTask(CurTask))
    elseif OnlyValue(CurTask) $\neq$ 0 then
        Moveto(TrueTask(CurTask))
ENDIF
```

Figure 22: Transition rule for *AND* tasks.

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

Figure 23: Transition rule for simple assignment tasks.

With two exceptions, all these assignment operators have transition rules like that shown in Fig 24 for the multiplicative assignment operator (“*=”).

```
if TaskType(CurTask) = multiplicative-assignment then
    EVALUATE OPERANDS WITH
        DoAssign( LeftValue(CurTask),
                    MemoryValue[LeftValue(CurTask), ValueType(CurTask)]
                        * RightValue(CurTask),
                    ValueType(CurTask))
        ReportValue(MemoryValue[LeftValue(CurTask), ValueType(CurTask)]
                        * RightValue(CurTask))
    END EVALUATE
endif
```

Figure 24: Transition rule for multiplicative assignment.

The two exceptions are the additive and subtractive assignment operators, “+=” and “-=”. In C, one may add an integer i to a pointer expression p, with the result being a pointer which is i units forward in memory from p. Thus, to process ‘‘a += b’’, we must perform different actions if a is a pointer variable. The transition rule for additive assignment involving pointers is shown in Fig 25. The rule for additive assignment involving non-pointers is like that shown above for multiplicative assignments, with the additional condition *PointerType(LeftChild(CurTask)) = false* inserted in the guard of the rule. The rules for subtractive assignment are similar and thus omitted.

```
if TaskType(CurTask) = additive-assignment
            and PointerType(LeftChild(CurTask)) = true then
    EVALUATE OPERANDS WITH
        DoAssign( LeftValue(CurTask),
                    MemoryValue(LeftValue(CurTask), ValueType(CurTask))
                     + Size(PointsToType(CurTask)) * RightValue(CurTask),
                    ValueType(CurTask)
        ReportValue(MemoryValue(LeftValue(CurTask), ValueType(CurTask))
                     + Size(PointsToType(CurTask)) * RightValue(CurTask))
    END EVALUATE
endif
```

Figure 25: Transition rule for additive assignment with pointers.

2.11 The sizeof Operator

Expressions involving **sizeof** have one of the following forms:

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

The *ValueType* function converts either operand 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 rule for size operators is shown in Fig. 26.

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

Figure 26: Transition rule for *sizeof* tasks.

2.12 Constants

For constant expressions, the *ConstVal* function returns the appropriate value. (Note that we treat enumerated type values as constants.) The transition rules for constants is shown in Fig. 27.

```
if TaskType(CurTask) = constant then
    ReportValue(ConstVal(CurTask))
    Moveto(NextTask(CurTask))
endif
```

Figure 27: Transition rule for *constant* tasks.

2.13 General Mathematic Expressions

There are a large number of mathematic expressions in C involving binary operators (“`*`”, “`+`”, “`-`”, *etc.*) whose behaviors are similar. (We treat the bit-wise operators (*e.g.*, `|`, `&`) as ordinary mathematic operators.) We assume that infix functions corresponding to these C functions are present within our algebra. Thus, to evaluate one of these expressions, evaluate both operand expressions and apply the appropriate function. We present the transition rule for multiplication in Fig.28 as a representative of this category of expressions, and omit the rules for other binary operators of this form for brevity.

Rules for the addition and subtraction operators are slightly more complicated, since one may add or subtract an integer to a pointer. One may add an integer i to a pointer variable p with the result being a pointer which is i units forward in memory from p. Consider, for example, a pointer p to an `int` in a system where `int`s require 4 bytes in memory. The expression “$p+1$” refers to the location in memory 4 bytes after p.

Similarly, one may subtract an integer i from a pointer variable p, with the result being a pointer i units in memory preceding p. (In our example, “$p-1$” refers to the location in memory 4 bytes before p.) Further, one may subtract two pointers of the same type, resulting in the number of units of memory lying between

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

Figure 28: Transition rule for *multiplication* tasks.

the two pointers. (Thus, $((p + i) - p) = i$.) In each case, the size of a "unit" of memory is determined by the size of the object type to which the pointer points.

This requires specialized rules for 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; a static partial function *PointerType: tasks* → {*true, false*} contains this information. For tasks for which *PointerType* returns *true*, a static partial function *PointsToType : tasks* → *typename* indicates the object type to which the pointer points.

The transition rules for the addition and subtraction operators are shown in Fig. 29 and Fig. 30.

```
if TaskType(CurTask) = addition then
    EVALUATE OPERANDS WITH
        if PointerType(LeftTask(CurTask)) = true then
            ReportValue(LeftValue(CurTask)
                + (Size(PointsToType(CurTask)) * RightValue(CurTask)))
        elseif PointerType(RightTask(CurTask)) = true then
            ReportValue(RightValue(CurTask)
                + (Size(PointsToType(CurTask)) * LeftValue(CurTask)))
        else
            ReportValue(LeftValue(CurTask) + RightValue(CurTask))
        endif
        Moveto(NextTask(CurTask))
    END EVALUATE
endif
```

Figure 29: Transition rule for *addition* tasks.

2.14 Mathematical Unary Operators

Unary operator expressions have one of the following forms:

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

```
if TaskType(CurTask) = subtraction then
    if PointerType(LeftTask(CurTask)) = true and
            PointerType(RightTask(CurTask)) = true then
        ReportValue((LeftValue(CurTask) - RightValue(CurTask))
                / Size(PointsToType(CurTask)))
    elseif PointerType(LeftTask(CurTask)) = true and
            PointerType(RightTask(CurTask)) = false then
        ReportValue(LeftValue(CurTask)
            - (Size(PointsToType(CurTask)) * RightValue(CurTask)))
    else
        ReportValue(LeftValue(CurTask) - RightValue(CurTask))
    endif
    Moveto(NextTask(CurTask))
endif
```

Figure 30: Transition rule for *subtraction* tasks.

Evaluating these expressions takes a form similar to that for binary mathematical operators. We present the transition rule for the negation operator in Fig. 31 as a representative example.

```
if TaskType(CurTask) = negation then
    ReportValue( - OnlyValue(CurTask))
    Moveto(NextTask(CurTask))
endif
```

Figure 31: Transition rule for *negation* tasks.

2.15 Casting Expressions

A casting expression has the following form:

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

A static function *CastType: tasks* → *typename* indicates the old type from which the value of the expression is to be cast; *ValueType* indicates the new type into which the value will be cast. A static function *Convert: typename* × *typename* × *values* → *values* converts elements from one universe into the corresponding elements of another universe. For example, *Convert(float,int,X)* is the closest integer to X (assuming X is a floating-point value). Note that the meaning of "closest" is implementation-defined.

To perform a cast, evaluates the argument expression and use the *Convert* function to generate the proper return value. Our task sequence places the expression to be cast before the task which performs the casting; thus, the argument of the cast has been evaluated already and its value is available. The transition rule for casting expressions is shown in Fig. 32.

```
if TaskType(CurTask) = cast then
    ReportValue(Convert( CastType(CurTask),
                        ValueType(CurTask), OnlyValue(CurTask)))
    Moveto(NextTask(CurTask))
endif
```

Figure 32: Transition rule for *cast* tasks.

2.16 Pre-Increment and Pre-Decrement

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

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

To evaluate a pre-increment (resp. pre-decrement) expression, increment (decrement) the value stored at the indicated memory location by one and store the new value into that memory location; the incremented (decremented) value is the value of the parent expression. Note that the expression to be modified may be a pointer; in this case, the value in memory is incremented (decremented) by the size of the object to which the pointer points (as with normal pointer addition and subtraction). The transition rule for pre-increment expressions is shown in Fig 33. The transition rules for pre-decrement expressions are similar to those presented here and thus omitted.

```
if TaskType(CurTask) = pre-increment then
    if PointerType(CurTask) = true then
        DoAssign(OnlyValue(CurTask),
            MemoryValue(OnlyValue(CurTask), ValueType(CurTask))
                + Size(PointsToType(CurTask)),
            ValueType(CurTask))
        ReportValue(  MemoryValue( OnlyValue(CurTask),
                                   ValueType(CurTask))
                      + Size(PointsToType(CurTask)))
    else
        DoAssign(OnlyValue(CurTask),
            MemoryValue(OnlyValue(CurTask), ValueType(CurTask)) + 1,
            ValueType(CurTask))
        ReportValue(MemoryValue(  OnlyValue(CurTask),
                                  ValueType(CurTask))+1)
ENDIF
```

Figure 33: Transition rule for *pre-increment* tasks.

2.17 Post-Increment and Post-Decrement

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

postincr-expression → *expression* **++**
postdecr-expression → *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: *i.e.*, the value of the parent expression is established before the incrementing (decrementing) takes place. The transition rule for the post-increment operator is shown in Fig. 34. (As before, the transition rules for the post-decrement operator are similar and thus omitted.)

```
if TaskType(CurTask) = post-increment then
    if PointerType(CurTask) = true then
        DoAssign(OnlyValue(CurTask),
            MemoryValue(OnlyValue(CurTask), ValueType(CurTask))
                + Size(PointsToType(CurTask)),
            ValueType(CurTask))
    else
        DoAssign(OnlyValue(CurTask),
            MemoryValue(OnlyValue(CurTask), ValueType(CurTask))+1,
            ValueType(CurTask))
    endif
    ReportValue(MemoryValue(   OnlyValue(CurTask),
                               ValueType(CurTask)))
endif
```

Figure 34: Transition rule for *post-increment* tasks.

2.18 Addresses

An addressing expression has the following form:

addressing-expression → **&** *expr1*

where *expr1* is an expression.

The & operator in C passes back as its result the address of the memory location indicated by the argument expression.

As we evaluate expressions which refer to objects in memory, we need to know whether we need to use the address of an object or the object itself in our calculations. (For example, in the assignment statement "`a = b;`", the address or *lvalue* of variable `a` is needed, but the object being referenced or *rvalue* of variable `b` is needed.) A static partial function *ValueMode: tasks* → {*lvalue, rvalue*} indicates which of the two pieces of information should be computed for a given task.

We assert that *ValueMode(e) = lvalue* for the sub-expressions of *expr1*; thus, the value returned through evaluation of the argument expression is the address (and not the value) of the argument expression in memory. We simply pass this address up the task graph. The resulting simple transition rule for the addressing operator is shown in Fig. 35.

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

Figure 35: Transition rule for *address* tasks.

2.19 De-Referencing

A de-referencing expression has the following form:

de-reference-expression → * *expression*

If the parent expression is an rvalue, evaluate the argument and use the *Memory* function to return the value stored in memory at the indicated location. Otherwise, return the address indicated by the argument (since the expression is an lvalue and requires that a pointer be returned to the parent expression). The transition rule for de-referencing is shown in Fig. 36.

```
if TaskType(CurTask) = de-referencing then
    if ValueMode(CurTask) = rvalue then
        ReportValue(MemoryValue(  OnlyValue(CurTask),
                                  ValueType(CurTask)))
    elseif ValueMode(CurTask) = lvalue then
        ReportValue(OnlyValue(CurTask))
    endif
    Moveto(NextTask(CurTask))
endif
```

Figure 36: Transition rule for *de-referencing* tasks.

2.20 Array References

An array reference has the following form:

array-ref-expression → *expr1* [*expr2*]

where *expr1* and *expr2* are expressions.

According to [KR], an array reference of the form `a[b]` is identical, by definition, to the expression `*((a)+(b))`.[3] 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.

We assert that any array references present in the program being modeled in our algebra are represented as an expression of equivalent form involving addition and de-referencing. Thus, we do not need to present any additional rules to handle array references.

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

One may prefer to think of arrays as objects in their own right, and present an algebra intermediate to Algebras One and Two where expressions like `a[b]` could be treated at a higher level of abstraction. While such a presentation is possible, we choose not to do so here.

2.21 Function Invocations

A function invocation has the following form:

func-invocation-expression → *expression (expression-list)*

Since we have disallowed function invocations for the moment, we will obtain the value of a function invocation expression from an external function *FunctionValue: tasks* → *results*. The transition rule for function invocations is shown in Fig. 37.

```
if TaskType(CurTask) = function-invocation then
    ReportValue(FunctionValue(CurTask))
    Moveto(NextTask(CurTask))
endif
```

Figure 37: Transition rule for *function-invocation* tasks.

2.22 Identifiers

An external function *FindID: tasks* → *addresses* maps identifier expression tasks to the corresponding memory location used by the associated variable. (In Algebra Three we shall eliminate the use of this function.) Thus, to handle an identifier expression, one returns the appropriate address or value from memory, as specified by the *ValueMode* function. The transition rule for identifiers is shown in Fig. 38.

```
if TaskType(CurTask) = identifier then
    if ValueMode(CurTask) = lvalue then
        ReportValue(FindID(CurTask))
    elseif ValueMode(CurTask) = rvalue then
        ReportValue(MemoryValue(  FindID(CurTask),
                                  ValueType(CurTask)))
    endif
    Moveto(NextTask(CurTask))
endif
```

Figure 38: Transition rule for *identifier* tasks.

2.23 `struct` or `union` References

A `struct` or `union` reference has the following form:

struct-expression → *expr1* . *identifier*

where *expr1* is an expression evaluating to a `struct` or `union`. There is also another form:

struct-expression → *expr2* `->` *identifier*

where *expr2* is an expression evaluating to a pointer to a `struct` or `union`. Expressions of the form "`a->b`" are equivalent to those of the form "`(*a).b`". Thus, we will only consider references of the form "`a.b`", asserting that references of the other form are represented using their equivalent expansions.

The *ConstVal* function applied to the `struct` reference task returns the offset in memory to be used in obtaining the address or value of the specified field of the structure. The transition rule for struct references is shown in Fig. 39.

```
if TaskType(CurTask) = struct-reference then
    if ValueMode(CurTask) = lvalue then
        ReportValue(OnlyValue(CurTask) + ConstVal(CurTask))
    elseif ValueMode(CurTask) = rvalue then
        ReportValue(MemoryValue(OnlyValue(CurTask)
                                                + ConstVal(CurTask),
                                        ValueType(CurTask)))
    endif
    Moveto(NextTask(CurTask))
endif
```

Figure 39: Transition rule for *struct-reference* tasks.

2.24 Bit Fields

Bit fields are members of `structs` which use a user-specified number of bits for their representations. Bit fields are used to minimize the space used by a `struct` or to represent accurately input or output values with bit-level significance. Much about bit fields behave is implementation-dependent: *e.g.* how bit fields are packed into adjacent bytes, whether or not unnamed "holes" will appear in `structs` between bit fields, or whether bit fields are read left-to-right or right-to-left. As a rule, operaptions are done with bytes (even bit operations); we show how to handle bit-fields in a byte-based model of memory.

An example of a `struct` using bit fields is shown in Fig. 40. Here, the field `b` holds a 4-bit unsigned integer and `c` holds a 15-bit unsigned integer. Since bytes usually comprise 8 bits, `c` will probably lie in two or three consecutive bytes in memory, possibly sharing a byte with `b`.

Suppose that we want to execute `bitty.c = 12`. We need to obtain the bytes which hold `c`'s bits and modify those bits accordingly while leaving all other bits unchanged. This gives rise to a static function *BitAssign: results* × *typename* × *results* → *results* which indicates the change occurring in the value of the appropriate collection of contiguous bytes when a bit field is modified. Given the content *oldval* of an appropriate piece of memory, *BitAssign(oldval, bittype, 12)* returns the new value; here *bittype* is the type of `c` in `bitty`.

```
struct {
   unsigned int a;
   unsigned int b:4;
   unsigned int c:15;
   } bitty;
```

Figure 40: Example of a `struct` using bit fields.

```
if TaskType(CurTask) = bit-assignment then
   EVALUATE OPERANDS WITH
      DoAssign(LeftValue(CurTask),
         BitAssign(MemoryValue(    LeftValue(CurTask),
                                   ValueType(CurTask)),
               ValueType(CurTask),
               RightValue(CurTask)),
         ValueType(CurTask))
      ReportValue(RightValue(CurTask))
   END EVALUATE
endif
```

Figure 41: Transition rule for *bit-assignment* tasks.

Assigning to bit-fields is thus slightly different than usual. We present the transition rule for simple assignments to bit fields in Fig. 41.

Evaluating bit-field references is also slightly different, since we need to extract the value of the bit field from the (usually) larger enclosing value. A static function *BitExtract: results* × *typename* → *results* performs this extraction. A static partial function *BitType: tasks* → *typename* indicates the type of bit field being references in such situations.

The transition rule for structure bit-field references is shown in Fig. 42.

3 Algebra Three: Allocating and Initializing Memory

Algebra Three refines Algebra Two and focuses on memory allocation and initialization of variables.

3.1 Declarations

We represent declarations in C as elements of the *tasks* universe, linked in the proper order with statement tasks by *NextTask*.

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 task 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, 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.

```
if TaskType(CurTask) = struct-bit-reference then
    if ValueMode(CurTask) = lvalue then
        ReportValue(OnlyValue(CurTask) + ConstVal(CurTask))
    elseif ValueMode(CurTask) = rvalue then
        ReportValue(BitExtract(
                MemoryValue(OnlyValue(CurTask) + ConstVal(CurTask),
                BitType(CurTask)),
            ValueType(CurTask)))
    endif
    Moveto(NextTask(CurTask))
endif
```

Figure 42: Transition rule for *struct-bit-reference* tasks.

A static partial function *DecType: tasks* → {*static, non-static*} indicates what type of variable is being declared. (Note that there are also `extern` variable 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.) A static partial function *Initializer: tasks* → *tasks* indicates the appropriate initializing expression (if any). We will store the value of the initializing expression using *RightValue*.

A partial function *StaticAddr: tasks* → *addresses* stores the current address (if any) that has been assigned to a static variable. In Algebra Three, *StaticAddr* is not really needed, since the *OnlyValue* function would provide the proper storage for the address of the static variable. However, it will simplify other rules to be presented. An external function *NewMemory: tasks* → *addresses* returns an address in memory to be used for the given declaration task. The transition rules for declarations are shown in Fig. 43 and Fig. 44.

3.2 Automatic Variables and Non-Local Jumps

Automatic (or local) variables are allocated memory not only when a block is entered normally, but also if a non-local `goto` statement transfers control into a block. As an example, consider the following code fragment:

```
int flag;                      for (;;) {
 .                                int local = 1;
 .
if (flag) goto target;            printf("%i" ,local);
 .
 .                                target:  local = 0;
                                  printf("%i" ,local);
                               }
```

In this fragment, if the `for` statement is entered normally, variable `local` will be allocated memory and initialized to 1. If the `goto` statement is executed, variable `local` should be allocated memory before the statement labeled `target` is executed, although no initialization is performed.

Our rules for constructing task graphs only provide for a unique moment when a given variable may be allocated. Consequently, we introduce a new task type *indirect-declaration*. Previously, *NextTask* mapped each *jump* task directly to its target. Now, we put an *indirect-declaration* task for each local variable between

```
if TaskType(CurTask) = declaration and DecType(CurTask) = static then
    if StaticAddr(CurTask) $\neq$ undef then
        OnlyValue(CurTask) := StaticAddr(CurTask)
        Moveto(NextTask(CurTask))
    elseif StaticAddr(CurTask) = undef then
        if Initializer(CurTask) $\neq$ undef and
            RightValue(CurTask) = undef then
            Moveto(Initializer(CurTask))
        else
            OnlyValue(CurTask) := NewMemory(CurTask)
            StaticAddr(CurTask) := NewMemory(CurTask)
            if Initializer(CurTask) $\neq$ undef then
                DoAssign(NewMemory(CurTask),
                    RightValue(CurTask), ValueType(CurTask))
            else
                Moveto(NextTask(CurTask))
ENDIF
```

Figure 43: Transition rule for static declarations.

```
if TaskType(CurTask) = declaration and DecType(CurTask) $\neq$ static then
    if Initializer(CurTask) $\neq$ undef and
        RightValue(CurTask) = undef then
        Moveto(Initializer(CurTask))
    else
        OnlyValue(CurTask) := NewMemory(CurTask)
        if Initializer(CurTask) $\neq$ undef then
            DoAssign(NewMemory(CurTask),
                RightValue(CurTask), ValueType(CurTask))
        else
            Moveto(NextTask(CurTask))
ENDIF
```

Figure 44: Transition rule for non-static declarations.

the *jump* task and its target. The *Decl* function will map each new task to the original declaration of that variable. The rule for handling indirect declarations (shown in Fig. 45) is pretty obvious.

```
if TaskType(CurTask) = indirect-declaration then
    OnlyValue(Decl(CurTask)) := NewMemory(Decl(CurTask))
    Moveto(NextTask(CurTask))
endif
```

Figure 45: Transition rule for indirect declarations.

3.3 Revision: Identifiers

A static partial function *Decl: tasks* → *tasks* maps tasks corresponding to occurrences of an identifier to the task corresponding to the declaration task for that variable. The revised rule for identifiers (generated by replacing each previous occurrence of *FindID()* with *OnlyValue(Decl())*) is shown in Fig. 46.

```
if TaskType(CurTask) = identifier then
    if ValueMode(CurTask) = lvalue then
        ReportValue(OnlyValue(Decl(CurTask)))
    elseif ValueMode(CurTask) = rvalue then
        ReportValue(MemoryValue(   OnlyValue(Decl(CurTask)),
                                   ValueType(CurTask)))
    endif
    Moveto(NextTask(CurTask))
endif
```

Figure 46: Revised transition rule for *identifier* tasks.

3.4 Initializers

Initializers in C come in two forms: expressions (for variables of the basic types) and lists of expressions (for variables representing arrays and structures).

Our previous rules for evaluating expressions will handle initializers for simple expressions. To assist in handling aggregate expressions, a static function *AddTo: typename* × *results* × *results* → *results* appends a value onto the end of an aggregate structure of the specified type. (For example, if $[42, 8]$ is an integer array, then $AddTo(array, [42, 8], 4) = [42, 8, 4]$.)

We illustrate how expressions with aggregate initializers (*i.e.* expressions whose initializer is an expression list) are represented in our algebra by the graph in Fig. 47. Our previous transition rules will insure that each expression in the initializer list will be evaluated; we need to provide rules that combine the results of these evaluations into the proper aggregate value.

The *WhichChild* function returns the value *aggregate* when the expression being evaluated is a component of an aggregate initializer. We extend our *ReportValue* macro as shown in Fig. 48 to correctly combine aggregate expressions.

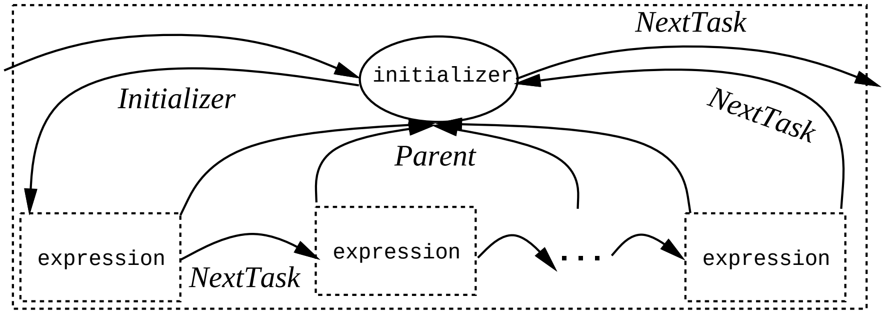

Figure 47: Aggregate initializer.

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

Figure 48: Extension of the *ReportValue* macro.

3.5 Initial State

We assert that initially, *CurTask* indicates the first declaration task for the program, or the first task of the first statement of the program if no declaration tasks exist.

4 Algebra Four: Handling Function Definitions

Algebra Four revises Algebra Three and focuses on C function definitions. In this way, we also (implicitly) present rules for starting a C program, since the starting function `main` is an ordinary C function (with externally provided parameters).

(There are also function declarations in C, which are used to specify syntactic information. Since their purpose is wholly syntactic in nature, we ignore them.)

4.1 Modeling The Stack

C functions may have several active incarnations at a given moment. Thus, we must have some means for storing multiple values of a function for a given task.

The universe *stack* comprises the positive integers, with a distinguished element *StackRoot = 1*. Static functions *StackPrev: stack* → *stack* and *StackNext: stack* → *stack* are the predecessor and successor functions on the positive integers. A dynamic distinguished element *StackTop* indicates the current top of the stack.

To store state-associated information on the stack, we modify the various *Value* functions *LeftValue, RightValue, OnlyValue,* and *TestValue* to be binary functions from *tasks* × *stack* to *results*. This requires us to rewrite almost every rule that has appeared previously; we simplify matters by stating that every previous reference to *V(X)* should be replaced by *V(X, StackTop)*, where *V* is one of the *Value* functions listed above. Similarly, we modify *Visited* to be a binary fruntion from *tasks* × *stack* to *tags*, replacing all previous occurrences of *Visited(X)* with *Visited(X, StackTop)*.

4.2 Function Invocations: Caller's Story

A function invocation has the following form:

func-invocation → *func-name (expression-list)*

In Algebra Two we used an external function *FunctionValue* to obtain the value of a function invocation. Here, we eliminate the use of this function.

The name of a function is often an identifier, but in general it is an expression referring to the address of the function. (What resides in memory at that address is implementation-dependent.) A static partial function *AddrToFunc: addresses* → *tasks* maps function addresses to the first task of the function definition.

While processing a function invocation, we wish to copy the value[4] of each parameter to an appropriate place for the callee to process. The *Parent* function (utilized by our *ReportValue* macro) maps each argument-expression task to its corresponding function parameter task. A partial function *ParamValue: tasks* × *stack* → *results* indicates the values of parameters being passed. We append the rule shown in Fig. 49 to the *ReportValue* macro.

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

Figure 49: Extension of the *ReportValue* macro.

We need to store the current task in order to resume execution at this point after the callee has finished. A dynamic function *ReturnTask: stack* → *tasks* indicates the new value of *CurTask* when the execution of the current function terminates. We assert that when *CurTask = 1*, *ReturnTask(CurTask) = undef*, which will cause the algebra to terminate when the top-level function terminates.

To process a function invocation, evaluate the name of the function along with all of the arguments in the expression list, and then transfer control to the specified function. At the same time, "push another frame onto the stack"; *i.e.*, increment *StackTop* by 1. When control returns from the function, the stack value will be "popped" (*i.e.*, decremented by 1) and the function's return value will be passed to the parent expression.

As with the operands to most arithmetic operators, the ANSI standard [KR] does 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. The external function *ChooseTask* 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 will occur when *ChooseTask* returns *undef*. The transition rule for function invocation is shown in Fig. 50.

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

```
if TaskType(CurTask) = function-invocation then
    if ChooseTask(CurTask) $\neq$ undef then
        Moveto(ChooseTask(CurTask))
    elseif ChooseTask(CurTask) = undef then
        if OnlyValue(CurTask,StackTop) = undef then
            StackTop := StackNext(StackTop)
            ReturnTask(StackNext(StackTop)) := CurTask
            Moveto(AddrToFunc(LeftValue(CurTask,StackTop)))
        elseif OnlyValue(CurTask,StackTop) $\neq$ undef then
            ReportValue(OnlyValue(CurTask,StackTop))
            Moveto(NextTask(CurTask))
ENDIF
```

Figure 50: Revised transition rule for *function-invocation* tasks.

4.3 Function Invocations: Callee's Story

A function definition in C consists of a list of parameter declarations and a compound statement. To process a parameter declaration, we 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 rule for parameter declarations is shown in Fig. 51.

```
if TaskType(CurTask) = parameter-declaration then
    DoAssign(NewMemory(CurTask),
        ParamValue(CurTask,StackTop), ValueType(CurTask))
    OnlyValue(CurTask,StackTop) := NewMemory(CurTask)
endif
```

Figure 51: Transition rule for *parameter-declaration* tasks.

A `return` statement has one of the following forms:

return-statement → `return` ;
return-statement → `return` *expression* ;

If an expression is present, copy the value of the expression to the task which invoked the current function (as indicated by *ReturnTask* and *StackPrev*). Whether or not an expression is present, return control to the invoking task. The transition rule for `return` statements is shown in Fig. 52.

If a `return` statement is not explicitly present at the end of a function, our algebra will still contain a *return* task as the last task of the function, as if the statement "`return ;`" was present as the last statement of the original C function.

```
if TaskType(CurTask) = return then
    OnlyValue(ReturnTask(StackTop),StackPrev(StackTop)) :=
        OnlyValue(CurTask,StackTop)
    StackTop := StackPrev(StackTop)
    Moveto(ReturnTask(StackTop))
endif
```

Figure 52: Transition rule for *return* tasks.

4.4 Global Variables

Since function definitions may not contain other function definitions, a given variable identifier refers either to a variable local to the current function or to a variable declared outside any function. A static partial function *GlobalVar: tasks* → *Bool* indicates whether or not a given identifier refers to a global variable. We present the modified transition rule for identifiers in Fig. 53.

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

Figure 53: Revised transition rule for identifiers.

4.5 Initial State

We assert that initially, *CurTask* indicates the first declaration task for the global variables of the program, or the first task of the first statement `main()` if no declaration tasks exist.

References

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

[Gu] Yuri Gurevich, "Evolving Algebras: An Introductory Tutorial", Bulletin of European Assocation for Theoretical Computer Science, February 1991. (A slightly updated version will appear in the EATCS Book of Columns, World Scientific Publishers.)

[GH1] Yuri Gurevich and James K. Huggins, "The Evolving Algebra Semantics of C: Preliminary Version", CSE-TR-141-92, EECS Department, University of Michigan, 1992.

[GH2] Yuri Gurevich and James K. Huggins, "The Semantics of the C Programming Language", Selected papers from *CSL'92* (Computer Science Logic), Springer Lecture Notes in Computer Science 702, 1993, 274–308.

[GH3] Yuri Gurevich and James K. Huggins, "Errata to 'The Semantics of the C Programming Language' ", Selected papers from *CSL'93*, Springer Lecture Notes in Computer Science, to appear.

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

Index