FACTS

A **F**ormal **A**nalysis of **C++**: **T**ype rules and **S**emantics

by

Adam Seligman

A Thesis
Submitted in partial fulfillment
of the requirements for the Degree of Bachelor of Arts with Honors
in Computer Science

WILLIAMS COLLEGE
Williamstown, Massachusetts
May 22, 1995

Abstract

This thesis presents a formal analysis of the core of C++, the method binding mechanism. The analysis will include features such as inheritance, overloading, overriding, hiding, and conversions. Our model will clearly explain method binding under all these features, without the ambiguity that colloquial descriptions can introduce.

Many of the intricacies of our formal model will demonstrate that the informal descriptions used to explain the language are not sophisticated enough to explain certain examples. The type rules and operational semantics we present should provide the basis for a more accurate conceptual model of the language.

Contents

Chapter 1

Background

1.1 Programming Languages: Between Industry and Research

The field of programming languages exists in a peculiar position, at the intersection of users and researchers with often very different demands. There has been a rift between the efficient, straightforward programming languages long preferred by developers, and the elegant, unimplemented beauties generally designed in academic environments. Programmers might like to take advantage of modern programming language developments, but are unable to justify the training overhead, or unable to find an efficient compiler. The semantic elegance of other languages is often quite attractive on paper, but some distance from an implemented reality.

As the gap has widened between the languages used in the "laboratory" and the "field," software packages have continued to grow to bloated sizes. Even the ubiquitous word processing programs on microcomputers have ballooned to megabytes of source code. (The next generation of the most popular operating system for microcomputers is compiled from almost a gigabyte of source code!)

The rampant growth of program code has begun to force developers into using programming languages with features that allow for efficient development and maintenance. Developers must be able to break their large projects into a collection of small, manageable projects. Interfaces need to be established between different parts of the programs. Low-level imperative programming languages such as C do not offer the features necessary to make such large projects feasible. The industry has decided that object-oriented features will provide a solution to their dilemma.

1.2 The Object-Oriented Paradigm

The object-oriented paradigm is a radical shift from simple imperative programming languages. Let us first consider the paradigm in the abstract. We must consider the three basic features: abstraction, encapsulation, and polymorphism.

The first feature, abstraction, is the notion that it is important for a programmer to work with constructs at the highest possible conceptual level. The commands a programmer writes should reflect the problem she is working with rather than be constrained to shuffle bits around manually. For example, when writing a payroll package, it would be convenient to group all the data associated with each employee in a sort of package. A programmer would create the general framework for an employee record, a "class," and then instantiate an "object" for each employee with the appropriate personal data.

The second feature, encapsulation, builds on the notion of abstraction by grouping data and the functions that operate on that data together with the object. If a raise is to be given to an employee, it would be convenient to program it with a simple message of "raise" sent to the appropriate employee object. A function that is defined for a particular kind of object is called a "method." While encapsulation is often seen as a simple organizational convenience, we will shortly examine the way in which the type of objects can be checked to ensure that a raise is never given to the wrong kind of object, such as the office coffee machine.

Inheritance is the mechanism by which classes can inherit data fields and methods from another class (or classes). Inheritance allows large classes to be built up from smaller classes, but more importantly allows other classes to masquerade as their base classes. For example, an employee record class might inherit linked-list functionality. Inheritance will become much more useful when combined with polymorphism.

Polymorphism is a very important part of the object-oriented paradigm. Simply, a polymorphic piece of code is one that can act on different kinds of data. More precisely, a polymorphic function is one that can operate on objects of multiple types. Different implementations of polymorphism allow for various levels of flexibility on the classes (the *kinds* of objects) that might be accepted by a single function.

Programming in the object-oriented paradigm is quite different than traditional imperative languages. The programmer will break the entire problem into smaller problems, and the nature of classes enforces an interface between the parts. It is generally straightforward to design these interfaces in an intuitive, consistent manner. This trait of object-oriented languages is conducive to software projects implemented by multiple programmers. Class interfaces are standardized by the team, and then programmers may implement each part as they choose. In some ways, good object-oriented programming technique encourages (or even forces) programmers not to intrude into the data of other classes directly, but rather access that data through the appropriate class interface, which generally isolates errors to a small piece of project.

1.3 The Demand for Object-Oriented Methodology

The features of object-oriented programming languages are understandably quite attractive to developers. Well-documented class interfaces are more maintainable than cryptic function calls, and the isolation of bugs to within a small portion of the project is also invaluable.

The choice of object-oriented languages available on the market is limited because of the requirements for most software development. Many object-oriented languages have been designed, and implemented to various degrees. They vary in syntax, feature sets, and degree of flexibility, but few are both implemented efficiently and well supported, which software development use demands.

A large amount of software development is done with the C programming language. Flexible, terse, and conceptually simple, C is a benchmark of efficiency, and is widespread enough to be considered a portable assembly language. These characteristics of C, however, are not conducive to the development and maintenance of large software projects. There are legions of programmers who have been writing in C for years, and have come to rely on various external tools and test suites to check for the problems that often go unchecked with C programs. However, because C is so "low level," it becomes very difficult to manage large projects. Ideally, by using the features of the object-oriented programming paradigm, developers can help make these large projects more manageable.

1.4 The Birth of C++

1.4.1 In the Beginning: Abstraction, Encapsulation, and Inheritance

C++ was a product of Bjarne Stroustrup out of Bell Labs, in Murray Hill, NJ. Stroustrup introduced the object-oriented features abstraction and encapsulation in "C with Classes," in the early 1980s. The simple C record type, the "struct," was extended to include operations on that data. For example, functions like "give a raise to person x" could be grouped with each employee object, simplifying the connection between data and the appropriate corresponding functions. Parts of a class, such as internal data, could be kept from access outside of the class, enforcing abstraction through the class interface. C with Classes also supported inheritance, where a class could be "derived" from another class, and would include all of the "base" (parent) class' data *fields* and methods.

A decision was made to enhance C with Classes into a commercial language, C++, and more object-oriented features were added. Until this point the form of polymorphism supported by the language was somewhat limited. A derived class would often be a more specialized version of a general base class. A subtyping rule was introduced that allowed objects of a derived class to be used anywhere an object of a parent class was expected. For example, a single set of linked list manipulation routines could operate on any specialization of the linked list class without having to be rewritten.

Let's look at an example. Suppose we would like to write a generic list class, and then create a more specialized list of employee data records. We decide on two classes for the list: a class `list_elt` that stores pointers to the next and previous elements in the list (doubly-linked), and a class `list` that keeps track of the head and tail of the list, and contains all the methods we can use on the list.

Example

```
class list_elt {
public:
  list_elt *next,*prev;
  int val;
  void display(void) { /*print our val*/ }
};

class employee_elt : public list_elt {
public:
  char *name,*title;
  void display(void) { /*print out name and title*/ }
};

class list {
public:
  list_elt *head,*tail;
  list_elt *return_first_elt(void);
};

void trouble(list L) {
  (L.return_first_elt)->display();
}
```

Notice that the class `employee_elt` inherits the `next` and `prev` pointers, and the data field `val` from class `list_elt`. Since `employee_elt` is derived from `list_elt`, C++ allows the employee element objects to masquerade as list_elts in list objects. Therefore, we can instantiate a list object and fill it with employee_elt objects.

Consider what happens when we pass the function trouble a list object whose elements are list_elts. The first element is returned by the method, and then the display message is sent to the object. But, notice that return_first_elt returns a pointer to a list_elt, and so display method called will come from list_elt, not employee_elt. This is not a type error, as the method list_elt::display works fine on employee_elts, but we are not calling the expected employee display method.

In order to return an employee_elt from the return_first_elt method, a more specialized version of the list routines would have to be written. This example uses subtyping well, but will be more effective when combined with polymorphism.

1.4.2 C++'s Flavor of Polymorphism: the Virtual Function

The virtual function is C++'s flavor of polymorphism. When we redefine a method in a derived class (like we did with display in the previous method), we would like the derived class version to be called whenever we send the draw message to a derived object. A virtual function is a method that is (usually) selected dynamically. If we redefine our display methods above as follows, the call to display would act in a polymorphic manner:

```
class list_elt {
public:
  //...
  void virtual display(void) { /*print our val*/ }
};

class employee_elt : public list_elt {
public:
  //...
  void virtual display(void) { /*print out name and title*/ }
};

class list {
public:
  list_elt *head,*tail;
  *list_elt return_first_elt(void);
};

void trouble(list L) {
  (L.return_first_elt)->display(); // Method will be selected dynamically
```

```
}
```

The `virtual` keyword is used to signify that the function should be bound dynamically. It is important to note that the method needs to only be considered `virtual` in the base class, and the keyword `virtual` does not need to be rewritten in the derived class. In fact, despite the confusion its omission might cause, the keyword's appearance in the derived class is sometimes referred to as "extraneous"!

In the example, if an employee list class was derived from a general list class, a generic routine working with general list objects would not have to be specialized to work with employee lists. The `display` message can be sent, and the actual method invoked will come from the appropriate class, the type of the object itself (called the dynamic type), not the type of the pointer (called the static type). The decision made of which method to actually call can generally only be determined at run time, and has given rise to an implementation of functions known as a "virtual function table" for every class.

This form of polymorphism is powerful, and conducive to the minimization of code replication. Other classes could be derived from the `list_elt` class, each including a `display` method conforming to the interface in the base class, and the general list routines would not have to be modified! Libraries of abstract data types can serve as base classes for specialized versions, rather than being coded in over and over again.

1.4.3 Overloading

A feature called "overloading" was introduced to C++, which allows functions and class methods to share names, but operate on different types. Almost every operator in the C++/C language can be overloaded to handle new types. A user could create, for example, a 3×3 matrix class and new versions of the addition and multiplication operators. Even the assignment operator can be overloaded. A programmer could simplify her code by replacing expressions of the form

```
matrixC.setValue(matrixA.multiply(matrixB))
```

by the simplified

```
matrixC = matrixA * matrixB
```

Overloading is not parametric polymorphism, as functions that share a name but have different argument types are different functions, and therefore have separate function bodies. Overloading can be seen primarily as a syntactic convenience, but has great semantic consequence on the language because of its tenuous relationship with inheritance.

1.5 Overloading and Inheritance: The Dangers of Mixing and Matching

Overloading and the virtual function mechanism are both straightforward when considered by themselves, but often have surprising interactions when used simultaneously. The complexity lies both in the subtlety of the different purposes of the mechanisms, and programmer misunderstanding of the way the mechanisms are actually implemented. In order for overloading to work properly, and also to encourage the feature's appropriate use, reusing the name of a method in a derived class obscures any base class methods with the same name. This seemingly simple rule is known as "hiding."

Hiding can cause much confusion in C++ programs, as methods might disappear in intermediate derived classes, but some of their properties (such as virtual-ness or access control properties) persist in downstream derived classes. A major indictment of C++ is that the interaction of virtual functions, overloading, and hiding is cryptic without an understanding of how the compiler implements those functions.

For example, consider the following C++ code:

```
class B {             //base class
public:
   void m(B b) { }
};

class D: public B {   //derived from B
public:
   void m(D d) { }
};

main() {
  B b;
  D d;
  d.m(b);  //illegal call - no method D::M(B)
}
```

Clearly, this behavior might surprise many novice C++ programmers (as well as a few more proficient programmers). The method `B::m(B)` is not inherited into class `D`, but rather is hidden by `D::m(D)`. When we try to call `d(b)`, there is no method that takes a `b` object. (Note that this is not subtyping, where an object from a *derived* class can be used anywhere a *base* class object is expected.)

A decision was made by the designers of C++ not to allow overloading across levels of inheritance. If the user reuses the method name, the base class version is hidden. *This trips up many new C++ users.* Overloading is primarily intended to be used as a convenience to make code more readable. Another benefit is that libraries (such as the `iostream` library) can be extended to operate on new user types. It is common practice to add new versions of the output operator, <<, so that programmers can easily output objects of their own types.

1.5.1 Multiple Inheritance

Eventually, multiple inheritance was added to C++. Multiple inheritance is the provision for a class to inherit from multiple parent classes. Those classes, in turn, might inherit from multiple parents. Objects of a particular type can masquerade as an object of any of its base classes. Pointers to an object of the derived class can be used anywhere pointers any of the base classes are used.

1.6 C++ and OO

At this point we should stand back and analyze C++'s relationship to the object oriented paradigm. Clearly the basics of encapsulation and abstraction are handled reasonably well. Data hiding is possible through access controls, but is not enforced. The programmer is free to leave internal data fields accessible outside her classes, even though this is not in the spirit of object-oriented programming.

1.7 OO for Developers

The object oriented features of C++ are a mixed bag. While very powerful, the semantics of their interaction is often cryptic. The object oriented features were "tacked on" to a low-level imperative language. Polymorphic class families are used at the same time as pointer arithmetic, offering more than enough "rope" for a programmer to get herself into trouble. C++ is not a strictly object-oriented language, but is multi-paradigmatic. Many users initially use C++ as a better C, and make little use of the sophisticated object-oriented features.

C++ has been described by my colleague Ham Richards as a "747," the behemoth jet airplane: very large, very complex, and very powerful, but best trusted to a skilled expert. Jumping from familiarity with a powerful, but simple imperative paradigm, to the intricacies of object-oriented polymorphism is not easy. It is important to approach the language from an object-oriented standpoint, rather than a familiarity with C. The ability to code C++ programs with a C background entices developers, who would like the benefits of object-oriented features, but who have programmers versed in C. Because of the paradigm shift to an object-oriented language, combined with the idiosyncratic relationship of C++ to the object-oriented language paradigm, C++ can be very difficult to master.

1.8 Shortcomings of C++ Reference Materials

The most important document used to decipher C++ is the ARM. The Annotated Reference Manual [ES91], by Margaret Ellis and Bjarne Stroustrup, is the base document for ANSI C++. It consists of the Reference Manual and annotations from the discussions of the ANSI committee. The ARM has all of C++ within its covers, but is so punctuated by cross-references and comments that it is difficult to read, particularly by beginners. The ARM spends considerable energy suggesting (in general terms) how the various features of C++ are implemented by most compilers. This discussion is our first hint that some language issues are sufficiently complex to require an understanding of compiler internals to decipher their semantics.

Stroustrup's other books include "The C++ Programming Language,"[Str91] and "The Design and Evolution of C++."[Str94] The former provides a general introduction to C++ and includes the (un-annotated) Reference Manual, while the latter offers an insightful discussion of the evolution of the language. Stroustrup clarifies his concerns and priorities when designing and implementing the language (in its various manifestations over the years). It is particularly insightful to see his comment, "I was very concerned that improved program structure was not achieved at the expense of run-time overhead compared to C. The explicit aim was to match C in terms of run-time, code compactness, and data compactness. To wit: Someone once demonstrated a 3% systematic decrease in overall run-time efficiency compared with C caused by the use of a spurious temporary....This was deemed unacceptable and the overhead promptly removed." C++ lives in the realm of compromise lying between speed, compactness, object-oriented features, and intimate relationship to C.

However, one of the major flaws in C++ is the complexity induced by multiple features interacting, such as hiding, overloading, and virtual functions. As a programmer begins to use the language more aggressively, combining these features, surprising results often occur.

For example, C++ can surprise novice users with the following:

```
class B {
public:
  void virtual vf(void) {}
};

class D: public B {
// no methods declared
};

class E: public D {
  void vf(void) {} //notice no virtual keyword
};
```

```
class F: public E {
  void vf(void) {}
};

main() {
  F f;
  E *ep = &f;
  ep->vf(); //which method is called?
}
```

The ARM says: "If a class `base` contains a `virtual` function `vf`, and a class `derived` derived from it also contains a function `vf` of the same type, then a call of `vf` for an object of class `derived` invokes `derived::vf` (even if the access is through a pointer or reference to base). The derived class function is said to *override* the base class function." (ARM 10.2)

This passage is a good example of the way ambiguities can creep into informal language specifications. This passage introduces the virtual function mechanism, and describes how dynamic binding is used whenever the base class has a virtual function with the same "type." The rule is being *given* by an example, as opposed to being illustrated by an example. So, when she tries to understand C++ behavior in general cases, a programmer must extrapolate from the example.

The rule does *not* mention the case of a method being declared as `virtual` in an *indirect* base class, as in our example. In fact, if a method is declared in a class as `virtual`, it will be virtual in all classes derived directly or indirectly.

The May 20 working paper of the ANSI/ISO standards committee was revised to address this issue:

"If a virtual member function `vf` is declared in a class `Base` and in a class `Derived`, derived directly or indirectly from `Base`, a member function `vf` with the same name and same parameter list as `Base::vf` is declared, then `Derived::vf` is also virtual (whether or not it is so declared) and it *overrides* `Base::vf`. For convenience we say that any virtual function overrides itself. Then in any well-formed class, for each virtual function declared in that class or any of its direct or indirect base classes there is a unique *final overrider* that overrides that function an every other overrider of that function." (20 May Working Paper, 10.3.2)

This seems to address the issue the previous example raised. However, consider the following example:

```
class B {
public:
  void virtual vf(void) {}
};
```

```
class D: public B {
  void vf(int) {}
};

class E: public D {
  void vf(void) {}
};

class F: public E {
  void vf(void) {}
};

main() {
  F f;
  E *ep = &f;
  ep->vf(); //which method is called?
}
```

Which method is bound to the call `ep->vf()`? Recall that the keyword `virtual` is only needed in the base class, and a method is virtual in the derived class. But we did not mention whether this also held for indirect base classes (i.e.: a base of a base class) when there is an intervening class that **hides** the base class method.

So which method should be called? Not all compilers agree which method should be called by `ep->vf()`. Some preserve the virtuality of `vf(void)` in class E and F, and `ep->vf()` results in `F::vf(void)` being called. Others compilers resolve the call to `E::vf(void)`, meaning that the virtuality was hidden by the intervening function. But shouldn't the language specification explain this example clearly?

Notice that the issue of intervening methods hiding the "virtuality" is never addressed directly, but rather is inferred from the working paper's passage. Informal language specifications leave "holes" such as these in the language. As this thesis neared conclusion, a new version of the working paper was released. This section has again been rewritten. To our surprise, an example suspiciously like the preceding one was included to make it clear that virtuality is, indeed, preserved through intermediate method hiding. Although this issue may be addressed by the example, there are an inordinate number of other "holes" that have failed to be addressed. The addition of more and more examples yields a sprawling language standard. Hopefully, the shortcomings of such informal approaches to language specification will encourage the use of formal techniques.

1.9 Semantics are precise!

The various OO features of C++ interact in complex ways. One cannot readily extrapolate from a trivial example given in a reference manual to the more complicated task at hand. Succinctly, the semantics of the programming language are not clear from C++ language specification. This raises an interesting philosophical question: what is C++? If the standard itself is ambiguous, are those ambiguities part of the language (i.e., unspecified behavior)?

We argue that the ambiguity of the semantics lies with the informal approach used for the language specification. The language designers intend to specify the semantics of the language, but are unable to do so precisely without a formal "tool set". Some behavior of the language is sufficiently undefined to suggest that the omission was accidental. We believe that the semantics we have determined for C++ describe the core of C++ precisely, including specifying behavior that was omitted in the language specification. The semantics of a programming language are the heart of the language, and should not lie submerged as they are in C++.

The confusing semantics of C++ cause grief among compiler writers struggling to generate correct output. Many significant differences exist between compilers and even versions of compilers. Users do not often realize the decisions behind the different compiler outputs, and simply adjust their internalized understanding of the language to match the semantics of their favorite compiler. One C++ book author put it succinctly when he said, "Do not attempt to learn the semantics of multiple inheritance from the compiler."

Without a formal semantic analysis, most texts on the C++ language either rehash the basic explanation for the various object-oriented features, or provide almost anecdotal solutions for common problems. These texts are invaluable for the need they address: the beginning user of C++, and the experienced user wishing to polish her style. Hopefully, the type-checking rules and natural semantics presented in this paper will assist the user of C++ to understand its object-oriented features more completely.

It is important to note that our formal rules are not intended to simply present the material in the ARM formally. Our intention was to create a semantic model for the language we think C++ should be. For the most part, the C++ we represent concurs with the language specification. However, some of the behavior that we specify is unspecified in the ARM. We feel that the ambiguities in the ARM were accidental, and therefore our semantics are a more accurate specification of the language. On the other hand, some would argue that since the ARM *is* the language specification itself, any discrepancy with our rules indicate that our model is of some language other than C++. We submit that the language represents what we feel C++ should be, given our understanding of the designers' intentions.

1.10 Other work in the field

This thesis sits at the intersection created by applying a formal analysis to an informal language specification. C++ was originally designed at a time when much of the theoretical work on object-oriented languages was just beginning. The theoretical developments in the study of object-oriented languages can offer insight into the behavior of a language like C++.

Work by Cardelli resulted in [Car88] and [Car85], which set the groundwork for the use of formal techniques to model the behavior of object-oriented programming languages. Such formal techniques would then facilitate important developments in the understanding of object-oriented languages, such as the realization that subtyping and inheritance are distinct, as presented in [CHC90]. Bruce's work, in the papers [BL90],[BM92],[BCD+93], [Bru93], [Bru94], and [BSvG94] (among others), has furthered the theoretical development of programming languages and strengthened the connection between formal analysis of the object-oriented features of a language and the development of a high-level conceptual model of the language.

Abadi's analysis of the core features of Modula-3, in [Aba93], has served as a model for the analysis we have applied to C++. In addition, van Gent's thesis, [vG93], and Schuett's thesis, [Sch94b], have set very high standards for the depth of semantic analysis appropriate to an undergraduate thesis.

1.11 A Guide to this Thesis

This thesis is designed to be approachable by a broad range of audiences. For those not familiar with C++, chapter 2 contains a brief introduction to the main features of the language. The following chapter, 3, introduces the formal analysis, including a discussion of the "flavor" of C++ that our analysis deals with.

Chapter 4 introduces the reader to the type-checking rules we have developed. Chapter 5 presents the type-checking rules themselves, in a form most suitable for reference. The chapter immediately following presents a detailed "walkthrough" of the type-checking rules.

Chapter 7 begins our presentation of the semantics we have determined for C++, by introducing the various constructs our semantics utilize. Chapter 8 presents the semantic rules in a compact form, and Chapter 9 discusses them fully.

Chapter 10 discusses some of the implications of the formal model of C++ for users of the language. The chapter attempts to utilize the formal rules to build a high-level conceptual model of the language. The semantics of the language appear to be sufficiently complex that a simple high-level conceptual model of the language is not possible. The conclusion of the thesis, chapter 11, attempts to put the language and the formal analysis into a perspective approachable by both programming language researchers and users.

Appendix A contains a glossary of the various constructions used in the formal model.

Chapter 2

A Brief C++ Primer

C++ is a big language. Its intricacies include very powerful features, which are sometimes redundant. This chapter should serve as an introduction to C++ itself for those unfamiliar with it. In the introduction, we saw the object oriented features that C++ supports, albeit in a relatively brief form. In this section we will examine those features more carefully.

C++ programs follow the syntactic style of C programs quite closely. The source code includes definitions of functions, concluding with a final function definition called `main`. This function is the first to be executed, and the program will terminate when `main` terminates.

2.1 Functions

Functions generally look like the following example, the successor function:

```
int succ(int x) {
   return(x+1);
}
```

The return type, `int` (an integer), precedes the function name, in this case "`succ`". The argument type and name go inside the parentheses after the function name. Our successor function takes an integer argument, which will be referred to as `x` within the body of the function. C++ uses call by value semantics, meaning that `x`, within the function, refers to a copy of whatever was passed in as an actual parameter. We will discuss this later more carefully, but suffice it to say that this is the same behavior that C uses.

The body of the function may contain other function calls (including a recursive one), assignment statements, and the creation of objects, among others.

2.2 Local Variables

Local variables can be created with statements like: `int x; char* cp;`. The first declares an integer variable named `x` in the current scope, while the second stores a pointer to a character. These definitions can be used outside of any class, method or function definition, in which case they are global (within the source file), or they can be created inside a function or method body, in which case they live in the scope of that body.

2.3 Free-store Variables

Variables and objects can also be allocated off the heap, with the `new` operator. The argument to the new operator is the type of object to allocate space for, and the call returns a pointer to the location just allocated. A common form of the call to `new` is the creation of an object for some pointer to point to: `B *bp = new(B);`. This statement allocates space for a new `B` object on the heap, and assigns the location to `bp`, a pointer to `B`.

2.4 Classes

One of the fundamental features of C++ (as with all object oriented languages) is the class. A class is a type, containing a collection of typed data fields (much like a record), and a set of "methods" that operate on those fields. Classes are a form of abstraction, allowing the programmer to encapsulate a particular data type with the functions that operate on that type. In addition, these functions (the methods of a class) can serve as an interface to the internal values of the class. If the internal representation of the data is modified, but the interface is maintained in a consistent manner, the changes will only affect the class internally. This is one of the main benefits of object oriented programming languages.

Here is an example of a class declaration:

```
class Point {
public:
  int x,y;
  int getX(void);
  int getY(void);
  void setXY(int newX, int newY);
};
```

The first class, `Point`, has two integer fields, `x` and `y` (the comma allows for multiple fields to be declared on the same line). The next line, `int getX(void)`, is the *declaration* of a method named `getX`, that takes no argument and returns an integer. getX is an example of an interface method, and would be used to retrieve the x component of the Point class, rather than some other code accessing the field itself. This way, the programmer has the flexibility to change the internal representation of the class (perhaps to polar coordinates) without worrying about "breaking" code outside the class.

The public keyword establishes a certain level of access control over the components of the class. Access controls are restrictions generally used to enforce the use of a class interface, and prevent other functions from accessing (and potentially garbling or becoming dependent on) class internals.

The method int getY(void) is the declaration for the method to fetch (or compute) the y value of the class. The next method, void setXY(int newX, int newY), would be a consistent way to set the internal values of the class, rather than having other pieces of code modifying Point's internals directly.

Notice that the methods mentioned in the class were method *declarations*, announcements that the method is visible (present) in the class, and that we will define them shortly. The next lines are a method *definition*:

```
int Point::getX(void) {
   return(x);
}
```

The syntax for method definitions is quite similar to that of function definitions, with the addition of a class name and a pair of colons, specifying which class the method was declared in. Methods are expected to be declared in the class, and may be defined either inside or outside of the class.

2.5 Access Controls

One of the features of C++ is access controls. Access controls are a way of restricting who may call the methods of a class. In addition, instance variables can be made internal to the class. The public, private, and protected keywords are used to specify the access controls for a class' methods and instance variables. For example:

```
class Point {
private:
  int x,y;
protected:
  void debug();
public:
  int getX(void);
  int getY(void);
  void setXY(int newX, int newY);
};
```

In this class we have the private instance variables `x` and `y`. These variables may only be accessed by methods of the class. If the user wants to find a `Point`'s `x` and `y` values, she will have to call the public `getX` and `getY` functions. The function `debug` was declared with the `protected` keyword, which means that it cannot be called from outside the class, although has the potential to be called by a class derived from `Point`. We have not seen inheritance yet, so for now consider `protected` to be equivalent to `private`.

The `public` keyword on the last three methods simply mean that there are no restrictions on who may call the methods. The public methods of the class can be considered the class *interface*. Good class interface design is important in object-oriented programming. Using class interfaces effectively greatly benefits program modularity, which is important for the design and implementation of large software projects.

2.6 Objects

Objects are instantiations of classes, meaning that they are variables with values for each of their fields. Objects can be sent a message to call one of their methods. There are various kinds of method calls, but the general principle is that one needs a receiver and a message to call a method. Method calls might look like the following:

```
Point p;
p.setXY(1,7);
int x = p.getX;
```

The first line creates a `Point` object named `p`. The second line tells `p` to invoke its `setXY` method with the arguments 1 and 7. Presumably this method will update the internal fields of `p` appropriately. Note that we are changing the values in the fields of the object `p`, not the class `Point`, which is just a type (and hence *never* has values for its fields).

The last line creates a variable `x` and assigns it the output of the `getX` method called on `p`. The variable `x` will be initialized with the `x` *field* of `p`.

2.7 Inheritance

One of the features supported by C++ is the notion of inheritance, the ability for a class to be "derived" from another class. Deriving a class from a "base" class allows methods and fields in the base class to also be present in the derived class. In this manner, a derived class (child, or subclass) can be an augmented versions of a base (parent) class by inheriting fields and methods from the base class and defining fields and methods of its own. This is a form of code reuse, but it will also be the basis for subtyping and polymorphism, which we will see shortly. First let us consider a typical class derivation, from `Point` to a `ColorPoint`:

```
class Point {
public:
   int x,y;
   void setXY(int newX, int newY);
};

class ColorPoint: public Point {
public:
   int color;
   void setXYC(int newX, int newY, int newC);
};
```

This code defines a base class `Point` and a class `ColorPoint` derived from it. Since `ColorPoint` is derived from `Point`, it will inherit all of `Point`'s fields an methods. So, the visible components of a `ColorPoint` object would be the integer fields `x,y,color`, and the methods `setXY` and `setXYC`.

2.8 Subtyping

One of the most interesting uses of inheritance is the potential for subtyping. Subtyping in C++ is the ability of an object of a derived class to masquerade as an object of one of its base classes. For example, any function expecting a `Point` object could be sent a `ColorPoint` object. There are some surprises when one subtypes with objects (as opposed to pointers to objects) which we will discuss in great detail later in this paper.

Subtyping with pointers is quite useful because when combined with dynamic method binding, we achieve polymorphism. Firstly, subtyping with pointers hinges on the ability for C++ to convert from a pointer to a derived object to a pointer to one of its base objects. In this manner, when a function expects a pointer to some base class as its argument, it can be passed a pointer to an object of a derived class and operate normally.

2.9 Dynamic Method Binding and Polymorphism: the Virtual Function

Dynamic method binding is the name for selecting an appropriate method to use at run-time, depending on the type of the object being sent the message. In C++, this is used when one has a pointer to an object, and one sends it a message. Because of subtyping, a pointer to some base class might actually (at run time) be assigned an object of a derived class. If we want to use a method that has been specialized for that derived class, we need some way to select the actual method at run time. C++ uses a construction known as the virtual function table to select the appropriate method at run time.

A virtual function is a method declared with the `virtual` keyword. This keyword signals to the compiler that a (virtual) method invocation cannot be bound to a particular function body until runtime. For example:

```
class Point {
public:
   void virtual draw(void);
};

class ColorPoint : public Point {
public:
   void virtual draw(void);
};
```

The `draw` method takes no arguments and returns nothing, it is run for its side effect, ostensibly drawing a point or a colorpoint on the display. Notice that the `draw` routine for a `Point` would simply want to display a monochrome point on the display, while the `ColorPoint` version would draw a point with the appropriate color. In this way, the method `ColorPoint::draw` is a specialization of the base class's `draw` method. When a method in a derived class has the exact same name and types as a method in a base class, the derived method is said to *override* the base class method. It is an error for a method in a derived class to have the same name and argument types as a virtual function in a base class, but a differing return type. C++ enforces this restriction so that a virtual function call will not return an unexpected type at run time.

Now, consider those class definitions with the function below:

```
function draw2points(Point *p1, Point *p2) {
   p1->draw();
   p2->draw();
}

main() {
   Point p;
   ColorPoint cp;
   draw2points(&p,&cp);
}
```

Observe how `draw2points` is called, with a first argument that is a pointer to a `Point` object, and a second argument that is a pointer to a `ColorPoint` object. The first argument, `p1`, is sent the `draw` message. Since the method `draw` is virtual, we look at the actual type of the receiver. Since we passed in the `Point` `p` as the first argument to `draw2points`, the actual type of the object `p1` points to (within the method body), is an object of class `Point`, and so `Point::draw` is called.

The second argument to `draw2points` is a pointer to a `ColorPoint` object. When we send the `draw` message to `p2` within the body of `draw2points`, C++ will dynamically select the method `ColorPoint::draw()` to execute.

C++ has a clever mechanism to implement dynamic binding, known as the virtual function table. Each class has an associated virtual function table that keeps track of its virtual methods. Objects keep a pointer to the virtual function table of their type at all times. In this manner, when a virtual function is actually executed, an object can be told to look that method up in its virtual function table.

2.10 Overloading

Overloading is the ability for multiple methods to share the same name. As long as the types of these arguments differ, the compiler can differentiate between which method to use. For example, the method `set` below is overloaded:

```
class Point {
public:
   int x; int y;
   void set(void);
   void set(int newX, int newY);
};

void Point::set(void) {
  x=0; y=0;
}

void Point:set(int newX, int newY) {
  x=newX; y=newY;
}
```

The `set(void)` method is designed to reset the fields of the class to 0's, while the `set(int,int)` method sets the fields to particular values. The compiler will resolve calls to `set` by figuring out the argument types:

```
void test(Point p) {
  p.set();
  p.set(1,7);
}
```

This function tests the two `set` methods of the class `Point`. The first call, `p.set()`, will result in the compiler deciding that the appropriate method is `void Point::set(void)`. The second call, `p.set(1,7)`, will result in the compiler's selection of `void Point::set(int,int)`. This process of selecting the appropriate method is known as "method binding," because we are binding a bit of source code that calls a method to the actual method to execute.

Overloading is only allowed within a class, not across levels of the inheritance hierarchy. If we derive a `ColorPoint` class from the `Point` class, and introduce a new flavor of the `set` method, it will *hide* the set methods from `Point`:

```
class Point {
public:
   int x; int y;
   void set(void);
   void set(int newX, int newY);
};

class ColorPoint: public Point {
public:
   int color;
   void set(int newX, int newY, int newColor);
};

void ColorPoint::set(int newX, int newY, int newColor) {
   x=newX; y=newY; color=newColor;
}
```

With this example, we see that the integer fields `x` and `y` are inherited into class `ColorPoint`, and we would expect the same to be true for `set(void)` and `set(int, int)`. This turns out not to be the case. The method `ColorPoint::set(int, int, int)` hides the `set` methods declared in `Point` that share the name `set`. This often surprises novice C++ users and is worth familiarizing oneself with.

The only way a method is inherited from a base class to a derived class is if the method's name is *not* reused in the derived class. The rationale behind this behavior is the observation that the methods `Point::set(void)` and `Point::set(int,int)` might be called on a `ColorPoint` object, and the users would have unwittingly set only *part* of the `ColorPoint` object! So, the decision was made to be somewhat restrictive on inheritance of methods in the language.

2.11 Constructors

2.11.1 Overview of Constructors

Constructors are special methods of a class that are used to initialize objects of that class. There are several different kinds of constructors, and a class may (and often does) have more than one. Constructors come in several different flavors: default, copy, and conversion constructors.

- A *default constructor* handles initialization of instance variables whenever an object of a class is created, and takes no arguments from the user.

- A *copy constructor* initializes an object with values of another object of the same type. This is useful when a deep copy needs to be performed, for example.

- A *conversion constructor* is a constructor that initializes an object with values from arguments of some other types.

Constructors can be user specified or can be generated by the compiler *automatically*. Let's look at each more carefully.

A default constructor is code that is executed whenever an object is created. For example, a linked list object might need its pointers set to nil or some initial element; the user would write a constructor that handled the needed initialization. If the user does not specify a constructor with this signature, the compiler will generate one automatically. The default constructor generated by the compiler would simply allocate a chunk of storage for the object, but not any of the data structures it might point to.

A copy constructor is a method that performs a copy of an object to another object of the same type. This is used when initializing an object with the value of another object, and for argument passing to functions, which we will see later. Notice that the user may specify a copy constructor that behaves in a more useful way, such as performing a deep copy, or the compiler will generate one that uses a simple bitwise copy of all fields.

A conversion constructor is used to initialize an object of one type with values extracted (in some way) from an object of another type. A very common form of this kind of constructor is the one generated by the compiler to convert from a derived class object to an object of a base class. These conversions may be specified by the user, including conversions from any type.

2.11.2 Using Constructors in C++

Constructors must have the same name as the class name, and differ in the argument types.

```
class Point {
   int x,y;
   Point();
   Point(Point &otherPoint);
   Point(int newX,int newY);
   Point(double rho, double sigma);
};
```

Whenever a `point` object is created, a constructor is called. The first constructor, `Point()`, takes no arguments, and is known as a default constructor.

The second constructor, `Point(Point &other Point)`, constructs a point as a copy of another point, and is known as a copy constructor. The "&" after the argument type means that the argument is a "reference" to a `Point`, which looks similar to an object being passed in, but behaves as if a pointer to the object were passed in. This is done rather than simply using a `Point` object as argument to the copy constructor because of a tricky recursive problem.[1] C++ uses the call by value mechanism to substitute arguments for formal parameters. Whenever a function or method is called with an object argument, the compiler is expected to copy the argument in, rather than use the argument itself. (Otherwise, modifications to the argument in the body of the function would would remain after the function returned.) We will see this more carefully in a moment.

First let us examine the next two constructors, `Point(int newX, int newY)` and `Point(double rho, double sigma)`, which are known as conversion constructors. Conversion constructors construct an object with arguments of some type(s) other than `Point`. `Point(int newX, int newY)` would be used to build a `Point` object from the integer pair given. `Point(double rho, double sigma)` would be used to build a `Point` from the polar coordinates given, presumably doing the necessary calculations.

Let's see how the copy constructor is used to implement call by value. Given the preceding declaration of the `Point` class:

```
Point mean(Point A, Point B) {
   int xmean = (A.x + B.x) / 2;
   int ymean = (A.y + B.y) / 2;
   return(Point(xmean,ymean));
}
```

This function takes two `Point`s as arguments, and returns a `Point` whose `x` and `y` fields contain the mean of the arguments' `x` and `y` fields. When we invoke this method with two point arguments, we will see the copy constructor used to implement the argument passing mechanism.

```
Point p1(1,7);
Point p2(1.0,45.0);
Point p3 = mean(p1,p2);
```

The first statement, `Point p1(1,7)`, creates a new `Point` object named `p1`, and calls the constructor `Point(int newX, int newY)` to initialize it. The second call, `Point p2(1.0,45.0)`, creates a new `Point` object from the polar coordinates given. Notice that this (conversion) constructor call would convert the polar coordinates into cartesian coordinates to be stored in the `x` and `y` fields of the object.

[1] Since constructor calls are essentially the same as method calls, if the copy constructor took an object, instead of an object reference, the compiler would have to copy the argument into the copy constructor, causing infinite recursion.

The last line, `Point p3 = mean(p1,p2)`, uses the return value of a function call being used to initialize a new `Point` object. First, let's consider the function call. The function has two arguments, `p1` and `p2`. Since C++ uses call by value semantics, copies of these objects will be used in the body of the function, rather than the objects themselves. So, the copy constructor, `Point::Point(&Point)`, will be called twice, the first time to copy in `p1`, and the second time to copy in `p2`.

Now, within the body of the function (`mean`), the point that lies in between the two arguments is computed, and returned. Notice how it is returned: as a point. The constructor call to `Point(xmean,ymean)` creates a new object, which will be returned by the function.

The statement, `Point p3 = mean(p1,p2)`, is one of the ways an object may be created. A new object `p3` is created out of the new (temporary) `Point` returned by the call of `mean`.

2.12 Call by Value and Constructors

Constructors serve multiple purposes: they handle initialization, copying, and conversion. Our semantics present constructors in a surprising way, but actually represents a consistency often unnoticed by C++ users. Each of the various constructor functions is actually the *same* behavior used in different ways.

For example, subtyping in C++ relies on conversion constructors. Assignment statements, whether object assignment or argument passing, use constructors to simplify the assignment rule in the semantics.

C++ uses an argument passing technique known as "call by value," meaning that objects passed as arguments to function are actually copied in, not passed directly. Since each class has a copy constructor, whenever an object is passed as an argument the copy constructor (or some other constructor) is used to copy the parameter in.

Interestingly, this is how our model of subtyping works. The type checker will specify conversion constructors for a class from each of its derived classes. In this manner, when a function is invoked taking arguments of class B, but passed an argument of a derived class D, the constructor $\breve{B}_{D\rightarrow B}$ will be called to convert it into a B object. (Note that this is a very limiting form of subtyping, as once the conversion is performed, the object *actually is* an object of class B, and has lost any specialization from class D! This is a major criticism of C++: that it's subtyping mechanism is primitive.)

2.13 C++ Resources

As a programmer learns any programming language, primers, reference manuals, and style guides can all be helpful. A large number of books have published on

C++ recently; each has its strengths and merits.

2.13.1 Primers

[Lip91] is arguably the seminal introduction to the language. Lippman makes use of many examples to illustrate the various features of the language in a tutorial style. This primer, however, will not suffice as a reference guide for the experienced C++ user. [Str91] is Bjarne Stroustrup's combination primer and reference guide. The primer is relatively terse, and the examples lack a continuity that simplify their comprehension. The reference guide component lacks the ANSI annotations, which could be seen as a pro or a con! Overall, Stroustrup's guide is a well-rounded mix of primer and reference guide. [WF92] is another good primer, and elucidates many issues and distinctions in manner less tersely than Stroustrup.

2.13.2 Reference

The seminal reference guide to C++, [ES91], is the Annotated Reference Manual, the so called "ARM." The ARM is quite difficult to read, particularly for a novice C++ programmer. The ARM is the language specification itself (and the ANSI standard base document) in a hypertext form committed to print. The reference manual is punctuated with comments and resolutions intended to clarify subtleties in the language. Unfortunately, by flooding the page with descriptions of the subtleties of C++, each written in a different style with different kinds of examples, the *rhythm* of the guide was lost, making it extremely difficult to read. Finding a basic definition of some feature might start on one page, be interrupted by several pages of commentary, and continue on another page.

It is easy to criticize the ARM, but credit must also be given. The ARM specifies the language sufficiently well that *most* compilers agree on C++'s behavior in *most* cases. The infamous complexity of C++ has made this an achievement for an informal language specification.

2.13.3 Miscellaneous

One of the problems with learning C++ and working through its contortions is that so many guides all repeat what the others have to say. I would recommend Scott Meyers book, [Mey92], as a useful set of pointers for an intermediate C++ user. Meyers sometimes fails to explain *why* one should use C++ in a particular way, but the suggestions are generally good nonetheless.

A very interesting book to read is Stroustrup's "The Design and Evolution of C++," [Str94], which appears to have evolved from the first few pages of [Str91] into a full book. The book offers anecdotal explanations of how and why various features were added to C with Classes, and then C++, over the years of its development. As one analyzes the language carefully, there are many situations in which the design of the language was quite tricky. As [Str94] gives insight into those decisions, it also sheds light on the inner workings of the language. I would highly recommend it.

Chapter 3

A General Introduction to the Formal Rules

3.1 Motivation

In this and following chapters we will introduce a formal analysis of the core of C++ under single inheritance. This analysis will include features such as inheritance, overloading, overriding, hiding, and conversions. Ideally, our model will clearly explain method binding under all these features, without the ambiguity that colloquial descriptions can introduce.

Informal language descriptions, reference manuals, and style guides abound. For many languages, these materials are sufficient. Assembly language, for example, is a sufficiently rigid language that a user can understand the semantics of the language without treating them formally.

As programming languages have grown in sophistication, the various features supported sometimes are less intuitive. The notions of inheritance and subtyping can be tricky, and vary greatly in different OO languages. The goals of type-safety and expressibility often come at the expense of each other; the "holes" in the type system of Eiffel come to mind as an example of this tradeoff.

3.2 What *are* Semantics?

The semantics of a programming language are a set of rules that describe the meaning of source code: what execution results. Most users of most programming languages have not seen the semantics of the language presented precisely (formally), but have built a conceptual model of how the language operates. This conceptual model of the language is often sufficient to understand the semantics of the language fully, but not always.

Specifying semantic rules precisely, as our research has done, is significant for two basic reasons. Firstly, we intend to provide a basis for the user to develop an accurate conceptual model of the language. C++ is a sufficiently complex and "contorted" language that its behavior often surprises even experienced users. The second reason for specifying the semantics of a programming language is to provide a basis for criticism and comparison between languages. C++ has both positive and negative features, in our estimation; a judgement that we can justify by examining the semantics. Without such a formal basis for comparison, it is easy for such discussions to degrade into subjective advocacy or dislike without much substantiated evidence.

The semantics of a language are really comprised of two parts, type checking rules and semantic rules. Type checking rules are the first stage of a two-stage process, and in C++ determine to a large part the execution that will actually take place. The second stage is the processing of the source code through semantic rules, that describe precisely how the source will be executed.

The design of C++ is such that most of the object-oriented features, such as overloading, inheritance, and overriding are all resolved statically. In our model, this behavior manifests itself in a large amount of decision-making on the part of the type checker. The type checking rules correspond to the first part of compilation (and symbol table building). Our type checking rules resolve method and function calls into exact function references, meaning that overloading, inheritance, and overriding have all been resolved (decided on) by the time we execute the program. The semantics, in turn, have the potential to be simpler, and hence provide efficient run time performance.

The model itself is comprised of several parts, including rules for the type checking of the program, semantic processing of declarations, and semantic run-time evaluation.

1. The first stage, *type checking*, finds the type of all identifiers, and then checks for the type legality of all statements. Various constructs are generated to keep track of type and other information accumulated during this stage.

2. The second stage, *the semantic processing of declarations*, builds various constructs that are used during the execution of the program, such as the virtual function table.

3. The third stage is *the semantic processing and execution of the program.* These rules precisely specify what the effect of the execution of a given program is. Some of the constructs created in earlier stages are saved to allow for dynamic method binding.

The first stage is encapsulated in a set of type checking rules. The second and third stage are encapsulated in the semantic rules.

The type-checking rules ensure that a program is legal C++. In addition, the type-checking rules bind method calls with actual methods; that is, they determine which method will actually be called at run time. In a later chapter we will introduce the type rules and present a more detailed description of the various constructions built to help type check the program and resolve method and function calls. In addition, we will see the notion of annotation, and how this mechanism is used to simplify the semantic rules.

The semantic rules describe how a C++ program is actually executed. Because our type checker resolves most of the various decisions to be made, the semantics are relatively straightforward.

3.3 Our Version of C++

C++ is sufficiently complex that a complete semantics of the language was larger than the scope of our research. We have limited the domain of the language that we accept, but we hope to have maintained the essential elements of the language.

We have restricted our language to single inheritance, with all derivations (explicitly) public. We do not allow the user to specify any access controls, and we assume all access controls to default to public (rather than private, the default for C++).

For notational simplicity, all functions are limited to one parameter, although methods have the implicit `this` parameter.

We have not included references in our type rules or semantics. Our copy constructor is simply a constructor for a class `C` that expects an object of class `C` as its argument, rather than a `C&`. (We ignore the potential recursion problem, outlined in section 12.1 of the ARM, introduced by this simplification.)

We require the class to contain the method declaration, and keep the definition of the method outside of the class body. For example, rather than:

```
class Point {
  int x, y;
  void draw(void) { BODY }
};
```

We use the following style:

```
class Point {
  int x, y;
  void draw(void);
};

void Point::draw(void) { BODY }
```

When executing the body of a method, there is an implicit `this` pointer to the object that the method was called in. When another method is called, we require the explicit use of `this->methcall()`, specifying which object the message is to be sent to. For example, in code of the form:

```
class Point {
  int x, y;
  void draw(void);
  int getX(void);
};

void Point::draw(void) {
   int temp;
   temp = getX();
   //...
}
```

The call to `getX`, within the body of the `draw` method, would require qualification with the `this`, i.e.: `temp = this->getX()`. This requirement does not affect the semantics of the language, it simply helps keep the size of our formal model to a minimum.

Comments are not processed, so we do not allow them in our input.

3.4 A Grammar for our C++

The following grammar illustrates the manner in which our type-checking rules process a C++ program:

```
PROGRAM = CLASSDECS DEFS

CLASSDECS = CLASSDEC CLASSDECS | <empty>
CLASSDEC = class name {COMPONENTDECS}; | class name: public basename {COMPONENTDECS};

COMPONENTS = COMPONENT COMPONENTS | <empty>
COMPONENT = METHDEC | VIRT-METHDEC | IV-DEC | CTOR-DEC
METHDEC = returntype name(argtype)
VIRT-METHDEC = returntype virtual name(argtype)
IVDEC  = type name
CTORDEC = class-name argtype

DEFS = DEF DEFS | <empty>
DEF = FUNCDEF {BODY} | METHDEF {BODY} | VIRT-METHDEF {BODY} | CTOR-DEF {BODY}

FUNCDEF = returntype funcname(argtype) {BODY}
```

```
METHDEF = returntype classname::methname(argtype) {body}
VIRT-METHDEF = returntype virtual classname::methname(argtype) {body}

BODY = CALL BODY | <empty>
CALL = return(returnval) | OBJ-CREATION |
   FUNC-PTR  | DIRECT-PTR  | INDIRECT-PTR   |
   FUNC-OBJ  | DIRECT-OBJ  | INDIRECT-OBJ   |
   OBJ-ASSIGNMENT | PTR-ASSIGNMENT

OBJ-CREATION = NEW | AUTOMATIC-OBJECT | PTR-CREATION
NEW = new(type)
AUTOMATIC-OBJECT = type variable-name
PTR-CREATION = type * pointer-name

FUNC-PTR = funcname(ptrname)
DIRECT-PTR = obj.methname(ptrname)
INDIRECT-PTR = ptr->methname(ptrname)

FUNC-OBJ = funcname(objname)
DIRECT-OBJ = obj.methname(objname)
INDIRECT-OBJ = ptr->methname(objname)

OBJ-ASSIGNMENT = obj1 = obj2
PTR-ASSIGNMENT = ptr1 = ptr2
```

This grammar describes C++ programs as having two main parts, `CLASSDECS`, declarations of classes, and `DEFS`, method and function definitions. `CLASSDECS` expands to include the declarations of classes, including methods, instance variables and constructors. `DEFS` expands to the definitions of methods and functions. Methods and functions have bodies made up of a series of calls, which include the return statement, the creation of an object, function and method calls, and the assignment operation. In the type-checking rules we will see the distinctions between pointer arguments (of function and methods), object arguments with an exact match, and objects that require an implicit conversion.

Chapter 4

An Introduction to the Type Checking Rules

4.1 What *are* Type Checking Rules?

Type checking rules are a formal description of an algorithm to "accept" a program, indicating that it is legal C++ code.

Types are used in programming languages to specify the form of data. Data–1's and 0's–are amorphous, and are meaningless without some notion of how they are to be decoded. The type of a variable is a "map" that describes what each part of the binary sequence means. Types are also used statically to catch errors in a program. If a user had a value of one type, but tried to interpret it as another type, she would likely get meaningless output.

Type checking is the process by which the compiler verifies that the actual types of data being passed around within a program all match what is expected. This ensures that when running, the program will not somehow misinterpret a piece of data, resulting in a garbled output, a program crash, or the shutdown of all telephone switches in New England!

Type-checking rules are a special way of specifying the type checking that the language requires as part of its definition. Our type-checking rules describe how C++ programs should be type checked by a compiler. This chapter will introduce some constructs that our type checking rules use and familiarize the reader with type checking rules in general.

4.2 A Brief Introduction to Type Rules

Type rules consist of a group of "premises" that justify one or more "conclusions." Premises are requirements; conditions that must be true for the rule to process. Conclusions present the information the compiler can logically deduce from the premises. Type rules are similar to logical rules of inference, in that satisfaction of all the premises allow the type checker to consider the conclusion to be valid.

The general form of a type rule is as follows:

$$\text{a generic type rule} \qquad \frac{\begin{array}{c} premise_1 \\ premise_2 \end{array}}{conclusion}$$

This rule states that if $premise_1$ and $premise_2$ are true, *conclusion* is true. Conclusions come in several different flavors. For our purposes, we will use two: productions and type resolutions.

A type rule with a production as a conclusion is used to process declarations, and build up some useful construction with information. For example, when a class declaration is processed, we would like to store the appropriate information in a construct known as $Symbols$. Production type rules look like the following:

$$\text{a generic production} \qquad \frac{\begin{array}{c} B\ satisfies\ premise_1 \\ B\ satisfies\ premise_2 \end{array}}{(B_{sym}, classB\{CLASSBODY\}) \Rightarrow B'_{sym}}$$

This psuedo-rule states that given a construct B_{sym} and the source code *class B* $\{CLASSBODY\}$, the type checker can conclude that B_{sym} should be updated to B'_{sym}. The left side of the conclusion contains the constructions the rule is passed, followed by the source code to process. As we process source code, these constructs accumulate type information used to type-check subsequent parts of the program. B_{sym} is a construction that stores the symbol table entry for class B. B'_{sym} is presumably B_{sym} updated with the information we have just confirmed about B, namely $premise_1$ and $premise_2$. So, this rule updates a construct B_{sym}, provided the two premises are true, by allowing the type checker to conclude B'_{sym}.

The other kind of type rule, type resolution, is the type checker's way of establishing the type of an expression. The left side of the conclusion indicates the constructions used to type check an expression, which follows the $\vdash$. The type of the expression is established to be the type that follows the colon.

Note that we will use "*ok*" to refer to something whose type is determined to pass the type checking rules. Let's see an example:

$$\text{generic type resolution} \qquad \frac{\begin{array}{c} expression\ satisfies\ premise_1 \\ expression\ satisfies\ premise_2 \end{array}}{(E, Symbols, Bases) \vdash expression : \tau}$$

This rule concludes that *expression* can be determined by the constructions $(E, Symbols, Bases)$ to have the type τ, provided that it satisfies $premise_1$ and $premise_2$. So, these kinds of rules are used to ensure that expressions resolve to the type that is expected, helping the programmer catch errors.

4.3 General Methodology

The general approach to modeling C++'s behavior has been to decompose the process into small, understandable steps. Our presentation of the model will begin with a brief overview, then present some syntactic definitions, and then examine the various constructs employed more carefully.

As we said earlier, the type checking rules effectively describe an algorithm to process a C++ program. First, class declarations are processed, and information about those classes is stored in a construct called *Symbols*. This set will keep track of all the instance variables and methods of a class. In addition, it maintains a running list of methods that have been defined as virtual. The *Symbols* construct is used to store the type information that will be needed to bind method calls with method bodies.

Another construct generated is the *Bases* set, which keeps track of which classes were derived from which other classes. The type checker we have described uses this information to ensure that only valid subtyping is allowed.

E is the construct known as the environment, which stores the types of variables for use within the type-checker. In addition, the environment stores the complete-signatures of functions defined in the program.

Before we describe these sets in detail, we need to discuss a syntactic convention that will facilitate describing the language's behavior precisely.

4.4 Definitions and Notation

4.4.1 Signature Definitions

Methods within a class can share the same name, provided they have different argument types. C++ compilers use a process called "mangling" or "decorating" to generate an augmented function name that contains (argument) type information.

A method has an associated name, argument-signature, and complete-signature. A method-ID is just an elaboration of the complete signature that specifies what class a method was defined in.

In the following discussion, C refers to an arbitrary class.

1. The *argument-signature* of a method is the name of the method, decorated with the types of its arguments. M_x is the argument-signature of a method `M` that takes an argument of type `x`.

2. The *complete-signature* of a method is the name of the method, decorated with both the types of its arguments and its return type. $M_{x \to y}$ is the complete-signature of a method `M` that takes an argument of type `x` and returns an object of type `y`.

3. The *method-id* of a method is the complete-signature of the method, augmented with the name of the class the method was defined in.$M^C_{x \to y}$ is a method ID, and elaborates on the previous definition to specify that the method was defined in a class `C`.

The notion of overloading hinges on the fact that several different argument-signature may be associated with a given method name. We will present later a restriction which formalizes overloading.

4.5 A Primer on our Constructs

The *Symbols* construct associates important type-checking information with each class. *Symbols* is a semantic function, taking the name of a class C and returning an entry, C_{sym}.

C_{sym} is a triple composed of several constructs: C_{meth}, C_{vset}, and C_{iv}. The first records the methods visible to a class, the second records information about which methods are virtual, and C_{iv} keeps track of the instance variables declared for the class. Let's look at each of these in turn.

4.5.1 Method Visibility: C_{meth}

Each class has an associated set of method-IDs that are the methods visible to the class. These sets are used at type checking time to both ensure that a call is legal and bind the call to the appropriate method (if possible). Objects may only call methods in their methset.

We will notate the methset of a class C as C_{meth}. C_{meth} is comprised of the method-IDs of the methods visible in the class. For example, $C_{meth} = \{M^C_{\sigma \to \tau}, M^D_{x \to y}\}$ represents the *methset* for C with two visible methods, $M^C_{\sigma \to \tau}$ (defined in C), and $M^D_{x \to y}$ (defined in a class D). Keep in mind that the *methset* for a class contains the methods visible in the class, which might include methods inherited from some base class.

4.5.2 Virtual Methods: C_{vset}

Each class has an associated set of complete-signatures called a *vset*. The *vset* associated with a class specifies which methods will be considered virtual.

We will notate the *vset* of a class C by C_{vset}. For example, $C_{vset} = \{M_{p \to q},\ N_{x \to y}\}$ represents a *vset* with two signatures that signify virtual methods, $M_{p \to q}$, and $N_{x \to y}$. Note that these complete-signatures are not methods guaranteed to be visible in the class, but we will see that if they *are* visible, their entry in C_{vset} indicates that they are virtual. Also, notice that entries in the *vset* are complete-signatures, rather than method-IDs, because the *vset* is just used to determine which method signatures will be virtual.

4.5.3 Instance Variables: C_{iv}

Each class has an associated set of instance variables and their types, known as the *iv-set*. An *iv-set* is a semantic function that takes instance variable names in the domain, and returns their types in the range. We will notate the *iv-set* of a class C by C_{iv}.

For example, the *iv-set* of C might be $C_{iv} = \{x : \tau, y : \sigma\}$, representing two instance variables for class C: x with type τ, and y with type σ.

4.6 Restrictions

In order to accurately model the C++ method binding mechanisms, our model must include the restrictions that C++ imposes.

The first restriction is that within a class, two methods that share a name can not have the same argument types. This property reflects the fact that overloading of methods is based on argument-signatures.

The second restriction concerns the return types of virtual functions. In order to prevent run-time type errors, C++ restricts the return types of virtual functions. It is an error to declare a method that matches the argument-signature in the `vset`, yet has a different complete-signature. This requirement is to prevent a pointer to a base class invoking a method dynamically that returns a value of unexpected type. We summarize these formally as:

1. Given method definitions M_x^C and M_y^C, $x \neq y$.

2. If $M_{p \to q}^A$, and $M_{x \to y}^B \in C_{vset}$, and if $p = x$, then $q = y$.

4.7 Looking Forward

In the next chapter we have listed all the type-checking rules as a compact reference. The following chapter is a presentation and discussion of each of the rules. The reader is advised to skip to the "walkthrough" of the type checking rules when first reading them.

Chapter 5

Type Checking Rules for C++

5.1 Processing of Declarations

5.1.1 Program Parsing

$$program\ parsing \qquad \frac{\begin{array}{c}(\epsilon, \epsilon, CLASSDECS) \Rightarrow Symbols, Bases \\ (\epsilon, Symbols, Bases) \vdash DEFS : ok\end{array}}{PROGRAM \Rightarrow ok}$$

5.1.2 Class Declaration

$$class\ dec\ parsing \ \frac{\begin{array}{c}(Symbols, Bases, CLASSDEC) \Rightarrow Symbols', Bases' \\ (Symbols', Bases', OTHERS) \Rightarrow Symbols'', Bases''\end{array}}{(Symbols, Bases, CLASSDEC;\ OTHERS) \Rightarrow Symbols'', Bases''}$$

$$base\ class\ dec \ \frac{(\{\epsilon, \epsilon, \epsilon\}, METHS) \Rightarrow B_{sym}}{(Symbols, Bases, \texttt{class}\ B\{METHS\};) \Rightarrow Symbols[B \mapsto B_{sym}], Bases}$$

$$\textit{derived class dec}\ \frac{\begin{array}{c}(\{\epsilon, B_{vset}, B_{iv}\}, METHS) \Rightarrow D_{sym} \\ D'_{meth} = D_{meth} \cup \{M^{C}_{\alpha\to\beta} \in B_{meth} |\ \nexists \sigma, \gamma\ s.t.\ M^{D}_{\sigma\to\gamma} \in D_{meth}\} \\ D'_{sym} = D_{sym}[D_{meth} \mapsto D'_{meth}] \\ Symbols' = Symbols[D \mapsto D'_{sym}] \\ Bases' = Bases[D \mapsto Bases(B) \cup \{B\}] \\ \forall C \in Bases(D), C'_{meth} = C_{meth} \cup \{\check{C}_{D\to C}\} \\ C'_{sym} = C_{sym}[C_{meth} \mapsto C'_{meth}] \\ Symbols'' = Symbols[C \mapsto C'_{sym}]\end{array}}{\begin{array}{c}(Symbols, Bases, \texttt{class}\ D :\ \texttt{public}\ B\ \{METHS\};) \Rightarrow \\ Symbols', Bases'\end{array}}$$

Where B_{meth},D_{meth}, etc. are shorthand for $Symbols(B)_{meth}$,$Symbols(D)_{meth}$.

5.1.3 Class Component Declaration

$$\textit{component parsing}\quad \frac{\begin{array}{c}(B_{sym}, COMPONENT) \Rightarrow B'_{sym} \\ (B'_{sym}, OTHERS) \Rightarrow B''_{sym}\end{array}}{(B_{sym}, COMPONENT; OTHERS) \Rightarrow B''_{sym}}$$

$$\textit{meth dec}\quad \frac{if\ M_{\sigma\to x} \in B_{vset},\ then\ \tau = x}{(B_{sym}, \tau\ M(\sigma);) \Rightarrow B_{sym}[B_{meth} \mapsto B_{meth} \cup \{M^{B}_{\sigma\to\tau}\}]}$$

$$\textit{virtual meth dec}\ \frac{if\ M_{\sigma\to x} \in B_{vset},\ then\ \tau = x}{\begin{array}{c}(B_{sym}, \texttt{virtual}\ \tau\ M(\sigma);) \Rightarrow \\ B_{sym}[B_{meth} \mapsto B_{meth} \cup \{M^{B}_{\sigma\to\tau}\}, B_{vset} \mapsto B_{vset} \cup \{M_{\sigma\to\tau}\}]\end{array}}$$

$$\textit{iv dec}\quad \frac{}{(B_{sym}, \tau\ \texttt{x};) \Rightarrow B_{sym}[B_{iv} \mapsto B_{iv} \cup \{x : \tau\}]}$$

$$\textit{ctor}\quad \frac{}{(B_{sym}, B(\sigma)) \Rightarrow B_{sym}[B_{meth} \mapsto B_{meth} \cup \{\check{B}_{\sigma\to B}\}]}$$

5.2 Processing of Method and Function Bodies

5.2.1 Method and Function Definition

$$\textit{methdef parsing}\quad \frac{\begin{array}{c}(E, Symbols, Bases, METHDEF) \Rightarrow OK \\ (E, Symbols, Bases, OTHERS) \Rightarrow OK\end{array}}{(E, Symbols, Bases, METHDEF;\ OTHERS) \Rightarrow OK}$$

$$\textit{func def} \qquad \frac{(E \cup \{x : \sigma\}, Symbols, Bases) \vdash BODY : \tau}{(E, Symbols, \tau\ \mathtt{F}(\sigma\ \mathtt{x})\{BODY\}) \Rightarrow E \cup \{F_{\sigma \to \tau}\}, Symbols}$$

$$\textit{meth def} \qquad \frac{\begin{array}{c} B_{meth} \in Symbols(B) \\ M^{B}_{\sigma \to \tau} \in Symbols(B)_{meth} \\ (E \cup \{this : B, x : \sigma\}, Symbols) \vdash BODY : \tau \end{array}}{(E, Symbols, Bases, \tau\ B :: M(\sigma\ x)\{BODY\}) \Rightarrow OK}$$

$$\textit{virtual meth def} \ \frac{\begin{array}{c} B_{meth} \in Symbols(B) \\ M^{B}_{\sigma \to \tau} \in Symbols(B)_{meth} \\ (E \cup \{this : B, x : \sigma\}, Symbols) \vdash BODY : \tau \end{array}}{(E, Symbols, Bases, \mathtt{virtual}\ \tau\ B :: M(\sigma\ x)\{BODY\}) \Rightarrow OK}$$

$$\textit{ctor def} \qquad \frac{\begin{array}{c} B_{meth} \in Symbols(B) \\ \breve{M}^{B}_{\sigma \to B} \in Symbols(B)_{meth} \\ (E \cup \{this : B, x : \sigma\}, Symbols) \vdash BODY : OK \end{array}}{(E, Symbols, Bases, B :: B(\sigma\ \mathtt{x})\{BODY\}) \Rightarrow OK}$$

5.2.2 Method Body

$$\textit{meth body parsing} \ \frac{\begin{array}{c} (E, Symbols) \vdash CALL : \alpha\ for\ some\ \alpha \\ (E, Symbols) \vdash REST : \tau \end{array}}{(E, Symbols, Bases) \vdash CALL; REST : \tau}$$

$$\textit{return} \qquad \frac{(E, Symbols) \vdash N : \tau}{(E, Symbols, Bases) \vdash \mathtt{return}(N); : \tau}$$

$$\textit{object creation} \qquad \frac{(E \cup \{b : B\}, Symbols) \vdash REST : \tau}{(E, Symbols, Bases) \vdash B\ b;, REST : \tau}$$

5.2.3 Call Resolution and Annotation

Pointers

$$\textit{func} \qquad \frac{\begin{array}{c} (E, Symbols, Bases) \vdash val : \sigma * \\ F_{C* \to \tau} \in E,\ C \in Bases(\sigma) \end{array}}{(E, Symbols, Bases) \vdash F(val) : \tau}$$

Where for all other $F_{C*' \to \tau} \in E$, if $C' \in Bases(\sigma)$ then $C' \in Bases(C)$. If such an $F_{C* \to \tau}$ does not exist then $F(val)$ does not type check.

$$annotate: \ F_{C* \to \tau}(val)$$

$$direct \qquad \frac{\begin{array}{c}(E, Symbols, Bases) \vdash val : \sigma* \\ (E, Symbols, Bases) \vdash obj : B \\ \exists M^{A}_{C* \to \tau} \in Symbols(B)_{meth}, \ C \in Bases(\sigma)\end{array}}{(E, Symbols, Bases) \vdash obj.M(val) : \tau}$$

Where for all other $M^{A'}_{C*' \to \tau} \in Symbols(B)_{meth}$, if $C' \in Bases(\sigma)$ then $C' \in Bases(C)$. If such an $M^{A}_{C* \to \tau}$ does not exist in $Symbols(B)_{meth}$, then $M(val)$ does not type check.

$$annotate: \ M^{A}_{C* \to \tau}(val)$$

$$indirect \qquad \frac{\begin{array}{c}(E, Symbols, Bases) \vdash val : \sigma* \\ (E, Symbols, Bases) \vdash bp : B* \\ \exists M^{A}_{C* \to \tau} \in Symbols(B)_{meth}, \ C \in Bases(\sigma)\end{array}}{(E, Symbols, Bases) \vdash bp{-}{>}M(val) : \tau}$$

Where for all other $M^{A'}_{C*' \to \tau} \in Symbols(B)_{meth}$, if $C' \in Bases(\sigma)$ then $C' \in Bases(C)$. If such an $M^{A}_{C* \to \tau}$ does not exist in $Symbols(B)_{meth}$, then $bp{-}{>}M(val)$ does not type check.

$$if \ M_{C* \to \tau} \in B_{vset}, \ annotate: \ \hat{M}_{C* \to \tau}(val)$$

$$if \ M_{C* \to \tau} \notin B_{vset}, \ annotate: \ M^{A}_{C* \to \tau}(val)$$

Objects - Exact Match

$$func \qquad \frac{\begin{array}{c}(E, Symbols, Bases) \vdash val : X \\ F_{X \to Y} \in E \\ \check{X}_{X \to X} \in Symbols(X)_{meth}\end{array}}{(E, Symbols, Bases) \vdash F(val) : Y}$$

$$annotate: \ F_{X \to Y}(\check{X}_{X \to X}(new(X), val))$$

$$direct \qquad \frac{\begin{array}{c}(E, Symbols, Bases) \vdash val : X \\ (E, Symbols, Bases) \vdash obj : B \\ \exists C, \tau \ s.t. \ M^{C}_{X \to \tau} \in Symbols(B)_{meth} \\ \check{X}_{X \to X} \in Symbols(X)_{meth}\end{array}}{(E, Symbols, Bases) \vdash obj.M(val) : \tau}$$

$$annotate:\ M^{C}_{X\to\tau}(\check{X}_{X\to X}(new(X), val))$$

$$indirect \qquad \frac{\begin{array}{c}(E, Symbols, Bases) \vdash val : X \\ (E, Symbols, Bases) \vdash bp : B* \\ \exists C, \tau\ s.t.\ M^{C}_{X\to\tau} \in Symbols(B)_{meth} \\ \check{X}_{X\to X} \in Symbols(X)_{meth}\end{array}}{(E, Symbols, Bases) \vdash bp\text{->}M(val) : \tau}$$

$$if\ M_{X\to\tau} \in B_{vset},\ annotate:\ \hat{M}_{X\to\tau}(\check{X}_{X\to X}(new(X), val))$$

$$if\ M_{X\to\tau} \notin B_{vset},\ annotate:\ M^{C}_{X\to\tau}(\check{X}_{X\to X}(new(X), val))$$

Objects - With Conversion

$$func \qquad \frac{\begin{array}{c}(E, Symbols, Bases) \vdash val : \sigma \\ F_{\sigma\to\tau} \notin E \\ F_{C\to\tau} \in E \\ \check{C}_{\sigma\to C} \in Symbols(C)_{meth} \\ \check{\sigma}_{\sigma\to\sigma} \in Symbols(\sigma)_{meth}\end{array}}{(E, Symbols, Bases) \vdash F(val) : \tau}$$

Where if no $F_{C\to\tau}$ and $\check{\sigma}_{C\to\sigma}$ exist, $F(val)$ does not type check.

$$annotate:\ F_{C\to\tau}(\check{C}_{\sigma\to C}(newC), \check{\sigma}_{\sigma\to\sigma}(new(\sigma), val)))$$

$$direct \qquad \frac{\begin{array}{c}(E, Symbols, Bases) \vdash val : \sigma \\ (E, Symbols, Bases) \vdash obj : B \\ M_{\sigma\to\tau} \notin Symbols(B)_{meth} \\ M^{A}_{C\to\tau} \in Symbols(B)_{meth} \\ \check{C}_{\sigma\to C} \in Symbols(C)_{meth} \\ \check{\sigma}_{\sigma\to\sigma} \in Symbols(\sigma)_{meth}\end{array}}{(E, Symbols, Bases) \vdash obj.M(val) : \tau}$$

Where if no $M^{A}_{C\to\tau}$ and $\check{\sigma}_{C\to\sigma}$ exist, $obj.M(val)$ does not type check.

$$annotate:\ M^{A}_{C\to\tau}(\check{C}_{\sigma\to C}(new(C), \check{\sigma}_{\sigma\to\sigma}(new(\sigma), val)))$$

$$\textit{indirect} \qquad \frac{\begin{array}{c}(E, Symbols, Bases) \vdash val : \sigma \\ (E, Symbols, Bases) \vdash bp : B* \\ M_{\sigma\to\tau} \notin Symbols(B)_{meth} \\ M^A_{C\to\tau} \in Symbols(B)_{meth} \\ \check{C}_{\sigma\to C} \in Symbols(C)_{meth} \\ \check{\sigma}_{\sigma\to\sigma} \in Symbols(\sigma)_{meth}\end{array}}{(E, Symbols, Bases) \vdash bp{-}{>}M(val) : \tau}$$

Where if no $M^A_{C\to\tau}$ and $\check{\sigma}_{C\to\sigma}$ exist, $bp{-}{>}M(val)$ does not type check.
if $M_{C\to\tau} \in B_{vset}$, annotate:

$$\hat{M}_{C\to\tau}(\check{C}_{\sigma\to C}(new(C), \check{\sigma}_{\sigma\to\sigma}(new(\sigma), val)))$$

if $M_{C\to\tau} \notin B_{vset}$, annotate:

$$M^A_{C\to\tau}(\check{C}_{\sigma\to C}(new(C), \check{\sigma}_{\sigma\to\sigma}(new(\sigma), val)))$$

5.2.4 Object and Pointer Creation

$$\textit{new} \qquad \frac{B \in domain(Symbols)}{(E, Symbols, Bases) \vdash new(B) : B}$$

$$\textit{object creation} \qquad \frac{B \in domain(Symbols)}{(E, Symbols, Bases) \vdash B\ b : B}$$

$$\textit{ptr creation} \qquad \frac{B \in E}{(E, Symbols, Bases) \vdash B*\ b : B*}$$

$$\frac{\begin{array}{c}(E, Symbols, Bases) \vdash c : C \\ \check{B}_{C\to B} \in Symbols(B)_{meth}\end{array}}{(E, Symbols, Bases) \vdash B(c) : B}$$

$$annotate:\ \check{B}(new(B), c)$$

5.2.5 Assignment

obj assignment

$$\frac{\begin{array}{c}(E, Symbols, Bases) \vdash b : B \\ (E, Symbols, Bases) \vdash c : C \\ \breve{B}_{C \to B} \in Symbols(B)_{meth}\end{array}}{(E, Symbols, Bases) \vdash b = c : B}$$

$$annotate: \; b = \breve{B}_C(new(B), c)$$

(After this translation, = can be interpreted as bitwise object copy.)

ptr assignment

$$\frac{\begin{array}{c}(E, Symbols, Bases) \vdash bp : B* \\ (E, Symbols, Bases) \vdash dp : D* \\ B \in Bases(D)\end{array}}{(E, Symbols, Bases) \vdash bp = dp : D*}$$

Chapter 6

A "Walkthrough" of the Type Rules

Formal rules and semantics can be quite intimidating at first, but are much more accessible once one is familiar with the notation. We will first introduce the syntax for type rules, and then look at our C++ rules carefully.

6.1 The Walkthrough

6.1.1 Basic Program and Class Declaration Rules

The very first type checking rule has two premises, and shows how recursion is used in the type-checker:

$$\textit{program parsing} \qquad \frac{\begin{array}{c}(\epsilon, \epsilon, CLASSDECS) \Rightarrow Symbols, Bases \\ (\epsilon, Symbols, Bases) \vdash DEFS : ok\end{array}}{PROGRAM \Rightarrow ok}$$

This rule verifies that the entire program is free of type errors (the ones that C++ catches, anyway). A C++ program (as we define it) is composed of class declarations (*CLASSDECS*), followed by function and method definitions (*DEFS*). The first premise of this rule will recursively process the class declarations and build the *Symbols* and *Bases* constructs. The processing of *CLASSDECS* starts with *Symbols* and *Bases* constructs which initially start empty, notated by ϵ. The second premise will use those constructs to type check all the definitions (method and function bodies, including `main()`). Expressions are type checked with the environment, *Symbols*, and *Bases* constructs. Since only classes have been declared when we begin to process $DEFS$, the environment is initially ϵ. The conclusion of the rule, $PROGRAM \Rightarrow ok$ indicates that the entity $PROGRAM$ passes the type checker.

Let's look at the next rule:

$$\textit{class dec parsing}\ \frac{\begin{array}{c}(Symbols, Bases, CLASSDEC) \Rightarrow Symbols', Bases' \\ (Symbols', Bases', OTHERS) \Rightarrow Symbols'', Bases''\end{array}}{(Symbols, Bases, CLASSDEC;\ OTHERS) \Rightarrow Symbols'', Bases''}$$

This rule handles the parsing of a list of class declarations. The first premise handles the first class declaration, while the second premise handles the rest of the list recursively. Notice that the first premise starts with $Symbols$ and $Bases$, and produces the updated versions, $Symbols'$ and $Bases'$. This is because processing a class declaration results in updating those constructs to reflect the new information the type checker has just gathered. We then pass the primed versions into the next premise so that the new information can be employed during type checking of the rest of the class declarations. As a result, the conclusion produces the $Symbols''$ and $Bases''$ (that were the result of the recursive second premise) as the final $Symbols$ and $Bases$ that will be used during the definitions stage of the type checking rules.

The next rule handles the declaration of a class with no base classes:

$$\textit{base class dec}\ \frac{(\{\epsilon, \epsilon, \epsilon\}, METHS) \Rightarrow B_{sym}}{(Symbols, Bases, \texttt{class}\ B\{METHS\};) \Rightarrow Symbols[B \mapsto B_{sym}], Bases}$$

The premise, $(\{\epsilon, \epsilon, \epsilon\}, METHS) \Rightarrow B_{sym}$, produces an entry B_{sym} that the conclusion passes back in an updated $Symbols$. Note that $\{\epsilon, \epsilon, \epsilon\}$ is the initial, empty triple for B_{sym}. We will soon see how B_{sym} is generated, but for now just recall that it contains the *methset*,*vset* and *iv-set* of the class. Notice that since B does not have a base class, $Bases$ is unmodified.

The next rule handles the declaration of a derived class:

$$\textit{derived class dec}\ \frac{\begin{array}{c}(\{\epsilon, B_{vset}, B_{iv}\}, METHS) \Rightarrow D_{sym} \\ D'_{meth} = D_{meth} \cup \{M^{C}_{\alpha\to\beta} \in B_{meth} |\ \nexists \sigma, \gamma\ s.t.\ M^{D}_{\sigma\to\gamma} \in D_{meth}\} \\ D'_{sym} = D_{sym}[D_{meth} \mapsto D'_{meth}] \\ Symbols' = Symbols[D \mapsto D'_{sym}] \\ Bases' = Bases[D \mapsto Bases(B) \cup \{B\}] \\ \forall C \in Bases(D), C'_{meth} = C_{meth} \cup \{\check{C}_{D\to C}\} \\ C'_{sym} = C_{sym}[C_{meth} \mapsto C'_{meth}] \\ Symbols'' = Symbols[C \mapsto C'_{sym}]\end{array}}{\begin{array}{c}(Symbols, Bases, \texttt{class}\ D:\ \texttt{public}\ B\ \{METHS\};) \Rightarrow \\ Symbols'', Bases'\end{array}}$$

In this rule, B_{meth},D_{meth}, etc., are shorthand for $Symbols(B)_{meth}$,$Symbols(D)_{meth}$.

This rule is one of the largest in the entire type checker, but is actually relatively straightforward. The conclusion is of course an updated $Symbols$ and $Bases$, but *what* they are updated with is interesting.

The first premise, $(\{\epsilon, B_{vset}, B_{iv}\}, METHS) \Rightarrow D_{sym}$, doesn't make much sense at first, until we recognize that $(\{\epsilon, B_{vset}, B_{iv}\}$ is an initial D_{sym} that will be augmented with the information gathered by processing $METHS$. This initial D_{sym} starts out with an empty methset, D's parent class' *vset* and *iv-set*. What does this mean? It reflects the fact that a derived class' *vset* and *iv-sets* are supersets of the parent's sets. This allows a derived class to inherit all the instance variables of its' parent. In addition, the parent's vset, keeping track of which methods should be virtual, is also inherited. *This implies that once a method is declared virtual in a base class, it will always be virtual in all child classes, no matter how far removed!*

The second premise,

$$D'_{meth} = D_{meth} \cup \{M^{C}_{\alpha\to\beta} \in B_{meth} |\ \nexists \sigma, \gamma, C'\ s.t.\ M^{C'}_{\sigma\to\gamma} \in D_{meth}\}$$

is one of the trickiest in the entire type checking rules. However, it enunciates an oft-confused behavior of C++ precisely. A method in a derived class which has the same name as a method in a base class *hides* the parent's method. If the name is not reused, the method is inherited. This premise describes an update of D_{meth}, by including methods present in the base class, $M^{C}_{\alpha\to\beta} \in B_{meth}$, whose names are not reused in D: $\nexists \sigma, \gamma\ s.t.\ M^{D}_{\sigma\to\gamma} \in D_{meth}$. Notice that the types of the method in the derived class are not important, only the reuse of the name. Notice that the method-IDs of the base class methods specify that the method was defined in some class C. This just reflects the fact that the methods we are considering might, in turn, have been inherited from some earlier class, rather than defined in D.

The third premise, $D'_{sym} = D_{sym}[D_{meth} \mapsto D'_{meth}]$, updates D_{sym} with the new *methset*. Then, in the next clause, $Symbols' = Symbols[D \mapsto D'_{sym}]$ updates $Symbols$ with D'_{sym}. The fifth premise, $Bases' = Bases[D \mapsto Bases(B) \cup \{B\}]$ updates $Bases$ with an entry for class D. This entry for D keeps track of all of D's base classes, direct or indirect, and thus adds class B and all of B's bases, $Bases(B)$. $Bases$ is used to verify the legality of conversions between derived objects to an object of one of their base classes. The next clause will declare the necessary constructors to carry out those conversions.

The clause $\forall C \in Bases(D), C'_{meth} = C_{meth} \cup \{\breve{C}_{D\to C}\}$ updates the methset of a class C with the constructors needed to convert objects of class C into objects of all of C's base classes. C++ considers derived classes to be subtypes of their base classes, and thus needs these constructors to act as implicit conversions to allow for this subtyping. (We will later see that the semantics of such constructors is trivial, because only the type of the object is modified, as it is not necessary to throw any of the extra instance variables in the object (which are from the derived class) away.) Recall that the notation $\breve{C}_{D\to C}$ simply means a constructor of class C that takes an object of class D as its argument.

The last two premises update C_{sym} to reflect the new C_{meth}, and then update $Symbols'$ to reflect the new C_{sym}.

The conclusion of the rule,

$$(Symbols, Bases, \texttt{class}\ D : \texttt{public}\ B\ \{METHS\};) \Rightarrow Symbols', Bases'$$

simply passes out the information gathered in the augmented *Symbols'* and *Bases'*.

6.1.2 Class Component Declaration

The next group of rules handle the declaration of the components of a class: methods, instance variables, and constructors.

The first of the rules is a general parsing rule, just like the *CLASSDEC* parsing we saw earlier:

$$\textit{component parsing} \quad \frac{(B_{sym}, COMPONENT) \Rightarrow B'_{sym} \qquad (B'_{sym}, OTHERS) \Rightarrow B''_{sym}}{(B_{sym}, COMPONENT; OTHERS) \Rightarrow B''_{sym}}$$

The next rules handle the declaration of `virtual` and non-virtual methods, and accumulates the results from the *CLASSDEC*s. In both rules, the first premise is a check to ensure that if the method we are declaring is virtual (in the *vset*), then the return type must match the earlier declaration. Assuming this is so, the conclusion updates B_{meth} to include the method-id of the method just declared:

$$\textit{meth dec} \quad \frac{if\ M_{\sigma\to x} \in B_{vset},\ then\ \tau = x}{(B_{sym}, \tau\ M(\sigma);) \Rightarrow B_{sym}[B_{meth} \mapsto B_{meth} \cup \{M^{B}_{\sigma\to\tau}\}]}$$

In the second rule, we again check to see if the virtual method we are considering differs in return type from an earlier declaration. If not, B_{meth} is updated with the method-id, and B_{vset} is updated to include the complete-signature of the new method. (If the method's complete-signature were previously used, it would match, and hence the update to B_{vset} has no effect.)

Note that this rule processes method declarations whether virtual or not, and ensures that virtual functions meet the requirement that the return type matches the complete-signature in the *vset*. This is the primary reason why the *vset* stores complete-signatures, rather than argument-signatures: to ensure that the return type of a virtual function does not change in a derived class.

$$\textit{virtual meth dec} \quad \frac{if\ M_{\sigma\to x} \in B_{vset},\ then\ \tau = x}{\begin{array}{c}(B_{sym}, \texttt{virtual}\ \tau\ M(\sigma);) \Rightarrow \\ B_{sym}[B_{meth} \mapsto B_{meth} \cup \{M^{B}_{\sigma\to\tau}\}, B_{vset} \mapsto B_{vset} \cup \{M_{\sigma\to\tau}\}]\end{array}}$$

The next two rules are straightforward; the first updating B_{iv} with a new instance variable, and the second updating B_{meth} with a constructor declaration:

$$\textit{iv dec} \quad \frac{}{(B_{sym}, \tau\ \texttt{x};) \Rightarrow B_{sym}[B_{iv} \mapsto B_{iv} \cup \{x : \tau\}]}$$

$$\textit{ctor} \quad \frac{}{(B_{sym}, B(\sigma)) \Rightarrow B_{sym}[B_{meth} \mapsto B_{meth} \cup \{\check{B}_{\sigma\to B}\}]}$$

6.1.3 Processing of Method Definitions

The next rules handle the type checking of method definitions, which are written (syntactically) outside of class declarations in the program, and specify the actual method bodies.

The first rule is for the most part another straightforward parsing tool, except for the addition of E:

$$methdef\ parsing \quad \frac{\begin{array}{c}(E, Symbols, Bases, METHDEF) \Rightarrow OK \\ (E, Symbols, Bases, OTHERS) \Rightarrow OK\end{array}}{(E, Symbols, Bases, METHDEF;\ OTHERS) \Rightarrow OK}$$

The function definition rule updates E with its complete-signature as follows:

$$func\ def \quad \frac{(E \cup \{x : \sigma\}, Symbols, Bases) \vdash BODY : \tau}{(E, Symbols, \tau\ \mathtt{F}(\sigma\ \mathtt{x})\{BODY\}) \Rightarrow E \cup \{F_{\sigma \to \tau}\}, Symbols}$$

The next rule type checks a non-virtual method definition:

$$\frac{\begin{array}{c}B_{meth} \in Symbols(B) \\ M^{B}_{\sigma \to \tau} \in Symbols(B)_{meth} \\ (E \cup \{this : B, x : \sigma\}, Symbols) \vdash BODY : \tau\end{array}}{(E, Symbols, Bases, \tau\ B :: M(\sigma\ x)\{BODY\}) \Rightarrow OK}$$

First B_{meth} is found, and then the declaration of the method is looked for in B_{meth}. The third premise, $(E \cup \{this : B, x : \sigma\}, Symbols) \vdash BODY : \tau$, type checks the body of the method and verifies that the return type is indeed τ, and hence returns *ok*. Notice that the environment used to evaluate the body has been augmented with the additional information of the types of the formal parameters. The implicit `this` argument, the pointer to the object itself, has its own type. The second parameter has whatever type the function was declared (and defined) with.

The virtual version of this rule operates identically:

$$\frac{\begin{array}{c}B_{meth} \in Symbols(B) \\ M^{B}_{\sigma \to \tau} \in Symbols(B)_{meth} \\ (E \cup \{this : B, x : \sigma\}, Symbols) \vdash BODY : \tau\end{array}}{(E, Symbols, Bases, \mathtt{virtual}\ \tau\ B :: M(\sigma\ x)\{BODY\}) \Rightarrow OK}$$

The constructor definition rule is almost identical:

$$ctor\ def \quad \frac{\begin{array}{c}B_{meth} \in Symbols(B) \\ \breve{M}^{B}_{\sigma \to B} \in Symbols(B)_{meth} \\ (E \cup \{this : B, x : \sigma\}, Symbols) \vdash BODY : OK\end{array}}{(E, Symbols, Bases, B :: B(\sigma\ \mathtt{x})\{BODY\}) \Rightarrow OK}$$

Notice that the evaluation of the body of the constructor returns OK, rather than a type. This is because constructors are not allowed to return a value (although for our purposes, we have them *implicitly* return the object they constructed).

6.1.4 Processing of Method and Function Bodies

Method and function bodies need to be type checked to insure that they don't contain any illegal calls. The first rule in this part takes care of sequences of statements in the body. Notice, however, that the type of a method body is the type of the last (unrolling the recursion) expression. This type will be the type of the entire method body, allowing return statements to operate as expected.

$$meth\ body\ parsing \quad \frac{\begin{array}{c}(E, Symbols) \vdash CALL : \alpha\ for\ some\ \alpha \\ (E, Symbols) \vdash REST : \tau\end{array}}{(E, Symbols, Bases) \vdash CALL; REST : \tau}$$

The type checking rule for a return statement is trivial:

$$return \qquad \frac{(E, Symbols) \vdash N : \tau}{(E, Symbols, Bases) \vdash \mathtt{return}(N);: \tau}$$

The type checking rule for an object creation statement is straightforward, simply updating E to record the type of the new object. Because the type checking rules are used recursively, we achieve scoping behavior as expected.

$$object\ creation \qquad \frac{(E \cup \{b : B\}, Symbols) \vdash REST : \tau}{(E, Symbols, Bases) \vdash B\ b;, REST : \tau}$$

6.1.5 Processing of Method and Function Calls

When a function or a message is applied or sent, the compiler has to determine precisely which method is to be invoked. In languages such as Pascal, where there is no overloading or automatic conversions of types, the name of the function is sufficient to find the appropriate body. But C++ allows overloading (multiple functions or methods sharing the same name but with different types), inheritance (methods may be inherited from a base class), and dynamic binding (where the actual method invoked is determined by the type of an object at run time). These features would make the binding issue confusing enough, but in addition C++ supports implicit conversions from one type to another. These conversions are performed by constructors, meaning that constructor calls (and the allocation of temporary objects) are carried out by the compiler in places where they are not explicitly present in the source code.

We have determined a semantics for C++'s method binding mechanism. The mechanism is somewhat simpler than the actual algorithm described in the ARM because we have limited the domain of the language we accept to avoid some of the more contorted "features" of the mechanism. We allow for implicit conversions from a pointer of a derived class to a pointer of a base class. We allow for conversions of an object of a derived class to an object of a base class, through conversion constructors. We allow for the conversion of an object of any class to an object of another class if the appropriate conversion constructor was defined. However, we have simplified the mechanism by presuming that there is a unique method to call after we have considered all the possible conversions. Our type checking rules simply pick one that matches, rather than guaranteeing a unique one. We feel that this is a necessary compromise to make the type checking rules human-readable, as the added complexity of precedence between conversions (i.e.: built-in over user-defined) is not worth discussing.

The rules are divided into three subsections: calls with pointers as arguments, calls with objects as arguments for which an exact match is found, and calls with objects as arguments for which a conversion must be found. Within each of these three categories, there is a rule for function calls, a rule for direct calls (`object.M(x)`), and a rule for indirect calls (`pointer->M(x)`). This gives nine rules total:

Function pointer argument	Method (Direct) pointer argument	Method (Indirect) pointer argument
Function object arg. (exact)	Method (Direct) object arg. (exact)	Method (Indirect) object arg. (exact)
Function obj. arg. (conversion)	Method (Direct) obj. arg. (conversion)	Method (Indirect) obj. arg. (conversion)

First, let us discuss the notion of a pointer's static and dynamic types. The static type of a pointer is the type of the class it was declared to point to. The dynamic type of a pointer is the type of the object it is actually pointing to at run time. Note that the dynamic type can be a derived class of the static type, and hence a subtype.

Also, be aware that C++ makes decisions on visibility and virtuality by the *Symbols* entry for the static type of a pointer, not the dynamic type. For this reason, the determination of the "virtuality" of a method depends on the presence of a signature in B_{vset}.

Function Call Resolution

When the type checker reaches a function or a method call, the name of the function is specified in the source code, along with the receiver of the message in the case of an object. For a function, there could be a collection of functions with that name with various argument types (i.e.: it is overloaded). If one of the functions happens to take the same argument type as the argument in the source code, we have found the correct function to use. If the argument was a pointer, we simply pass it in. If the argument was an object, we annotate with the copy constructor. This annotation modifies the source code that the semantics will receive. When the semantics process this call, the argument to the function will be the output of the copy constructor. We will see this in a moment when we examine the rules themselves.

When an exact match is not found, the type checking rules consider all the functions with that name, and examines their argument types. Then the type checker looks for a conversion constructor from the argument the function was actually invoked with to one of the argument types that the function was defined with. If the constructor is found, then the conversion is annotated in.

Method Call Resolution

When the type checker considers a method, the general behavior is essentially the same. Instead of looking for a function in E, however, the type checker determines the type of the receiver of the message, and looks in that class' *methset*. Notice that the type of the receiver is determined statically, so in the case of the pointer, the actual type of the object–the dynamic type–might be different at run time, namely some derived class. If the argument is a pointer we look for a method that takes that pointer's type or a base class of that pointer's type. If the argument is an object, we first check to see if there is an exact match, i.e.: a method that takes the actual argument's type. If an exact match is found, we chose that method, annotating with a copy constructor.

If an exact match is not found, we first find all the methods that share the name in the *methset* of the receiver's type. Then, we look for conversions (constructors) from the actual argument type to any of the argument types of these methods. Assuming such a conversion is found, the call is annotated with that conversion.

These rules are hard to understand without looking at them, but hard to look at without some introduction. Also, some fine points have been glanced over in this introduction. Let's consider each one more carefully now.

Pointer Arguments

$$func \qquad \frac{\begin{array}{c}(E, Symbols, Bases) \vdash val : \sigma* \\ F_{C*\to\tau} \in E,\ C \in Bases(\sigma)\end{array}}{(E, Symbols, Bases) \vdash F(val) : \tau}$$

Where for all other $F_{C*'\to\tau} \in E$, if $C' \in Bases(\sigma)$ then $C' \in Bases(C)$. If such an $F_{C*\to\tau}$ does not exist then $F(val)$ does not type check.

$$annotate: \ F_{C*\to\tau}(val)$$

This rule handles the binding of a function call with a pointer argument type to the actual function to invoke. The first step is to determine that the argument type is indeed a pointer with static type σ. Now, one or more functions could be found in E with the name F; the type checker will determine which one will be called.

Notice that when we build the *Bases* construct for a class, we include the class itself, i.e. $\sigma \in Bases(\sigma)$. The requirement that follows the rule ensures that our type checker behaves as C++ is specified to behave. The ARM says (13.2): "...if `B` is publicly derived directly from `A`, ... [and] if `C` is publicly derived directly or indirectly from `B`, converting a `C*` to `B*` is better than converting to `A*`."

This means that if a conversion from a pointer to a derived class to a pointer to a base class must be made, the closer to it in the class hierarchy the better. In other words, the function whose argument is the "most derived" in a class hierarchy takes precedence over functions that take parent classes as arguments. In this case, if a function were present taking an argument of type $\sigma *$, it would be the most derived base of σ, and therefore executed. Recalling that σ is in $Bases(\sigma)$, and all the other argument types would be bases of σ, we can see that it will be selected. So, the exact match case is handled properly. But what if there were no function taking a $\sigma *$ as an argument? This system should select a function which takes an argument whose type was a base class of σ, namely C. Again, the requirement that follows the rule ensures that C is the most derived argument type in the class hierarchy (recall that all other C' were in $Bases(C)$).

Notice the *annotation* that follows the rule. As mentioned earlier, "annotation" is the syntactic replacement of source code by an annotated version. This source code is then handed off to the second phase of compilation, the semantics.

In this rule, we annotate the function call with:

$$annotate: \ F_{C* \to \tau}(val)$$

By annotating the complete-signature of the function into the source code, we resolve the overloaded function once and for all (at compile time). This kind of annotation will be used throughout the various kinds of function and method application. The type checking phase of compilation can statically resolve almost all method calls (with the exception of virtual methods, which we will explain shortly). This is a nice feature of C++, although oft-misunderstood.

The next rule handles direct calls of methods with pointer arguments:

$$direct \qquad \frac{\begin{array}{c}(E, Symbols, Bases) \vdash val : \sigma * \\ (E, Symbols, Bases) \vdash obj : B \\ \exists M^{A}_{C* \to \tau} \in Symbols(B)_{meth}, \ C \in Bases(\sigma)\end{array}}{(E, Symbols, Bases) \vdash obj.M(val) : \tau}$$

Where for all other $M^{A'}_{C*' \to \tau} \in Symbols(B)_{meth}$, if $C' \in Bases(\sigma)$ then $C' \in Bases(C)$. If such an $M^{A}_{C* \to \tau}$ does not exist in $Symbols(B)_{meth}$, then $M(val)$ does not type check.

$$annotate: \; M^{A}_{C*\to\tau}(val)$$

The first premise resolves the type of the argument to be $\sigma*$, a pointer to a σ. The second premise resolves the type of the object to be B. Since we are dealing with a *direct* call, the object receiving the message is specified, so we know its type.

The third premise,

$$\exists M^{A}_{C*\to\tau} \in Symbols(B)_{meth}, \; C \in Bases(\sigma)$$

finds a method in B_{meth} with an argument type that is a base class of σ (or σ itself). In B_{meth} we find the method-ID of the method, which includes both the class it was defined in and the return type. The conclusion returns this return type.

The requirement that follows the rule again ensures that we select the method with the most-derived base of σ. Notice that since $\sigma \in Bases(\sigma)$, by definition of *Bases*, the exact match case (a method taking an argument of type σ) is handled properly.

Let's look at the annotation of the call:

$$annotate: \; M^{A}_{C*\to\tau}(val)$$

The annotation of this method call passes the complete method-ID to the semantics. Notice that no conversion of any sort is done to the argument (i.e. $\sigma*$ to $C*$) because it is just a pointer: in the body of the method it will simply be treated as a C pointer statically. In fact, observe that we were checking the static type of the argument all along, not the dynamic type, which might be an object of some class derived from σ. The static type of the argument is used to resolve the overloaded method. We will see this repeatedly; it is very important: C++ resolves method calls statically, with the singular exception of virtual methods.

The next rule handles indirect calls, i.e. `bp->M(x)`:

$$indirect \qquad \frac{\begin{array}{c}(E, Symbols, Bases) \vdash val : \sigma* \\ (E, Symbols, Bases) \vdash bp : B* \\ \exists M^{A}_{C*\to\tau} \in Symbols(B)_{meth}, \; C \in Bases(\sigma)\end{array}}{(E, Symbols, Bases) \vdash bp\text{->}M(val) : \tau}$$

Where for all other $M^{A'}_{C*'\to\tau} \in Symbols(B)_{meth}$, if $C' \in Bases(\sigma)$ then $C' \in Bases(C)$. If such an $M^{A}_{C*\to\tau}$ does not exist in $Symbols(B)_{meth}$, then $bp\text{->} M(val)$ does not type check.

$$if \; M_{C*\to\tau} \in B_{vset}, \; annotate: \; \hat{M}_{C*\to\tau}(val)$$

$$if\ M_{C*\to\tau} \notin B_{vset},\ annotate:\ M^A_{C*\to\tau}(val)$$

This rule is functionally identical to the previous rule, for direct calls, except for the annotation. C++ resolves method calls statically, so the static type of the receiver of the message is used, in this case $B*$. Indirect calls, however, can result in the calling of a virtual method, and hence dynamic binding to get the right method at run time. Interestingly, it is the annotation that allows this to operate correctly.

If the complete-signature of the method found to call is in B_{vset}, then the call is virtual, and is annotated in a way that the semantics will recognize to bind dynamically:

$$if\ M_{C*\to\tau} \in B_{vset},\ annotate:\ \hat{M}_{C*\to\tau}(val)$$

On the other hand, if the complete-signature is not in B_{vset}, the call is non-virtual, and has been resolved statically:

$$if\ M_{C*\to\tau} \notin B_{vset},\ annotate:\ M^A_{C*\to\tau}(val)$$

The implication of this annotation algorithm is that the virtuality of methods (whether the dynamic binding mechanism will be used) is decided by the *static type of the receiver*, in this case B, not by the actual type of the object that *bp* points to. This confuses many beginning C++ users, and is important to understand clearly.

Object Arguments, with an Exact Match

Now we will examine the rules for function and method invocations with objects as arguments, when an exact match is found for actual and formal argument types. These rules are relatively straightforward, but because of C++'s call by value nature, include some annotation that can be surprising.

The first rule is for the resolution of a function call with an object argument:

$$func \qquad \frac{\begin{array}{c}(E, Symbols, Bases) \vdash val : X \\ F_{X\to Y} \in E \\ \check{X}_{X\to X} \in Symbols(X)_{meth}\end{array}}{(E, Symbols, Bases) \vdash F(val) : Y}$$

$$annotate:\ F_{X\to Y}(\check{X}_{X\to X}(new(X), val))$$

The first premise, $(E, Symbols, Bases) \vdash val : X$, resolves the type of the argument to be X. The second premise, $F_{X\to Y} \in E$, finds the function $F_{X\to Y}$ in the environment whose formal argument type matches the actual argument. The next premise, $\check{X}_{X\to X} \in Symbols(X)_{meth}$, verifies the existence of a copy constructor (taking an X and returning an X). Since C++ is call by value, a *copy* of the argument is actually substituted for the formal parameter in the function body. Notice that the return type of the function is returned in the conclusion.

Now look at the annotation:

$$annotate: \ F_{X\to Y}(\check{X}_{X\to X}(new(X), val))$$

The copy constructor, $\check{X}_{X\to X}$ was inserted, with a $new(X)$ as the object to construct, and val as the object to construct it with. In other words, we inserted a constructor to make a copy of val, and generate a new object to pass in to the function invoked.

The next rule handles method invocations with an object parameter where an exact match is found:

$$direct \qquad \frac{\begin{array}{c}(E, Symbols, Bases) \vdash val : X \\ (E, Symbols, Bases) \vdash obj : B \\ \exists C, \tau \ s.t. \ M^C_{X\to\tau} \in Symbols(B)_{meth} \\ \check{X}_{X\to X} \in Symbols(X)_{meth}\end{array}}{(E, Symbols, Bases) \vdash obj.M(val) : \tau}$$

$$annotate: \ M^C_{X\to\tau}(\check{X}_{X\to X}(new(X), val))$$

The first premise determines the type of the argument to be X, while the second premise determines the type of the receiving object to be B. The next premise, $\exists C, \tau \ s.t. \ M^C_{X\to\tau} \in Symbols(B)_{meth}$, finds a method whose argument is of type X. The $\exists C \ldots$ business is because the method we find might have been inherited into class B, and defined in some earlier class C (hence $M^C \ldots$).

The next premise, $\check{X}_{X\to X} \in Symbols(X)_{meth}$, verifies that there is a copy constructor for class X. We will see it used in the annotation:

$$annotate: \ M^C_{X\to\tau}(\check{X}_{X\to X}(new(X), val))$$

to copy val in to the method body (call by value).

The next rule handles indirect method calls with an object argument for which an exact match is found. The premises are almost identical to the previous direct invocation rule:

$$indirect \qquad \frac{\begin{array}{c}(E, Symbols, Bases) \vdash val : X \\ (E, Symbols, Bases) \vdash bp : B* \\ \exists C, \tau \ s.t. \ M^C_{X\to\tau} \in Symbols(B)_{meth} \\ \check{X}_{X\to X} \in Symbols(X)_{meth}\end{array}}{(E, Symbols, Bases) \vdash bp\text{->}M(val) : \tau}$$

$$if \ M_{X\to\tau} \in B_{vset}, \ annotate: \ \hat{M}_{X\to\tau}(\check{X}_{X\to X}(new(X), val))$$

$$if\ M_{X\to\tau} \notin B_{vset},\ annotate:\ M_X^C(\check{X}_{X\to X}(new(X), val))$$

The first premise establishes the type of the argument to be X, and the second premise establishes the static type of the receiver to be $B*$. Recalling that C++ does most of the method resolution statically, we see that the third premise and fourth premises (identical to the previous rule) find a method in B_{meth} that matches exactly and verify the existence of a copy constructor for class X, respectively.

The annotation algorithm checks to see whether or not the method found is in B_{vset}. Notice that B is the static type of the pointer, and the actual object receiving the message might be of some class derived from B. Again, C++ decides on method virtuality statically, and hence looks in B_{vset}. If the complete-signature $M_{X\to\tau}$ is found in the *vset*, the method will be virtual:

$$if\ M_{X\to\tau} \in B_{vset},\ annotate:\ \hat{M}_{X\to\tau}(\check{X}_{X\to X}(new(X), val))$$

The copy constructor is inserted to take care of C++'s call by value behavior. On the other hand, if the complete-signature were not found in B_{meth}, it is not considered virtual and is bound statically:

$$if\ M_{X\to\tau} \notin B_{vset},\ annotate:\ M_X^C(\check{X}_{X\to X}(new(X), val))$$

Object Arguments, with a Conversion

The next group of rules is somewhat more complex, as an exact match between formal and actual argument types is not found. A conversion (constructor) must be used to convert the argument into a type that a function can be found for. Let's look at the rule for functions first.

$$func \qquad \frac{\begin{array}{c}(E, Symbols, Bases) \vdash val : \sigma \\ F_{\sigma\to\tau} \notin E \\ F_{C\to\tau} \in E \\ \check{C}_{\sigma\to C} \in Symbols(C)_{meth} \\ \check{\sigma}_{\sigma\to\sigma} \in Symbols(\sigma)_{meth}\end{array}}{(E, Symbols, Bases) \vdash F(val) : \tau}$$

Where if no $F_{C\to\tau}$ and $\check{\sigma}_{C\to\sigma}$ exist, $F(val)$ does not type check.

$$annotate:\ F_{C\to\tau}(\check{C}_{\sigma\to C}(newC), \check{\sigma}_{\sigma\to\sigma}(new(\sigma), val)))$$

This rule handles function calls with an object argument that requires a conversion. The first premise establishes the type of the argument to be σ. The second premise, $F_{\sigma\to\tau} \notin E$, verifies that there is no function that matches exactly, meaning a conversion will have to be used.

The next premise, $F_{C \to \tau} \in E$, establishes that there is a function $F_{C \to \tau}$. The following premise,

$$\breve{C}_{\sigma \to C} \in Symbols(C)_{meth}$$

establishes the existence of a (conversion) constructor for class C that takes an argument of class σ. This conversion constructor will be used to generate a new C object to be passed into the body of $F_{C \to \tau}$. This is the one level of conversion that C++ allows, as described in the ARM, section 13.2. Notice that when we defined classes, we included constructors that converted in from an object to any of its base classes. Because of those constructors, functions expecting a base class object can be passed a derived class object, and this rule will handle inserting the constructor to convert from the derived object to the base class.

This algorithm, however, is a simplification of the algorithm presented in the ARM in one very important respect. In the ARM, there exist various subtle differences of precedence of user-defined and the various built-in conversions. In our model, we presume that there is a *unique* conversion that yields an object some function was defined for.

The final premise,

$$\breve{\sigma}_{\sigma \to \sigma} \in Symbols(\sigma)_{meth}$$

finds the copy constructor for class σ, which will be used to *copy* in the the (object) argument into the conversion constructor for C.

Notice the annotation:

$$annotate: \; F_{C \to \tau}(\breve{C}_{\sigma \to C}(newC), \breve{\sigma}_{\sigma \to \sigma}(new(\sigma), val)))$$

This annotation instructs the semantics to first make a new copy of *val* with the copy constructor for class σ, which is then passed in to the conversion constructor from σ to C. The copy constructor is inserted because all calls in C++, whether function, method, or constructor, have call-by-value semantics. Recall that whenever a function is called with an object parameter, the copy constructor is automatically used to copy in the argument. This means that the copy constructor must be inserted to pass the σ argument to the $\breve{C}_{\sigma \to C}$ conversion constructor! The conversion constructor is then called to create a new C object to be passed into the function.

Let us consider the rule for direct method calls with an object argument that requires a conversion:

$$direct \qquad \frac{\begin{array}{c} (E, Symbols, Bases) \vdash val : \sigma \\ (E, Symbols, Bases) \vdash obj : B \\ M_{\sigma \to \tau} \notin Symbols(B)_{meth} \\ M^A_{C \to \tau} \in Symbols(B)_{meth} \\ \breve{C}_{\sigma \to C} \in Symbols(C)_{meth} \\ \breve{\sigma}_{\sigma \to \sigma} \in Symbols(\sigma)_{meth} \end{array}}{(E, Symbols, Bases) \vdash obj.M(val) : \tau}$$

Where if no $M^A_{C\to\tau}$ and $\breve{\sigma}_{C\to\sigma}$ exist, $obj.M(val)$ does not type check.

$$annotate: \; M^A_{C\to\tau}(\breve{C}_{\sigma\to C}(new(C), \breve{\sigma}_{\sigma\to\sigma}(new(\sigma), val)))$$

The first two premises resolve the type of the argument to be σ, and the type of the receiver object to be B. Since the receiver was a specified object in the call, its static and dynamic types are the same.

The next premise, $M_{\sigma\to\tau} \notin Symbols(B)_{meth}$, verifies that there is no method whose formal parameter type matches the actual parameter's type exactly.

The next premise, $M^A_{C\to\tau} \in Symbols(B)_{meth}$, looks up a method that takes an argument of class C. The following premise, $\breve{C}_{\sigma\to C} \in Symbols(C)_{meth}$, looks up the constructor that converts from an object of class σ to an object of class C. The final premise, $\breve{\sigma}_{\sigma\to\sigma} \in Symbols(\sigma)_{meth}$, looks up the copy constructor for class σ, which will be used to pass *val* to the conversion constructor $\breve{C}_{\sigma\to C}$, as in the previous rule.

The requirement after the rule ensures that a method and conversion constructor can be found such that the call is legal C++.

The annotation in this rule is again somewhat complex, but quite similar to the preceding function rule:

$$annotate: \; M^A_{C\to\tau}(\breve{C}_{\sigma\to C}(new(C), \breve{\sigma}_{\sigma\to\sigma}(new(\sigma), val)))$$

A copy constructor copies *val* into a new σ object to be passed to the conversion constructor, which generates a new C object to be used in the body of the method.

It is surprising that this one method call results in three method calls being executed. It is surprises like this one that hurt the performance of some C++ programs not coded carefully.

The next (and final) method resolution rule is for an indirect method call with an object argument for which there is no exact match. The premises of the rule are essentially identical to the previous rule, but the annotation is different:

$$indirect \qquad \frac{\begin{array}{c}(E, Symbols, Bases) \vdash val : \sigma \\ (E, Symbols, Bases) \vdash bp : B* \\ M_{\sigma\to\tau} \notin Symbols(B)_{meth} \\ M^A_{C\to\tau} \in Symbols(B)_{meth} \\ \breve{C}_{\sigma\to C} \in Symbols(C)_{meth} \\ \breve{\sigma}_{\sigma\to\sigma} \in Symbols(\sigma)_{meth}\end{array}}{(E, Symbols, Bases) \vdash bp\text{->}M(val) : \tau}$$

Where if no $M^A_{C\to\tau}$ and $\breve{\sigma}_{C\to\sigma}$ exist, $bp\text{->}M(val)$ does not type check.
if $M_{C\to\tau} \in B_{vset}$, annotate:

$$\hat{M}_{C\to\tau}(\breve{C}_{\sigma\to C}(new(C), \breve{\sigma}_{\sigma\to\sigma}(new(\sigma), val)))$$

if $M_{C\rightarrow\tau} \notin B_{vset}$, annotate:

$$M^{A}_{C\rightarrow\tau}(\check{C}_{\sigma\rightarrow C}(new(C), \breve{\sigma}_{\sigma\rightarrow\sigma}(new(\sigma), val)))$$

Except for the second premise resolving the receiver's type to be a pointer to a B object, the premises are identical to the previous rule. Since the call is indirect, however, the annotation is different to allow for dynamic binding.

The annotation algorithm checks to see if the complete-signature $M_{C\rightarrow\tau}$ is in B_{vset}. If so, the method is virtual and the semantics need to be informed to bind it dynamically:

$$\hat{M}_{C\rightarrow\tau}(\check{C}_{\sigma\rightarrow C}(new(C), \breve{\sigma}_{\sigma\rightarrow\sigma}(new(\sigma), val)))$$

Note that the copy constructor for σ is inserted inside to implement pass by value to the C (conversion) constructor.

On the other hand, if the method is determined to be non-virtual, then it is annotated as in the direct method call rule:

$$M^{A}_{C\rightarrow\tau}(\check{C}_{\sigma\rightarrow C}(new(C), \breve{\sigma}_{\sigma\rightarrow\sigma}(new(\sigma), val)))$$

Again, notice that the σ copy constructor is inserted as the argument to the c_j constructor.

In summary, these rules are quite complex, although it is a product of the complexity of the language itself, rather than an artificial complexity from the type checking rules we have devised. It is critical to notice that almost all binding information about method and function calls is determined statically, with the exception of a dynamic look-up for a particular signature at run-time for virtual methods. C++ can resolve all this statically because the language was designed to locate information about class visibility and virtuality in the static type of a pointer to an object, rather than an object itself. In this manner, since at compile time the static type of a pointer is known, all the needed method resolution information can be had without waiting for a run-time lookup. By the time the program is actually fed to the semantics to be executed, overloading and inheritance are all "gone" (computed away at compile time), making for efficient run time execution.

6.1.6 Object Creation

The type checking rules for object creation are quite straightforward. The only premise is that the type actually has already been declared in *Symbols*. If so, object creation statements simply return the type of the object just created.

$$new \qquad \frac{B \in domain(Symbols)}{(E, Symbols, Bases) \vdash new(B) : B}$$

object creation $$\frac{B \in domain(Symbols)}{(E, Symbols, Bases) \vdash B\ b : B}$$

ptr creation $$\frac{B \in E}{(E, Symbols, Bases) \vdash B*\ b : B*}$$

6.1.7 Constructor Call

The rule for a constructor call is interesting, as it potentially could be used as a conversion function:

$$\frac{\begin{array}{c}(E, Symbols, Bases) \vdash c : C \\ \breve{B}_{C \to B} \in Symbols(B)_{meth}\end{array}}{(E, Symbols, Bases) \vdash B(c) : B}$$

$$annotate: \ \breve{B}(new(B), c)$$

The first premise resolves the type of the argument to the constructor call to be C. The second premise verifies the existence of a constructor in B that takes an argument of type C. Finally, the type returned by the constructor call is of course B, the type we have just constructed.

The annotation is somewhat tricky: $annotate: \ \breve{B}(new(B), c)$. We will pass code to the semantics that includes another argument, a `new(B)` object. Notice that constructors actually take two arguments, the object to be constructed (here a new B object), and an argument (the C object to "convert" into a B object). We will see how this technique allows subtyping in function arguments and assignment statements to behave in a straightforward and consistent manner.

6.1.8 Assignment

Assignment is surprisingly straightforward with the constructor rules as above. All assignments require a constructor to convert between the types. The compiler will automatically generate a copy constructor (a copy of another object of the same class) and constructors to a class from any of its derived classes (allowing subtyping).

obj assignment $$\frac{\begin{array}{c}(E, Symbols, Bases) \vdash b : B \\ (E, Symbols, Bases) \vdash c : C \\ \breve{B}_{C \to B} \in Symbols(B)_{meth}\end{array}}{(E, Symbols, Bases) \vdash b = c : B}$$

$$annotate: \; b = \breve{B}_C(new(B), c)$$

(After this translation, $=$ can be interpreted as bitwise object copy.)

The first two premises establish the types of b and c. The third premise verifies the existence of a constructor of class B that takes an argument of class C. This could have been a copy constructor (if $B = C$), a derived to base constructor, or a user defined constructor. If there is a constructor with the right argument type, the assignment is legal. Notice the annotation: $annotate: b = \breve{B}_C(new(B), c)$. This transforms the assignment into an assignment of b to the output of a constructor call that generates a new B object. Then, the semantics of assignment can simply be a bitwise copy between the new B object and b. This works consistently whether the constructor was a copy constructor, a derived-to-base conversion, or a user defined conversion.

Notice, however, that the previous assignment rule was for object assignment, not for pointer assignment, which is specified by:

$$\textit{ptr assignment} \qquad \frac{\begin{array}{c}(E, Symbols, Bases) \vdash bp : B* \\ (E, Symbols, Bases) \vdash dp : D* \\ B \in Bases(D)\end{array}}{(E, Symbols, Bases) \vdash bp = dp : D*}$$

The first two premises establish the types of bp and dp. The third premise establishes that the conversion is legal: that D is a subtype of B, or equivalent to B. If so, then the assignment is legal, and is a simple bitwise copy of pointers.

Notice now that in the semantics, an assignment statement will always mean a bitwise copy. In the case of object assignment, the call would have been annotated into the output of a constructor call of the same type, while in the pointer assignment case no conversion is needed, just a copy of the pointer. This simplifies the semantics of assignment considerably.

Chapter 7

An Introduction to the Semantics

The semantics of a programming language describe precisely the manner in which a program will behave when executed. The set of semantics has two parts: the processing of declarations and run-time execution.

The processing of declarations results in the generation and storage of information that will then be used to execute the program. In C++, the virtual function tables used to implement dynamic binding are built at compile time by semantic processing of the class declarations.

At execution time, the semantics describe a precise meaning for source code, and often involve the various constructs created when the declarations were processed. Notice that the code actually processed is the output of the type checking rules, and in our case has annotations (revisions) that cause the semantics to execute the source code appropriately.

7.1 Dynamic Binding in C++: the *vtbl*

We will use a construction known as the virtual function table, the *vtbl* to implement dynamic method binding. The *vtbl*-based technique we describe is quite similar to many C++ compilers' implementations of dynamic binding, although not identical in form. The behavior, however, accurately reflects the behavior of C++.

7.1.1 *vtbl* Syntax

In our model, each class has an associated table, the *vtbl*. These tables are used at run time to bind method calls with method bodies dynamically. Each table can be thought of as a semantic function, taking an argument-signature, and returning a method-ID of the appropriate function. These tables will be used at run time to determine which method body will be executed. We will denote these tables as follows:

1. C_{vtbl} is a class C's *vtbl*.
2. The domain of C_{vtbl} is argument-signatures, of the form M_x. The range is over method IDs, of the form $M^B_{x\to\sigma}$.
3. Application of the function to an argument-signature is notated by $C_{vtbl}(M_x)$.
4. $C_{vtbl} = \{M_{void} \mapsto M^C_{void\to void},\ M_{int} \mapsto M^B_{int\to void}\}$ will be used to signify the `vtbl` of class `C`, where $C_{vtbl}(M_{void}) = M^C_{void\to void}]$ and $C_{vtbl}(M_{int}) = M^B_{int\to void}$.

The *vtbl* is essentially a table that keeps track of the virtual methods for an object. Invocations of a virtual method will result in looking up the actual method through the table, and dynamic binding is achieved. We will see this more carefully later; let us first consider how the *vtbl* is built.

7.1.2 *vtbl* Construction

The *vtbl* is a semantic function that takes the argument-signature (which has no class information) of a method, and returns the appropriate method-ID. Once a method call is determined to be a virtual call, the *vtbl* will be used to determine which method to execute. Construction of the *vtbl* is done with a very simple algorithm, but might not seem intuitive at first. It is possible that the *vtbl* for a class will contain methods that are not present in the class' *meth* set, which means that they can not be called. In other words, the *vtbl* might include some methods that the type-checker will not allow the user to call. Surprising as this may be, the behavior of the model accurately reflects C++'s behavior.

For a class `C`, derived directly from class `A`, we build `C`'s *vtbl* as follows:

- $C_{vtbl} = A_{vtbl}[M_x \mapsto M^C_{x\to y} | M_{x\to y} \in C_{vset}]$

This describes how C's *vtbl* is built from A's *vtbl*. Recall that all methods defined in C with the `virtual` keyword will have their complete-signatures placed in C_{vset}. The methods defined in C whose signatures are found in C_{vset} are used to update the *vtbl*. By updating the *vtbl* whenever a virtual method is overridden in a class, we ensure that the dynamic method selection mechanism will work properly.

7.1.3 Run Time Behavior

Once the type checking and semantic processing of the declarations is complete, semantic processing of the program commences and the program executes. Let us examine how the constructs we have generated are used to bind method calls with the appropriate bodies.

An object does not keep track of its own type at run-time, in contrast to some other object-oriented languages. Each object simply maintains a pointer to its type's *vtbl*. We will see shortly that this information is sufficient to implement dynamic binding of methods.

First, recall that our annotations in the type checking phase determined whether the dynamic method binding mechanism would be used or not. If not virtual, then the call would be annotated with the complete method-ID. If it were virtual, then the call was annotated as an argument-signature of the form $\hat{M}_x(x)$. The semantics will interpret these different annotations appropriately.

Recall the two different kinds of method invocations, direct and indirect. Direct calls, such as `b.M(x)`, do not invoke the virtual function mechanism, and hence are bound statically. The compiler, at type checking time, would resolve the type of `x` to be `X`, and `b` to be an object of class `B`. A method $M^C_{x \to y}$ would be looked up in B_{meth}. Notice that the method found might have been declared in a class `C` and inherited in `B`, and that we will also find the return type to be y. Once the method ID is found in B_{meth}, the method-ID is annotated into the source code. When the semantic rules handle execution of that source, the method-ID is present, and specifies precisely which method body to execute.

The compiler can determine the type of an object at type checking time, but what about the type of an object pointed to by some pointer? C++ distinguishes between the static and dynamic type of a pointer. (Note that "static" here refers to "knowable at compile time", as opposed to the multiple meanings of the C++ keyword `static`.) The static type of a pointer is the type a pointer was created as, while the dynamic type is the type of the object the pointer actually points to. For example, given a class `D` derived from a base class `B`:

```
B *bp1 = new(B);  //bp1 has static type B, dynamic type B
B *bp2 = new(D);  //bp2 has static type B, dynamic type D
```

Recall that C++ treats subclasses as subtypes, so pointers to a class might be assigned addresses of objects of a derived class, as in this example. Consequently, the dynamic type of an object cannot always be known at compile time.

With this in mind, let us consider indirect method calls, such as `bp->M(x)` Assuming the *static* type of `bp` is a pointer to an object of class `B`, we must somehow (at run time) select the method from the dynamic type (class) of the object pointed to.

However, C++ does not always bind the method dynamically when the receiver is a pointer. A determination is made in the type-checking rules of whether the call is virtual or not. This determination is made on the basis of the *vset* of the *static* type of the pointer. If the pointer `bp` is of type `B`, and $M_{x \to y} \notin B_{vset}$, then the call will not use dynamic binding. The argument-signature M_x will be looked up in B_{meth}, and a method ID $M^C_{x \to y}$ will be found, and that method executed by the semantic rules.

On the other hand, if $M_{x \to y} \in B_{vset}$, then the method should be bound dynamically. The type-checking rules will "annotate" the call with the instruction to use the virtual function mechanism, along with the complete-signature of the method. At run time, the complete-signature of the method to call is looked up in the *vtbl* the object. The *vtbl* then returns the method-ID of the method to execute. Notice that the appropriate method can be executed without the storage of the actual type of the object at run time.

7.2 The Form of Semantic Rules

Semantic rules are similar in form to type-checking rules, and are made up of a set of premises that validate a conclusion. If the premises, which are above the vertical bar, can all be satisfied, then the conclusion (below the bar) is valid.

$$\textit{generic declaration} \quad \frac{\begin{array}{c} premise_1 \\ premise_2 \end{array}}{(declarations, \rho, s) \downarrow (\rho', s')}$$

This generic example would be the rule for the semantic processing of *declarations*, in the environment ρ and state s. If $premise_1$ and $premise_2$ hold, then this rule results in the modification of ρ and s (notice the primes). Premises are read from top to bottom, as in the type-checking rules.

$$\textit{generic statement eval} \quad \frac{\begin{array}{c} premise_1 \\ premise_2 \end{array}}{(statements, \rho, s) \downarrow (V, \rho', s')}$$

The second generic rule is the an example of the semantic processing and execution of *statements*. The general form of rules for processing of statements is the same as rules for declarations, but return a value V.

What are ρ and s? ρ is the environment, where various run-time information is stored. ρ is a semantic function, which maps:

- Identifiers to locations (which will then be used to look up variables in the state).
- For each class D, the name D_{vtbl} to the virtual function table for a class D. The virtual function table for the class is a semantic function whose domain is argument-signatures and range is method-IDs.
- For each class D, the name D_{base} to the name of the direct base class of class D.
- For each class D, the name D_{vars}, to a set of instance-variable names

Updates to ρ are usually notated with the form $\rho' = \rho[x \mapsto new]$, meaning that an identifier x was updated to map to a value *new* in the domain of ρ. We have discussed the *vtbl* before, but we have not yet discussed D_{base} or D_{vars}.

D_{base} stores the name of the direct base class of a class D. If D has no base class, then D_{base} is empty.

D_{vars} maps to a set that contains the names of all instance variables for a class D. When an object of class D is instantiated, D_{vars} will be copied into the domain of a function that maps from the object's instance-variable names to their values.

The state, s, represents computer memory. It is a mapping from locations to objects or other values. Since C++ is an imperative language, most statements are evaluated for their side effects, namely the modification of the state.

To find the value of a variable x, first $\rho(x)$ would need to find the location, *loc*, where the value is stored. Then, *s(loc)* would return the actual value of the variable.

7.3 Overview of the Semantics

A C++ program consists of class declarations, function definitions, and method definitions. Each of these is processed by the semantics and used to build the *vtbl*'s (along with the *base* and *vars* sets). Then, the function `main()` is executed, resulting in the execution of the program.

Chapter 8

A Natural Semantics for C++

8.1 Semantic Processing of Declarations

8.1.1 Basic Program Parsing

$$\frac{(FUNCDEF, \rho, s) \downarrow (\rho', s) \quad (PROGRAM, \rho', s) \downarrow (\rho'', s')}{(FUNCDEF; PROGRAM, \rho, s) \downarrow (\rho'', s)}$$

$$\frac{(CLASSDEF, \rho, s) \downarrow (\rho', s') \quad (PROGRAM, \rho', s') \downarrow (\rho'', s'')}{(CLASSDEF; PROGRAM, \rho, s) \downarrow (\rho'', s'')}$$

$$\frac{(METHDEF, \rho, s) \downarrow (\rho', s) \quad (PROGRAM, \rho', s) \downarrow (\rho'', s')}{(METHDEF; PROGRAM, \rho, s) \downarrow (\rho'', s')}$$

8.1.2 Class Declaration

$$CLASSDEF \qquad \frac{(METHDECS, \rho', s) \downarrow (\rho'', s)}{(class\ B\ \{METHDECS\}, \rho, s) \downarrow (\rho'', s)}$$

Where $\rho' = \rho[B_{vtbl} \mapsto \epsilon, B_{vars} \mapsto \epsilon, B_{base} \mapsto \epsilon]$

$$Derived\ CLASSDEF \qquad \frac{(METHDECS, \rho', s) \downarrow (\rho'', s)}{(class\ D:\ public\ B\{METHDECS\}, \rho, s) \downarrow (\rho'', s)}$$

Where $\rho' = \rho[D_{vtbl} \mapsto clone(B_{vtbl}), D_{vars} \mapsto clone(B_{vars}), D_{base} \mapsto \{B\}]$

8.1.3 Method Declarations and Instance Variable Definition

implicitly virtual methdec
$$\frac{\begin{array}{c} if\ M_{\sigma\to\tau} \in B_{vtbl} \\ \rho' = \rho[B_{vtbl}[M_{\sigma} \mapsto M^{B}_{\sigma\to\tau}]]] \end{array}}{(\tau\ M(\sigma\ x);, \rho, s) \downarrow (\rho', s)}$$

nonvirtual methdec
$$\frac{if\ M_{\sigma\to\tau} \notin B_{vtbl}}{(\tau\ M(\sigma\ x);, \rho, s) \downarrow (\rho', s)}$$

virtual methdec
$$\frac{\rho' = \rho[B_{vtbl}[M_{\sigma} \mapsto M^{B}_{\sigma\to\tau}\}]]}{(\tau\ \texttt{virtual}\ M(\sigma\ x);, \rho, s) \downarrow (\rho', s)}$$

constructor dec
$$\frac{}{(B(\sigma\ x);, \rho, s) \downarrow (\rho', s)}$$

instance variable
$$\frac{\rho' = \rho[B_{vars} \mapsto B_{vars} \cup \{x\}]}{(\tau\ x, \rho, s) \downarrow (\rho', s)}$$

8.1.4 Function Definition

func def
$$\frac{}{(\tau\ F(\sigma\ x)\{BODY\}, \rho, s) \downarrow (\rho', s)}$$

Where $\rho' = \rho[F_{\sigma\to\tau} \mapsto < fun(x : \sigma)\{BODY\}, \rho >$.

8.1.5 Method Definitions

meth def
$$\frac{}{(\tau\ B{::}M(\sigma\ x)\{BODY\}, \rho, s) \downarrow (\rho', s)}$$

Where $\rho' = \rho[M^{B}_{\sigma\to\tau} \mapsto < fun(this : B*, x : \sigma)\{BODY\}, \rho >]$.

virtual keyword
$$\frac{}{(\tau\ \texttt{virtual}\ B{::}M(\sigma\ x)\{BODY\}, \rho, s) \downarrow (\rho', s)}$$

Where $\rho' = \rho[M^{B}_{\sigma\to\tau} \mapsto < fun(this : B*, x : \sigma)\{BODY\}, \rho >]$.

constructor def
$$\frac{}{(B{::}B(\sigma\ x)\{BODY\}, \rho, s) \downarrow (\rho', s)}$$

Where $\rho' = \rho[\breve{B}_{\sigma\to B} \mapsto < fun(this : B*, x : \sigma)\{BODY\}, \rho >]$.

8.1.6 Calls

$$\textit{func call} \qquad \frac{(N,\rho,s) \downarrow (V_0,\rho,s') \quad (BODY, \rho^f[x \mapsto V_0], s') \downarrow (V_2, \rho^f, s'')}{(F_{\sigma\to\tau}(N);, \rho, s) \downarrow (V_2, \rho, s'')}$$

Where $\rho(F_{\sigma\to\tau}) =< fun(x:\sigma')\{BODY\}, \rho^f >$.

$$\textit{direct} \qquad \frac{(b,\rho,s) \downarrow (V_0,\rho,s') \quad (N,\rho,s) \downarrow (V_1,\rho,s'') \quad (BODY, \rho^f[this \mapsto \&V_0, x \mapsto V_1], s'') \downarrow (V_2, \rho^f, s''')}{(b.M^C_{\sigma\to\tau}(N);, \rho, s) \downarrow (V_2, \rho, s''')}$$

Where $\rho(M^C_{\sigma\to\tau}) =< fun(this:C*, x:\sigma)\{BODY\}, \rho^f >$.

$$\textit{indirect nonvirtual} \qquad \frac{(b,\rho,s) \downarrow (V_0,\rho,s') \quad (N,\rho,s') \downarrow (V_1,\rho,s'') \quad (BODY, \rho^f[this \mapsto V_0, x \mapsto V_1], s'') \downarrow (V_2, \rho^f, s''')}{(b{-}{>}M^C_{\sigma\to\tau}(N);, \rho, s) \downarrow (V_2, \rho, s''')}$$

Where $\rho(M^C_{\sigma\to\tau}) =< fun(this:C*, x:\sigma)\{BODY\}, \rho^f >$.

$$\textit{indirect virtual} \qquad \frac{(b,\rho,s) \downarrow (V_0,\rho,s') \quad (N,\rho,s') \downarrow (V_1,\rho,s'') \quad B_{vtbl}(M_\sigma) = M^C_{\sigma\to\tau} \quad \rho(M^C_{\sigma\to\tau}) =< fun(this:C*, x:\sigma)\{BODY\}, \rho^f > \quad (BODY, \rho^f[this \mapsto V_0, x \mapsto V_1], s'') \downarrow (V_2, \rho^f, s''')}{(b{-}{>}\hat{M}^A_{\sigma\to\tau}(N);, \rho, s) \downarrow (V_2, \rho, s''')}$$

Where $\rho(*V_0.vptr) = B_{vtbl}$.

$$\textit{constructor} \qquad \frac{(b,\rho,s) \downarrow (V_0,\rho,s') \quad (N,\rho,s') \downarrow (V_1,\rho,s'') \quad (BODY, \rho^f[this \mapsto V_O, x \mapsto V_1], s'') \downarrow (V_2, \rho^f, s''')}{(\breve{B}_{\sigma\to B}(b,N);, \rho, s) \downarrow (V_2, \rho, s''')}$$

Where $\rho(\breve{B}_{\sigma\to B}) =< fun(this:B*, x:\sigma)\{BODY\}, \rho^f >$.

8.2 Object Creation

$$\textit{base or derived} \qquad \frac{newloc = GetNewLoc(s) \quad \rho' = \rho[b \mapsto newloc] \quad s' = s[newloc \mapsto \{(vptr \mapsto \epsilon), vars\}] \quad (B_{metaCTOR}(newloc), \rho', s') \downarrow (V, \rho', s'')}{(\texttt{B b};, \rho, s) \downarrow (V, \rho', s'')}$$

Where *vars* is a semantic function, whose domain is B_{vars} and whose range is initially ϵ.

$$\textit{new object} \qquad \frac{\begin{array}{c} newloc = GetNewLoc(s) \\ s' = s[newloc \mapsto \{(vptr \mapsto \epsilon), vars\}] \\ (B_{metaCTOR}(newloc), \rho, s') \downarrow (V, \rho, s'') \end{array}}{(\texttt{new(B)}; \rho, s) \downarrow (newloc, \rho, s'')}$$

Where *vars* is a semantic function, whose domain is B_{vars} and whose range is initially ϵ.

8.3 Constructor Call

$$\textit{metaCTOR call no base} \qquad \frac{\begin{array}{c} if\ \rho(B_{base}) = \epsilon \\ (\breve{B}(b), \rho, s) \downarrow (V, \rho', s') \end{array}}{(B_{metaCTOR}(b, void), \rho, s) \downarrow (V, \rho', s')}$$

Where $\rho' = \rho[b*_{vptr} \mapsto B_{vtbl}]$.

$$\textit{metaCTOR call with base} \qquad \frac{\begin{array}{c} A = \rho(B_{base}) \\ (A_{metaCTOR}(b), \rho, s) \downarrow (V_1, \rho', s') \\ (\breve{B}(b, void), \rho', s') \downarrow (V_2, \rho'', s'') \end{array}}{(B_{metaCTOR}(b, void), \rho, s) \downarrow (V_2, \rho'', s'')}$$

Where $\rho' = \rho[b*_{vptr} \mapsto B_{vtbl}]$

8.4 Assignment

$$\textit{assignment} \qquad \frac{\begin{array}{c} (b, \rho, s) \downarrow (L_0, \rho', s') \\ (N, \rho', s') \downarrow (V, \rho'', s'') \\ s''' = s''[(L_0) \mapsto V] \end{array}}{(b = N;, \rho, s) \downarrow (L_0, \rho'', s''')}$$

Note: We are assured that N was the output of some ctor call (building a new object clone), because of the annotation during type-checking time.

Also, note that the line $s''' = s''[(L_0)_{vars} \mapsto \rho'[L_1]_{vars}]$ simply means that L_0 is updated with a bitwise copy of L_1.

Chapter 9

A Walkthrough of the Semantic Rules

The semantics of C++ describe precisely how a program executes. In our model, the type-checking rules handle much of the compiler's decision making process, allowing the run-time semantics to be straightforward.

9.1 Basic Program Semantics

A C++ program is a collection of class declarations, function definitions, and method definitions. The first three semantic rules handle the semantics of an entire C++ program:

$$\frac{(FUNCDEF, \rho, s) \downarrow (\rho', s) \quad (PROGRAM, \rho', s) \downarrow (\rho'', s')}{(FUNCDEF; PROGRAM, \rho, s) \downarrow (\rho'', s)}$$

$$\frac{(CLASSDEF, \rho, s) \downarrow (\rho', s') \quad (PROGRAM, \rho', s') \downarrow (\rho'', s'')}{(CLASSDEF; PROGRAM, \rho, s) \downarrow (\rho'', s'')}$$

$$\frac{(METHDEF, \rho, s) \downarrow (\rho', s) \quad (PROGRAM, \rho', s) \downarrow (\rho'', s')}{(METHDEF; PROGRAM, \rho, s) \downarrow (\rho'', s')}$$

A program is composed of *FUNCDEF*s, *CLASSDEFS*, and *METHDEFS*. These three rules break the program up to allow individual components to be processed by the rest of the semantic rules.

9.2 Class Declaration

A class can be declared as a base class or can be derived from some other class. A class with no base does not inherit any methods or instance variables, so the semantic rule will start building from empty *vtbl* and *vars*:

$$CLASSDEF \qquad \frac{(METHDECS, \rho', s) \downarrow (\rho'', s)}{(class\ B\ \{METHDECS\}, \rho, s) \downarrow (\rho'', s)}$$

Where $\rho' = \rho[B_{vtbl} \mapsto \epsilon, B_{vars} \mapsto \epsilon, B_{base} \mapsto \epsilon]$

This rule handles the declaration of a class B, with no base classes, under environment ρ and state s.

We start by setting B_{vtbl} and B_{vars} to ϵ since B is not inheriting virtual methods or instance variables from another class. B_{vtbl} and B_{vars} will be built as the features are declared. Similarly, B_{base} is set to ϵ because B has no base class. The first premise computes an update of ρ with new values for B_{vtbl}, B_{vars}, and B_{base}.

The premise, $(METHDECS, \rho', s) \downarrow (\rho'', s)$, recursively processes the components of the class: instance variable and method declarations.

$$Derived\ CLASSDEF \qquad \frac{(METHDECS, \rho', s) \downarrow (\rho'', s)}{(class\ D:\ public\ B\{METHDECS\}, \rho, s) \downarrow (\rho'', s)}$$

Where $\rho' = \rho[D_{vtbl} \mapsto clone(B_{vtbl}), D_{vars} \mapsto clone(B_{vars}), D_{base} \mapsto \{B\}]$

This rule handles class declaration for derived classes. The rule is almost identical to the previous rule, with the exception of the update to ρ. Since D has a base class B, D inherits a copy of B_{vtbl}, B_{vars}, and the entry B in D_{base}.

$D_{vtbl} \mapsto clone(B_{vtbl})$ means that D_{vtbl} is assigned to a copy of B_{vtbl}. Recall that some of the methods in a base class can be hidden by reusing their names in the derived class. So why is the entire *vtbl* of the base class copied in? Wouldn't some of those methods be illegal to call (since they are hidden) in the derived class? Yes, they are hidden (and therefore illegal to call), but the type-checking rules decide on the *legality* of method calls. If the call were illegal because the method were hidden in the derived class, the type-checker would catch the error. This implies that some of the methods in D_{vtbl} might not be legally called by objects of class D. This strategy seems to be the simplest way of thinking about *vtbl* inheritance, since the type checker will decide on the legality of calls statically.

Instance variables are all inherited, and the *clone* function simply makes a copy of the B_{vars} function to use for D_{vars}.

D_{base} stores the name of the direct base class of D, class B. This will be used to call the constructors for a class' base classes properly.

The rest of the derived class declaration rule is identical to the base class declaration rule.

9.2.1 Method Declarations and Instance Variable Definition

The components of a class include the declarations of methods and instance variables. These rules will handle the building of the *vtbl* for a class and the *vars* construct for a class. Recall that the *vtbl* for a class is a mapping between complete-signatures and method-ID's.

implicitly virtual methdec
$$\frac{\begin{array}{c} if\ M_{\sigma\rightarrow\tau} \in B_{vtbl} \\ \rho' = \rho[B_{vtbl}[M_\sigma \mapsto M^B_{\sigma\rightarrow\tau}]]] \end{array}}{(\tau\ M(\sigma\ x);, \rho, s) \downarrow (\rho', s)}$$

This rule establishes that a method not explicitly declared as virtual *is* in fact virtual, if it overrides something in the *vtbl* already. Recall that we initialized this *vtbl* with the *vtbl* from the base class, and we do not ever remove any methods. Therefore, any method ever declared as virtual would be in the *vtbl* for all classes subsequently derived.

The first premise determines that the complete-signature of the method is indeed virtual. The second premise, $\rho' = \rho[B_{vtbl}[M_\sigma \mapsto M^B_{\sigma\rightarrow\tau}]]]$, updates the *vtbl* for B, with a mapping from the argument-signature of the method (M_σ) to the method-ID ($M^B_{\sigma\rightarrow\tau}$). The *vtbl* maps from argument-signatures to method-ID's, rather than closures themselves, to simplify the semantics. Later, when we call a virtual method, we will look up the method's argument-signature in the class' *vtbl*, and find a method-ID that will then be used to look up the method body in ρ.

The next rule determines that the method is *not* virtual, and therefore no change is needed to the virtual function table for the class.

nonvirtual methdec
$$\frac{if\ M_{\sigma\rightarrow\tau} \notin B_{vtbl}}{(\tau\ M(\sigma\ x);, \rho, s) \downarrow (\rho', s)}$$

The behavior of the rule for an method declared virtual explicitly is identical to that of the implicitly virtual method, with the exception of not needing to look for the complete-signature in the *vtbl*. Since the `virtual` keyword is present, it is obvious that the method is virtual.

virtual methdec
$$\frac{\rho' = \rho[B_{vtbl}[M_\sigma \mapsto M^B_{\sigma\rightarrow\tau}\}]]}{(\tau\ \texttt{virtual}\ M(\sigma\ x);, \rho, s) \downarrow (\rho', s)}$$

The constructor declaration is trivial; no special processing is needed at this time.

constructor dec
$$\frac{}{(B(\sigma\ x);, \rho, s) \downarrow (\rho', s)}$$

The declaration of an instance variable of a class simply updates the *vars* set of the class to include that entry.

$$\textit{instance variable} \qquad \frac{\rho' = \rho[B_{vars} \mapsto B_{vars} \cup \{x\}]}{(\tau\ x, \rho, s) \downarrow (\rho', s)}$$

9.2.2 Function Definition

The function definition rule is a good introduction to the way the bodies of functions are processed and stored. In the flavor of C++ we accept, a function is declared and defined simultaneously.

$$func\,def \qquad \frac{}{(\tau\ F(\sigma\ x)\{BODY\}, \rho, s) \downarrow (\rho', s)}$$

Where $\rho' = \rho[F_{\sigma\to\tau} \mapsto < fun(x : \sigma)\{BODY\}, \rho >$.

This rule updates ρ with a mapping from a complete-signature to a closure. The closure is a construction used to store the body of the function with the environment in which the function was defined. The environment that the function was defined in will be the environment it will be executed in.

9.2.3 Method Definitions

The rules for method definitions are similar to that of function definitions, but do vary somewhat. Methods have an additional argument, the *this* pointer, that may be used within the body of the method without explicitly passing it in. The *this* pointer is automatically inserted as one of the arguments to methods by the semantic rules.

$$meth\,def \qquad \frac{}{(\tau\ B{::}M(\sigma\ x)\{BODY\}, \rho, s) \downarrow (\rho', s)}$$

Where $\rho' = \rho[M^B_{\sigma\to\tau} \mapsto < fun(this : B*, x : \sigma)\{BODY\}, \rho >]$.

This is the general rule for a method definition. The method definition will cause the generation of a closure that the method-ID will map to in the environment. The closure takes two arguments, $this : B*$ and $x : \sigma$. The first argument is a pointer to the current object, the receiver of the method call. The second argument is the argument declared by the method declaration. The closure stores the body of the method and the environment the method was declared in. The current environment is updated to include this new method and then returned by the conclusion.

The other method definition rules are essentially identical. The next rule solely differs by accepting the `virtual` keyword:

$$\textit{virtual keyword} \qquad \frac{}{(\tau\ \texttt{virtual}\ B{::}M(\sigma\ x)\{BODY\}, \rho, s) \downarrow (\rho', s)}$$

Where $\rho' = \rho[M^B_{\sigma\to\tau} \mapsto < fun(this : B*, x : \sigma)\{BODY\}, \rho >]$.

The constructor rule also differs minimally, storing $\breve{B}_{\sigma\to B}$ in ρ to signify that the method is a constructor. This signification is not really necessary, but makes the semantics somewhat more readable, as it clarifies when we are calling a constructor, instead of a normal method.

constructor def
$$\frac{}{(B :: B(\sigma\ x)\{BODY\}, \rho, s) \downarrow (\rho', s)}$$

Where $\rho' = \rho[\breve{B}_{\sigma\to B} \mapsto < fun(this : B*, x : \sigma)\{BODY\}, \rho >]$.

9.2.4 Calls

When a function is actually called, the closure will be found in ρ, and the body executed in the environment it was defined in. The next rule describes precisely the manner in which a function call is executed. Notice that the type-checking rules have annotated the function call with the complete-signature of the function, which the semantics will use to find the function's closure in ρ.

func call
$$\frac{(N, \rho, s) \downarrow (V_0, \rho, s') \qquad (BODY, \rho^f[x \mapsto V_0], s') \downarrow (V_2, \rho^f, s'')}{(F_{\sigma\to\tau}(N);, \rho, s) \downarrow (V_2, \rho, s'')}$$

Where $\rho(F_{\sigma\to\tau}) =< fun(x : \sigma')\{BODY\}, \rho^f >$.

The first premise, $(N, \rho, s) \downarrow (V_0, \rho, s')$, evaluates the function's argument down to some value (presumably an object or a pointer). Notice that the environment has a mapping from the signature to a closure (the body of the method):

$$\rho(F_{\sigma\to\tau}) =< fun(x : \sigma')\{BODY\}, \rho^f >$$

Notice there is no issue of overloading to worry about in the semantics: the annotation describes precisely which method will be called.

The second premise, $(BODY, \rho^f[x \mapsto V_0], s') \downarrow (V_2, \rho^f, s'')$, evaluates the body of the function (as stored in the closure). The function is evaluated in the environment stored in the closure with the additional mapping of formal parameter to actual parameter. In other words, when the source code of $BODY$ is evaluated in the updated ρ^f, the formal parameter x will map to the value of the actual parameter passed in, V_0. The result of the function evaluation is V_2, which is then returned by the conclusion. Notice that both the evaluation of the argument (the first premise), and the evaluation of the function body (the third premise), have the side effect of (possibly) updating the state. When a function is evaluated, it uses the *environment* stored with it, and the current *state*. The environment determines which variables and functions were known about when the function was defined, and the current state stores their values. The environment does not change when we are executing the program, as its contents are only modified when we are processing declarations or defining instance variables. Instance variables may be defined within a method or a function, and the rule for instance variables, which we saw earlier, updates ρ in the current scope. When that function or method returns, its scope ends and the changes to ρ are not kept. (The state, on the other hand, maintains the side effects of the function's execution.)

The rules for method calls handle the execution of the body of a method, possibly including looking up a virtual method in a class' *vtbl*. The first rule is for a direct call, where the receiver of the message is specified directly:

$$\textit{direct} \qquad \frac{\begin{array}{c}(b,\rho,s)\downarrow(V_0,\rho,s') \\ (N,\rho,s)\downarrow(V_1,\rho,s'') \\ (BODY,\rho^f[this \mapsto \&V_0, x \mapsto V_1],s'')\downarrow(V_2,\rho^f,s''')\end{array}}{(b.M^C_{\sigma\to\tau}(N);,\rho,s)\downarrow(V_2,\rho,s''')}$$

Where $\rho(M^C_{\sigma\to\tau}) =< fun(this : C*, x : \sigma)\{BODY\}, \rho^f >$.

The first premise, $(b,\rho,s)\downarrow(V_0,\rho,s')$, evaluates the receiver to be V_0. The second premise, $(N,\rho,s)\downarrow(V_1,\rho,s'')$, evaluates the argument to be V_1. Next, the closure for the method is looked up in the environment. Notice that the method is annotated with the method-ID of the method, $M^C_{\sigma\to\tau}$. We find the closure corresponding to that method to be $< fun(this : C*, x : \sigma)\{BODY\}, \rho^f >$.

The third premise, $(BODY,\rho^f[this \mapsto \&V_0, x \mapsto V_1],s'')\downarrow(V_2,\rho^f,s''')$, evaluates the body of the method in the environment ρ^f (stored in the closure) updated with the actual parameters. Notice that the *this* parameter of the method expects a pointer to the class that the object was actually defined in, $C*$. When we bind formal parameters to actual parameters in the updated ρ^f, we bind $this \mapsto \&V_O$; to the *address* of V_0.

The method evaluates to return a value of type V_2, which is returned by the conclusion, along with the updated state.

The next pair of rules handle indirect calls: non-virtual and virtual. The non-virtual case is quite similar to the previous direct call rule:

$$\textit{indirect nonvirtual} \qquad \frac{\begin{array}{c}(b,\rho,s)\downarrow(V_0,\rho,s') \\ (N,\rho,s')\downarrow(V_1,\rho,s'') \\ (BODY,\rho^f[this \mapsto V_0, x \mapsto V_1],s'')\downarrow(V_2,\rho^f,s''')\end{array}}{(b{-}{>}M^C_{\sigma\to\tau}(N);,\rho,s)\downarrow(V_2,\rho,s''')}$$

Where $\rho(M^C_{\sigma\to\tau}) =< fun(this : C*, x : \sigma)\{BODY\}, \rho^f >$.

Notice that the rule handles calls of the form $(b{-}{>}M^C_{\sigma\to\tau}(N)$, in which b is a pointer to an object. In addition, notice that the method has been annotated by the type checker with a method-ID, rather than the special virtual marking, meaning that the call is non-virtual.

The first premise evaluates b to V_0, which we know to be a pointer to an object, the receiver of the message. The second premise evaluates the argument to be V_1. Then we can find the closure bound to the method-ID $M^C_{\sigma\to\tau}$, as in the previous rule. The evaluation of the body of the method occurs in the third premise:

$$(BODY, \rho^f[this \mapsto V_0, x \mapsto V_1], s'') \downarrow (V_2, \rho^f, s''')$$

Notice that the *this* argument is bound to V_0 itself, since V_0 is a pointer in this indirect call. The result of the method call and the rest of the conclusion are identical to the direct call rule.

The next rule is quite interesting, as it implements dynamic method binding by using the virtual function table.

$$\textit{indirect virtual} \quad \frac{\begin{array}{c}(b, \rho, s) \downarrow (V_0, \rho, s') \\ (N, \rho, s') \downarrow (V_1, \rho, s'') \\ B_{vtbl}(M_\sigma) = M^C_{\sigma\to\tau} \\ \rho(M^C_{\sigma\to\tau}) =< fun(this : C*, x : \sigma)\{BODY\}, \rho^f > \\ (BODY, \rho^f[this \mapsto V_0, x \mapsto V_1], s'') \downarrow (V_2, \rho^f, s''')\end{array}}{(b{-}{>}\hat{M}^A_{\sigma\to\tau}(N);, \rho, s) \downarrow (V_2, \rho, s''')}$$

Where $\rho(*V_0.vptr) = B_{vtbl}$.

Here we are evaluating an indirect method call that the type checker has informed us will be virtual, by annotating the call with $\hat{M}^A_{\sigma\to\tau}$. The first two premises evaluate the receiver and argument just as in the previous rule.

The requirement following the rule, $\rho(*V_0.vptr) = B_{vtbl}$, reminds us that the object's *vptr* is dereferenced to find the *vtbl* for that object. Recall that each object keeps a *vptr* to the virtual function table it will use for dynamic binding. Once we have found the *vtbl* we will use, we will look up the method-ID of the method to invoke.

The fourth premise, $B_{vtbl}(M_\sigma) = M^C_{\sigma\to\tau}$, determines that the complete-signature passed to us by the type-checking rules (by annotation), binds to the method-id $M^C_{\sigma\to\tau}$. By then calling $M^C_{\sigma\to\tau}$, selected at run time from V_0's virtual function table, we have implemented dynamic binding.

The fifth premise evaluates the method as in the previous rule, and returns the method return value appropriately.

$$\textit{constructor} \quad \frac{\begin{array}{c}(b, \rho, s) \downarrow (V_0, \rho, s') \\ (N, \rho, s') \downarrow (V_1, \rho, s'') \\ (BODY, \rho^f[this \mapsto V_O, x \mapsto V_1], s'') \downarrow (V_2, \rho^f, s''')\end{array}}{(\breve{B}_{\sigma\to B}(b, N);, \rho, s) \downarrow (V_2, \rho, s''')}$$

Where $\rho(\breve{B}_{\sigma\to B}) =< fun(this : B*, x : \sigma)\{BODY\}, \rho^f >$.

This rule handles the call of a constructor. Constructors take two arguments: a pointer to the object which is being constructed (not explicitly passed in by the user), and the argument to construct from (as specified by the user). The first premise $(b, \rho, s) \downarrow (V_0, \rho, s')$ evaluates b to yield a pointer to the object to construct, V_0. The second argument is evaluated in the second premise, $(N, \rho, s') \downarrow (V_1, \rho, s'')$, yielding the value V_1.

The appropriate closure is found in ρ:

$$\rho(\breve{B}_{\sigma \to B}) =< fun(this : B*, x : \sigma)\{BODY\}, \rho^f >$$

The final premise will evaluate the body of the constructor:

$$(BODY, \rho^f[this \mapsto V_O, x \mapsto V_1], s'') \downarrow (V_2, \rho^f, s''')$$

Notice that the constructor call rule is almost identical to the other method call rules. When we create an object, however, by calling its constructor, we recall that the constructor for the base classes of the object are are also called. These base class constructors will be called by a "meta-constructor" called by the various object creation rules. We will see this scheme in action in more detail shortly.

9.3 Object Creation

The semantics of object creation are somewhat tricky. When an object is created, C++ specifies that the constructor for the base of a class is executed first. This behavior is applied recursively: first the constructor of the root of the inheritance hierarchy is called, then the constructor of the next derived class, all the way down to the constructor for the class of the object we are creating itself (last!). Our object creation rules work in conjunction with a "meta-constructor" rule to model this properly. Let's look at the object creation rules to see this in action:

$$\textit{base or derived} \qquad \frac{\begin{array}{c} newloc = GetNewLoc(s) \\ \rho' = \rho[b \mapsto newloc] \\ s' = s[newloc \mapsto \{(vptr \mapsto \epsilon), vars\}] \\ (B_{metaCTOR}(newloc), \rho', s') \downarrow (V, \rho', s'') \end{array}}{(\texttt{B b};, \rho, s) \downarrow (V, \rho', s'')}$$

Where *vars* is a semantic function, whose domain is B_{vars} and whose range is initially ϵ.

This rule handles the creation of an object in the current scope with the command `B b`. The first step is to get a new chunk of storage space from the store. In our model, we assume each chunk to be whatever size necessary to store the object, and do not concern ourself with this issue in more detail. The first premise, $newloc = GetNewLoc(s)$, calls a function *GetNewLoc(s)*, which we define to give us a new chunk of memory in the state s. We will call the location returned to be *newloc*. The contents of $s(newloc)$ are undefined at this point.

The second premise, $\rho' = \rho[b \mapsto newloc]$, updates the environment with the binding of the identifier b with the new location we just determined, *newloc*. The third premise,

$$s' = s[newloc \mapsto \{(vptr \mapsto \epsilon), vars\}],$$

updates the store with the object. The form of objects in the store is a set containing a *vptr*, and a semantic function that maps from instance variables to their values. The *vptr* points to the virtual function table that will be used for the object. The *vptr* is set here to ϵ because of the manner in which the meta-constructor rules work. The instance-variable mapping function domain is B_{vars}, which contains the names of all the instance-variables of the class. At this point, every instance-variable name of the class maps to ϵ in the *vars* function, and the body of the constructor will contain the user code that actually initializes these variables.

The last premise, $(B_{metaCTOR}(newloc), \rho', s') \downarrow (V, \rho', s'')$, calls the meta-constructor we keep alluding to. We will see it discussed shortly. For now, notice that the constructor call returns a value V, which is the object that was just constructed. This is useful for some kinds of object initializations, such as those of the form: `B b = B(17);`.

The next kind of object creation is the explicit call to the `new()` function, which returns a pointer to the space that it reserves for an object.

$$\textit{new object} \qquad \frac{\begin{array}{c} newloc = GetNewLoc(s) \\ s' = s[newloc \mapsto \{(vptr \mapsto \epsilon), vars\}] \\ (B_{metaCTOR}(newloc), \rho, s') \downarrow (V, \rho, s'') \end{array}}{(\texttt{new}(\texttt{B}); \rho, s) \downarrow (newloc, \rho, s'')}$$

Where *vars* is a semantic function, whose domain is B_{vars} and whose range is initially ϵ.

The first premise again reserves a space for the object in the state. The second premise, $s' = s[newloc \mapsto \{(vptr \mapsto \epsilon), clone(B_{vars})\}]$, updates the state with a new object. The third premise calls the meta-constructor on the new object.

Notice that we lack a premise that updates ρ, because we used `new()` to create the object, rather than an object creation of sorts. The only way for the user to keep track of this object will be to assign some pointer with the value *newloc*.

The value returned by the constructor is the location of the object that was just constructed.

9.4 Meta-Constructor Calls

The semantics of constructor calls were straightforward, but when we create an object, we must also call the constructors for the base classes of that object, leading to another group of constructor rules.

C++ dictates that when an object derived from some other class is created, the constructor for the "most" base class (earliest on the inheritance hierarchy) is called first. Then, the constructor for the class derived from that base class is called, continuing down to the constructor for the object's class itself. For example, given that a class C is derived from a class B which in turn is derived from a class A, when we create a C object, the constructors are called in the order: $\breve{A}, \breve{B}, \breve{C}$.

Within the body of the constructor, virtual functions may be called, but dynamic binding is not used. The method is simply resolved statically. If this were not the case, then the constructor of a base class might call (dynamically) a method in a derived class, where not all of the requisite fields had been initialized properly yet! Here's an example of the potential for error, patterned after an example in section 12.7 of the ARM:

```
class B {
public:
   virtual void f();
   B() { f(); }   //calls B::f()
};

class D: public B {
public:
   int *p;
   void f() {
     cout << *p;   // disaster if 'p' is uninitialized
   }
};
```

When the programmer wrote the code for class B, she cannot be responsible for instance variables in other classes. From a language design standpoint, the constructor for B objects should be specific to B. If a base-class constructor is called for a base-class part of some derived object, than the constructor should only concern itself with the base-class part of the object.

This behavior posed the tricky question of how to formulate the method resolution rules in the type checker to differentiate between the bodies of constructors and the bodies of normal methods. The rules for constructor bodies would need to shut off dynamic binding. Dynamic binding would have to be disabled solely for the particular object that was being constructed. If the constructor for that object sent a method call to itself, which in turn sent *another* method call, the type checker would have to be clever enough to prevent dynamic binding on the object (but not interfere with any dynamic binding occurring on method calls on *other objects*).

Rather than further complicate the already contorted rules, we arrived at a simple solution that avoids much undue complexity in the type checking rules.

Rather than shutting off the dynamic binding mechanism, we simply change the virtual function table for the object being constructed. We give the object being constructed the virtual function table of the class that it is being constructed as. In this way, the virtual function table used knows nothing about virtual methods overridden in derived classes. For example, using the previous example (adding a constructor for `D`):

```
class B {
public:
   virtual void f();
   B() { f(); }   //calls B::f()
};

class D: public B {
public:
   int *p;
   void f() {
     cout << *p;   // disaster if 'p' is uninitialized
   }
   D() { f(); }  //calls D::f()
};
```

If we create an object of class `B`, we call the constructor on the new object with its *vptr* set to B_{vtbl}. The call to `f()` in the constructor body will be considered virtual, looked up in B_{vtbl}, and bound to $f^{B}_{void \to void}$.

If we create an object of class `D`, we first call the constructor for the base class of the new object, with its *vptr* set to B_{vtbl}. When the method `f()` is called in the constructor for `B`, the method call is found to be virtual, and looked up in the *vtbl* pointed to by the object's *vptr*, which we just set to be B_{vtbl}. The object is resolved dynamically, but from the virtual function table for class `B`, rather than class `D`. In this manner, even if another virtual method were called from within the body of the method `f()`, the *vtbl* of the object being constructed would still be that of `B`, and dynamic binding would be bypassed properly.

When the constructor for the `B` part of the object returns, the *vptr* of the object is set to D_{vtbl}, and the constructor for class `D` is called. Again, dynamic binding will be used to call methods, and the *vtbl* has been set in a way that methods in `D` will be called.

So, when an object is created, the meta-constructor is called to handle calling all the constructors for the base-class parts of the object. Note that when an object is copied or converted via a copy or conversion constructor, the meta-constructor rules are not used because base class constructors are not called. Only instances of object creation require this behavior. Let's look at the meta-constructor rules that make all of this fit together:

$$\textit{metaCTOR call no base} \qquad \frac{\begin{array}{c} if\ \rho(B_{base}) = \epsilon \\ (\breve{B}(b), \rho, s) \downarrow (V, \rho', s') \end{array}}{(B_{metaCTOR}(b, void), \rho, s) \downarrow (V, \rho', s')}$$

Where $\rho' = \rho[b{*}_{vptr} \mapsto B_{vtbl}]$.

This first meta-constructor rule handles the constructor calls for an object with no base class. The first premise, $if\ \rho(B_{base}) = \epsilon$, verifies that B has no base classes, by checking the construct B_{base} stored in the environment. The requirement following the rule indicates how we will set the object's *vptr* to point to B_{vtbl}. The second premise, $(\breve{B}(b, void), \rho, s) \downarrow (V, \rho', s')$, actually calls the constructor for class B, and returns a pointer to the object constructed, which is then in turn returned in the conclusion of the rule. Constructors takes two arguments, the object to construct, and the object two construct it from, which in this case is *void* since we are dealing with the default constructor.

The next rule handles the construction of an object that has a base class (which might in turn have a base class). The rule can be thought of the inductive step of an algorithm, where the previous rule handles the base case.

$$\textit{metaCTOR call with base} \qquad \frac{\begin{array}{c} A = \rho(B_{base}) \\ (A_{metaCTOR}(b), \rho, s) \downarrow (V_1, \rho', s') \\ (\breve{B}(b, void), \rho', s') \downarrow (V_2, \rho'', s'') \end{array}}{(B_{metaCTOR}(b, void), \rho, s) \downarrow (V_2, \rho'', s'')}$$

Where $\rho' = \rho[b{*}_{vptr} \mapsto B_{vtbl}]$

The first premise establishes that the object of class B that is being constructed has a base class A. The meta-constructor for A is called to handle the base class(es) of B. The requirement that follows the rule, $\rho' = \rho[b{*}_{vptr} \mapsto B_{vtbl}]$, indicates how we will set the object's *vptr* to point to the *vtbl* of the class for which the constructor is about to be called. Since we always make this assignment before we call a constructor, in both the inductive case and the base case rules, we always set the *vptr* to be the same class as the constructor is for. In this manner, we always prevent any unwanted dynamic binding of messages sent to the object being constructed.

The last premise, $(\breve{B}(b, void), \rho', s') \downarrow (V_2, \rho'', s'')$, calls the default constructor for the class B part, and returns the object just constructed.

9.5 Assignment

This rule handles the assignment statement. Assignments, in C++, are quite complex because of all the implicit conversion rules that are used to determine the legality of assignments. Our type-checker handles all the conversions (through constructor calls), allowing for the semantic rule for assignment to be simple. If the assignment was originally between objects, the type checker would have annotated the necessary constructor conversions and copy constructor, so that the assignment rule *is* simple assignment. For pointers, the type-checking rules verified the legality of the assignment (i.e.: the static type of the pointer on the right side of the operator is a subtype of the pointer on the left side).

$$\textit{assignment} \qquad \frac{\begin{array}{c}(b, \rho, s) \downarrow (L_0, \rho', s') \\ (N, \rho', s') \downarrow (V, \rho'', s'') \\ s''' = s''[(L_0) \mapsto V]\end{array}}{(b = N; , \rho, s) \downarrow (L_0, \rho'', s''')}$$

The first premise, $(b, \rho, s) \downarrow (L_0, \rho', s')$, evaluates the left side off the equation to be the l-value L_0. The second premise, $(N, \rho', s') \downarrow (V, \rho'', s'')$, evaluates the right side of the equation to be the value V. The third premise,

$$s''' = s''[(L_0) \mapsto V]$$

updates the state to bind the l-value L_0 to map to value V.

Chapter 10

The Big Picture

Studying the formal type-checking and semantic rules of a language may seem like an obtuse way to master it, but the approach yields insights that would be hard to come by outside of a formal framework. The most important result of our research, however, is simply to help build a conceptual model that explains the behavior of the core of C++.

10.1 Inheritance and Overloading

One of the benefits of OO languages is code reuse. Inheritance facilitates code reuse by allowing methods to be included in a derived class without recoding it. Unfortunately, inheritance seems complicated when considered with overloading of method names.

Usage Tip 1 *The reuse of a method name in a derived class causes method hiding.*

In order for a method to be inherited from a base class into a derived class, the *name* of the method must not be reused. If the name is reused, then the method in the base class is hidden. For example:

```
class B {
public:
  void M(void);
};

class D {
public:
  void M(int); //hides B::M(void) - it will not be inherited
};
```

```
main() {
  D d;
  d.M(17); // D::M(int) called
  d.M();   // illegal, D::M(int) hides B::M(void)
}
```

Observe that by reusing the name `M` in the derived class, we prevent *any* methods of that name from being inherited.

This hiding rule seems straightforward, but what about the notion of overriding, as described in every C++ guide? In reality, overriding is just a particular form of hiding. Let us elaborate.

Overriding is the reuse of a method name in a derived class, with argument types that match the base class declaration, i.e.: matching argument-signatures. Consider the following:

```
class B {
public:
  void M(void);
  void M(int);
};

class D {
public:
  void M(int); //hides all B::M; overrides B::M(int)
};

main() {
  D d;
  d.M(17); // D::M(int) called
  d.M();   // illegal, D::M(int) hides B::M(void)
}
```

So, in class `B` we have two methods of the name `M`, and we reuse the name `M` in class `D`. The simple conceptual model is that `D::M(int)` hides all methods in the base class with the same name (in this case, `D::M(void)` and `D::M(int)`). So, rather than thinking of this as "overriding," it is useful to realize that it is simply hiding the base class methods. *If the method name is reused, the base class method is hidden.* The type-checking rules we have developed demonstrate this clearly. One of the premises of the *derived-class-dec* rule shows how a derived *methset* is built from a base class *methset*: any methods whose names are reused in the derived class are not inherited into the derived *methset* (which determines method visibility).

10.2 Subtype Polymorphism

One of the strongest incentives to use inheritance, of course, is to take advantage of subtype polymorphism. Polymorphism is the ability for a piece of code to operate on a variety of types. In C++, the flavor of polymorphism is subtype polymorphism, which allows certain types to masquerade as objects of another type. More specifically, anywhere an object of a base class can be used, an object could also be used. Likewise, anywhere a pointer to a base class was expected, a pointer to a derived class object can be used.

Polymorphism is an important OO feature, and C++'s form of subtype polymorphism can be somewhat tricky. Anywhere an object of a base class `B` is expected, an object of a derived class `D` can be substituted. This holds even if `B` is an indirect base class of `D`, meaning that `D` might have inherited from some other class that inherited from `B` (also possibly indirectly).

The common method of implementing subtype polymorphism is to define a base class that will serve as the root of the inheritance hierarchy, with generic (or pure virtual) forms of the methods the class *family* will support. Polymorphic functions are coded to expect pointer to instances of the base class, and then subtyping is used to allow those pointers to point to objects of any class in the family.

This is distinct from the subtyping that C++ uses with objects, where an *object* of a derived class is used when an object of a base class is expected. C++ actually implicitly converts the derived class object into an object of the base class. (Because of this, if a virtual method is called after an object has been converted, the version in the base class will be called.)

10.3 Dynamic Method Binding

The power of C++'s polymorphism hinges on dynamic method binding. To understand this feature, let us first consider the difference between static and dynamic binding of methods.

The **static binding** of methods is where the actual receiver of the message call object is known at compile time. For example, if our program explicitly creates an object, and then sends it a message, the compiler knows it's type. In fact, for all *direct* calls, the compiler will know the type of the receiver. All direct calls are resolved statically (at compile time), as explained by the type-checking rules.

The **dynamic binding** of methods is where the method invoked is determined at run time by the type of the receiver. For example, a pointer `bp` to a base class `B` might be assigned to point to an object of a derived class `D`. In general, the compiler cannot know what type of object `bp` actually points to[1].

[1]Recall that the static type of the pointer is the type it was declared to point to, and the dynamic type is the type of object the pointer actually points to, which may change as the

So, the type-checker determines the visibility and "virtuality" of messages that a pointer can receive. When we actually execute the program, we already know that the method calls are legal, and which ones will be bound dynamically. Our formal rules make heavy use of "annotation" to pass the information about classes we have determined on to the semantics. Notice that anything we annotated (such as the method-ID of a function to call, or the virtuality "top hat") was resolved statically (at type-checking time). Because almost all of method binding is determined statically, at type-checking time, decisions are made on the basis of the pointer's static type, the base class.

Usage Tip 2 *The base class determines the behavior of the entire class family.*

- *Method visibility is determined by the base class interface.*
- *Method virtuality is determined by the base class interface.*

This behavior leads directly to several principles for good C++ coding style. Firstly, if a programmer want to use polymorphism (which is one of the main benefits of an OOL), she has to build it into the base class. A polymorphic function should take pointers to the base class for its arguments, allowing it to take arguments from any class derived from the base class. Any methods she wants the entire family to use should be defined in the base class. *The base class interface determines method visibility.*

The base class also determines which methods will be virtual. Recall that dynamic binding is *only* used when we use an indirect method call. A method is virtual if it is declared with the `virtual` keyword, or a method with a matching argument-signature was declared as `virtual` in a base class. Recall the *vset* from the semantics. The *vset* stored the complete-signatures of methods declared to be virtual in the class, or were virtual in a base class. Once an argument-signature is declared virtual in the class hierarchy, it is forever virtual.

For dynamic binding to be used, the call must be indirect, and the receiver's static type must consider the method virtual. If both of these conditions hold, then dynamic binding will be used. Because the test for "virtuality" is done in the receiver's static type, we say that *the base class determines method virtuality.*

The actual mechanism used to implement dynamic method binding is the virtual function table. Every C++ class has a virtual function table, which stores pointers to the bodies of the virtual methods. Each object carries around a pointer to the virtual function table of its class. For example, a `B` object carries around a `vptr` to B_{vtbl}.

The virtual function table (*vtbl*) used at run-time decides which method body actually executes. Recall that the decision of *when* to use dynamic binding was made at compile time. We passed along the complete-signature of the method, with the virtual hat, to the semantics for execution. All that is left to do at

program runs.

run-time is to look up the method body that corresponds to the annotated signature. First, the pointer to which the message was sent is dereferenced to find the object it points to. Then, the *vptr* of that object is dereferenced to find the right *vtbl*, which has a mapping from the signature (that the type checker passed the semantics) to the actual method body to call.

The benefit of this implementation is that an object does not need to carry around any information about its type, just a pointer to the appropriate *vtbl*.

10.4 Constructors

The constructor is a class method that is called when an object is created. Its purpose is to initialize the object according to some user-specified code. This seems like a reasonable feature to add to a programming language. What makes constructors somewhat confusing, however, is C++'s call by value semantics and conversions. When an object is passed to a function, the object is copied into the formal parameter by a copy constructor. Because of this, a single method call might result in three method invocations: the method, a conversion constructor, and a (essentially unnecessary) copy constructor.

Recall that C++ has three kinds of constructors:

- A *default constructor* is a constructor with no arguments.
- A *copy constructor* is a constructor that takes an argument of the same class that it is constructing.
- A *conversion constructor* is a constructor that takes arguments of other types.

A default constructor is specified without any arguments, and is called when an object is created without any arguments passed to the constructor. The body of the constructor will initialize the instance variables of the new object to whatever the user wants.

A copy constructor is a constructor for a class that builds a new object with values from some other object of the same class. Copy constructors are called whenever a programmer passes an object as an argument: the copy constructor for the argument's class is used to create a new object.

A conversion constructor is used to create a new object from arguments of some other types, or to convert from one type to another. Let's look at some examples:

```
class B {
public:
   int x,y;
   B(); //default constructor
   B(B &otherB); //copy constructor
```

```
   B(int newX,int newY); //conversion constructor
};

void foo(B input) {
  ...
}

main() {
   B b;           //default constructor called
   B b2(b);       //copy constructor called
   B b3 = B(b);   //alternate form; copy constructor called
   B b4(17,32);   //conversion constructor called

   foo(b);        //copy constructor implicitly called
}
```

The comments in the example indicate the default, copy, and conversion constructor. We define the function `foo` to take a `B` *object* as its parameter. In the main program, `B b` creates a `B` object using the default constructor. Whatever code the user specifies for `B::B(void)` will be executed when the object is created.

The second line, `B b2(b)`, creates a new `B` object, constructed with a copy of `b`. The next line, `B b3 = B(b)`, is an alternate form of "initialize a new object with the copy constructor," as in the previous line. The last line, `B b4(17,32)`, is the creation of a new object with the conversion constructor.

The compiler will automatically generate certain constructors, even if the user does not specify them. A default constructor will be generated. The copy constructor–which will perform a bitwise copy–will be generated automatically. Also, for a class `B`, conversions constructors that convert from every class derived from `B` will be generated too. This will allow derived *objects* to be passed to be substituted for base class *objects* in function calls.

A Conceptual Model of Constructors

As we developed the formal rules, a clearer understanding of constructors emerged. Here is a conceptual model that may help the reader understand the behavior of constructors. Much of this mirrors what we discussed in the introduction to the type-checking rules.

Usage Tip 3 *Constructors can be thought of as methods that take as parameters:*

- *the object to construct*
- *the arguments to construct the object with*

Constructors implicitly return the object they just constructed.

Constructors can be thought of as methods that take two parameters: the object to construct and the object(s) to construct it from. The "object" to construct is *implicitly* passed in by the compiler. When we are creating a new object of class `B`, the type-checker inserts a `new(B)` to be executed by the semantics.

The type-checking rules tell us that whenever we pass an object into a function, a constructor is called. The process is actually quite consistent, even if we are passing the same type as expected, a derived type, or some other type (with a user-defined conversion). The type-checking rules simply look at the type of the argument. If there is a constructor from the type of the actual argument type to the formal parameter type, the call is legal. The appropriate constructor is used with a `new`. (Call by value necessitates the use of a new object inside the function body.)

Let's see an example of this behavior:

```
class B {
public:
   int x,y;
   B(); //default constructor
   B(B &otherB); //copy constructor
   B(int newX,int newY); //conversion constructor
   B(Other o); //conversion constructor
   // compiler generates a constructor with D object as argument
}

class D: public B {
public:
   int z;
   //compiler will generate default and copy ctors automatically
};

class Other {
public:
   //...
};

void foo(B arg) {
   //...
}

main() {
   B b;      //default ctor for B called
   D d;      //default ctor for D called
```

```
    foo(b);    //copy ctor for B called
    foo(d);    //copy ctor for D and conversion B(D) ctor called

    Other Ot; //default ctor for class Other called
    foo(Ot);  //copy ctor for Other and converstion B(Other) called
}
```

In this example we have the base class `B`, and a class derived from it, `D`. We also defined another class `Other`, that is *not* derived from `B` or `D`. We defined a function, `foo`, whose argument is an object of type `B`.

In the main program, we declare objects of type `B` and `D`. Then, we call `foo(b)`. The type checking rules look for a constructor for class `B` (the type of the formal parameter), that accepts an object of the type of the actual parameter, in this case also `B`. Of course, the copy constructor for `B` accepts a `B`, so the copy constructor is called, to construct a `new(B)` object from the object we passed in, `b`.

Usage Tip 4 *When we pass an object to a function or method, a constructor is called to convert from the type of the actual parameter to the type of the formal parameter.*

This guideline consistently explains the usage of constructors to pass parameters into function bodies. The constructor that converts from the argument's actual type to the expected type will be used. If the types are the same, then the constructor called will of course be the copy constructor. If the types are different, a conversion constructor is called. The conversion constructor *might* be the conversion from a derived class to a base class. We will see that behavior in the next call: `foo(d)`.

The call `foo(d)` results in first the calling of the (compiler-generated, in this case) copy constructor for class `D`, followed by the conversion constructor of class `B` that takes an argument of class `D`. The compiler will automatically generate both of these constructors. The conversion constructor has the responsibility of converting the `D` object into a `B` object.[2]

Let's consider the last two lines now:

```
    //...
    Other Ot; //default ctor for class Other called
    foo(Ot);  //copy ctor for Other and converstion B(Other) called
```

[2]It turns out that such conversions are easy; the type-checker can simply type check the object as a `B` object in the body of the function, and the semantics will simply update the *vtbl* to a B_{vtbl}, so methods bound dynamically will appropriately come from `B`. The extra instance-variables can simply be kept around, as the type checker will prevent anybody from accessing them (since they aren't part of class `B`). These features of the formal rules are discussed in the "walkthrough" sections in much greater detail.

We create an object `Ot` of class `Other`, which calls the compiler-generated default constructor for class `Other`. Although `Other` is not derived from either of the other classes (`B` or `D`), notice the constructor the user defined for class `B` that takes an object of class `Other`. This is a user-specified conversion constructor, that the compiler would *not* have automatically generated (since `Other` is not derived from `B`). When we then call `foo(Ot)`, the compiler will look for a constructor that converts from class `Other` to class `B`. The appropriate user specified conversion constructor is found, so the call is legal, and the constructor calls is "annotated" in. The (compiler-generated) copy constructor of class `Other` is called to copy in a new `Other` object into the conversion constructor of class `B`, which in turns generates an object of class `B` to be used in the body of `foo`.

Clearly, the user may be getting a lot more constructor calls than she expected!

Chapter 11

Conclusion

This thesis was an attempt to apply formal language analysis techniques to a large, complex programming language. We focused our research on the the core of the object-oriented features of C++, the method binding mechanism. The impact of conversions on the method binding mechanism was analyzed. We devised a set of type-checking rules that statically resolve the bulk of the method binding process, and used "annotation" to pass information to the semantic rules. The semantic rules were relatively straightforward, thanks to the resolution handled by the type checker. Finally, we attempted to draw together the knowledge we gained about the behavior of the language into a set of usage guidelines, qualified by features of the formal rules.

C++ is a large, complex language. Near the completion of this thesis, a new version of the ANSI/ISO working paper was released. This new version of the language specification is much better than the ARM, and strives to be more readable, consistent, and complete. This researcher is pleased to find an example in section 10.3 suspiciously similar to one he posted to the `comp.std.c++` Usenet group. The example revealed behavior that the language specification did not address, and many compilers behaved differently when fed the example. The new working paper now presents the example, and describes the behavior that should result. By using examples to explain the language, rather than some sort of formal framework, the language specification will undoubtably always have holes. Compilers will still occasionally generate code that behaves differently, and the discrepancy will reveal a new unanswered question about the true semantics of the language.

C++ is particularly prone to such informal-description problems because it is a big language, and it is multi-paradigmatic. The informal nature of the specification creates the trade-off between features and consistency. If you just "throw in" more features to a language, they will invariably interact in strange ways with existing features. If a language designer only adds features that are extremely consistent with all the others, the language invariably will end up sparsely featured. (A language that lacks a pile of features is not necessarily a bad thing, however!)

The power of formal analysis, when used as a tool in language design, is the ability to build up a consistent, useful feature base for a language. Why does C++ bother differentiating between default, conversion, and copy constructors? It turns out there is a consistent way to explain the behavior of all these (presumably different) features as one simple method selection whenever an object is passed.

A user can code good object-oriented code in C++, taking advantage of the benefits of object-oriented language. But without a *deep* understanding of the inner workings of the language, the user will often find herself up the proverbial creek of strange behavior. In the past, programmers have been forced to think like a compiler writer to make sense of the language's behavior. I personally spent a few months working on a C++ compiler. I often had to make sense of the language's behavior by *tracing through the compilation!*

Many of the formal rules we use might seem similar to the internals of C++ compilation. The *vptr* and *vtbl*, for example, seem like low-level compiler details. We have not found an abstraction above this level. C++ books try to present an abstraction from internals like *vptr*s and *vtbl*s, but fail, in my estimation, in providing useful conceptual models. The presence of the ARM on the desk of virtually every C++ developer attests to the shortcomings of high level abstractions for C++; programmers keep having to come back to the ARM.

In closing, there is still much work to be done in analysis of C++. We did not investigate access controls, conversion functions (in contrast to conversion *constructors*), public/private inheritance, the multiple meanings of the `static` keyword (a perennial confusion-causer), templates, multiple inheritance, and many other features of the language. At some point, with a language as large and contorted as C++, the benefits of such research would probably be greater to compiler writers than helping users to build a conceptual model of the language, because there appears to be no simple high-level conceptual model possible.

Appendix A

Glossary

A.1 General Definitions

M_σ is the argument-signature of a method that takes a parameter of type σ.

$M_{\sigma\to\tau}$ is the complete-signature of a method that takes a parameter of type σ, and returns an object of type τ.

$M^B_{\sigma\to\tau}$ is a method ID, and elaborates on the previous definition by indicating that the method was defined in class B.

$\breve{B}_{D\to B}$ is the complete-signature of a constructor, defined in class B, that accepts a parameter of class D, and returns a constructed object of class B.

A.2 Type Checking Definitions

C_{meth} is a set that contains the method-IDs of methods visible in class C, for example: $C_{meth} = \{M^C_{\sigma\to\tau}, M^D_{x\to y}\}$. Notice that methods in C_{meth} are decorated with the class they were defined in.

C_{vset} is a set that contains complete signatures of methods that will be virtual for a class C, for example: $C_{vset} = \{M_{p\to q},\ N_{x\to y}\}$.

C_{iv} is a semantic function that takes instance variable names in the domain, and returns their types in the range, for example: $C_{iv} = \{x : \tau, y : \sigma\}$.

C_{sym} is a triple generated for each class C, which contains C_{meth}, C_{vset}, and C_{iv}.

Symbols is a semantic function that maps from each class C to its C_{sym} construct.

E is the type checking environment, which stores the types of variables for use within the type-checker. In addition, the environment stores the complete-signatures of functions defined in the program.

annotation is the substitution of source code with a modified version. The annotation will then be recognized at semantic processing time, and dealt with appropriately.

A.3 Semantic Definitions

ρ is the environment, which contains identifiers of the form:

f_σ the argument-signature of a function, which maps to a closure of the form $f_\sigma \mapsto < fun(x : \sigma)BODY, \rho >$.

$M^B_{\sigma\to\tau}$ a method-ID, which maps to a closure $M^B_{\sigma\to\tau} \mapsto < fun(this : B, x : \sigma)\{BODY\}, \rho >$.

B_{vtbl} a semantic function, which maps from argument-signatures to method-ID's, i.e.: $\rho[B_{vtbl}]M_{sigma} = M^B_{\sigma\to\tau}$.

B_{vars} a set of variables associated with a class B.

D_{base} the name of a class' direct base class, if any.

b the name of an object, which maps to a location in the store, i.e.:$b \mapsto L_1$.

s is the state, which contains identifiers of the form:

L_1 a location, which maps to an object in the store, i.e.: $L_1 \mapsto \{vptr, vars\}$.

$\{vptr, vars\}$ an object, containing a $vptr$ and the set $vars$. $vptr$ is a mapping to a $vtbl$ identifier, and $vars$ is a set of mappings between variable names and their values.

Bibliography

[Aba93] Martin Abadi. Baby modula-3 and a theory of objects, April 1993.

[BCD+93] K. Bruce, J. Crabtree, A. Dimock, R. Muller, T. Murtagh, and R. van Gent. Safe and decidable type checking in an object-oriented language. In *Proc. ACM Symp. on Object-Oriented Programming: Systems, Languages, and Applications*, pages 29–46, 1993.

[BL90] K. Bruce and G. Longo. A modest model of records, inheritance and bounded quantification. *Information and Computation*, 87(1/2):196–240, 1990.

[BM92] Kim B. Bruce and J. Mitchell. PER models of subtyping, recursive types and higher-order polymorphism. In *Proc. ACM Symp. on Principles of Programming Languages*, pages 316–327, 1992.

[Bru93] Kim Bruce. A paradigmatic object-oriented programming language: Design, static typing and semantics. *Journal of Functional Programming*, January 1993.

[Bru94] Kim Bruce. Binary methods, subsumption, and matching. December 1994.

[BSvG94] Kim Bruce, Angela Schuett, and Robert van Gent. PolyTOIL: A type-safe polymorphic object-oriented language. DRAFT, December 1994.

[Car85] L. Cardelli. Basic polymorphic typechecking. *Polymorphism Newsletter*, 2(1), January 1985.

[Car88] L. Cardelli. A semantics of multiple inheritance. *Information and Computation*, 76:138–164, 1988. Special issue devoted to *Symp. on Semantics of Data Types,* Sophia-Antipolis (France), 1984.

[Cas94] Giuseppe Castagna. Covariance and contravariance: conflict without a cause. page 12, April 1994.

[CHC90] William R. Cook, Walter L. Hill, and Peter S. Canning. Inheritance is not subtyping. pages 125–135, 1990.

[CW85] L. Cardelli and P. Wegner. On understanding types, data abstraction, and polymorphism. *Computing Surveys*, 17(4):471–522, 1985.

[Dav93] Stephen R. Davis. *C++ Programmers Companion: Designing, Testing, and Debugging.* Addison-Wesley, 1993.

[ES91] Margaret A. Ellis and Bjarne Stroustrup. *The Annotated C++ Reference Manual.* Addison-Wesley, 1991. ANSI base document.

[Gun92] Carl A. Gunter. *Semantics of Programming Languages: Structures and Techniques.* Foundations of Computing. MIT Press, 1992.

[Joy92] Ian Joyner. C++??: A critique of C++. 1992.

[Lip91] Stanley B. Lippman. *C++ Primer.* Addison-Wesley, 2nd edition, 1991.

[Mey92] Scott Meyers. *Effective C++: 50 Specific Ways to Improve Your Programs and Designs.* Addison-Wesley, 1992.

[NN92] Hanne Riis Nielson and Flemming Nielson. *Semantics with Applications: A Formal Introduction.* John Wiley & Sons, 1992.

[Sch94a] David A. Schmidt. *The Structure of Typed Programming Languages.* Foundations of Computing. MIT Press, 1994.

[Sch94b] Angela Marie Schuett. Parametric polymorphism in a type-safe, object-oriented programming language. Honors thesis, Williams College, Williamstown, Massachusetts, May 1994. Thesis Advisor: Dr. Kim Bruce, Williams College CS Department.

[Str91] Bjarne Stroustrup. *The C++ Programming Language.* Addison-Wesley, 2nd edition, 1991.

[Str94] Bjarne Stroustrup. *The Design and Evolution of C++.* Addison-Wesley, 1994.

[vG93] Robert van Gent. TOIL: An imperative type-safe object-oriented language. Honors thesis, Williams College, Williamstown, Massachusetts, May 1993. Thesis Advisor: Dr. Kim Bruce, Williams College CS Department.

[Wal93] Jim Waldo, editor. *The Evolution of C++: Language Design in the Marketplace of Ideas.* MIT Press, 1993.

[WF92] Keith Weiskamp and Bryan Flamig. *The Complete C++ Primer.* Harcourt Brace Jovanovich, Publishers, 2nd edition, 1992.