The Semantics of the C++ Programming Language

Charles Wallace*

November 23, 1993

1 Introduction

In this paper we extend the evolving algebra presented in [GH] to give formal operational semantics for the C++ programming language. The evolving algebra of [GH] is a specification for the C programming language; in this paper we propose modifications to it to accommodate the features of C++. We refer the reader to [GH] for a description of the C algebra, and to [Gu] for an introduction to evolving algebras. [KR] describes the ANSI standard for the C language, on which the algebra of [GH] is based. We assume the informal specification for C++ in [ES] as guidelines for our semantics. Knowledge of C++ will ease comprehension but is not necessary, as we shall explain the new features of C++ and illustrate their use with examples as we proceed.

The C++ programming language is designed to be an "extension" of C, retaining all of C's language facilities and adding new ones. On a syntactic level, the differences between C and C++ consist entirely of language constructs allowable in C++ but not in C. Our algebra for C++ will alter the rules for C in [GH], maintaining their functionality while extending them to capture the new features of C++. Many of the extensions we shall consider do not require any changes at all to the rules. Some extensions, such as derived classes, affect only static information about the program, determined when the program is compiled and never changed during the running of the program. Other extensions, such as the keyword `class`, simply constitute syntactic alternatives to constructs present in C. In addition to proposing changes to the rules, we shall discuss the extensions which do not require any rule changes and explain how we can handle them.

1.1 C++ and object-oriented programming

The new features of C++ support the "object-oriented" programming paradigm. The term "object" can be defined simply as the instantiation of a type. This approach to programming is a synthesis of several principles, which we summarize as follows. First, object-oriented programming supports the inclusion of *operations* performed on an object within the definition of the object's type. New types can be defined in terms of preexisting types through *inheritance*. Finally, access to an object's data is localized through *encapsulation*.

As our algebra must specify not only the new features of C++ but also the C subset, it shares with the algebra of [GH] the relatively low level of abstraction necessary to specify the features of C. For example, C allows the programmer direct access to memory, so we must model memory as a linear ordering of locations. Moreover, objects must be modeled as contiguous sequences of locations. While this attention to memory considerations is necessary for an accurate specification, it obscures the object-oriented nature of C++. In fact, the concept of "object" *per se* is absent from the algebra; there is a universe of memory locations but no universe of objects.

An interesting alternative to this algebra would be one in which the details of memory are removed. Objects could then be modeled as such, *i.e.*, as elements of a universe of objects. These objects would have

*Partially supported by NSF grant #029862 and ONR grant #028355. The author's address is: EECS Department, University of Michigan, Ann Arbor, MI 48109-2122. Electronic mail address: wallace@eecs.umich.edu.

locations associated with them, but of a more abstract nature; there would be no assumption of an ordering of locations. Such an algebra could be considered more "object-oriented" in nature. Of course, such an algebra would represent only a fragment of C++, as the object-oriented paradigm can be easily subverted in C++. For instance, the assumption that all data accessible to a program is in the form of objects can be violated; C++ allows direct reference to any memory location, even locations not associated with any object. The form of this high-level algebra for C++ and the restrictions on C++ it would require are topics for further investigation.

1.2 Outline

The features required to implement encapsulation and inheritance are presented in section 2, while those required to combine type and operation definitions are presented in section 3. Section 4 deals with the features supporting creation and destruction or objects. In sections 5 and 6 we discuss extensions that are not object-oriented in nature; section 5 concerns overloading and parameterized type definitions, and section 6 covers the remaining extensions.

In the interest of readability, we define a set of macros for commonly used rule expressions. The definitions of these macros appear in appendix A.

1.3 Acknowledgments

I would like to thank Yuri Gurevich for inspiring me to write this paper and guiding me during its development. I would also like to thank Jim Huggins and Solomon Foster for their many helpful comments.

2 Class structure and encapsulation

The central notion of the object-oriented programming paradigm is the encapsulation of data types and operations associated with them. Encapsulation ensures that the data stored within an object an instantiation of a given type is accessed only by the operations associated with the type. This localization of access is conducive both to data security and to good programming style. Encapsulation is achieved in C++ through the "class" construct, which defines aggregate types similar to "struct" types in C. A class type combines the components of a programmer-defined data type with the functions and operators to manipulate it. The notion of a class is introduced in section 2.1. The level of localization of access is achieved by specifying access status for the components of the type; access status is discussed in section 2.4.

In object-oriented programming, redundant code may be eliminated by allowing one type to "inherit" the data structure and operations of another. The inherited structure and operations may then be modified or extended to suit the new type. The notion of inheritance is called "derivation" in C++; the features supporting class derivation are discussed in sections 2.2 and 2.3.

2.1 Classes

C++ introduces a new keyword `class` which indicates the definition of a new class type. This is almost identical in functionality to the C keyword `struct`. Both define types whose instantiations are contiguous sequences of ordered fields (or "members") in memory; in C++, both may have operations, or "member functions," associated with them. The only difference between the two is in the default "access status" assigned to their fields.[1] As we shall see in section 2.4, access status is itself a C++ extension of a purely syntactic nature; thus we can treat class-types in the same way we treat struct-types in the C algebra, with no rule modifications necessary. We hereby adopt C++ terminology: we shall use the term "class" to refer to both struct- and class-types and the term "member" as a synonym for "field." In addition, we shall use

[1]The default access status is "public" for a struct-type and "private" for a class-type. The default status is assigned to a field if no status is specified.

the term "object" to refer to a contiguous area of memory serving as an instantiation of a particular type; in particular, the term "class object" refers to an area allocated as an instantiation of a class- or struct-type.

2.2 Derived classes

In addition to the members defined explicitly in its declaration, a class may inherit the members of a set of other classes. A class that inherits members is said to be "derived"; the classes whose members it inherits are its "base" classes. The base classes of a derived class are specified in its declaration. For example, given the class `person` containing the members `name` and `age`:[2]

```
// declaration of (nonderived) class person
class person {                                                   // (no members inherited)
  char name[25];                                                 // person's name (string)
  int age;                                                       // person's age (integer)
  void printPerson();                                // function to print person's name and age
};
```

we can add the derived classes `professor` and `student`:

```
// declaration of derived class professor
class professor:  person {                               // (inherits members of class person)
  int salary;                                                 // professor's salary (integer)
  void printProfessor();                                  // function to print professor's info
};
// declaration of derived class student
class student:  person {                                 // (inherits members of class person)
  int year;                                                       // student's year (integer)
  float GPA;                                                      // student's GPA (decimal)
  void printStudent();                                      // function to print student's info
};
```

Professors and students in the real world are individuals with names and ages, as well as professor- and student-specific attributes. The classes `professor` and `student` represent this by inheriting the members of the class `person`. Both derived classes contain `name`, `age` and `printPerson`, the members of their base class; in addition, the class `professor` contains the members `salary` and `printProfessor`, while the class `student` contains `year`, `GPA` and `printStudent`.

From these two derived classes we can create another derived class, `teachingAssistant`:

```
// declaration of derived class teachingAssistant
class teachingAssistant:  professor, student {
  // (inherits members of classes professor and student)
  professor *worksFor;       // professor that TA works for (pointer to professor object)
  int section;                                          // section that TA teaches (integer)
  void printTA();                                             // function to print TA's info
  };
```

In the real world, a teaching assistant is a single individual with attributes of both a professor and a student. We represent this via "multiple inheritance": class `teachingAssistant` contains the members of both `professor` and `student`, as well as the members `worksFor`, `section` and `printTA`.

An object of a nonderived class consists of a sequence of members arranged contiguously in memory; the members are arranged according to the order in which they appear in the class declaration. An object of

[2] In C++, a pair of slash characters (//) indicates the beginning of a one-line comment. Our C++ examples include comments for clarificational purposes; the text of these comments should not be confused with C++ code.

a derived class also consists of a sequence of members; some members are inherited from the base classes, and some are declared in the derived class itself. Unlike nonderived classes, there is no way of determining the ordering of a derived class' members from the class declaration. In particular, the relative order of base- and derived-class members is implementation-dependent; base members may precede derived members, or *vice versa*.

Regardless of the relative ordering chosen by a given implementation, a derived class shares with a nonderived class the property of a fixed ordering of members over all objects of its class. That is, given a derived class `D` with a set of derived members and a set of underived members, each object of class `D` will order these members in the same sequence $m_1..m_n$. Thus given a nonderived class `ND` which declares the same members of `D` in the order $m_1..m_n$, objects of type `ND` will be structurally equivalent to those of type `D`. We shall therefore treat derived classes as if they were declared in a nonderived form; given a base class `B` and a class `D` derived from it, we simply add `B` and `D` as distinct elements in the types universe.

2.3 Virtual base classes

A class may be designated as "virtual"; the virtual status of a class affects the way in which its members are inherited by other classes. Consider our class `teachingAssistant`: this class inherits members from both `professor` and `student` classes. Since both `professor` and `student` classes in turn inherit members from the class `person`, `teachingAssistant` inherits `person`'s members from two bases. For the class `teachingAssistant` as it is currently defined, this means that the class contains two disjoint sets of `person` members. Each object of this class will have two `name` members and two `age` members, conceivably with different values. For certain applications this is desirable,[3] but if we wish to constrain objects of type `teachingAssistant` to a single `name` and `age` value, our definition of `teachingAssistant` as it stands is unsatisfactory.

To remedy this problem, we declare class `person` as a virtual class. Declaring a class as virtual ensures that any class derived from it will contain only one set of its members. We modify our declaration of class `person`, prefixing it with the keyword `virtual`:

```
// modified declaration of class person, designating class as virtual
virtual class person {
  char name[25];
  int age;
  void printPerson();
};
```

With `person` declared as `virtual`, the class `teachingAssistant` still inherits the members of `person` from two bases, but objects of class `teachingAssistant` will contain only a single `name` member and a single `age` member.

Virtual base classes do not require any changes to the algebra. The virtual status of a base class affects only static information about a class subsequently derived from it: namely, the sequence of members it contains. Following our example, if we were to declare `person` nonvirtual, the members contained in `teachingAssistant` may be arranged as the sequence {`name`, `age`, S, `name`, `age`, P, TA}, where S, P and TA

[3]For an example in which duplication of inherited members is desirable, consider a derived class representing a research project between a professor and a student:

```
// declaration of class researchProject
class researchProject:  professor, student {
  char topic[25];                                              // research topic
  int funding;                                  // amount of funding for research
};
```

This class contains information representing a professor and a student, two distinct people. Here it is necessary to inherit separate copies of the `person` members; the `name` and `age` members corresponding to the professor and student will have distinct values. Thus the class `person` should be declared as nonvirtual in this case.

are sequences of the members defined in classes `student`, `professor` and `teachingAssistant`, respectively.[4] Declaring `person` virtual would simply truncate this sequence to {`name`, `age`, S, P, TA}. As the number and types of a class' members are statically determined, and the effect of a class' virtual status extends only to this static information, we may safely ignore it in our algebra.

2.4 Access control

Class members may be specified as "public," "private" or "protected." These specifications restrict the "accessibility" of the members, *i.e.*, the set of functions which may access them. A "private" member may only be accessed by a member function of the class in which it is declared; a function that is not a member of any class or is the member of a different class, even a derived class which inherits the private member, may not refer to it. A "protected" member is less restricted: it may be accessed by a member function of any class in which it is declared or inherited. A "public" member may be accessed by any function, regardless of the function's class membership. For example, let us assign private status to the `name` and `age` members of class `person`:

```
// modified declaration of class person, with access status specified
virtual class person {
private:
  char name[25];
  int age;
public:
  void printPerson();
};
```

Since `name`, `age` and `printPerson` are declared in the class `person`, the private status of `name` and `age` does not prevent `printPerson` from accessing these members; `printPerson` may refer to them within its function body:[5]

```
// definition of printPerson function for class person
person::printPerson() {
  // print the name and age members of the person object
  output("name:", name);                             // note reference to name
  output("age:", age);                                // note reference to age
}
```

On the other hand, `printProfessor` is not declared in the same class as `name` or `age`. Therefore, `name` and `age` cannot be accessed within the body of `printProfessor`. If we had assigned `name` and `age` protected status, `printProfessor` would have been able to access these members, as `printProfessor`'s class `professor` is a derived class containing `name` and `age` members.

Access to a class' private members may be granted to nonmember functions by giving them "friend" status within the class declaration. For example, we can define a global version of our `printPerson` function that is not a member of the `person` class:

```
// definition of global function globalPrintPerson
void globalPrintPerson(person p) {
  output("name:", p.name);                                // reference to name
  output("age:", p.age);                                   // reference to age
}
```

[4]This is only one possible ordering of members; as noted in section 2.2, the relative ordering of derived and underived members is implementation-dependent.

[5]In this and following examples, we assume that the function `output` simply takes a sequence of arguments, of any number, and sends their values to an output device. We do not define the function explicitly.

For the nonmember function `globalPrintPerson` to access the private members `name` and `age`, we must declare it as a friend to `person` within the class declaration:

```
// modified declaration of class person,
// allowing function globalPrintPerson to access private members
virtual class person {
private:
  char name[25];
  int age;
public:
  void printPerson();                                              // member function
  friend void globalPrintPerson(person);                           // global function
};
```

Access status is a purely syntactic feature; since each member's status is assigned in the declaration of the class and cannot be changed, access restrictions can be enforced before the program is run and need not be enforced later. A member's status has no further effect on either the member itself or the functions which access it. We may therefore ignore this feature in our algebra; no rules need to be changed to accommodate it.

2.5 Scope resolution operator

In both C and C++, names may differ in their scope. In C, a name may be either global or local; in C++, the situation is more complex, as a non-global name may have scope over any of a number of nested classes. The possibility of overlapping scopes leads to potential ambiguity. For example, let us add a member `course` to the class `professor`; `course` will itself be a class, containing the members `name`, `studentsEnrolled` and `print`:

```
// modified declaration of class professor, with new member course
class professor:  private person {
private:
  int salary;
  class course {                                    // course that professor teaches
  private:
    char name[25];                                      // name of course (string)
    int studentsEnrolled;                            // number of students (integer)
  public:
    void print();                                // function to print info about course
  };
public:
  void nonvirtualPrint();
  void virtualPrint();
};
```

The class `professor` contains two instances of the member name `name`: one inherited from the base class `person`, and one nested inside the class `course`. Both instances have scope over the nested class `course`. If we introduce a global variable `name`:

```
// declaration of global variable name
char name[25];                                                  // name of university
```

we now have three identical names with scope over the class `course`.[6] Within the body of `course`'s member function `print`, there is a reference to `name`:

[6] Of course, this is bad programming practice; the confusion here could be easily eliminated by choosing more descriptive labels for the three `name` variables.

```
// definition of print function for class course
void professor::course::print() {
  output("course name:", name);                                  // reference to name
  output("students enrolled:", studentsEnrolled);
}
```

The identifier `name` here could conceivably refer to two possible variables: the member `name` defined inside `course` or the global variable `name`. In the event of such a reference, the more local referent of `name`, *i.e.*, the member of `course`, is selected. The global variable `name` is said to be "hidden." Note that the member `name` defined inside `professor` is not a possible referent; within a member function body, only members of the function's class may be referred to by a simple identifier. Thus there are two problems in our example: given the set of C++ features we have considered so far, there is no way to refer to either the global variable or the member of `professor` from within the class `course`.

The scope resolution operator `::` solves both of these problems. Its unary form allows for references to hidden global variables; the single operand is the name of a global variable, and the expression refers to the global variable of that name. Its binary form allows for references to members of enclosing classes. The left-hand operand is the name of an enclosing class, and the right-hand operand is the name of a member of the enclosing class; the expression refers to the member of the given name within the enclosing class of the given name. For example, the function `print` within `course` can refer to the global variable `name` using the unary form of the scope resolution operator, and to the `name` member of `person` via the binary form of the operator:

```
// modified definition of print function for course class,
// using scope resolution operator
professor::course::print() {
  output("university:", ::name);                            // refers to global variable
  output("professor:", professor::name);                   // refers to professor member
  output("course name:", name);                               // refers to course member
  output("students enrolled:", studentsEnrolled);
}
```

Neither form of the scope resolution operator requires changes to the algebra. An expression consisting of an variable name preceded by the unary operator is simply a reference to the global variable of that name; such an expression corresponds to a simple identifier task. An expression involving the binary form of the operator corresponds to a "data-member" task, which we discuss in sections 3.1 and 3.2. We treat such expressions as class-reference tasks, referring to a member within an object and involving a statically determined offset to the member, provided by the *ConstVal* function. The assumptions made in section 3.2 to handle data-member tasks will also handle binary scope resolution expressions.

3 Programmer-defined class operations

In object-oriented programming, the operations that access a given data type are included as part of the definition of the type. As C does not allow functions to be included as part of a type definition, C++ introduces this possibility for class types; this is discussed in sections 3.1 and 3.2. Functions associated with a class may be declared as "virtual." If a function is so declared, references to it will be resolved based on dynamic rather than static type resolution. The features supporting virtual functions are presented in sections 3.3 and 3.4.

3.1 Member functions

The first extension requiring a change in the algebra is the ability of classes to have functions as members. This is not allowed in C; inclusion of a function field in a C struct-type is syntactically illegal. This extension involves changes to the algebra because the way in which member functions are accessed does not parallel the way in which other members are accessed. Unlike members of other types, referred to as "data members," a member function does not occupy a memory area at some predetermined offset from the starting location of its class; thus our rule for member references as it stands is incapable of handling a reference to a member function.

The value returned by a reference to a member function must be the starting address of the function. For nonvirtual functions,[7] this address is statically determined and cannot be changed. Thus a given nonvirtual member function reference refers to a particular, unchangeable function address; in other words, a particular memory address is associated with each such reference task. We therefore define a partial function *FunctionLoc: tasks* $\rightarrow$ *addresses* which maps a member function reference to the corresponding starting address of the member function. To distinguish between member functions and data members, we add the values *data* and *nonvirtual-function* to the *tags* universe, and a function *MemberStatus: tasks* $\rightarrow$ *tags* to determine whether a given member reference is a reference to a member function or to a data member.

We change the task-type tag *struct-reference* to *class-reference*, in keeping with our new terminology. Our new rule for class references is shown in Fig. 1.

```
if TaskType (CurTask) = class-reference then
    if ValueMode (CurTask) = lvalue then
        ReportValue (OnlyValue (CurTask, StackTop) + ConstVal (CurTask))
    elseif ValueMode (CurTask) = rvalue then
        if MemberStatus (CurTask) = data then
            ReportValue (ObjectValue (OnlyValue (CurTask, StackTop) + ConstVal (CurTask)))
        elseif MemberStatus (CurTask) = nonvirtual-function then
            ReportValue (FunctionLoc (CurTask))
        endif
    endif
    Moveto (NextTask (CurTask))
endif
```

Figure 1: Transition rule for class reference tasks.

Allowing functions as members means that members may in some cases be accessed without an explicit class reference. In particular, inside the body of a member function a reference to a member of the function's class may be made simply by an expression consisting of the member's name; no class name or class-reference operator need appear. Consider our example in section 2.4: within the body of the member function `printPerson`, the class' `name` and `age` members are referred to simply as "name" and "age." This is possible because `name`, `age` and `printPerson` are members of the same class.

To handle such references within member functions, we shall treat them in the same way as explicit class references; an expression of this form will correspond to a class-reference task. As the left operand is missing from these implicit class-reference expressions, the question arises as to what the left-operand value as defined by *OnlyValue* should be. As this requires consideration of certain issues that we have not yet addressed, we shall wait until the next section before answering this question.

3.2 Implicit this parameters

Every member function has a hidden argument that is not included explicitly in either the function's list of parameter declarations or the list of argument expressions in a call to the function. This hidden argument's

[7]The situation is somewhat more complicated for "virtual" functions, which we discuss in section 3.4.

value is always the address of the class object of which the function is a member. Thus in our example in section 2.4, the function `person::printPerson` is explicitly defined as a nullary function, and no arguments are supplied when it is called; nevertheless, it is in fact a one-place function whose sole argument is a pointer to the class object.

To support this in our algebra, we assume that each member function does indeed take a class-pointer argument in addition to the arguments explicitly defined by the programmer: in particular, we assume the existence of an implicit parameter declaration in the function body. We wish to be able to distinguish this parameter as the implicit class-pointer parameter; we do this by adding the partial function *IsImplicitParm: tasks* $\rightarrow$ {*true, false*}, which determines whether a given declaration task is the declaration of an implicit parameter. We also add the partial function *ImplicitParm: stack* $\rightarrow$ *tasks*, which returns the declaration task of the implicit class pointer for a given level of the stack. When the parameter declaration task for the implicit parameter is encountered, we change the *ImplicitParm* function to return this declaration task. Our new rule for parameter declarations is shown in Fig. 2.

```
if TaskType (CurTask) = parameter-declaration then
    DoAssign (NewMemory (CurTask), ParamValue (CurTask, StackTop), ValueType (CurTask))
    OnlyValue (CurTask, StackTop) := NewMemory (CurTask)
    if IsImplicitParm (CurTask) = true then
        ImplicitParm (StackTop) := CurTask
ENDIF
```

Figure 2: Transition rule for parameter declaration tasks.

We also assume the existence of an expression task returning the address of the function's class in each call to a member function. We introduce the macro *ThisPtr*, shown in Fig. 3, to express the value of the implicit class-pointer parameter:

```
macro ThisPtr:
MemoryValue (OnlyValue (ImplicitParm (StackTop), StackTop),
                ValueType (ImplicitParm (StackTop)))
```

Figure 3: Definition of macro `ThisPtr`.

Within a member function, the value of the implicit parameter can be accessed via an expression consisting of the keyword `this`. To handle this new type of expression, we introduce a tag, *this*, and a corresponding rule shown in Fig. 4.

With the *ThisPtr* macro returning the implicit class pointer parameter value, we are now able to handle a member function's references to members of its own class. For a non-function member, the value to return for such a reference is the memory address value of the implicit parameter, offset by some value determined by *ConstVal*. For a reference to a function member, the value to return is the function's memory location, determined via *FunctionLoc* with either the static or object type of the implicit parameter. Assuming that we treat member references within a member function as class-reference tasks, we simply define the task's left-operand value to be the value *ThisPtr*. Thus an implicit class-reference such as `name` in our example in section 2.4 will be equivalent to the explicit class reference `this->name`.

```
if TaskType (CurTask) = this then
    ReportValue (ThisPtr)
    Moveto (NextTask (CurTask))
endif
```

Figure 4: Transition rule for `this` tasks.

3.3 Object type

In C++, each class object has a particular type associated with it. If the object has been allocated as the memory location for a variable of a certain class type, the object's type will simply be the predetermined "static type" of the variable. In the simple case, a class object's type also corresponds to the static type of a pointer variable pointing to it; the variable is declared as a pointer to a particular class, and the object that it points to is of that class. However, this is not necessarily the case; under certain conditions an object's type may not correspond to the static type of a variable pointing to it.

First, a pointer with a given static type, say `A*`, may be assigned the address of an object of a different static type, say `B`, by explicitly casting the address as type `A*`. In this case, the static type of `A*` would dictate that the static type of its dereferencing is `A`; however, the type of the object it points to is `B`. Second, given a hierarchy of a base class and one or more classes derived from it, and a variable declared as a pointer to an object of the base class, the pointer may be assigned to point to an object of either the base class or any of the classes derived from it, without any casting. For example, let us assume the following variable declarations, using our predefined `person`, `professor` and `student` classes:

```
// declaration of variables personObject, profObject and studentObject
person personObject;
professor profObject;
student studentObject;
```

We shall also assume the declaration of `personPtr`,[8] defined as a pointer to an object of class `person`:

```
// declaration of variable personPtr
person *personPtr;
```

The object type of `personObject`, `profObject` and `studentObject` is fixed at the time of their declaration: the object type of `personObject` is `person`; the object type of `profObject` is `professor`; the object type of `studentObject` is `student`. `personPtr`'s value can be changed to point to any of the objects defined by `personObject`, `profObject` or `studentObject`,[9] without having to cast the new values as type `person*`:

```
personPtr = &personObject;              // points to personObject; object type is person
personPtr = &profObject;               // points to profObject; object type is professor
personPtr = &studentObject;          // points to studentObject; object type is student
```

Furthermore, the object type of each object pointed to by `personPtr` will remain the same; it will not be affected by the assignment expressions.

An object's type is information stored in the object itself and determined at the time of initialization of the object. Thus while a variable can point to objects of different types, the type of an object *per se*, as a sequence of fields at a particular memory location, cannot change.

[8] In general, given a type name `T` the type name `T*` is a pointer type that points to an object of type `T`. Thus the type name `person*` in `personPtr`'s declaration denotes a pointer type that points to an object of class `person`.

[9] The unary operator `&` returns the memory location of its operand; thus in our example, we set `personPtr` to the address of `personObject`, `profObject`, and so on.

To keep track of an object's type, we simply associate a type with the object's location in memory. We define a partial function *ObjectType: addresses* $\rightarrow$ *types*, which returns the type of the object at the given memory location. When a new object is initialized, either by a variable declaration or by use of the `new` operator (discussed in section 4.1), the *ObjectType* function is changed to reflect the type of the object at the new address.

Our new rules for static and non-static variable declarations are shown in Fig. 5 and Fig. 6, respectively.

```
if TaskType (CurTask) = declaration and DecType (CurTask) = static then
    if Defined (StaticAddr (CurTask)) then
        OnlyValue (CurTask, StackTop) := StaticAddr (CurTask)
        Moveto (NextTask (CurTask))
    elseif Undefined (StaticAddr (CurTask)) then
        if Defined (Initializer (CurTask)) and Undefined (RightValue (CurTask, StackTop)) then
            Moveto (Initializer (CurTask))
        else
            OnlyValue (CurTask, StackTop) := NewMemory (CurTask)
            ObjectType (NewMemory (CurTask)) := ValueType (CurTask)
            StaticAddr (CurTask) := NewMemory (CurTask)
            if Defined (Initializer (CurTask)) then
                DoAssign (NewMemory (CurTask), RightValue (CurTask, StackTop), ValueType(CurTask))
            else
                Moveto (NextTask (CurTask))
ENDIF
```

Figure 5: Transition rule for static variable declarations.

```
if TaskType (CurTask) = declaration and DecType ≠ static then
    if Defined (Initializer (CurTask)) and Undefined (RightValue (CurTask, StackTop)) then
        Moveto (Initializer (CurTask))
    else
        OnlyValue (CurTask, StackTop) := NewMemory (CurTask)
        ObjectType (NewMemory (CurTask)) := ValueType (CurTask)
        if Defined (Initializer (CurTask)) then
            DoAssign (NewMemory (CurTask), RightValue (CurTask, StackTop), ValueType (CurTask))
        else
            Moveto (NextTask (CurTask))
ENDIF
```

Figure 6: Transition rule for automatic variable declarations.

3.4 Virtual functions

The importance of object type is manifested in its interaction with "virtual functions." A member function may be labeled "virtual" by placing the keyword **virtual** before the definition of the function. A function defined as virtual for a base class is also virtual for all classes derived from the base class, even if the function is redefined in a derived class. The virtual nature of a member function manifests itself in the case where the function is originally defined in a base class and redefined in a derived class. In such a case, an access of the member name may refer to either the base-class version or the derived-class version of the function; the difference between a virtual and a nonvirtual function is in the way in which the correct version is chosen. For a nonvirtual function, the choice is based on the static type associated with the function's class; for a virtual function, the choice is based on the type associated with the class object.

As an example, let us add member functions to the classes **person** and **professor**. In place of the functions **printPerson**, **globalPrintPerson** and **printProfessor**, we add a function **virtualPrint** and a function **nonvirtualPrint** to both classes. The **virtualPrint** functions will be tagged as virtual functions:

```
// modified declaration of class person, with virtual and nonvirtual function members
virtual class person {
private:
  char name[25];
  int age;
public:
  virtual void virtualPrint();                              // virtual print function
  void nonvirtualPrint();                                // nonvirtual print function
};
// modified declaration of class professor, with virtual and nonvirtual function members
class professor:  person {
private:
  int salary;
public:
  virtual void virtualPrint();                              // virtual print function
  void nonvirtualPrint();                                // nonvirtual print function
};
```

For each class, **nonvirtualPrint** and **virtualPrint** perform the same actions: they simply print the member values of the class:

```
// definition of nonvirtual print function for class person
void person::nonvirtualPrint() {
  output("Nonvirtual print function for person object");
  // print name and age members
  output("name:", name);
  output("age:", age);
}
// definition of virtual print function for class person
void person::virtualPrint() {
  output("Virtual print function for person object");
  // print name and age members
  output("name:", name);
  output("age:", age);
}
// definition of nonvirtual print function for class professor
void professor::nonvirtualPrint() {
  output("Nonvirtual print function for professor object");
  // call print function for class person
  person::nonvirtualPrint();
  // print salary member
  output("salary:", salary);
}
// definition of virtual print function for class professor
void professor::virtualPrint() {
  output("Virtual print function for professor object");
  // call print function for class person
  person::nonvirtualPrint();
  // print salary member
  output("salary:", salary);
}
```

Let us assume the declaration of a variable `profObject`, of type `professor`. This will initialize an object of type `professor`. A variable of type `person*` may then be assigned to point to this object:

```
// declaration of variables profObject and personPtr
professor profObject = {"Jennifer Olmsted", 32, 50000};
person *personPtr = &profObject;                      // personPtr points to profObject
```

Thus according to the static type of the variable `personPtr`, the type of the object it points to is `person`; however, the object type of `personPtr`'s dereferencing is `professor`. The member functions `personPtr->nonvirtualPrint` and `personPtr->virtualPrint` will now exhibit different behaviors.[10] Since `nonvirtualPrint` is a nonvirtual function, a function call of the form `personPtr->nonvirtualPrint()`will call the version of `nonvirtualPrint` as defined by the static type of `personPtr`'s dereferencing, namely `person`. The resulting output will be

```
Nonvirtual print function for person object
name:  Jennifer Olmsted
age:  32
```

On the other hand, `virtualPrint` is a virtual function, so a function call of the form `personPtr->virtualPrint()` will call the version of `virtualPrint` as defined by the object type of `personPtr`'s dereferencing, namely `professor`. The output will be

```
Virtual print function for professor object
name:  Jennifer Olmsted
age:  32
salary:  50000
```

Virtual functions require a change to the rule for class references to accommodate virtual functions. The rules for accessing virtual member functions will be different from that for accessing data members; while accessing a data member merely requires the address of the class object and an offset to the correct member, accessing a virtual function member involves the type of the object. To determine the correct address of a virtual function reference, we redefine *FunctionLoc* as a binary function: *tasks* $\times$ *types* $\rightarrow$ *addresses*, which determines the address for a given member function identifier and class type. In the case of a nonvirtual function, the type argument provided will be the static type associated with the member's class, as determined by the *ValueType* function; in the case of a virtual function, the argument will be the class object's type, as determined by *ObjectType*. We also introduce a new element *virtual-function* to the *tags* universe, to signify a reference to a virtual function member. Our new rule for class references is shown in Fig. 7.

[10] The class member access operator `->` returns the value of the field specified by its right-hand operand, in the class object that its left-hand operand points to. In other words, it performs a member access on the dereferencing of its left-hand operand. Thus the expressions `personPtr->nonvirtualPrint` and `(*personPtr).nonvirtualPrint` are equivalent.

```
if $TaskType\ (CurTask) = class\text{-}reference$ then
    if $ValueMode\ (CurTask) = lvalue$ then
        $ReportValue\ (OnlyValue\ (CurTask,\ StackTop) + ConstVal\ (CurTask))$
    elseif $ValueMode\ (CurTask) = rvalue$ then
        if $MemberStatus\ (CurTask) = data$ then
            $ReportValue\ (ObjectValue\ (OnlyValue\ (CurTask,\ StackTop) + ConstVal\ (CurTask)))$
        elseif $MemberStatus\ (CurTask) = nonvirtual\text{-}function$ then
            $\boldsymbol{ReportValue\ (FunctionLoc\ (CurTask,\ ValueType\ (CurTask)))}$
        **elseif** $\boldsymbol{MemberStatus\ (CurTask) = virtual\text{-}function}$ **then**
            $\boldsymbol{ReportValue\ (FunctionLoc\ (CurTask,}$
                                    $\boldsymbol{ObjectType\ (OnlyValue\ (CurTask,\ StackTop))))}$
        endif
    endif
    $Moveto\ (NextTask\ (CurTask))$
endif
```

Figure 7: Transition rule for class reference tasks.

4 Object creation and destruction

C++ introduces convenient mechanisms for creating and destroying objects. The operator `new` allocates memory for a new object, while the operator `delete` deallocates memory already associated with an object. These operators are covered in sections 4.1 and 4.2 respectively. In addition, the programmer may define functions to be invoked implicitly when an object is created or destroyed. Discussion of these "constructor" and "destructor" functions appears in sections 4.3 and 4.4 respectively.

4.1 The `new` operator

C++ introduces an operator `new` for dynamic object creation. This operator takes a type name as an operand; it allocates a region of memory whose size corresponds to that of the indicated type and then returns the memory location of this newly allocated memory. In other words, it creates an object of a given type and returns a pointer value to it. For example, the expression `new person` allocates enough space for the `name` and `age` members of a `person` object and returns a pointer value to this newly allocated space. Memory allocation is accomplished by a call to the global function `operator new`; if the object being allocated contains a member function `operator new`, the member function is called instead.

The `new` operator introduces a new task type tag, *new-object*, to the universe of tags. As with variable declarations, an initializing expression, if one exists, is evaluated; the evaluation task for this initializer is determined by the function *Initializer*. We introduce a partial function *Allocator: tasks* $\rightarrow$ *tasks*, which maps each expression involving the `new` operator to a task which calls the appropriate version of `operator new`. This function call changes the *OnlyValue* function, setting the `new` expression's value to a new memory location. In addition, we modify the *ObjectType* function to indicate that a new object of a particular type has been initialized at the new memory location. Once a memory location has been established for the new object, it is returned as the value of the `new` expression; if an initializer is provided, the object is assigned its value. Our new rule is shown in Fig. 8.

```
if $TaskType\ (CurTask) = new\text{-}object$ then
    if $Defined\ (Initializer\ (CurTask))$ and $Undefined\ (RightValue\ (CurTask,\ StackTop))$ then
        $Moveto\ (Initializer\ (CurTask))$
    elseif $Undefined\ (OnlyValue\ (CurTask,\ StackTop))$ then
        $Moveto\ (Allocator\ (CurTask))$
    else
        $ObjectType\ (OnlyValue\ (CurTask,\ StackTop)) := PointsToType\ (CurTask)$
        $ReportValue\ (OnlyValue\ (CurTask,\ StackTop))$
        if $Defined\ (Initializer\ (CurTask))$ then
            $DoAssign\ (OnlyValue\ (CurTask),\ RightValue\ (CurTask,\ StackTop),$
                        $PointsToType\ (CurTask))$
        else
            $Moveto\ (NextTask\ (CurTask))$
ENDIF
```

Figure 8: Transition rule for `new`-operator tasks.

4.2 The `delete` operator

The `delete` operator reverses the effects of the `new` operator: given the address of an object as an operand, it deallocates the memory allocated for the object, allowing subsequent memory allocations to use the object's space. For example, given the declaration

```
// declaration of pointer variable personPtr
person *personPtr;
personPtr = new person;
```

the expression `delete person` deallocates the memory allocated by the `new` operator; the pointer `personPtr` no longer points to an object. Memory deallocation is accomplished by calling the global function `operator delete`; if the deallocated object contains a member function of this name, its member function will be called. An expression with the `delete` operator returns a value of type `void`.

We add a new task type tag *delete-object* to the universe of tags. We add a partial function *Destructor: tasks* $\rightarrow$ *tasks*, which maps each expression involving the `delete` operator to a task which calls the appropriate version of `operator delete`. The rule for the `delete` operator, shown in Fig. 9, returns the operator expression's `void` value, sets the object type of the operand's memory location to an undefined value, and passes control to the task invoking the function `operator delete`.

```
if $TaskType\ (CurTask) = delete\text{-}object$ then
    $ReportValue\ (Void)$
    $ObjectType\ (RightValue\ (CurTask,\ StackTop)) := undef$
    $Moveto\ (Destructor\ (CurTask))$
endif
```

Figure 9: Transition rule for `delete`-operator tasks.

4.3 Constructors

When defining a class, the programmer may define special member functions to be invoked when an object of the class is created; these functions are called "constructor functions." Constructors are commonly used to initialize newly created objects with default values. It is important to note that a constructor function does not actually create a new object, in the sense of allocating new memory to be used as a class object. The name of a constructor function member within a class is simply the name of the class itself; like other function names, it may be overloaded. To illustrate, we add two constructor functions to our class `person`:

```
// modified declaration of class person with constructor functions
virtual class person {
private:
  char name[25];
  int age;
public:
  person();                               // "default" constructor:  requires no argument
  person(const char*, int);                         // constructor taking string and int
  virtual void virtualPrint();
  void nonvirtualPrint();
};
```

The first constructor function takes no arguments; an invocation of this function will simply fill in the `name` and `age` members with default values:[11]

```
// definition of default constructor for class person
person::person() {
  strcpy(name, "");                                    // set name member to null string
  age = -1;                                          // set age member to invalid value
}
```

The second constructor function takes two arguments and fills in the `name` and `age` members with these argument values:

```
// definition of binary constructor function for class person
person::person(const char *n, int a) {
  strcpy(name, n);                        // copies contents of string n to name member
  age = a;
}
```

A constructor may be invoked when an automatic or static variable is declared; in the case of a static variable, it is invoked only the first time the declaration is encountered. For example, the declarations

```
// declaration of person objects p1 and p2
person p1();
person p2("Chuck Wallace", 26);
```

initialize variable `p1` with the first constructor function and variable `p2` with the second constructor function. When a constructor function with no arguments, a so-called "default constructor," is defined, it may be invoked without the use of argument parentheses; thus the declaration of `p1` above is equivalent to

```
// alternate, equivalent declaration of p1
person p1;
```

[11] We assume the definition of the function `strcpy` from the `<string.h>` library. The `strcpy` function takes two string pointers as arguments and copies the first argument's string to that of the second argument.

Alternatively, the declaration of a variable may initialize the variable's new object in the standard C fashion, via direct assignment; in this case, the constructor function is not called. Thus the declaration:

```
// alternate declaration of variables p1 and p2 using direct assignment for initialization
person p1 = {"Susanna Peters", 31};
person p2 = p1;
```

initializes the variables `p1` and `p2` without calling either constructor function; the values of `p1`'s members are assigned explicitly, while those of `p2`'s members are copied from `p1`'s members.

A constructor function may also be called when a new object is allocated using the `new` operator. For example, the expression `new person("Matt Parker", 27)` allocates memory for a new object of class `person` and initializes it by calling the binary constructor function for `person`. Finally, a constructor may be called explicitly in the standard form for functions. Given the declaration of `p1` above, the subsequent expression `p1.person()` calls the appropriate constructor function, reinitializing `p1`'s members.

Programmer-defined constructor functions require changes to our rules for tasks which create new objects: namely, variable declarations and expressions involving the operator `new`. We add a partial function *Constructor: tasks* $\rightarrow$ *tasks* which maps a class-variable declaration task to a task calling the class' constructor function. After memory is allocated for the variable, the constructor function is called to initialize the new object. The modified rules are shown in Fig. 10 and Fig. 11.

```
if TaskType (CurTask) = declaration and DecType (CurTask) = static then
    if Defined (StaticAddr (CurTask)) then
        OnlyValue (CurTask, StackTop) := StaticAddr (CurTask)
        Moveto (NextTask (CurTask))
    elseif Undefined (StaticAddr (CurTask)) then
        if Defined (Initializer (CurTask)) and Undefined (RightValue (CurTask, StackTop)) then
            Moveto (Initializer (CurTask))
        else
            OnlyValue (CurTask, StackTop) := NewMemory (CurTask)
            ObjectType (NewMemory (CurTask)) := ValueType (CurTask)
            StaticAddr (CurTask) := NewMemory (CurTask)
            if Defined (Initializer (CurTask)) then
                DoAssign (NewMemory (CurTask), RightValue (CurTask, StackTop), ValueType (CurTask))
            elseif Defined (Constructor (CurTask)) then
                Moveto (Constructor (CurTask))
            else
                Moveto (NextTask (CurTask))
ENDIF
```

Figure 10: Transition rule for static variable declaration tasks.

We make a similar change to our rule for the operator `new`. The *Constructor* function maps an instance of the operator to a constructor-function call; this function-call task is performed after allocation of memory for a new object. The modified rule is shown in Fig. 12.

4.4 Destructors

Just as constructor functions can be defined to handle initialization of new class objects, special member functions may also be defined to perform certain actions when a class object is destroyed. These functions,

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

Figure 11: Transition rule for automatic variable declaration tasks.

```
if $TaskType\ (CurTask) = new\text{-}object$ then
    if $Defined\ (Initializer\ (CurTask))$ and $Undefined\ (RightValue\ (CurTask,\ StackTop))$ then
        $Moveto\ (Initializer\ (CurTask))$
    elseif $Undefined\ (OnlyValue\ (CurTask,\ StackTop))$ then
        $Moveto\ (Allocator\ (CurTask))$
    else
        $ObjectType\ (OnlyValue\ (CurTask,\ StackTop)) := PointsToType\ (CurTask)$
        $ReportValue\ (OnlyValue\ (CurTask,\ StackTop))$
        if $Defined\ (Initializer\ (CurTask))$ then
            $DoAssign\ (NewMemory\ (CurTask),\ RightValue\ (CurTask,\ StackTop),\ PointsToType\ (CurTask))$
        elseif $Defined\ (Constructor\ (CurTask))$ then
            $Moveto\ (Constructor\ (CurTask))$
        else
            $Moveto\ (NextTask\ (CurTask))$
ENDIF
```

Figure 12: Transition rule for `new`-operator tasks.

called “destructor functions,” are invoked implicitly by a variable going out of scope or by use of the `delete` operator, or explicitly by a simple function call to the destructor. As an example, let us add a destructor function to the class `person`:

```
// modified declaration of class person, with destructor function
virtual class person {
private:
  char name[25];
  int age;
public:
  person();
  person(const char*, int);
  ~person();                                          // destructor function
  virtual void virtualPrint();
  void nonvirtualPrint();
};
```

We also add a global variable `totalPeople` to keep track of the number of `person` objects currently in existence:

```
// declaration of global counter variable totalPeople
int totalPeople = 0;
```

This global variable can be incremented and decremented in the class' constructor and destructor functions; then once the class is defined, the programmer need not perform any explicit incrementing or decrementing outside the class. We modify our constructor functions, adding a statement incrementing `totalPeople`:

```
// modified definition of default constructor for class person,
// including increment of global object counter
person::person() {
  strcpy(name, "");
  age = -1;
  totalPeople++;                                      // counter incremented
}
// modified definition of binary constructor function
// for class person, including increment of global object counter
person::person(const char *n, int a) {
  strcpy(name, n);
  age = a;
  totalPeople++;                                      // counter incremented
}
```

Now each time a new object of class `person` is created, the counter `totalPeople` is incremented. The opposite action is performed by the destructor function: when an object is destroyed, the counter is decremented:

```
// definition of destructor function for class person
person::~person() {
  totalPeople--;                                      // counter decremented
}
```

As mentioned above, the destructor function is called implicitly when a variable goes out of scope. For a local automatic variable, this is the point at which the function in which it is declared ends. For a static or global automatic variable, it is the end of the program. The destructor is also called implicitly when the `delete` operator is used to deallocate a class object. Finally, the programmer may call the destructor member function explicitly: the function name is simply the class name preceded by a tilde (`~`). Thus the expression `p1.~person()` will call the destructor function and decrement the global counter.

Explicit calls to destructor functions are handled by the existing rule for function invocations. However, as not all destructor-function calls are explicit in the program code, we must make them so in the representation of the program. We simply add a destructor function call task at each point where a class variable with a destructor function defined goes out of scope: this will be either at the end of a function or the end of the program, depending on the variable type. We shall refer to the sequence of implicit destructor function calls followed by a `return` task at the end of a function or program as a "return sequence."

The `delete` operator may also invoke a destructor function. To handle this, we alter our definition of the *Destructor* function: in `delete`-expression tasks involving an object of a class with a destructor function defined, the *Destructor* function maps to a task that calls the destructor function. The task following this function call, as defined in *NextTask*, is a task calling the appropriate `operator delete` function.

5 Overloading and parameterized types

C++ allows function names and operators to be "overloaded." Overloading is a loosening of the restrictions on associating names with functions. While in C a particular function name may refer to at most one function, in C++ a name may refer to a family of functions. The particular function referred to by an instance of a name is determined by the types of the arguments and return type associated with the name instance. Overloading allows the programmer to refer to conceptually similar functions with the same name, thereby easing comprehension. Overloading is discussed in sections 5.1 and 5.2.

In a similar vein, the template mechanism allows the programmer to define a family of conceptually similar types through a parameterized type definition. An instantiation of a type from the family is attained by supplying values for the parameters. As with overloading of functions and operators, this allows the programmer to refer to similarly defined types with a single name. Template definitions are discussed in section 5.3.

5.1 Function overloading and default arguments

In C++, function names may be "overloaded": a function name may refer to more than one function declaration within the same scope. When an overloaded name is used in an expression, it refers to a particular function; the function it refers to is determined by matching the actual arguments of the function reference with the formal arguments of a function declaration.[12] As an example, let us add an overloaded function name, `monthlySalary`, as a friend to the `professor` class defined in section 2.2. Within the class definition, we declare two functions, both named `monthlySalary`: the first `monthlySalary` function takes a single `int` argument, while the second `monthlySalary` function takes two `int` arguments. The first `monthlySalary` function calculates a monthly salary for the `professor` object by dividing its `yearlySalary` argument by `12`:

```
// definition of unary monthlySalary function
  int monthlySalary(int yearlySalary) {
    return yearlySalary / 12;
}
```

The second monthlySalary function divides its integer yearlySalary argument by its integer months argument:

```
// definition of binary monthlySalary function
  int monthlySalary(int yearlySalary, months) {
```

[12]There need not be an exact match between actual and formal arguments. [ES] lays out a set of rules to determine the best match when no exact match exists. For the sake of simplicity, we shall only consider examples where formal and actual arguments match exactly.

```
    return yearlySalary / months;
}
```

A subsequent function call using the name `monthlySalary` is disambiguated by considering the arguments supplied in the function call. An expression `monthlySalary(40000)` is a call to the first function declaration, as its arguments match the formal arguments of the first declaration exactly. An expression `monthlySalary(30000, 9)` is likewise a call to the function defined in the second declaration.

Function overloading does not require any changes to the algebra because the mapping between function references and function declarations is static. When an overloaded function name is used, the function it refers to is determined by the types of its arguments; since these types are statically determined, the referent of the overloaded name is as well. For any expression task T consisting of an overloaded function name, we simply determine the best match for the function reference and assign *Decl(T)* the declaration task of the best-match function.

Another C++ addition, related to function overloading, is the ability to supply default values for the formal arguments of a function. A function with a default value specified for one of its arguments may be called either with or without a value for that argument; if no actual argument is supplied, the default value is used. For instance, rather than defining separate unary and binary `monthlySalary` functions, we can define the function once as a binary function and give the `months` argument a default value of `12`:

```
// modified definition of binary monthlySalary function,
// with default value for months member
  int monthlySalary(int yearlySalary, months = 12) {
    return yearlySalary / months;
}
```

The result is identical to that of defining unary and binary `monthlySalary` functions. The function may be called with two arguments, in which case the formal argument `months` receives the value of the second argument; it may also be called with one argument, in which case `months` receives the default value `12`.

There is no standard method for implementing default argument values; however, none of the different possible approaches require changes to the algebra. Functions with default argument values can be thought of as special cases of function overloading. The definition of a function with formal arguments $a_1..a_n$ and a_n assigned a default value is then essentially a definition of two functions: one with formal arguments $a_1..a_n$, and another with formal arguments $a_1..a_{n-1}$ and a local variable a_n set to the default value. Thus the above definition of `monthlySalary` would be equivalent to the following definitions:

```
// binary monthlySalary function
int monthlySalary(int yearlySalary, months) {
    return yearlySalary / months;
}
// unary monthlySalary function
int monthlySalary(int yearlySalary) {
  int months = 12;
  return yearlySalary / months;
}
```

An alternate approach to implementing default arguments is to define a single function and modify calls to the function, supplying default values as actual arguments if need be. For instance, our definition of `monthlySalary` above would instantiate a single binary function, and a unary function call like `monthlySalary(40000)` would be changed to `monthlySalary(40000, 12)`.

The overloaded-function approach requires no changes to the algebra, as we have seen that function overloading is a purely syntactic feature. The single-function approach does not even require function overloading; it simply involves calls to a non-overloaded function. Thus our algebra as it stands is able to handle default argument values, regardless of their implementation.

5.2 Operator overloading

Operators may also be overloaded: in particular, they may be extended to have special meanings when applied to class objects. The user may define an "operator function" for a particular operator, taking at least one class object as an argument. When an operator is used with no class objects as operands, the result is the standard action for the operator as defined in C; when used with a class object as one of its operands, the result is a call to the operator function defined for that class. Operator functions taking different argument types may be defined for the same operator. As with overloaded functions, the operator function for a given occurrence of an operator is determined by matching the actual operands with the formal arguments of the operator functions.

As an example, we shall overload the relational operator `>` to accommodate our class `student`. Within the class definition, we declare two friend functions, both denoted by `operator>`:

```
// modified declaration of class student,
// allowing operator functions to access private members
class student:  private person {
private:
  int year;
  float GPA;
public:
  void printStudent();
  friend int operator>(student, student);                        // operator >
  friend int operator>(student, int);                            // operator >
};
```

Both operator functions take a `student` class object as a left operand; the first declaration defines a function taking a `student` object as a right operand, while the second defines a function taking an `int` object as a right operand. We define the first version of `operator>` so as to return a "true" value if the `year` member of the left operand is greater than that of the right operand:[13]

```
// definition of operator function > for (student, student) operands
int operator>(student s1, s2) {
    return s1.year > s2.year;
}
```

We define the second version of the operator function so as to return a "true" value if the `year` member of the left operand is greater than the second operand:

```
// definition of operator function > for (student, int) operands
int operator>(student s, int p) {
    return s.year > p;
}
```

When the operator `>` is used with a `student` object as its left operand, the appropriate version of the operator function is chosen based on the type of the right operand. Thus given the declaration of `student` objects `s1` and `s2`:

```
// declaration of variables s1 and s2
student s1, s2;
```

[13]It should be noted that this definition is somewhat problematic. The values of the `year` members is not the only possible basis for a "greater-than" ordering of `student` objects; they could just as easily be ordered by the values of their `age` members, for instance. It may not be clear to a programmer using the `operator>` function what the basis for the ordering is. This is a common problem with defining operators for classes; one way of avoiding this confusion would be to define a function with a meaningful name, like `atSchoolLonger`, rather than an overloaded operator.

an expression of the form `s1 > s2` will result in a call to the first function, since the type of the actual argument `s2` matches that of the first function's formal argument. An expression of the form `s1 > 5` will result in a call to the second member function, for similar reasons.

Any programmer-defined operator function can also be invoked by an explicit function call. In our example, we declared two operator functions with the name `operator>`; a function call using this name is equivalent to using the operator `>`. Thus the function calls `operator>(s1, s2)` and `operator>(s1, 5)` are equivalent to the two operator expressions above.

Like function overloading, operator overloading does not necessitate any changes to the algebra rules as it requires only statically determined information. The use of a given operator requires one or more operand expressions of a given type: this static type information is all that is needed to determine the correct meaning of the operator. Using a programmer-defined operator function corresponds to a function- invocation task, with the function to invoke determined statically by the types of the actual arguments. For each task T involving an overloaded operator function name, we determine the best match for the function reference and assign to *Decl(T)* the result of this best match, as with overloaded non- operator functions.

5.3 Templates

The template mechanism in C++ allows the programmer to define "container classes," classes containing members whose types are specified outside the class definition. A container class defines a family of classes differing in the types of some of their members but sharing common structure. "Abstract data types" such as stacks and queues can be represented as container classes. For example, the notion of a list defines the way in which list items, or nodes, are linked to one another and methods of manipulating the items but leaves undefined the type of information stored in a node. A family of list types can be defined simply by specifying different values for this type information. A template separates the structure common to all members of the family from the type information specifying a particular member of the family. A list of type arguments is supplied first, followed by a declaration; the specific type information is supplied as arguments, while the common structure is defined in the declaration.

A common example of a container class is the "singly-linked list." This abstract data type consists of two data items and a set of manipulation functions. The data items are the information contained in a node of the list and a pointer to the next node in the list, and typical manipulation functions include a print function and a node-addition function. The term "singly-linked list" denotes a family of data types all sharing the above characteristics but differing in the type of information stored in each node. We define the common characteristics within the template's declaration:

```
// definition of template listNode for singly-linked list node
template <class T>
class listNode {
private:
  T data;                                                  // data contained in node
  listNode *next;                        // ptr to node following this node in list
public:
  void print();                                 // print data for all nodes in list
  void addNode();                                                 // add node to list
};
```

The only information not specified in the declaration, the type of `T`, is supplied as an argument whenever the template is used. For example, to create a singly-linked list of `person` objects, we declare a variable using the `listNode` template:

```
listNode<person> *personList;
```

Nodes in this list have `data` members of type `person`. A linked list of `int` objects can be created in a similar way:

```
listNode<int> *intList;
```

Nodes in this list have data members of type int.

A template's declaration need not be a type declaration; a family of functions can be defined by a function declaration within a template. For instance, we can create a function template `max` which, given two objects of the same type as arguments, returns the greater of the two. The function declaration within the template specifies everything except the return type of the function and the type of its arguments:

```
// definition of function template max
template<class T>
T max (T a, b) {
  return a > b ?  a :  b;
}
```

The type information is supplied in a particular invocation of the template; for instance, the function call `max<int>(1, 2)` will compare the two `int` objects and return the `int` value `2`. The function call `max<student>(s1, s2)` will compare two `student` objects, using our definition of `>` in section 5.2, and return a `student` value.

The template feature is another language facility that affects only the static information associated with a program. As stated earlier, a template defines a family of types; in terms of static type information, defining a template is equivalent to defining each type in its associated family separately. Thus we may treat types like `listNode<int>` and `listNode<person>` as entirely distinct types and functions like `max<int>` and `max<student>` as distinct functions; the fact that they are generated by the same template has no effect on the operation of the program. As templates affect only the way in which a program's static information is determined, we do not need to alter our algebra to accommodate them.

6 Other extensions

Apart from the extensions we have considered so far, C++ introduces several language features which do not fit well into any category. The extensions discussed here round out the set of C++ extensions.

6.1 Constant objects

When an object is created, it may be specified as "constant." An object so specified may be given an initial value, but this value may not be subsequently modified. Constant status is assigned by prefixing the keyword `const` to the object's type. For example, once a constant `person` object has been created from the declaration of variable `p1`,

```
const person p1 = {"Charles Wallace", 26};
```

an expression that simply accesses a member of the object, such as `p1.age`, is valid, but an expression that would modify the value of a member, such as `p1.age++`, results in an error at compile time.[14]

Apart from special considerations during compile time, constant objects are treated no differently from non-constant objects. Expressions and statements that would alter the value of a constant object are simply rejected during compilation; once a program is compiled, constant and non-constant objects are equivalent. Thus our algebra need not be changed to accommodate constant objects.

[14] The postfix operator `++` takes a variable reference as its sole operand, returns the variable's value, and increments that value by 1. It is this last step that violates the `const` constraint on `p1`.

6.2 Inline functions

A function may be declared as "inline" by prefixing the keyword `inline` to its declaration; this indicates to the compiler that it should try to handle calls to this function without using the standard function-call mechanism. Rather than creating a single memory location for the function and subsequently passing control to this location each time it is called, the compiler will try to replace each call to the function with the sequence of code contained within the function. The function name then acts much like a macro. For example, let us declare our `monthlySalary` function as inline:

```
// modified definition of function monthlySalary as inline
inline int monthlySalary(int yearlySalary, months = 12) {
  return yearlySalary / months;
}
```

If the compiler accepts the request to treat `monthlySalary` as an inline function, it will transform an expression like `monthlySalary(30000, 9)` into an expression not involving a function call. Replacing the formal arguments of the inline function with actual arguments results in the expression `30000 / 9`. An optimizing compiler will simplify this further to `3333`. The inline option affects only the static structure of a program's code; it may modify expressions within the code, but it has no effect on the code once it is compiled. Our algebra applies only to the compiled version of a program; thus changes to the code made by inlining are assumed. As the inline status of a function has no further effect on the program, we need not change our algebra to accommodate inlining.

6.3 References

C++ introduces the "reference" as a means of attaching a name to an object. A reference is declared in a way similar to the declaration of a variable: a declaration contains a name for the reference and a type specification followed by the symbol `&`. A reference declaration must also contain an initializing expression determining the object that the reference refers to. For instance, given the declaration of `personObject` in section 3.3, we may subsequently declare a reference `personReference`:

```
// declaration of reference personReference
person& personReference = personObject;
```

This creates a reference which returns the value of the object referred to by `personObject` each time it is used. Note that the declaration does not create a new object of type `person`; `personReference` simply refers to the existing object `personObject`. Thus a modification to `personReference`'s object is a modification to `personObject`'s object; after the assignment

```
personReference.age = 22;
```

the expressions `personObject.age` and `personReference.age` will both return the value `22`. References can be used within functions to implement "call by reference," in which the value of an actual parameter may change as the result of a function call. For example, we can create a function `birthday`, a friend to the class `person`, which increments the `age` member of its `person` argument:

```
// modified declaration of class person, granting access to global function birthday
virtual class person {
  private:
    char name[25];
    int age;
  public:
    person();
    person(const char*, int);
    ~person();
    virtual void virtualPrint();
    void nonvirtualPrint();
    friend void birthday(person);                      // new global function birthday
  };
```

We declare the single parameter of `birthday` as a reference:

```
// definition of function birthday
void birthday(person& p) {
  p.age++;
}
```

Given the current state of the variable `personObject`, with `age` value `22`, a function call `birthday(personObject)` assigns the object denoted by `personObject` to the parameter `p`. This object is modified within the function; the `age` member is incremented to `23`. After the function call, the expression `personObject.age`will return `23`.

Our existing rules for declarations can accommodate references, with one additional stipulation. In a reference declaration, the reference is assigned the address, *i.e.*, the "lvalue," of an object specified in the required initializer expression. We therefore stipulate that the initializer-expression task associated with a reference via the function *Initializer* returns the lvalue of its expression. In addition, we must alter the rules for identifiers and class references. The use of a reference as an identifier or class-reference expression should return the lvalue or rvalue of the reference's object. This is determined indirectly by the address stored when the reference is declared. Thus an lvalue access should return the address stored in the reference, while an rvalue access should return the value of the object stored at this address. We add a partial function *IsReference: tasks* $\rightarrow$ {*true, false*} to determine whether a given expression task is a reference expression. If the value of this function is *true*, we simply follow an extra level of indirection in accessing the identifier's value. The modified version of the identifier and class-reference rules are shown in Fig. 13 and Fig. 14, respectively.

6.4 Exception handling

C++ adds exception handling as a means of recovering from run-time errors. A set of "exception catchers" may be associated with a block of code, a "try block." These catchers are themselves blocks of code, intended to be used as a means of recovering smoothly from run-time errors occurring within the try block. A catcher is invoked by "throwing an exception"; this results in control being passed to an exception catcher associated with an enclosing try block. An "exception" is an object, and different exception catchers are associated with different object types; thus the catcher invoked for a given exception is determined by matching the exception object's type and the type associated with a catcher. If an exception is thrown within a function and no catcher is defined for the exception within the function, the function invocation is popped off the stack, destructor functions are called for any objects local to the function, control returns to the next function invocation on the stack, and a catcher is searched for there.

To illustrate, we add exception handling to the member functions of our `person` class. As this class contains a string member, `name`, it harbors a potential run-time error common to all types containing strings: namely, the possibility of string overflow. Consider adding a member function `inputPerson` which accepts name and age values from the user and then calls the constructor function with these values. A user could enter a string longer than 25 characters, exceeding the bounds of the name member. In this case, we would like to issue a warning to the user and truncate the string to the 25- character limit. We first add a new member `nameTooLong` to serve as an "exception class"; this is the type of object to be thrown when a string overflow exception is encountered:

```
// modified declaration of class person, with exception class nameTooLong
virtual class person {
```

```
if $TaskType\ (CurTask) = identifier$ then
    if $ValueMode\ (CurTask) = lvalue$ then
        if $GlobalVar\ (CurTask) = true$ then
            if $IsReference\ (CurTask) = true$ then
                $ReportValue\ (ObjectValue\ (GlobalVarLoc))$
            else
                $ReportValue\ (GlobalVarLoc)$
            endif
        elseif $GlobalVar\ (CurTask) = false$ then
            if $IsReference\ (CurTask) = true$ then
                $ReportValue\ (ObjectValue\ (LocalVarLoc))$
            else
                $ReportValue\ (LocalVarLoc)$
            endif
        endif
    elseif $ValueMode\ (CurTask) = rvalue$ then
        if $GlobalVar\ (CurTask) = true$ then
            if $IsReference\ (CurTask) = true$ then
                $ReportValue\ (Deref\ (ObjectValue\ (GlobalVarLoc)))$
            else
                $ReportValue\ (ObjectValue\ (GlobalVarLoc))$
            endif
        elseif $GlobalVar\ (CurTask) = false$ then
            if $IsReference\ (CurTask) = true$ then
                $ReportValue\ (Deref\ (ObjectValue\ (LocalVarLoc)))$
            else
                $ReportValue\ (ObjectValue\ (LocalVarLoc))$
            endif
        endif
    endif
    $Moveto\ (NextTask\ (CurTask))$
endif
```

Figure 13: Transition rule for identifier tasks.

```
if $TaskType\ (CurTask) = class\text{-}reference$ then
    if $ValueMode\ (CurTask) = lvalue$ then
        **if $\boldsymbol{IsReference\ (CurTask) = true}$ then**
            $\boldsymbol{ReportValue\ (ObjectValue\ (OnlyValue\ (CurTask,\ StackTop) + ConstVal\ (CurTask)))}$
        **else**
            $ReportValue\ (OnlyValue\ (CurTask,\ StackTop) + ConstVal\ (CurTask))$
        **endif**
    elseif $ValueMode\ (CurTask) = rvalue$ then
        if $MemberStatus\ (CurTask) = data$ then
            **if $\boldsymbol{IsReference\ (CurTask) = true}$ then**
                $\boldsymbol{ReportValue\ (Deref\ (ObjectValue\ (OnlyValue\ (CurTask,\ StackTop)}$
                                                    $\boldsymbol{+ ConstVal\ (CurTask))))}$
            **else**
                $ReportValue\ (ObjectValue\ (OnlyValue\ (CurTask,\ StackTop) + ConstVal\ (CurTask)))$
            **endif**
        elseif $MemberStatus\ (CurTask) = nonvirtual\text{-}function$ then
            $ReportValue\ (FunctionLoc\ (CurTask,\ ValueType\ (CurTask)))$
        elseif $MemberStatus\ (CurTask) = virtual\text{-}function$ then
            $ReportValue\ (FunctionLoc\ (CurTask,\ ObjectType\ (OnlyValue\ (CurTask,\ StackTop))))$
        endif
    endif
    $Moveto\ (NextTask\ (CurTask))$
endif
```

Figure 14: Transition rule for class reference tasks.

```
private:
  char name[25];
  int age;
public:
  class nameTooLong {                              // exception class for string overflow
  };
  person();
  person(const char*, int);
  ~person();
  void inputPerson();                                 // new member function inputPerson
  virtual void virtualPrint();
  void nonvirtualPrint();
  friend void birthday(person);
};
```

Next, we modify the `person` constructor function so as to throw a `nameTooLong` exception if it encounters a string of length greater than 25:

```
// modified definition of binary constructor function
// for class person, with exception nameTooLong
person::person(const char *n, int a) {
  if (strlen(n) > 25) throw nameTooLong;
  strcpy(name, n);
  age = a;
}
```

Finally, we add the `inputPerson` function; we enclose the call to the constructor function within a try block and add a catcher for a `nameTooLong` exception:[15]

```
// definition of function inputPerson, with catcher for exception nameTooLong
person::inputPerson() {
  char n[80];
  int a;
  try {
    input(n);
    input(a);
    person(n, a);
  }
  catch(nameTooLong) {
    output("Warning:  truncating name to 25 characters");
    n[24] =
    0;                                     // set end-of-string marker after character 25
    person(n, a);                         // call constructor again with truncated string
  }
}
```

Exception handling is now in place for the `inputPerson` function. If the function is invoked and the user enters an overly long string, the `nameTooLong` exception will be thrown when the constructor function is invoked. At this point, memory is allocated for a temporary `nameTooLong` exception object. As the constructor function has no `nameTooLong` exception catcher, the function terminates and control returns to the `inputPerson` function. This function does contain a `nameTooLong` catcher, so control passes directly to

[15]We assume that the function `input` reads input from a device and sets the value of its argument to this input value. As with the function `output`, we do not define the function explicitly.

the catcher; the warning is displayed, the name member truncated, and the constructor function invoked again.

In extending the algebra to include exception handling, we add a new task type to handle `throw` statements and a corresponding tag name *throw*. In the rule for such statements, shown in Fig. 15, we check to see whether memory has been allocated for the exception object; if no space has been allocated, we move to a task calling the `operator new` function. Once memory has been allocated, we assign to it the exception object's value; in addition, we set the value of two new nullary functions. *Unwinding:* $\{$*true, false*$\}$, which determines whether the stack is being unwound as the result of an exception, is set to *true*; *Exception: tasks*, which returns the `throw`-statement task that has been executed, is set to the current task:

```
if TaskType (CurTask) = throw then
   if Undefined (OnlyValue (CurTask, StackRoot)) then
      Moveto (Allocator (CurTask))
   else
      Unwinding := true
      Exception := CurTask
      DoAssign (OnlyValue (CurTask, StackRoot), RightValue (CurTask, StackTop),
                ValueType (CurTask))
ENDIF
```

Figure 15: Transition rule for throw tasks.

With the introduction of exception handling, a program can be in one of two states: an exception may have been thrown and not yet handled, in which case the stack must be unwound and control passed to the nearest catcher, or it may be that no unhandled exception has been thrown, in which case control passes from one task to another as already defined. We add a new rule to be executed when a program is in the former state, *i.e.*, when the value of *Unwinding* is *true*. We also add two new functions. *Catcher: tasks* $\times$ *types* $\rightarrow$ *tasks* maps each task to the catcher associated with it for the given exception-object type. If no catcher with a given exception-object type is defined for a given task, the value of *Catcher* is *undef* for that task and type. *Return: tasks* $\rightarrow$ *types* maps each task to the first task of the return sequence, *i.e.*, the sequence of destructor function calls followed by a `return` statement at the end of the task's function.[16] Our rule for the "unwinding" state, shown in Fig. 16, will pass control to an exception catcher if one of the appropriate type is defined for the current task and will pass control to the return sequence at the end of the function if no catcher is defined:

This is the only rule that should be executed when in the unwinding state; thus we must place an extra constraint on all our other rules so that they are not executed. We make the following changes: for each rule of the form "if *G* then*[rule body]*" where *G* is a truth-functional guard condition, we change the rule to: "**if** ***Unwinding = false*** **and** *G* then*[rule body]*."

Finally, we make an assumption about the tasks within an exception catcher. As the exception object is eliminated when the catcher terminates, the destructor function for this object must be called. Thus at the end of the catcher we add a function-invocation task which calls the destructor of the exception object.

7 Conclusion

Our algebra as it stands represents all the features of C++ as described in [ES]. Unfortunately, we cannot claim that our specification constitutes a standard version of C++, as no standard has been established for

[16] See section 4.4 for a further discussion of "return sequences."

```
if Unwinding = true then
    if Defined (Catcher (CurTask, ValueType (Exception))) then
        Unwinding := false
        Moveto (Catcher (CurTask, ValueType (Exception)))
    else
        Moveto (Return (CurTask))
ENDIF
```

Figure 16: Transition rule for unwinding state.

the language. [ES] has been chosen as a "starting point" for an ANSI standard; thus it seems likely that our specification will closely approximate any eventual standard.

A Macro definitions

We assume the macro definitions shown in Fig. 17. *Defined* and *Undefined* are used to test whether a given value is defined or undefined, *i.e.*, whether its value is *undef*. The macros *GlobalVarLoc* and *LocalVarLoc* are used to determine the memory locations of global and local variables. *ObjectValue* returns the value of an object, given the object's memory location. Finally, *Deref* takes a pointer value and returns the value of the object pointed to by the pointer.

```
macro Defined(Value):
Value ≠ undef

macro Undefined(Value):
Value = undef

macro GlobalVarLoc:
OnlyValue (Decl (CurTask), StackRoot)

macro LocalVarLoc:
OnlyValue (Decl (CurTask), StackTop)

macro ObjectValue(MemoryLoc):
MemoryValue (MemoryLoc, ValueType (CurTask))

macro Deref(Value):
MemoryValue (Value, PointsToType (CurTask))
```

Figure 17: Initial macro definitions.

References

[ES] Ellis, Margaret A. and Bjarne Stroustrup. *The Annotated C++ Reference Manual*. Addison-Wesley, 1990.

[Gu] Gurevich, Yuri, "Evolving Algebras: An Attempt to Discover Semantics," in *Current Trends in Theoretical Computer Science* (ed. G. Rozenberg and A. Salomaa), World Scientific, 1992, 266-292.

[GH] Gurevich, Yuri and James Huggins, "The Semantics of the C Programming Language," in *Lecture Notes in Computer Science*, v. 702 (ed. E. Börger *et al.*), Springer-Verlag, 1993, 274-308.

[KR] Kernighan, Brian and Dennis Ritchie. *The C Programming Language*. Prentice-Hall, 1988.