TOIL: An Imperative Type-Safe Object-Oriented Language

by

Robert van Gent

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

WILLIAMS COLLEGE
Williamstown, Massachusetts
22 May 1993

Contents

1 Introduction **1**

2 Characteristics of Object-Oriented Languages **4**

3 A Survey of Other Object-Oriented Languages **8**
3.1 The "Don't Cares" 8
3.2 The Subtle Failures 9
3.3 Limitations 11

4 Background **13**
4.1 TOOPL 13
4.2 Justification for TOIL 17
4.3 Issues in Imperative Languages 18

5 Type Checking in TOIL **23**
5.1 Types in TOIL 23
5.2 The type constraint system 26
5.3 Subtyping in TOIL 26
5.4 Inheritance in TOIL 32
5.5 Typing rules and abstract syntax 33

6 The natural semantics of TOIL **41**
6.1 The basics 41
6.2 Declarations 43
6.3 Intricacies 44
6.4 Formal rules 49

7 Denotational semantics **53**
7.1 The basics 53

7.2 Meanings of Types 55
7.3 Meanings of terms 58
7.4 Definitions and lemmas on consistency 65

8 Proof of Type Safety **67**
8.1 Significance of the proof 67
8.2 Theorem 67

9 The Implementation **78**
9.1 Overview of the prototype interpreter 78
9.2 Syntactic sugar 80
9.3 Formal grammar for TOIL 81
9.4 How to use the interpreter 85

10 Complexity of Type Checking **86**
10.1 About complexity 86
10.2 The Subtyping Algorithm 87
10.3 Complexity of Subtyping Algorithm 87
10.4 Complexity of Type Checking 91

11 Summary and Future Work **96**
11.1 Summary 96
11.2 Future Work 98

A Sample TOIL program **100**
A.1 Code for a maze program 100
A.2 The maze program in action 110

Abstract

The object-oriented programming language paradigm offers a useful approach toward data abstraction. However, conventional object-oriented languages fail to provide safe static typing systems, a serious drawback especially for large scale programming projects. Bruce presents TOOPLE ([Bru93]), a functional object-oriented language which supports structural subtyping, class-based inheritance, and updateable instance variables, while still maintaining completely safe and deterministic type checking and subtyping algorithms.

This paper details the design and prototype implementation of TOIL, an imperative version of TOOPLE. As an imperative language, TOIL hopes to provide a stronger and more efficient platform for realistic object-oriented programs. The semantics of TOIL are described in a manner similar to that used for TOOPLE. Although many of the basic semantic techniques remain the same, TOIL's imperative features require new syntactic constructs as well as fundamental changes in the semantics of the language. A semantic model is used to prove that the type-checking algorithm is consistent.

Chapter 1

Introduction

As the ultimate mode of communication between user and computer, programming languages attempt to make the programmer's job as easy as possible. At the lowest level, the computer only understands strings of 1s and 0s, but high level programming languages provide hardware independence and allow the programmer to concentrate on the top level algorithms being implemented instead of the details involved in how the machine executes them. Varying levels of abstraction make communication with the machine more natural, and therefore more productive. High level constructs are translated by a compiler or interpreter into a hardware-dependent format which can be executed directly. Procedures, modules, abstract data types, and even simple commands are all examples of such constructs.

A more recent approach toward abstraction is to group data and operations on that data together into so-called objects. For example, an object representing a point in space would not only store its coordinates, but could also provide methods to move the point relative to its old position, move it to new absolute coordinates, or compare it with other points. Furthermore, the form in which the data is stored (e.g. Cartesian versus polar coordinates for the point) can be hidden from the user, since the object itself should provide all necessary operations. The implementation of the Move operation differs for a point in Cartesian and polar coordinates, but the interface for the user remains unchanged. In this way, object-oriented languages provide a high level of abstraction and modularity. Furthermore, object-oriented languages naturally support polymorphism through subtyping, and encourage the reuse of code through inheritance. These strengths have contributed to the recent surge of interest in object-oriented programming.

In recent years the increasing importance and complexity of software engineering relative to hardware engineering has brought attention to new software development issues. Whereas in the past, the cost of hardware vastly overshadowed programming costs, pro-

grammer time is the most expensive input in most modern projects. With programmer time at a premium, research and common sense point to the importance of carefully designed programming languages in effective large scale software development projects. The strengths of object-oriented languages as described above particularly suit them to such projects. However, many aspects of object-oriented languages are not well understood theoretically, and this combination of promise and unfamiliarity may provide a unique opportunity for programming language theorists to make significant contributions not only to theory but to applications as well. In fact, there is definite room for improvement in the object-oriented languages presently available. See Chapter 3 for an overview of other object-oriented languages, most of which fail to provide a satisfactory framework for object-oriented programming.

Bruce's functional language TOOPL([Bru93]) demonstrates that the tradeoff between type safety, flexibility, and the power of object-oriented features is unnecessary. It provides inheritance, subtyping, and self-referential objects with hidden instance variables, all within the context of a completely safe static typing system which is maintained through the separation of the inheritance and subtyping hierarchies. Inheritance is specified by the syntactic derivation of classes, whereas subtyping is determined by an algorithm. This thesis involves continued work with the ideas presented in [Bru93], namely the development of an imperative version of TOOPL called TOIL. Such an exercise is interesting and useful because of the potential applications for a type-safe object-oriented language. The functional language is easier to model theoretically, and useful for demonstrating the fundamental techniques used by Bruce to achieve type safety in an object-oriented environment. But functional languages give up in efficiency what they gain in cleanliness, due to costly copy semantics, and therefore an imperative version is more relevant to potential applications of object-oriented languages. Many of the same techniques demonstrated in the semantics of TOOPLE are used in TOIL, but the imperative features add a great deal of complexity and bring up new issues which interact with the object-oriented ones in interesting ways.

There is still a good deal of work to be done with TOIL. As will be discussed later, a prototype interpreter has been developed, but an efficiently implemented compiler would be a useful exercise to test the true applicability of the language. The prototype interpreter avoided many details which a true implementation would have to deal with. Although these details are unlikely to prove impossible to achieve, many of them will probably involve solutions of considerable complexity. Also, there is still some theoretical work to be done, in particular a subject reduction proof and a proof of the consistency of the denotational semantics with the natural semantics. Hopefully one or both of these goals will be achieved during further research at Williams College with Professor Bruce.

Chapter 2 describes some general characteristics of object-oriented languages, including some examples. Chapter 3 provides a survey of other object-oriented languages, including some in use today, and other theoretical ones. Chapter 4 surveys previous work on this

topic, including work on TOOPLE, and justifies the current research on TOIL. Chapter 5 introduces types, subtyping, inheritance, and the abstract syntax for TOIL. Chapter 6 presents TOIL's natural semantics and in doing so attempts to explain many of the potentially confusing solutions to the difficulties raised by the implementation. Chapter 7 attempts to explain TOIL's denotational semantics. Chapter 8 presents a proof of type safety. Chapter 9 discusses the prototype implementation of TOIL, including a full grammar. Chapter 10 considers the complexity of the type-checking algorithm in TOIL. Chapter 11 summarizes and outlines possibilities for future research.

Chapter 2

Characteristics of Object-Oriented Languages

In this chapter we present some common features of object-oriented languages, and include a sample program using an approximation of TOIL's syntax. In the interest of space, we assume a general familiarity with the basic concepts of object-oriented languages and provide only a quick review.

Often it is easiest to think about object-oriented languages using examples, metaphors, and even personifications. Objects themselves are simply data and operations on the data grouped together. Ideally, the data cannot be accessed except through the operations provided by the object itself. Thus, an object serves as the "owner" of the data, servicing requests from the program or from other objects. Object-oriented languages represent an advanced form of data abstraction because the implementation of the object is opaque to the user. The object's customers do not need to have any information about how the operations provided by the object are implemented. Note that this abstraction allows the implementation of objects to be changed (e.g. to increase efficiency) without affecting their use. Using an example from the introduction, Point objects can be implemented using Cartesian or polar coordinates. Furthermore, the two Point objects are interchangeable from the point of view of the user, as long as they provide the same operations. The following is a sample dialogue between two differently-implemented Point objects and their customer:

Customer to Point1 : "Please move to (0,1)."

(Point1 responds by setting x=0, y=1)

Customer to Point2 : "Please move to (0,1)."

(Point2 responds by setting sets r=1, θ=90)

Like most programming languages, TOIL classifies data according to types. These types

can be used to detect many possible errors before a program is even run. For example, if an integer value is being compared to a value which represents a function, something is seriously wrong. The process used to determine whether a program is well-typed is called type checking. TOIL is statically typed, meaning that it is type-checked while it is being compiled, and that all values in the program are associated with types at that time. This method has the advantage of catching a high percentage of errors at compile time instead of at run time, increasing program reliability. Furthermore, TOIL is type-safe, meaning that *all* type errors can be caught at compile time, and none will slip by and cause problems while the program is running. In particular, for objects, it means that an object is guaranteed to know how to implement any operation which is requested of it.

The interface for an object, which specifies only which operations it has available, is provided by its type. Libraries can therefore be generated for TOIL simply by supplying the interfaces for a set of commonly-used objects, along with compiled code to implement them.

In our language, objects are created from templates called classes. A class will contain the initial values for data associated with the object, as well as the code used to implement its operations. Each class can be viewed as a factory which can produce an infinite number of objects. However, the factory itself is not useful for much other than producing objects. Initially, the produced objects are all the same, but they can change over time if they provide self-modifying operations. In TOIL, the `new` operation is used to manufacture an object from a class. New classes can be defined from old ones by simply adding new operations (called methods) or by modifying old ones. The original class is called the superclass, or parent of the modified one, which in turn is called the subclass. The incorporation of methods from a parent class into a subclass is called inheritance. The use of inheritance to generate new classes from old ones is common in object-oriented languages. It allows for rapid prototyping of many different kinds of objects, because classes defined using inheritance automatically retain all of the capabilities of the old class. Also, inheritance increases the power of libraries, since the programmer can define new classes from those supplied in the library. In TOIL the use of inheritance is limited in order to preserve type safety. However, one can determine whether a subclass is type-safe solely by inspecting the type of its parent. This property is ideal for libraries which are supplied only with interfaces and compiled code, because it is not necessary to have access to the original code in order to support inheritance.

Many object-oriented languages choose to use the class from which an object is formed as its type, or interface, as well. However, in TOIL the interface of an object is provided by an explicit type separate from the class definition. This separation is consistent with our desire for data abstraction, since the type of an object can omit details of implementation which are necessary in the class.

Another feature which is common in object-oriented languages is subtyping. We say

that a type A is a subtype of another type B (written A $\leq$ B) if and only if we can use something of type A in any context which expected something of type B. Note that the subtyping relates to types, whereas inheritance relates to the mechanism used to define classes. Many languages use the two interchangeably, but we will provide an example later which shows that this decision can lead to holes in a language's typing system.

Finally, TOIL allows an object to recursively refer to itself through a bound variable called *self*, and to its own type through a bound variable called MT (for MyType). Access to an object's data is impossible from anywhere except within its own methods. Therefore, *self* is available only within an object's methods. The variable *self* can be used to recursively call the object's methods, or to access or modify the object's data. Note that the use of *self* is the **only** way that the object's data can be directly modified. Another bound variable, *super*, is available in objects created from subclasses, and provides access to the methods of the parent class. Note, however, that the parent class has no data associated with it as such. Therefore, it would not make sense to attempt to access or modify a parent class's data.

A little notation will be necessary to understand the example in Figure 2.1. Data associated with an object is stored in various appropriately named *instance variables*. To request an object to perform a method, the format "Object <= method name" is used. In the example, the `extend` command is used to generate a ColorPoint class from a Point class using inheritance. `Extend` simply allows new methods and instance variables to be added to a class. Inheritance is also possible using an `update` command, which allows the modification of inherited methods, with some limitations. We assume that the reader is familiar with the use of procedures and functions. Note that the example is not meant to be a rigorous example of the syntax of TOIL. In fact, some constructs used in the sample program, such as the user-defined type ColorType, are not available with the current implementation. However, it should provide an initial introduction to the flavor of object-oriented programming.

```
program Points;

const  PointClass =
         class
           var  x = 0 : num;
                y = 0 : num
           methods
                GetX = Function():num
                       begin
                          return self.x;
                       end;
                GetY = Function():num
                       begin
                          return self.y;
                       end
                Move = Procedure(NewX:num,NewY:num)
                       begin
                          self.x := NewX;
                          self.y := NewY;
                       end
         end;
       ColorPointClass =
         extend PointClass with      -- inheritance!
           var
                c = Black : ColorType
           methods
                GetC = Function():ColorType
                       begin
                          return self.c;
                       end;
                SetC = Procedure(NewC);
                       begin
                          self.c := NewC;
                       end
         end

var    point       : PointObjectType
       color_point : ColorPointObjectType
begin
       point := new PointClass;
       color_point := new ColorPointClass;
       color_point <= Move(10,10);     -- using an inherited method
       print (point <= GetX);          -- should produce a 0
       print (color_point <= GetX);    -- should produce a 10
end.
```

Figure 2.1: Sample TOIL program

Chapter 3

A Survey of Other Object-Oriented Languages

Despite their many successes, most popular object-oriented languages have been unable to achieve type safety. The most common way to break the type system of an object-oriented language is to send a message to an object which does not know how to implement it, resulting in a run-time type error. Type safety is clearly desirable, for large scale projects in particular, since it can radically increase the likelihood that a program is correct. Most of the errors in a program will tend to be caught at compile time by a safe static typing system, making debugging easier and increasing reliability.

Unfortunately, object-oriented features such as inheritance and subtyping make type safety very difficult to achieve, and in fact most object-oriented languages currently in use fail to achieve it. Some languages have obviously abandoned type safety as a goal and therefore make no effort to maintain it in the use of object-oriented features, while others fall victim to somewhat subtle counterexamples. None, however, manage to support a full array of object-oriented features as well as type safety.

3.1 The "Don't Cares"

A fairly large class of languages have simply avoided the entire issue of type safety by making no attempt to preserve it. Despite the fact that this choice places a large burden in the lap of the programmer in terms of debugging and ensuring reliability, several of the most popular object-oriented languages in use today fall into this class. For example, in normal C, a safe type system is so little of a priority that values can be arbitrarily cast to any type. The type system which results is so loose that it makes little sense to attempt to ensure that the object-oriented features added in C++ ([Str86]) are type safe. In fact,

no such attempt was made. Simula ([Kir89]), an early object-oriented language, provides a another example. Object variables in Simula are essentially dynamically typed, meaning that the type of the object stored in the variable is kept track of at run-time with a tag, rather than being determined statically at compile-time. Furthermore, the tags are not even carefully checked at run-time, so Simula's type system is relatively weak. We can easily produce run-time type errors. Smalltalk([GR83]) is another example of an essentially untyped object-oriented language.

3.2 The Subtle Failures

Another class of languages fail to produce type-safety but make more of a concerted effort to at least come close. Eiffel 2.0([Mey88]) falls into this class. Eiffel runs into difficulties largely because of its simplifying assumption that the inheritance hierarchy is equivalent to the subtyping hierarchy. Recall that the subtyping relationship depends on the types of constructs, whereas inheritance is based solely on the implementation of classes. In TOIL, if one object inherits from another, then they may or may not prove to be subtypes. However, in Eiffel, the fact that one object inherits from another means that they are presumed to be subtypes as well. It is easy to write counter-examples which show that this is not necessarily the case. For example, recall our Point/ColorPoint example from Chapter 2, Figure 2.1. Now let us add the following method to the Point class:

```
Equal: function(comparePoint:MT):bool
       begin
          return ((comparePoint.x = self.x) &
                  (comparePoint.y = self.y));
       end;
```

`Equal` takes a parameter of type Point and checks to see whether it is the same as the current object. For ColorPoints, we want to add an additional condition:

```
ColorPointClass =
   update PointClass by
     method
     Equal: function(comparePoint:MT)
            begin
               return ((super.Equal(comparePoint) &
                       (comparePoint.c = self.c));
            end
   end
```

Remember that `super` refers to the methods of the parent class. Thus, the Equal method for ColorPoints uses the Equal method for Points to check whether the x and y coordinates are the same, and then performs an additional check on the new instance variable for color. Because of the use of the variable MT, the Equal method in ColorPoints now takes a parameter of type ColorPoint. So far everything we have done seems reasonable. However, programming languages which conflate subtyping with inheritance, and allow the definition of ColorPoint as we have written it, will fail to be type-safe. Since ColorPoints inherit from Points, these languages will presume that they are subtypes as well. But then consider the following function:

```
function Break(p:PointType):bool
begin
   return (p<=Equal(new(PointClass)));
end;
```

The procedure takes a parameter P of type PointType, and calls its Equal method with a brand new Point object as a parameter. In effect the procedure should return true if it is given a Point at (0,0). However, since we are assuming that ColorPoints are subtypes of Points, we can also send Break a ColorPoint CP as an actual parameter. Break will call the object's Equal method, which for ColorPoints includes a comparison of the color instance variable. However, the new Point object sent as a parameter contains no such variable, and therefore we will get a type error. Furthermore, this error cannot be caught at compile-time, leaving the language type-unsafe. The failure is due to the assumption that the ColorPoint class generates subtypes of PointType.

A later version of Eiffel, Eiffel 3.0 ([Mey92]), attempts to patch the hole. It performs a link-time global-level check which analyses all of the source code used in a particular program and ascertains whether any combination of statements (such as the Break function above) could be executed which might break the type system. The algorithm appears to work in that it will prevent run-time type errors, but this patchy approach leads to other difficulties. Since it is combinations of statements rather than single ones which produce system-level type errors, it is possible that a set of modules which all type correctly and work fine independently would suddenly break when placed together. This is a clearly undesirable result, especially considering that one of the main strengths of object-oriented languages is the ease with which implementations for different objects can repeatedly be glued together to form new systems. The system level type checking in Eiffel 3.0 is essentially a patch attempting to fix the problems generated by a much deeper issue, the confusion between subtyping and inheritance.

3.3 Limitations

Many object-oriented languages, including most of the ones which are not type-safe, have tight restrictions on the redefinition of inherited methods. Often, a method from a parent class will have to be reimplemented, not to prevent type errors, but to ensure that the method is doing what it naturally should do. The Equal method in ColorPoints is an example. It would be misleading not to redefine the Equal method used by Points because intuitively two ColorPoints should not be seen as equal unless they have the same color value as well as being in the same position. Some languages, such as Object Pascal ([TRea92]) and C++, require that the type of a redefined method in a subclass must be exactly the same as the type of the method in the parent class. Clearly, one does not want to allow arbitrary redefinitions of inherited methods, but an example will show why the above requirement is too restrictive. Suppose we have a Point class with a new method called Copy, which merely returns an exact replica of the object. So Copy is a method which returns something of type Point. The restrictions on inheritance in the languages mentioned above would require that the Copy method in the ColorPoint class return a value of type Point, which is not at all what we want. Somehow, we should be able to update the Copy method in our ColorPoint class so that it returns a ColorPoint.

Other object-oriented languages, including TOIL, require that the type of a redefined method be a subtype of the original type from the superclass. This technique works as desired in Eiffel. The Equal method could be redefined in ColorPoints to take a ColorPoint parameter instead of a Point parameter since ColorPoints are considered to be subtypes of Points. In TOIL, however, they are not considered subtypes. They cannot be subtypes because the Break function written above causes a run-time type error. Thus, it may seem that the restriction on redefinition of inherited methods is more constricting in TOIL than in other languages. However, this difficulty is avoided in TOIL through the use of the bound variable MT. This issue is discussed in greater depth in Section 4.1).

Trellis-Owl([SCB+86]) takes another approach, requiring that the types of classes defined using inheritance be subtypes of their parent class. Therefore, the assumption that the inheritance hierarchy and the subtype hierarchy are equivalent is justified. However, this restriction is overly tight. For example, the ColorPoint class we used before could not even be defined through inheritance in Trellis-Owl because the ColorPoint type does not turn out to be a subtype of the Point type. A similar approach is used in Modula-3 ([CDJ+89], [Aba92]) in order to ensure that objects defined using inheritance are always subtypes. Objects are treated as extensible records, where each method implicitly takes a value of the object type as a parameter. This parameter is used as *self*. Since inheritance is only permitted through extension, it always results in object types which are subtypes. Methods can be redefined using assignment, but in this case the types of explicit parameters to methods cannot be changed, and the implicit *self* parameter must be a supertype of

the old one. Hence, the ColorPoint class mentioned earlier would not be definable through inheritance in Modula-3 because the Equal method would have to be redefined to implicitly take a ColorPointType instead of a PointType, where ColorPointType is a subtype of PointType instead of a supertype.

Rapide 1.0 ([Tea92]) is like TOIL in that it separates the subtyping relation from the mechanism of inheritance. In fact, it goes even farther than TOIL, separating the interfaces (types) for objects from their generation completely. Rapide does not appear to provide a mechanism for inheritance in the implementation of objects. However, inheritance can be used in the definition of interfaces. Since implementations must be described explicitly without the benefit of inheritance, Rapide avoids the problems with inheritance mentioned above. However, it does not take advantage of inheritance as a mechanism for flexible code reuse. Rapide 1.0 appears to be type-safe. It is also a more complete language than TOIL, with a full implementation and support for abstract data types, type constructors, self-references within objects, and more. However, inheritance in Rapide is more awkward because it does not have any construct equivalent to MT in TOIL. Imagine that our Point object includes a function Distance which takes a Point P as a parameter and returns the distance between itself and P. In TOIL, the parameter type could be written as MT instead of using explicitly writing PointType. Therefore, when the method is inherited to the ColorPoint, it will continue to work when supplied a ColorPoint. However, in Rapide, in the interface for ColorPoints, the Distance method would have to be redefined to take a ColorPoint parameter. Thus, the inheritance mechanism in Rapide can be more awkward than TOIL's.

We have seen that many object-oriented languages, including the most popular ones, are either not type-safe or lack flexibility because of severe restrictions. TOIL manages to be both type-safe and provide a great deal of flexibility and power. TOIL separates the subtyping and inheritance hierarchies, and thus is able to place the least binding restrictions on inheritance which are necessary to keep it type-safe.

Chapter 4

Background

4.1 TOOPL

In [Bru93], Bruce presents a statically typed functional language, eventually named TOOPL for Type-Safe Object-Oriented Programming Language. TOOPL includes most of the features one would desire in an object-oriented language, while remaining type-safe. Since TOIL (for Type-safe Object-Oriented Imperative Language) is an imperative version of TOOPL, many of the base features are the same. Thus, most of the following discussion about TOOPL is relevant to TOIL as well.

The object-oriented constructs in TOOPL are classes and objects. Objects consist of data items (*instance variables*) and operations on that data (*methods*). Methods are accessed by sending the object a message. In the full version of the language, an object's instance variables are hidden – they cannot be changed except by sending the object a message and accessing a method which changes them. For example, it would be impossible to change the coordinates of a Point object without sending it a "Move" message, an operation which would presumably be provided by all Points. Actually, since TOOPL is functional, changing an instance variable results in the creation of a new object with the desired values instead of an actual change in the original object. Classes are essentially templates for the construction of objects, and contain initial values for instance variables as well as the bodies of methods. Thus, a single class can be used to create multiple objects, each of which will be initially the same.

TOOPL also supports inheritance. As we saw in Chapter 3, the unrestricted use of inheritance leads to holes in the type system. Therefore, inheritance in TOOPL is restricted in some ways specifically to preserve type safety. When used legally, it can be used to create hierarchies of related classes which share many of the same operations. For a sample inheritance hierarchy using classes which would hypothetically describe types of vehicles,

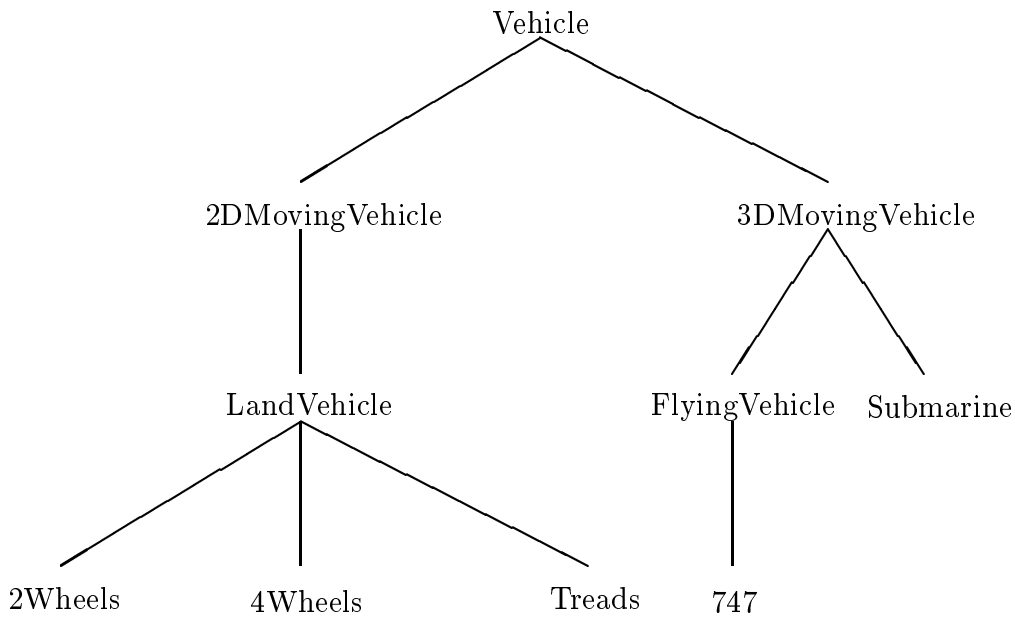

Figure 4.1: Sample inheritance hierarchy for vehicles

see Figure 4.1. When extending a parent class to form a "subclass" through inheritance, it is not necessary to include descriptions of methods which were already in the parent. Unless they are modified, inherited methods will automatically be present and accessible in the subclass. Intuitively, this technique can be thought of as code sharing. TOOPL's static typing and type safety combine to ensure that every object is guaranteed to know how to implement any message which is sent to it. In other words, consider any class which the type checker accepts as legal, and any object O created from that class. Then the only faulty messages sent to O are not included in its type, and can be caught at compile time. It is impossible to send O a message which will generate a run-time type error.

Like many other popular object oriented languages, TOOPL supports references within a method to the object which is executing it (*self*), that object's type (*MT* for My Type), and to the methods of the object's parent if it is constructed from an inherited class (*super*). These features are necessary in order to allow for the flexible reuse of inherited methods. For example, examine Figure 4.2. We want the Point class to include a method called Equal, a function intended to take some object as a parameter and determine if it is exactly the same as the current object. TOOPL requires that the types of parameters be specified by the programmer, so we could naively write down that the type of the parameter is just

the type of a Point object. But now consider a new class for ColorPoints made from the original Point class using inheritance. The Equal method no longer works properly since intuitively it should now take as a parameter a ColorPoint. We want to change the type of the parameter to ColorPointType, but this kind of change can result in holes in the type system. For a more complete description of why this is the case, see Section 5.3.

In TOOPL this problem is solved through the use of MT. Instead of specifying the type of the parameter to the Equal methods as a Point object, we allow it to be described using the bound variable *MT*. MT would be interpreted as the type of the current object, no matter how far removed through inheritance from the original code of the method itself. Thus, the Equal method will work even in a ColorPoint because MT will be interpreted as a ColorPoint. *Self* and *super* are also interpreted as bound variables – *self* refers to the current object itself, and thus can be used to get at the dynamic values of its instance variables, while *super* is a record containing the methods of the parent.

TOOPL also provides a set of rules which determine structurally whether a type σ is a subtype of another type τ. Recall that some of the languages presented in Chapter 3 identified subtyping with inheritance, resulting in holes in the type system. By separating the subtyping and inheritance relations, TOOPL avoids this pitfall. Since inheritance and subtyping are thus separated, it is possible that an extension of a class will not prove to be a subtype. On the other hand, it is equally possible that two different classes may produce objects which are subtypes, or even of exactly the same type. See Sections 5.3 and 5.4 for more details on subtyping and inheritance in TOIL.

The semantics of TOOPL are given in terms of a model of the bounded, higher-order, polymorphic lambda calculus. As we shall see only too soon, the semantics are extremely complex, though we believe that this complexity is inherent in the support of inheritance and subtyping. However, several extensions to the language such as recursion, unbounded parametric polymorphism, and higher-order types should be possible without much increase in complexity because of the nature of the model.

My initial work in conjuction with a summer research group on TOOPL at Williams College concentrated on implementing a prototype interpreter for TOOPL in Standard ML. Although the implementation to a large extent closely follows the abstract syntax and semantics provided in the language definition, it was useful in highlighting several theoretical and practical difficulties. Least upper bounds and greatest lower bounds were necessary to successfully type IF-THEN-ELSE expressions, which at first had seemed too trivial to merit close consideration. Also, it became clear that minimal types were problematic in TOOPL because of the self-referential nature of objects, and that type inference such as that used in ML would be difficult if not impossible for the same reason. Therefore, the language evolved to TOOPLE, which includes extra type information provided by the programmer in order to resolve these difficulties. The typing algorithm for TOOPLE to a large extent only has to confirm the type information given in the syntax of the program. This designation of

Naive Point class:

```
class
    var
        x:num;
        y:num
    methods
        getX :  func():num;
        getY :  func():num;
        Equal : func(p:PointObjectType):bool
end
```

Corresponding ColorPoint class:

```
class
    var
        x:num;
        y:num;
        color:num
    methods
        getX :  func():num;
        getY :  func():num;
        getC :  func():num;
        Equal : func(p:PointObjectType):bool
                    -- we want a ColorPoint parameter here instead
end
```

In TOOPL:

```
class
    var ...
    methods ...
        Equal : func(p:MT):bool  -- for both Points and ColorPoints
end
```

Figure 4.2: The use of MT in TOOPL

extra type information is also used in TOIL to avoid the problem with minimal types.

4.2 Justification for TOIL

The functional paradigm is appropriate for TOOPLE because its goal was to present the design of a type-safe object-oriented language with many of the features one would want, and to demonstrate the semantic complexity inherent in those features. As discussed above, some extensions to the language such as recursion and polymorphism should be relatively easy. Therefore, the major shortcoming of TOOPLE is the fact that it is functional. TOIL aims to reap the benefits of imperative languages without sacrificing any of the achievements of TOOPLE.

Imperative languages have several advantages over functional ones. The most obvious is efficiency. Although compiler technology for functional languages such as ML has advanced significantly, they are still slow relative to imperative languages. This difference in efficiency is especially relevant to developmental languages such as TOOPLE and TOIL since their compilers (when, if ever, they are implemented) will realistically not include large amounts of optimizations. Imperative languages tend to be more efficient than functional ones because of their differing approaches to data. Pure functional languages deal with data as immutable entities which can be accessed and passed from function to function without being changed. If large data structures are being used, then in the worst case the entire structure will have to be replicated every time it is updated. Of course, most functional languages include impure features which permit the kind of memory-specific updating used in imperative languages. However, the inclusion of these features destroys many of the nice semantic properties, such as lack of side effects, which made the functional paradigm such an attractive choice for TOOPLE.

Although many find programming in functional languages easier and more natural in many ways than programming in imperative languages, some key kinds of algorithms and data structures are difficult to implement in the functional paradigm. Some examples include:

1. I/O, which is difficult to simulate naturally without the use of side effects
2. Large, updateable data structures such as arrays and heaps
3. Tight, efficient loops, due to the overhead of calling functions in functional languages

For these reasons, imperative languages are seen as more applicable to real-world programming problems. Therefore, the adaptation of TOOPLE to an imperative version, TOIL, is a significant stage in the development of a type-safe object-oriented language

which could be interesting not only for its theoretical features, but also for its potential use in real-life applications.

4.3 Issues in Imperative Languages

The most noticeable difference between functional and imperative languages is the use of a memory to store information. All of the constructs in a functional language are expressions which evaluate to values which can be used in other expressions. There is no sense of temporality, as the entire program is one large expression which must be evaluated, where the order in which sub-expressions are evaluated does not matter. Imperative languages, however, include variables which may refer to different values at different points in the program, and commands which do not return any value, but change the memory store in some way which may affect the rest of the commands or expressions in the program. Furthermore, expressions may themselves have side-effects in the memory store. Thus, a mutable memory, or state, of some sort must be modelled in the semantics of TOIL. This addition leads to considerable complications, even though the commands which are added, such as an assignment operator and WHILE loops, may seem simple enough intuitively.

The presence of variables in TOIL requires some changes in the type system as well. A new type for variables, a reference type, must be added, and its behavior must be defined in various contexts:

1. How and where are variables declared?
2. How do reference types behave in the subtype relation?
3. How should variables be accessed and changed?
4. How should a reference type be interpreted semantically?

The reference type is used for the instance variables of objects as well as for regular user-defined variables. Therefore, we need to be very careful about our solutions to the above problems since they will be reflected in the behavior of objects. In particular, we need to make sure that subtyping (if any) for variables works out correctly with subtyping for objects. Furthermore, variables need to be scoped carefully so that they are accessed correctly, and do not interfere with each other. For example, it would be disastrous if an instance variable, x, for some object were confused with a local or global variable coincidentally named x as well.

The addition of variables and an assignment operator also necessitates a design decision about how to deal with memory. Variables are unique in that they can denote two different values, the address to which they refer and the value stored at that address. The effect

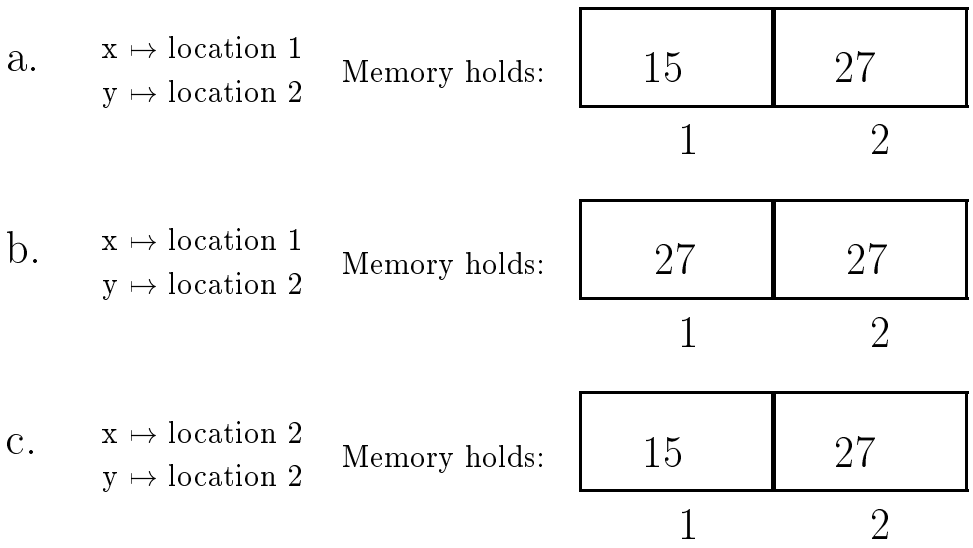

Figure 4.3: Two approaches to assignment.
a. Status after the execution of `x:=15; y:=27;`.
b. Status after `x:=y;` using assignment by copy.
c. Status after `x:=y;` using assignment by sharing.

of assignment depends on how and when these values are used. For one interpretation, an assignment statement of the form `x:=y` leads to the sharing of data, while for another it leads to a copying of data. More specifically, if y is interpreted as a location, then x and y will share a value which is stored at a single location in memory. Any operation affecting one variable's value would affect the value of the other as well. On the other hand, if variables are interpreted as locations only on the left hand side of assignment statements, the assignment results in a copying of a value from one location to another. Both variables will refer to separate values and later operations which affect the value of one variable will not affect the value of the other variable. See Figure 4.3 for a graphical demonstration of the difference between assignment by sharing and assignment by copy. In TOIL, the latter approach is used except in the case of objects.

Objects are modelled as references to a location in memory where the actual information for the object resides (see Figure 4.4). Since the value of an object variable is a location in memory, two object variables can refer to the same actual object, much as the variables x and y refer to the same location in part c of Figure 4.3. Therefore, changes directed at one object variable will be reflected in the other as well. It is necessary to model objects as references because the existence of subtyping precludes any upper bound on the amount of memory which must be allocated to an object variable. Even though a variable may be

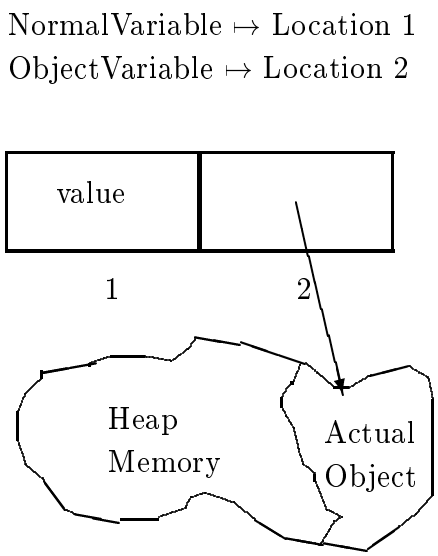

Figure 4.4: A sample object in memory compared to a regular variable.

declared as a Point, it is possible that its value at run-time may be a subtype of Point (e.g. a ColorPoint), which may have more methods and instance variables than a Point at run-time. Therefore, it is very difficult to allocate enough space on a run-time stack for all of the possible values of an object variable. Instead, the object variable will be a reference to some part of memory which can be dynamically expanded to any necessary size, presumably using heap-based memory allocation. Note that in Figure 4.4, the sample object can grow at run-time. The sizes of objects are bounded only by the free heap memory available. [1]

Finally, TOIL's imperative features require a semantic change which itself is relatively minor but leads to a whole series of problems and fundamental changes in the semantics of objects. The problem arises from our intuition that all objects created from the same class should initially be the same. In TOOPL this is the case because the initial values for the instance variables of the object are frozen in the class. However, if we model TOIL in the same way as TOOPL it will be possible to set the instance variable for an object equal to an expression involving a global variable. The question is whether the value of this expression should be computed when the class is defined or when an object is created. If we evaluate the expression when an object is created, and the global variable used in the expression

[1]The same problem with non-bounded sizes of types applies to TOOPL's record type. Since records are essentially non-recursive objects anyway, we decided to omit the record as such in TOIL.

```
x:=1;
uhoh = class
          var
            a = x   -- this instance variable may change value
end;
object1 = new (uhoh); -- ‘‘a’’ in object1 is set to x = 1
x:=99;
object2 = new (uhoh); -- ‘‘a’’ in object2 is set to x = 99
```

Figure 4.5: TOIL program fragment illustrating problem with lazy evaluation

is changed at some point, the initial values of instance variables for objects created before the change will differ from those of objects created after the change. See Figure 4.5 for a program fragment illustrating this difficulty.

The solution lies in the way we evaluate classes. TOOPL does not evaluate the instance variables or methods in a class (or an object for that matter) until they are absolutely required. Thus, the value of each instance variable is not inspected until an actual object is created. Similarly, methods in an object are not evaluated until an appropriate message is actually sent to the object. This technique of not evaluating expressions until they are required is called "lazy evaluation". Lazy evaluation has no affect on the results of a computation in a pure functional language such as TOOPL because there are no side effects. The time at which an expression is evaluated will not affect its resulting value. However, this property is lost in an imperative language with side effects. In order to solve the problem outlined above and prevent instance variables in classes from changing, TOIL utilizes "eager evaluation". As one might expect, this simply means that instance variables and methods are evaluated as soon as possible, with no additional work performed when they are actually requested. Figure 4.6 demonstrates the results of eager evaluation on the previous example.

Unfortunately, this move to eager evaluation necessitates a whole series of changes in the implementation and semantics of TOIL. The reason is obvious from a quick analysis of what happens when we declare a class and then create an object from it. The methods of the object were evaluated as soon as the class was declared. Therefore, to send a message we should be able to simply select the correct method and return it without performing any further evaluation. This solution proves to be too simple, however, because of the use of implicitly recursive bound variables such as *self* and *super*. If the functional abstraction for each method is evaluated when the class is created, how can it be expected to have any information about the object it is selected from? In particular, we should be able to

```
x:=1;
uhoh = class
          var
            a = x   -- this instance variable is frozen at the
                    -- current value of x, 1
end;
object1 = new (uhoh); -- ‘‘a’’ in object1 is set to 1
x:=99;
object2 = new (uhoh); -- ‘‘a’’ in object2 is set to 1
```

Figure 4.6: Previous example, now correct with eager evaluation

access the instance variables of the object through the use of *self*, but it is not clear how the method can know about *self* at all. The solution to this problem is somewhat involved, and will be further examined in Chapter 6.

Despite this list of difficulties which make TOIL seem perhaps too toilsome, it leads to a substantial increase in power and efficiency compared to TOOPL, while retaining its type-safe properties.

Chapter 5

Type Checking in TOIL

5.1 Types in TOIL

Almost all programming languages organize collections of data in terms of "types". Since at the lowest level every data object consists of a series of 1s and 0s in memory, the data's type helps indicate how the bits should be interpreted – as an integer, a real, a boolean, etc. In statically typed imperative languages, each variable also has a type associated with it, and only data which can be shown to have the same type are allowed to be stored in the variable. There are also more complicated types such as functions, classes, and objects. What follows is a formal definition of the types in TOIL, followed by a brief informal description of each one. [1]

Definition 5.1.1 *Let $\mathcal{V}$ be an infinite collection of type variables, $\mathcal{L}$ be an infinite collection of labels, and $\mathcal{C}$ be a collection of type constants which includes at least the type constants Bool and Num. The type expressions with respect to $\mathcal{V}$ and $\mathcal{C}$ are defined as follows:*

1. *If $t \in \mathcal{V} \cup \mathcal{C} \cup$ {COMMAND,UNIT,PROGRAM, $\bot, \top$} then t is a type expression.*
2. *If τ is a type expression, and $\tau \notin$ {COMMAND,UNIT, PROGRAM,$\bot, \top$} then* ref τ *is considered a type expression.*
3. *If $\sigma_1 \ldots \sigma_n$ and τ are type expressions, then so is $func(\sigma_1, \ldots, \sigma_n): \tau$.*
4. *If $m_1, \ldots, m_n \in \mathcal{L}$ and $\tau_1, \ldots, \tau_n$ are type expressions, then $\{m_1: \tau_1; \ldots; m_n: \tau_n\}$ is a (record) type expression.*

[1] For a more in depth introduction to the evolution and uses of types in programming languages, as well as their basis in the typed λ calculus, see [CW85].

5. *Let σ and τ be record type expressions and $MT \in \mathcal{V}$. Then (σ, τ), $TClass(MT)(\sigma, \tau)$, and $TObj(MT)\tau$ are type expressions. MT is considered to be a bound variable in the last two type expressions.*

The first group of types is relatively self-explanatory. Booleans and numbers (integers) are currently the only type constants, but this list could easily be extended to include others such as reals. In fact, the current implementation of a TOIL interpreter contains a STRING type constant which was added with little difficulty. COMMAND is the type of an imperative command expression. UNIT is a trivial type used when typing parameterless functions. PROGRAM is the type of an entire well-typed TOIL program. Reference types are simply the types of variables, so X: *ref* τ implies that X is a variable containing data of type τ. In the implemented language and in most of our examples, we omit the *ref* in variable declarations since it can always be inferred from the context. Thus, we would write `var` $X: \tau$, and a type of *ref* τ would be inferred for X automatically.

Function types are designated $func(\sigma_1, \ldots, \sigma_n): \tau$, where $\sigma_1, \ldots, \sigma_n$ are the types of the function's n parameters, and τ is the type returned by the function. Parameterless functions are modelled as taking only one parameter, of type UNIT, although in the actually implementation one would simply write `F()` instead of `F(Unit)` (where Unit is a value of type UNIT). Functions which return no value, commonly known as procedures, return the unique value of type COMMAND which is essentially ignored. Thus, although procedures as such are not mentioned in most of the formal definitions of the syntax and semantics of TOIL, they are actually present as a subset of functions.

In Chapter 4, some of the problems inherent to the implementation of imperative languages were described, including a rationale for the omission of records in TOIL. However, record types are useful in the construction of classes and objects since methods and instance variables can easily be modelled as records. Thus, we include the record type in definition 5.1.1 above although the current definition of TOIL does not contain records themselves. A record is simply a labelled tuple, or a group of named data items lumped together. The record type $\{m_1: \tau_1; \ldots; m_n: \tau_n\}$ simply implies that the piece of data associated with the label m_i has the type τ_i, for $1 \leq i \leq n$.

The remaining three types correspond to TOIL's object oriented features. The most intuitive is the class type: $TClass(MT)(\sigma, \tau)$ simply refers to a class whose record of instance variables has type σ and whose record of methods has type τ. The MT (for My Type) is parameterized to indicate that it is a bound variable within σ and τ.

The final two types included in TOIL follow from the fact that we wish to make an object's instance variables invisible to the outside world. The pair type (σ, τ) refers to the type of an object when seen from the inside, and so it, like the class type, includes both instance variables and methods. The object type $TObj(MT)\tau$ corresponds to the type of an object seen from the outside, and so contains only the types of its methods.

TOIL includes a `NIL` value which corresponds to an empty object without any methods. `NIL` has the type $\bot$. As we will see in the next section, $\bot$ is a subtype of every object type and therefore can be used wherever any object is expected. However, requesting a method from `NIL` will obviously result in an error. However, this kind of error does not compromise our promise of type safety because it is due to an empty object instead of a type error. One way to think of `NIL` is as an object with every possible method, each of which generates an error. It is used to provide recursive data structures such as trees and linked lists with base cases (i.e. leaves). The top type $\top$ is the opposite of the bottom type, in that every object type is a subtype of $\top$.

This hiding of instance variables is advantageous for intuitive and practical reasons. It provides a high level of data abstraction, preventing the user from viewing the details of how the object is implemented. Furthermore, it prevents the programmer from directly changing the values associated with an object, necessitating the use of methods provided. If some instance variables are kept entirely private to the object, it is impossible for the programmer to break the abstraction barrier. If the designer of the object wishes to allow some sort of access to one or more of its variables, he/she can simply provide a method to return its current values and/or allow a new value to be placed in it. Since the object type avoids any mention of how it is implemented, it can be seen as a public interface for the object. An object's type simply indicates the operations available to anyone who wants to use the object. For example, consider two different Point classes which are implemented in different ways, one in polar coordinates and the other in Cartesian coordinates. If both classes provide methods to return the position of the object in the Cartesian coordinate system, the types could be as follows:

PointClass1 : $TClass(MT)(\{r\colon num; \theta\colon num\}, \{GetX\colon func()\colon num; GetY\colon func()\colon num\})$

PointClass2 :
$$TClass(MT)(\{x\colon num; y\colon num\}, \{GetX\colon func()\colon num; GetY\colon func()\colon num\})$$

But objects created from both classes would have the type

PointObject1OR2 : $TObj(MT)\{GetX\colon func()\colon num; GetY\colon func()\colon num\}$.

In terms of interfaces, both objects would be treated exactly the same even though they are implemented differently, thus providing a high level of data abstraction.

5.2 The type constraint system

All of our proofs about the types of constructs and whether they are subtypes must be based on some context. This context may include information about relations between type variables or constants. For example, the initial context may include a clause which states that the type *Real* is subtype of *Number*, so that wherever a *Number* is expected a *Real* can be used. Note, however, that our current implementation includes only the type constants *Number*, *Boolean*, and *String*, none of which happen to be related. However, as we shall see shortly, constraints are added to the context by the type-checking rules. We define our context C as a type constraint system as follows:

Definition 5.2.1 *Relations of the form $\sigma \leq \tau$ and $\sigma \leq_{meth} \tau$, where σ and τ are type expressions, are said to be* type constraints. *A* type constraint system *is defined as follows:*

1. *The empty set, ϵ, is a type constraint system.*
2. *If* C *is a type constraint system and $t \leq \tau$ is a type constraint such that the type variable* t *does not appear in* C *or τ, then $C \cup \{t \leq \tau\}$ is a type constraint system.*
3. *If* C *is a type constraint system, τ is of the form $TObj(MT)\sigma$, and $t \leq_{meth} \tau$ is a type constraint such that the type variable* t *does not appear free in* C *or τ, then $C \cup \{t \leq_{meth} \tau\}$ is a type constraint system.*

5.3 Subtyping in TOIL

The concept of object interfaces is crucial to the understanding of the relationship between subtyping and inheritance in TOIL. As we saw in Chapter 3, many object-oriented languages equate the two, resulting in holes in their type systems.

Recall that one type σ is a subtype of another type τ (written $\sigma \leq \tau$) if everywhere that a construct of type τ is expected, a construct of type σ can be substituted. The formal subtyping rules will be presented in Section 5.3.1, and it may be useful to refer to them while going through the examples and intuition below. Also, it should be noted that [Pie92] presents an alternative formal model of subtyping and inheritance.

5.3.1 Subtyping rules and axioms

Here we present the formal rules for the subtype relation. Their justifications are provided in our informal discussion which follows. C is a type constraint system which contains assumptions about types. A formal definition for C will be provided in Section 5.2. The symbol $\vdash$ stands for "shows that". The hypotheses of each rule are listed above the line,

and the conclusion is written below the line. For example, the $Trans(\leq)$ rule should be read:

"If under the assumptions in C one can show that γ is a subtype of σ and that σ is a subtype of τ, then under the same assumptions one can show that γ is a subtype of τ."

We now present the formal subtyping rules. The reader should refer to them while reading the intuition behind them in the next section.

$Refl(\leq)$ $$C \vdash \tau \leq \tau,$$

$Var(\leq)$ $$C \cup \{t \leq \tau\} \vdash t \leq \tau,$$

$Trans(\leq)$ $$\frac{C \vdash \gamma \leq \sigma,\ C \vdash \sigma \leq \tau}{C \vdash \gamma \leq \tau},$$

$Func(\leq)$ $$\frac{C \vdash \sigma_i' \leq \sigma_i \ for\ 1 \leq i \leq n,\ C \vdash \tau \leq \tau'}{C \vdash Func(\sigma_1 \ldots \sigma_n) :\ \tau\ \leq\ Func(\sigma_1' \ldots \sigma_n') :\ \tau'},$$

$Rec(\leq)$ $$\frac{C \vdash \sigma_j \leq \tau_j, \quad \text{for } 1 \leq j \leq k \leq n}{C \vdash \{m_1 \colon \sigma_1; \ldots; m_k \colon \sigma_k; \ldots; m_n \colon \sigma_n\} \leq \{m_1 \colon \tau_1; \ldots; m_k \colon \tau_k\}},$$

$Pair(\leq)$ $$\frac{\sigma \leq \sigma',\ C \vdash \tau \leq \tau'}{C \vdash (\sigma, \tau) \leq (\sigma', \tau')},$$

$Obj(\leq)$, $$\frac{C \cup\ \{s \leq t\} \vdash \tau[s/MT] \leq \tau'[t/MT]}{C \vdash TObj(MT)\tau \leq TObj(MT)\tau'}.$$

where neither s nor t may occur free in C, τ, or τ',

$Bot(\leq)$ $$C \vdash \bot \leq TObj(MT)\tau,$$

$Top(\leq)$ $$C \vdash TObj(MT)\tau \leq \top,$$

5.3.2 Intuition behind subtyping

As one would expect, the subtype relation is reflexive and transitive, and these properties are reflected in the first few formal rules. The $Var(\leq)$ rule simply says that a type variable t is a subtype of some type τ if there is a constraint in the constraint system which says so.

Perhaps the most intuitive non-trivial subtyping relation corresponds to that for records. For example:

$$\{x\colon num; y\colon num\} \leq \{x\colon num\}$$

because whenever the latter is expected, we can simply insert the former and ignore the y field. Furthermore, two records can be subtypes when they share a field. In this case, the subtyping relation is used recursively: if the types held in the shared field are subtypes, then the records are subtypes as well. Thus,

$$\{x\colon \sigma\} \leq \{x\colon \tau\}$$

if $\sigma \leq \tau$. Finally, it should be noted that the order in which fields are listed is unimportant – records should be treated as sets.

The subtype relation can be more formally defined by requiring that if $\sigma \leq \tau$, there must exist a canonical function which converts elements of σ to elements of τ. The exact nature of these coercion functions will not be discussed here, but for a more in depth discussion see [BTCGS91] or [BL90]. For our purposes, it is enough to realize that any type can always be treated as any one of its supertypes, simply by applying the coercion function. In fact, the subsumption rule in the type checking rules, which will be introduced in Section 5.5.2, makes this property explicit.

The subtyping relation for functions is somewhat more complicated, but makes sense when it is viewed in the light of coercion functions. See Figure 5.3.2 for a graphical illustration of the following example. Consider the function f with type $func(\sigma)\colon \tau$. To construct a function $f'\colon (func(\sigma')\colon \tau')$, where $func(\sigma')\colon \tau' \leq func(\sigma)\colon \tau$, we should make sure that f' will work in any situation that f would have worked in. First of all, any parameter of type σ which is legal for f should also be legal for f'. Therefore, we need a coercion function which converts values of type σ to values of type σ'. Thus, we can ensure that any parameter which is in the domain of f will also be in the domain of f'. Secondly, we must also ensure that any value of type τ' returned from f' could also have been returned from f. We can achieve this by providing a coercion function which converts values of type τ' to values of type τ. Consider a parameter $p\colon \sigma$. If we apply f to p, we get a value $r\colon \tau = f(p)$. We can use f' in place of f by coercing p from σ to σ', applying f', and then coercing the result r' from τ' to τ. Thus, $r'\colon \tau =$Coerce(τ', τ) of $f'($Coerce(σ, σ') of $p)$. The existence of coercion

Figure 1.1:

both objects with s and t, respectively. [2] To make the assumption, we throw a constraint into C. Essentially, the use of s and t allows us to differentiate between the MTs of the two objects, which represent different types but happen to be similarly named.

It is easy to show that an attempt to prove that two object types are subtypes can fail when we assume that their MTs are subtypes. The failures are due to that fact that MT may appear in both the covariant and the contravariant position in the function spaces of methods. For example, consider the following extension of our Point object from earlier:

PointObject : $TObj(MT)$
$\{GetX\colon func()\colon num; GetY\colon func()\colon num; Eq\colon func(MT)\colon bool\}$

ColorPointObject : $TObj(MT)$
$\{GetX\colon func()\colon num; GetY\colon func()\colon num; GetC\colon func()\colon num; Eq\colon func(MT)\colon bool\}$

where a ColorPoint is simply a Point extended to include a Color variable which is a number, and a method to access it. If we assume that $MT_{ColorPoint} \leq MT_{Point}$, the types of the Eq methods for the two objects will fail to be subtypes since MT is used in the contravariant (domain) position. Thus, ColorPoints are not subtypes of Points, even though they may be related through inheritance. It is this counterexample which typically breaks the type systems of the object-oriented languages which equate the subtype and inheritance hierarchies without including heavy restrictions on the use of inheritance.

The last two axioms dealing with top ($\top$) and bottom ($\bot$) types are added to deal with NIL. NIL will end up having type $\bot$. Since we want to be able to use the empty object NIL wherever an object would be expected, we make $\bot$ a subtype of every object type. The top type was added largely to allow functions to be written which take any object type as a parameter. For example, a function which compares an object variable with NIL to determine if there is actually an object there before sending it messages is highly desirable. It should be noted that the top and bottom types, like records, the pair type, and the reference type, are not directly available to the programmer. Instead, they are used implicitly throughout by the type-checker. Thus, the programmer can use NIL, which has type $\bot$ without being able to actively declare a value of that type. Similarly, reference types are used implicitly every time a variable is declared, but cannot be used explicitly.

There is no non-trivial subtyping for class types or reference types. Subtyping fails for classes for complex reasons which we shall avoid going into here (see [Bru93] for more details). However, an example and our knowledge of subtyping of functions should be enough to demonstrate why the obvious subtyping rule fails for references. Consider the pseudocode in Figure 5.2.

[2] The notation used, x$[a/b]$, means "x with all occurrences of a replaced by b".

```
A = class          -- class with one method and no instance variables
       methods
          m:func():num
    end;
B = class          -- class with no methods or instance variables
    end

var a: TObj(MT){m:(func():num)};  -- holds an object made from class A
    b: TObj(MT){};                -- holds an object made from class B

b := new B    -- legal
```

Figure 5.2: How subtyping of references fails.

Let us assume that references types can be subtyped using one intuitive rule. The rule states that for two types σ and τ, *ref* $\sigma \leq$ *ref* τ if we can show that $\sigma \leq \tau$. It can easily be shown using the subtype rule for objects that:

`TObj(MT){m:(func():num)}` $\leq$ `TObj(MT){}`.

Thus, we can show that the type of a from the figure is a subtype of the type of b. Now, however, look what happens if we can use a wherever an object of type $TObj(MT)\{\}$ is expected, including in the assignment statement from Figure 5.2:

```
a := new B; -- legal since a's type is a subtype of the type of b
```

After this assignment, a holds an object made from the class B, which has no methods. Therefore, if we try to send a the message m, it will not know what to do and generate a run-time error. This behavior is fatal to our type system since, according to the type of a, the message should have been legal.

The above counterexample shows that allowing subtyping of references will create a hole in our typing system. Furthermore, the problem cannot be fixed by switching the direction of the subtyping rule, so it is not merely that the reference type is contravariant. What causes this hole? The problem lies in the two uses of variables. They need to support "get" and "set" operations: one to retrieve the current value stored in the variable, and the other to set the value. Of course, almost all programming languages, including TOIL, hide this inherent complexity and figure out which operation is required from the context of the

expression. An exception is ALGOL 68([Tan76]),which makes the distinction between the operations explicit. For a more complete discussion of the distinction, see [Rey81]. At any rate, the types of the two operations for a variable of type $ref\ \tau$ are as follows:

get x : func():τ
set x : func(τ):UNIT

Since both operations must be supported, to show that $ref\ \sigma \leq ref\ \tau$ we must first show that:

func():$\sigma \leq$ func():τ *and*
func(σ):UNIT $\leq$ func(τ):UNIT

Note that τ is used in both the domain of *get x* and range of *set x* in the two operations. It is for this reason that subtyping for references fails. Recall from our discussion of subtyping that the subtyping relation for function types is covariant in the range and contravariant in the range. Therefore, to show the above, we would have to show that $\sigma \leq \tau$ and also that $\tau \leq \sigma$. Such a case can only occur if in fact $\sigma = \tau$. This logic shows that subtyping for references only works when the two types being referenced are exactly the same. There is no need to include a rule specific to references, however, since the reflexivity rule takes care of this trivial case.

5.4 Inheritance in TOIL

We shall define a relation $\leq_{meth}$ for two object types A and B such that $A \leq_{meth} B$ if the class from which A was created could have been defined through inheritance from the class which created B. Although we have not yet shown exactly how inheritance can be accomplished in TOIL, for now it is enough to know that a class can be modified through inheritance to produce a subclass. The $\leq_{meth}$ relation extends throughout the inheritance hierarchy. From our Vehicle example in Figure 4.1, we can conclude that:

$2DMovingVehicleObject \leq_{meth} VehicleObject$,
$TreadsObject \leq_{meth} VehicleObject$,
$SubmarineObjectVehicleObject$,

and so on. Note, however, that the $\leq_{meth}$ relation is **not** equivalent to the subtype relation $\leq$. Recalling our Point/ColorPoint example from Section 5.3, ColorPointType $\leq_{meth}$ PointType, but ColorPointType $\not\leq$ PointType.

Next we present the formal rules for the inheritance relation. They are actually very simple. The first is similar to the $Var(\leq)$ rule for subtyping in that it accounts for infor-

mation about inheritance of type variables held in C. The second corresponds to reflexivity, while the third corresponds to a limited form of transitivity.

$$Var(\leq_{meth}) \qquad C \cup \{t \leq_{meth} \tau\} \vdash t \leq_{meth} \tau,$$

$$Refl(\leq_{meth}) \qquad C \vdash TObj(MT)\tau \leq_{meth} TObj(MT)\tau,$$

$$Trans(\leq_{meth}) \qquad \frac{C \vdash \gamma \leq_{meth} TObj(MT)\tau, C \vdash \tau \leq \tau'}{C \vdash \gamma \leq_{meth} TObj(MT)\tau'},$$

Note that the following rule can be deduced from $Refl(\leq_{meth})$ and $Trans(\leq_{meth})$:

$$\frac{C \vdash \tau \leq \tau'}{C \vdash TObj(MT)\tau \leq_{meth} TObj(MT)\tau'}$$

This rule provides a link between subtyping and inheritance. It looks a lot like the rule which we naively wanted to use for subtyping of objects. However, here the rule only reflects the restrictions we will have to make on inheritance. Specifically, in order to define a subclass B from an original class A, we will have to ensure (with no additional assumptions) that the type of B's methods is a subtype of the type of A's methods.

5.5 Typing rules and abstract syntax

5.5.1 More contexts

We need only one more definition before we can finally proceed to the type checking of terms. In both functional and imperative languages, variables are declared and then used later. In functional languages, the declaration is made implicitly through formal parameters of functions, while in imperative languages declarations are made either through formal parameters or explicitly in a declaration section. In both cases, in strongly typed languages such as TOIL and TOOPLE, the programmer is required to designate the type of each variable. The type checker must somehow remember the declared type of each variable. We shall use another context, similar to our type constraint system C, to remember the type associated with each variable. This context is called a *type assignment*.

Definition 5.5.1 *A type assignment* E *is a finite set of associations between variables and type expressions of the form* $x{:}\,\tau$*, where each* x *is unique in* E*. If the relation* $x{:}\,\tau \in E$*, then we write* $E(x) = \tau$*.*

5.5.2 Type assignment rules and axioms

Here we present the formal type assignment rules and axioms. The reader should refer to them while reading the intuition behind them in the section which follows.

Type Assignment Rules (Program):

$$Prog \qquad \frac{C_0, E_0 \vdash Block\colon COMMAND}{C_0, E_0 \vdash program\ p\ ;\ Block.\ :PROGRAM}$$

where $Block =$ const (*constant declarations*)
var (*variable declarations*)
begin (*statement list*) return (*expression*);
end

Type Assignment Rules (Blocks):

$$ConstDecl \qquad \frac{C, E \vdash M\colon \tau,\quad C, E \cup \{X\colon \tau\} \vdash rest\colon \sigma}{C, E \vdash const\ X = M; rest\ \ :\sigma}$$

$$VarDecl \qquad \frac{C, E \cup \{X\colon ref\ \tau\} \vdash rest\colon \sigma}{C, E \vdash var\ X\colon \tau; rest\ \ :\sigma}$$

$$Block \qquad \frac{C, E \vdash S\colon \text{COMMAND},\quad C, E \vdash M\colon \sigma}{C, E \vdash begin\ S\ return\ M\ :\ \sigma;\ end\ \ :\sigma}$$

Type Assignment Rules (Commands):

$$Assn \qquad \frac{C, E \vdash X\colon ref\ \tau,\ C, E \vdash M\colon \tau}{C, E \vdash X\colon = M\colon \text{COMMAND}}$$

$$Cond \qquad \frac{C, E \vdash B\colon Bool,\ C, E \vdash S\colon \text{COMMAND},\quad C, E \vdash T\colon \text{COMMAND}}{C, E \vdash if\ B\ then\ S\ else\ T\ end : \text{COMMAND}}$$

$$While \qquad \frac{C, E \vdash B\colon Bool,\ C, E \vdash S\colon \text{COMMAND}}{C, E \vdash while\ B\ do\ S\ end : \text{COMMAND}}$$

$$StmtList \qquad \frac{C, E \vdash S\colon \text{COMMAND},\ C, E \vdash T\colon \text{COMMAND}}{C, E \vdash S;\ T\colon \text{COMMAND}}$$

Type Assignment Rules (Expressions):

$$Nil \qquad C, E \vdash NIL\colon \bot$$

$$Var \qquad C, E \vdash X\colon \tau \text{ if } E(X) = \tau$$

$$Ref \qquad \frac{C, E \vdash X\colon ref\ \tau}{C, E \vdash val\ X\colon \tau}$$

$$Function \qquad \frac{C, E \cup \{v_1\colon \sigma_1, \ldots, v_n\colon \sigma_n\} \vdash Block\colon \tau}{C, E \vdash function\ (v_1\colon \sigma_1; \ldots; v_n\colon \sigma_n)\ Block\colon\ (func(\sigma_1, \ldots, \sigma_n)\colon \tau)}$$

$$FuncAppl \qquad \frac{C, E \vdash M\colon func(\sigma_1, \ldots, \sigma_n)\colon \tau,\quad C, E \vdash N_i\colon \sigma_i \text{ for all } 1 \leq i \leq n}{C, E \vdash M(N_1, \ldots, N_n)\colon \tau}$$

$$Record \qquad \frac{C, E \vdash e_i\colon \tau_i \text{ for all } 1 \leq i \leq n}{C, E \vdash \{m_1 = e_1, \ldots, m_n = e_n\}\colon \{m_1\colon \tau_1; \ldots; m_n\colon \tau_n\}}$$

$$Proj \qquad \frac{C, E \vdash e\colon \{m_1\colon \tau_1; \ldots; m_n\colon \tau_n\}}{C, E \vdash e.m_i\colon \tau_i \text{ for all } 1 \leq i \leq n}$$

$$Class \qquad \frac{\begin{array}{c} C^{IV} = C \cup \{MT \leq_{meth} Tobj(MT)\tau\}, E \vdash e_1\colon \sigma \\ C^{METH} = C^{IV} \cup \{ST \leq (RecToMem(\sigma), \tau)\}, \\ E^{METH} = E \cup \{self\colon ST, close\colon (func(ST)\colon MT)\} \vdash e_2\colon \tau \end{array}}{C, E \vdash class(e_1, e_2)\colon Tclass(MT)(\sigma, \tau)}$$

$$Object \qquad \frac{C, E \vdash c\colon Tclass(MT)(\sigma, \tau)}{C, E \vdash new\ c\colon Tobj(MT)\tau}$$

$$Message \qquad \frac{C \vdash \gamma \leq_{meth} Tobj(MT)\{m\colon \tau\},\ C, E \vdash o\colon \gamma}{C, E \vdash o \Leftarrow m\colon \tau[\gamma/MT]}$$

$$InstVar \qquad \frac{C \vdash \gamma \leq (\{v_1\colon \sigma_1; \ldots; v_n\colon \sigma_n\}, \tau),\ C, E \vdash o\colon \gamma}{C, E \vdash o.v_i\colon \sigma_i}$$

$$Update \qquad \frac{\begin{array}{c} C, E \vdash c\colon Tclass(MT)(\{v_1\colon \sigma_1; \ldots; v_m\colon \sigma_m\}, \{m_1\colon \tau_1; \ldots; m_n\colon \tau_n\}), \\ C \vdash \tau_1' \leq \tau_1,\quad C^{IV}, E \vdash a_1'\colon \sigma_1,\quad C^{METH}, E^{METH} \vdash e_1'\colon \tau_1' \end{array}}{\begin{array}{l} C, E \vdash update\ c\ by\ (\{v_1 = a_1'\}, \{m_1 = e_1'\})\colon \\ \qquad Tclass(MT)(\{v_1\colon \sigma_1; \ldots; v_m\colon \sigma_m\}, \{m_1\colon \tau_1'; m_2\colon \tau_2; \ldots; m_n\colon \tau_n\}) \end{array}}$$

where $C^{IV} = C \cup \{MT \leq_{meth} Tobj\ (MT)\{m_1\colon \tau_1'; m_2\colon \tau_2; \ldots; m_n\colon \tau_n\}\}$
$C^{METH} = C^{IV} \cup \{ST \leq (RecToMem(\{v_1\colon \sigma_1; \ldots; v_m\colon \sigma_m\}),$
$\{m_1\colon \tau_1'; m_2\colon \tau_2; \ldots; m_n\colon \tau_n\})\}$
$E^{METH} = E \cup \{self\colon ST, close\colon func(ST)\colon MT, super\colon ST \to \{m_1\colon \tau_1; \ldots; m_n\colon \tau_n\}\}$

$$Extend \qquad \frac{\begin{array}{c} C, E \vdash c\colon Tclass(MT)(\{v_1\colon \sigma_1; \ldots; v_m\colon \sigma_m\}, \{m_1\colon \tau_1; \ldots; m_n\colon \tau_n\}), \\ C^{IV}, E \vdash a_{m+1}\colon \sigma_{m+1},\quad C^{METH}, E^{METH} \vdash e_{n+1}\colon \tau_{n+1} \end{array}}{\begin{array}{l} C, E \vdash extend\ c\ with\ (\{v_{m+1} = a_{m+1}\}, \{m_{n+1} = e_{n+1}\})\colon \\ \qquad Tclass(MT)(\{v_1\colon \sigma_1; \ldots; v_{m+1}\colon \sigma_{m+1}\}\{m_1\colon \tau_1; \ldots; m_{n+1}\colon \tau_{n+1}\}) \end{array}}$$

where $C^{IV} = C \cup \{MT \leq_{meth} Tobj\ (MT)\{m_1\colon \tau_1; \ldots; m_{n+1}\colon \tau_{n+1}\}\}$,
$C^{METH} = C^{IV} \cup \{ST \leq (RecToMem(\{v_1\colon \sigma_1; \ldots; v_{m+1}\colon \sigma_{m+1}\}),$
$\{m_1\colon \tau_1; \ldots; m_{n+1}\colon \tau_{n+1}\})\}$
$E^{METH} = E \cup \{self\colon ST, close\colon func(ST)\colon MT, super\colon ST \to \{m_1\colon \tau_1; \ldots; m_n\colon \tau_n\}\}$
Neither MT nor ST may occur free in C, E in the rules for *class*, *extend*, and *update*.

$$Subsump \qquad \frac{C \vdash \sigma \leq \tau,\quad C, E \vdash e\colon \sigma}{C, E \vdash e\colon \tau}$$

5.5.3 The intuition behind the type rules

We use C and E to pronounce *typing judgements*, which essentially use the current context to determine the type of a construct. See [Gun92] for a more extensive discussion of type assignments and typing judgements. It is important to have a general sense of how the typing rules work, and pay particular attention to the ones related to objects and message-sending. Perhaps the easiest way to understand what is going on is to use an example. Refer to the formal rules presented in Section 5.5.2 while working through each example.

The rule for the assignment statement $X := M$ is relatively straightforward. First we need to be able to show from the current context *C, E* that X, presumably a variable, is of type *ref* τ, and that M is a value of type τ. Then we can conclude that $X := M$ is a legal command. For example, the following assignment is legal:

`var X : {a:num};` – recall that *ref* is inferred in variable declarations
`X := {a=5};` – `{a=5}` can be shown to have type $\{a: num\}$ for any C,E

Note that because of the *Subsumption* rule (the last rule), corresponding to our previous discussion of coercion functions and subtypes, we can interpret a value of a type as a value of any of its supertypes. Thus, the following assignment is also legal:

```
X := {a=5, b=10};
```

because `{a=5,b=10}` can be shown to have type $\{a: num; b: num\}$ for any C,E, and

$\{a: num; b: num\} \leq \{a: num\}$.

The *Function* rule provides a more complicated example. In TOIL, every function body has a *return* clause. For procedures, a trivial value of type *COMMAND* is returned. The type of a block is simply the type of the value it returns. Thus the body of a procedure has type *COMMAND*. Since the types of all of a function's parameters are given by the programmer, the goal of the *function* rule is to find out what the type of the value returned is. To do so, we assume that the types of all the parameters are what the programmer said they would be. Again, the variables are "assigned" to the proper types by throwing some assumptions into E. Finally, using the new context, we type-check the body of the function. The type of the body is the value which will be returned by the function.

It should be noted here that there is a difference between necessary type data supplied by the programmer and redundant type data. For example, the types of a function's parameters are needed to pronounce a typing judgement on it using our rules. However, in TOIL the programmer is also required to designate a type for the returned value. In fact, the return type can be inferred using the type rules and is therefore redundant. But the inferred type can be compared to the given one, effectively over-specifying the program and increasing the odds that what is happening is what the programmer expects.

Like Algol 68, TOIL makes the difference between accessing the value and the location of a reference type explicit in its syntax. By default, the use of a variable refers to its location. To see why, look at the *VarDecl* rule for variable declarations. If we write `var X:` τ, X is assigned to *ref* τ, which is essentially the location of X's data in memory. Later

on, when X is used, the *Var* rule will return $E(x) = ref\ \tau$, not τ. To actually get at the value of the variable X, the construct `val X` is necessary. The *Ref* rule corresponds to the use of `val`. Luckily, it is easy to determine from the context of an expression whether its value or location is relevant. In the language as implemented, the programmer can simply use the variable itself, and TOIL's parser will insert the `val` construct where necessary.

The use of type assignments in E produces static scoping. In other words, variables at one level in the program are hidden by similarly named variables at a lower level. For instance, a global variable x is visible everywhere in a program except where another x, perhaps in a procedure, has been declared (see [GJ87] for further information on static scoping). Since by our definition, each variable must be unique in E, we simply allow subsequent bindings of a variable to hide previous bindings, and this decision results in static scoping.

Finally, we shall attempt to explain the typing rules corresponding to the object-oriented features of TOIL. The object rule is straightforward, but the class and message sending rules are more complex because they depend on the use of inheritance. In Section 5.1, we introduced the difference between the "inside" (pair type, or (σ, τ)), and "outside" (TObj(MT)τ) views of an object. In the methods of an object, we need to support the use of *self* and MT. MT corresponds to the type of the object when seen from the outside, but we want *self* to correspond to the inside view of an object. Remember that the instance variables are hidden and thus unchangeable from the outside, but we definitely want to be able to change them from the inside. Changes from the inside are implemented using *self*. Thus, we introduce another bound variable, ST (for Self Type), to refer to the inner type of an object. We also provide a built in function, *close*, to move from the inside view of an object to the outside view. However, there is no corresponding function *open* to do the opposite, because that would destroy our data abstraction. Think of the inner type ST of an object as a castle with a drawbridge. The object's instance variables are stored safely away in some inner sanctum. From the outside, the instance variables are not accessible at all. We can leave the castle over the drawbridge and shut it behind us using the *close* function, but it is impossible to open it up and get back into the castle from the outside.

From what has already been said, the assumptions added to E in the *class* rule should make sense. *Self* is bound to the type variable ST, and *close* is bound to the type $Func(ST)\colon MT$. These assumptions are only used in the type checking of methods simply because the use of *self* is not permitted in the instance variables. So what is going on with the assumptions added to C? Well, in both cases we use the fact that:

$$MT \leq_{meth} TObj(MT)\tau.$$

In other words, MT is the type of an object created from a class which inherits from the

current one. If we use the simpler assumption that:

$$MT = TObj(MT)\tau.$$

the methods are guaranteed to work for objects created from the current class. However, to be completely type-safe, we need to ensure that the methods work for all possible classes which inherit from this one, since we will not get another chance to type-check inherited methods later on. Therefore, we make the first, more general, assumption.

While type checking methods, a final assumption is used about ST. It is similar to the assumption made on MT in that it is weakened by the possible use of inheritance. However, it uses straight subtyping instead of $\leq_{meth}$ since it is a pair type. The function *RecToMem* used on the instance variables is a function on types which takes a record type $\{v_1{:}\,\sigma_1; \ldots; v_n{:}\,\sigma_n\}$ and returns a similar record of type $\{v_1{:}\,ref\ \sigma_1; \ldots; v_n{:}\,ref\ \sigma_n\}$. This action corresponds to the operation which is performed when an actual object is created from a class and locations are assigned for all of its instance variables.

If, using all of these assumptions, we cannot determine a correct type for the instance variables and methods of a class, then the class is not well formed and should generate an error. It is not necessarily the case that the class would not have worked. Rather, we need to be more strict to preserve type safety using inheritance, and ensure that the methods of the class will work for all possible subclasses.

We can derive the restrictions in TOIL on redefinition of methods and instance variables in subclasses from the *Update* rule. First of all, when redefining an instance variable, we need to be able to show that it has exactly the same type as the old instance variable, under the assumption that MT $\leq_{meth}$ the type of an object generated from the new class being formed. It is easy to show that we run into trouble if we do not make this tight restriction. For example, we might allow the redefined instance variable to be a subtype of the old one. But then since objects are allowed to modify themselves, a method which sticks a value of the original type into the instance variable could be inherited into the new class, where it would no longer be legal. See Figure 5.3 for an example. In the figure, the method `putR` inherited in BogusClass does not work since it attempts to put a value of type $\{a{:}\,num\}$ into a variable of type $\{a{:}\,num; b{:}\,num\}$, and $\{a{:}\,num\} \not\leq \{a{:}\,num; b{:}\,num\}$. This simple counterexample demonstrates why we have to require that the types of instance variables remain unchanged in subclasses.

The restriction on the redefinition of methods is less tight, however. We merely need to be able to prove that the type of the new method is a subtype of the type of the old one. We can allow this since we are not allowed to assign values to methods and therefore the example used for instance variables is invalid.

```
OKClass = class
          var
              r={a=1} : {a:num}
          methods
              putR = procedure()
                     begin
                       r:= {a=5};
                     end
          end;
BogusClass = update OKClass by
             var r={a=2,b=2} : {a:num;b:num};

                          -- would be legal since {a:num;b:num} ≤ {a:num}

             end;
```

Figure 5.3: Class which breaks due to unrestricted redefinition of instance variables.

Chapter 6

The natural semantics of TOIL

6.1 The basics

This chapter presents a natural semantics for TOIL. The natural semantics is a series of reduction rules which recursively shows how to evaluate TOIL subexpressions. The prototype interpreter written in ML corresponds very closely to these rules. In fact, they are based on the working implementation. See [NN92] for more background on natural semantics, and [Wat91] for a discussion of rapid prototyping.

The semantic rules here are somewhat similar to the type checking rules in the last chapter in that they list hypotheses above a line, with the conclusion which can be drawn written underneath the line. Every legal program reduces to "Exit OK". Each rule for expressions evaluates a construct to its final, irreducible value. The irreducibles in TOIL are:

- Numbers,
- Booleans,
- Strings,
- Locations (references),
- Functions,
- Records with irreducible components,
- Classes,
- Object values, written $obj(\{instance\ variables\}, \{methods\})$, and

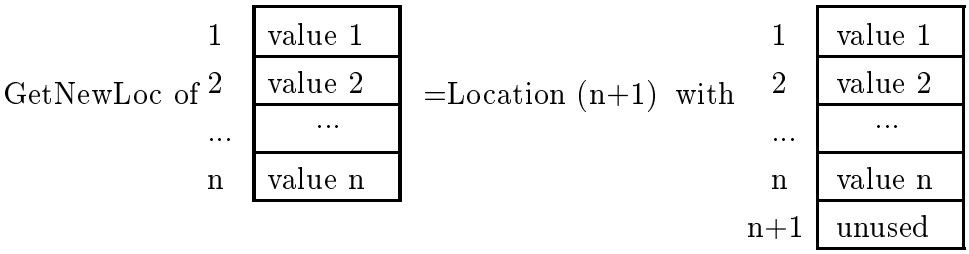

Figure 6.1: *GetNewLoc* function to get a new memory location in the store.

- Objects (recall that objects are actually locations which reference object values).

In our type rules, each expression and statement was typed in a semantics context C, E. Similarly, we need some kind of context when evaluating constructs. First, we need to know what the contents of memory are. The value of a variable depends on what is stored in the location in memory it is associated with. We use a store, s, to represent the state of the machine. The store is modelled as a function which maps locations in memory to values. The value found in a location can be accessed simply by applying the function. For example *s(10)* would return the current value of location 10. The store is infinitely extensible – the function *GetNewLoc* takes a store as a parameter, and returns an unused location l, along with a new store. The new store is exactly the same as the old one except that l is mapped to a memory location marked with an "unused" flag. The use of the *GetNewLoc* function is demonstrated in Figure 6.1. A run-time error will be produced if an "unused" location is referenced before it has a real value assigned to it. Thus, the programmer cannot count on the initial value of a declared variable, but must initialize it explicitly. The notation $s[Loc \mapsto V]$ is used to denote the updating of memory. The store s is updated so that the location *Loc* now contains the value V. Thus $(s[Loc \mapsto V])(Loc)$ should always return V.

We also need to keep track of the bindings of constants and variables. These values are stored in the environment, ρ. The environment is the semantic equivalent of the type assignments in E. However, it holds the actual values of variables and constants instead of just their types. ρ is modelled as a function from identifiers to values. To introduce a binding V for an identifier x, we write $\rho[V/x]$. For constants, the value will be a non-location irreducible value of some kind. For variables, the value will be a location. Thus, for a variable x, $s(\rho(x))$ returns the value stored in that variable.

Each reduction rule takes a subexpression, an environment, and a store, and returns an irreducible value along with a new store to keep track of side effects. Since side effects are impossible in pure functional languages, the natural semantics for TOOPL did not include a store, and were therefore much simpler.

Since TOIL is type-safe (see Chapter 8 for a proof), all of the reductions will proceed

Assuming we start with $\rho = \{\}$:

```
const x=1;
      y=x;
      x=2;
      z=3;
      z=x;
```

We end up with: $\rho = \{x \mapsto 2, y \mapsto 1, z \mapsto 2\}$.

Figure 6.2: Sample declaration section

without difficulty. We do not even have to consider types in the natural semantics because all of the typing is done statically instead of dynamically. In particular, whenever a field is selected from a record, the record is guaranteed to contain that field.

In the following sections, refer to the formal natural semantics presented in the final section of this chapter.

6.2 Declarations

Variables and constants can be declared at the start of the main program and in each function as well. The values of constants are frozen when they are declared. To remain consistent with this intuition, it is illegal to declare an object, or a reference, as a constant. A constant declaration results in the evaluation of the value, and the binding of the resulting irreducible value to the constant identifier in ρ. Instead of just returning a value, the declaration rules return a new environment which incorporates the new value of the constant along with a new store. Again, we need a new store because side effects may have been introduced during the evaluation of the values assigned to the constants. The rule for the declaration of constants simply goes along the list of declarations evaluating each expression and binding it to the correct identifier, all the while generating updated copies of the store. Finally, it returns the last environment and store generated. The order of evaluation chosen implies that identifiers listed early on can be used in the expressions assigned to subsequent constants. Furthermore, if the same identifier is declared twice, the second value will be used. See Figure 6.2 for a sample declaration section.

The declaration of variables is similar except that the name of the variable is mapped to a new location from the store. As previously mentioned, the value stored at the location initially is a flag which remains until a real value is placed there.

```
program lexical-scoping;
const x=10;
      f1 = function(y:num):num
           begin
              return x+y; -- which x to use?
           end;
      f2 = function(x:num):num
           begin
              return f1(1);
           end

begin
   print(f2(5));  -- prints 6 if dynamic scoping, 11 if static scoping
end.
```

Figure 6.3: Lexical versus dynamic scoping

6.3 Intricacies

Most of the rules follow fairly naturally. For example, to evaluate an IF-THEN-ELSE statement, we evaluate the boolean expression, and then evaluate either the THEN clause or the ELSE clause depending on its value. A WHILE statement is evaluated repeatedly while the boolean clause is true. The store keeps track of side effects which occur during each iteration, so that subsequent evaluations of the clause can end up as false, terminating the loop.

The eager evaluation of records is also reflected in the natural semantics. The values in a record's fields are evaluated when the record itself is reduced. Thus, when a particular component is referenced, we need only select the correct irreducible value. In contrast, TOOPL did not evaluate the value in a record field until it was actually selected. For the pure functional language, lazy evaluation did not make a difference because of the lack of side effects, but TOIL needs to be more sensitive to the actual time of evaluation.

The reduction rule for functions is complicated by the fact that we need to correctly implement static lexical scoping. Since functions are first class values, they can be passed as parameters and returned from functions. This ability implies that each function needs to carry with it the environment in which it was declared. To see why, examine Figure 6.3. Which value will end up being printed? If the value of x at the time that f1 is applied is taken to be that of the most recent binding (dynamic scoping), x will be interpreted as 1,

and a 6 will be printed. However, if the declaration of x from the nearest lexically enclosing block is used (lexical scoping), x will be interpreted as 10, and an 11 will be printed. To be consistent with lexical scoping, the sample program should print the value 11. This result can easily be modelled in the semantics by requiring that each evaluated function carry with it an environment which holds all of the correct lexical bindings of variables.

It should be noted that implementing lexical scoping with functions as first class values is not easy. The difficulty lies in the fact that local variables for a function f1 are normally stored on a run-time stack, implying that they may be popped off before they are referenced by a function f2 which is lexically enclosed in f1, but is called after f1 has returned. See Figure 6.4 for an example of such a program. The correct binding for x in f3 is the parameter to f1, but by the time f3 is actually applied, f1 has already returned. The value which was passed as a parameter to f1 is probably no longer available. Therefore, implementing this feature generally requires heap-based allocation of a function's frame. To apply a function, then, we simply:

1. Evaluate the function,
2. Evaluate its parameters,
3. Bind the resulting values $V_1 \ldots V_i$ to the corresponding identifiers in the environment ρ_f stored with the function, and
4. Evaluate the function body B in ρ_f with the parameters bound to the appropriate values to get a return value R.

TOIL does not allow VAR parameters because difficulties arise with assignment to variable parameters in conjuction with subtyping. Consider the procedure in Figure 6.5. It is well-typed. However, if we pass the function some subtype BetterPoint of Point as a VAR parameter, a Point value will be assigned to it. BetterPoint may be a specialization of Point, with new methods and instance variables. Thus, when we return from the function and request the BetterPoint variable to perform one of the new methods, the Point object stored there will not know how to execute it.

TOIL follows call-by-constant-value semantics for parameter passing. Thus, parameters are evaluated when the function is called, and the resulting values are then treated as constant. No parameter will change in value during the execution of the function. The use of call-by-constant-value semantics along with the omission of VAR parameters may seem to leave a large gap in the language, namely the ability of functions and procedures to affect any variables out of their scope. However, this limitation is eased by the use of objects. If a parameter is an object, the object being referenced cannot be changed, but its instance variables can. Thus, any use of VAR parameters can be achieved through the use of objects. In this case, we do not run into the problem demonstrated in Figure 6.5 for regular VAR

```
program lexical-scoping2;
const x=10;
      f1 = function(x:num):(func():num)
           const f2 = function():num
                      begin
                         return x;
                      end
           begin
              return f2;
           end
var f3:func():num;
    y:num
begin
   f3:= f1(5);
   y := f3();
   print(y);
end.
```

Figure 6.4: An application of lexical scoping

```
function Break(VAR p:PointObjectType):UNIT
begin
   p:= new(PointClass);
   return Unit;
end;
```

Figure 6.5: Why VAR parameters break the type system.

parameters because all assignments are carried out within objects, and the type checking can be carried out successfully in that context.

A few more intricacies are introduced by the need to have a value for *self* available. When a class is evaluated, it contains two records, one with the initial values for instance variables, and one with a set of functions corresponding to the methods. Recall that an object is modelled as a location, which references the actual value of the object (as shown in Figure 4.4). Therefore, when an object is instantiated from a class, the result is a location. At that location in the store, we want to place the value of the object. Aside from the location where the object will be stored, we also need to go out and get unique locations for all of the object's instance variables. Therefore, if the object has n instance variables, we need to get $n+1$ new locations total.

Note that the methods carry with them the environment at the time that the class was evaluated, in which *self* is not bound to anything. However, when we call a method, *self* needs to be bound. Therefore, when the object is instantiated from the class, we bind self to a value. The value for *self* is just o, an object with a record of locations corresponding to the instance variables, and a record of methods which is unchanged from the class. Now we generate a new copy of the methods, with *self* bound to o in each of the method environments. Thus, if a method is called, a binding for *self* will be found in the environment stored with the method. Finally, we place the completed object in the correct location, and return the location itself as a pointer to it.

We have only partially solved the problem. *Self* is now bound in all of the object's methods, but what if we use *self* to recursively call another method? The value for *self* contains the correct set of methods, but the environments stored along with them do not having the binding for *self*. Perhaps we should have bound *self* in their environments as well. But then what about a second recursive call using *self*? It seems as if we will always be one step behind. No matter how many levels deep we bind *self*, someone can always recursively call methods one more time. Furthermore, these multiple level bindings will probably take up a lot of space.

The solution is to keep one step ahead of the game. Recall from Chapter 5 that MT refers to the "outside" type of an object, while ST refers to the "inside" type. Now we can see that MT corresponds to the location at which the object is stored, while ST refers to the actual value of the object itself. The result of a `new` operation is an object type, corresponding to a location. The value stored in *self* is of type ST, and corresponds to an actual object value. Our solution to the dilemma presented above takes advantage of the fact that messages can only be sent to the values of object types. Thus, before we can send a message to *self*, we must use the *close* operation to convert it into an object type. Now it is clear that as long as we bind *self* to a value every time we use *close*, it will be impossible to call a method in whose environment *self* is not bound. In fact, the rule to perform *close* looks exactly like the rule for `new` except that it is not necessary to get new locations for

all of the object's instance variables.

The rule to send an object a message reflects the fact that the object is actually a location. The correct method can simply be plucked from the object value stored at that location. However, recall that instance variables can only be accessed using *self*. Therefore, they are selected directly from an object value without having to look up anything first.

The final complication in the natural semantics concerns the use of *super*. Like *self*, *super* needs to be bound to a value, but only in the environment of methods in objects formed from a subclass. *Super* can simply be modelled as a record of methods from the superclass, since it does not really make sense to access the parent's instance variables. However, the methods of the parent class may involve references to *self*, and therefore *super* is dependent on the value of *self*. Because of this dependency, we run into problems with timing:

- The value of *self* is not known until the object is instantiated with a `new` command.
- The methods of the superclass are known when the subclass is formed from it using `update` or `extend`.

Therefore, the value that *super* should be bound to is known when the class is created, but it cannot actually be bound until the class is created. Furthermore, we cannot just store the value of *super* with each class and then fill it in when an object is instantiated because we run into the same problem with circularity as we did before with *self*.

We solve this problem by making explicit *super's* dependency on *self*. Each reference to *super* in a method is modelled as a functional abstraction with *self* as a parameter. Consider:

$$super = \{m_1 = e_1, \ldots, m_n = e_n\} : \{m_1 : \tau_1; \ldots; m_n : \tau_n\}$$

Then we make the transformation:

$$\begin{array}{l} super \Longrightarrow \\ \quad \texttt{function(self:ST)} : \{m_1 : \tau_1; \ldots; m_n : \tau_n\} \\ \quad \texttt{begin} \\ \quad\quad \texttt{return}\ \{m_1 = e_1, \ldots, m_n = e_n\}; \\ \quad \texttt{end;} \end{array}$$

Whenever we actually want to use *super* in a method, we simply apply this function to *self* and get back what we wanted as *super* with *self* already bound. Note that *self* will always be available when we are trying to use *super* because we will always be inside a method.

6.4 Formal rules

Program

$$\frac{(B, \mathit{initial\ Env}, \mathit{initial\ Store}) \downarrow (\text{Command}, s)}{\mathit{program}\ p;\ B. \downarrow (\text{Exit OK}, s)}\ ,$$

Blocks

$$\frac{\begin{array}{c}(C, \rho, s) \downarrow (\rho', s'),\quad (V, \rho', s') \downarrow (\rho'', s''),\\ (S, \rho'', s'') \downarrow (\text{Command}, s'''),\quad (R, \rho'', s''') \downarrow (R_v, s'''')\end{array}}{(C\ V\ \mathit{begin}\ S\ \mathit{return}\ R;\ \mathit{end}, \rho, s) \downarrow (R_v, s'''')}\ ,$$

Declarations

$$\frac{(e_i, \rho^i, s^i) \downarrow (v_i, s^{i+1}),\quad \rho^{i+1} = \rho^i[v_i/id_i],\quad 1 \le i \le n}{(\mathit{const}\ id_1 = e_1; \ldots; id_n = e_n, \rho^1, s^1) \downarrow (\rho^{n+1}, s^{n+1})}\ ,$$

$$\frac{(\mathit{newLoc}_i, s^i) = (\mathit{GetNewLoc}\ s^{i-1}),\quad \rho^i = \rho^{i-1}[\mathit{newLoc}_i/id_i],\quad 1 \le i \le n}{(\mathit{var}\ id_1{:}\,\tau_1; \ldots; id_n{:}\,\tau_n, \rho^0, s^0) \downarrow (\rho^n, s^n)}\ ,$$

Commands

$$\frac{(X, \rho, s) \downarrow (\mathit{Loc}, s'),\quad (M, \rho, s') \downarrow (V, s'')}{(X := M, \rho, s) \downarrow (\text{Command}, s''[\mathit{Loc} \mapsto V])}\ ,$$

$$\frac{(B, \rho, s) \downarrow (\text{true}, s'),\quad (M, \rho, s') \downarrow (\text{Command}, s'')}{(\mathit{if}\ B\ \mathit{then}\ M\ \mathit{else}\ N\ \mathit{end}, \rho, s) \downarrow (\text{Command}, s'')}\ ,$$

$$\frac{(B, \rho, s) \downarrow (\text{false}, s'),\quad (N, \rho, s') \downarrow (\text{Command}, s'')}{(\mathit{if}\ B\ \mathit{then}\ M\ \mathit{else}\ N\ \mathit{end}, \rho, s) \downarrow (\text{Command}, s'')}\ ,$$

$$\frac{\begin{array}{c}(B, \rho, s) \downarrow (\text{true}, s'),\quad (S, \rho, s') \downarrow (\text{Command}, s''),\\ (\mathit{while}\ B\ \mathit{do}\ S\ \mathit{end}, \rho, s'') \downarrow (\text{Command}, s''')\end{array}}{(\mathit{while}\ B\ \mathit{do}\ S\ \mathit{end}, \rho, s) \downarrow (\text{Command}, s''')}\ ,$$

$$\frac{(B, \rho, s) \downarrow (\text{false}, s')}{(\mathit{while}\ B\ \mathit{do}\ S\ \mathit{end}, \rho, s) \downarrow (\text{Command}, s')}\ ,$$

$$\frac{(C_1, \rho, s) \downarrow (\text{Command}, s'), \quad (C_2, \rho, s') \downarrow (\text{Command}, s'')}{(C_1;\ C_2, \rho, s) \downarrow (\text{Command}, s'')}$$

Expressions

$$(\text{Num n}, \rho, s) \downarrow (\text{Num n}, s),$$

$$(\text{Bool b}, \rho, s) \downarrow (\text{Bool b}, s),$$

$$(\text{Command}, \rho, s) \downarrow (\text{Command}, s),$$

$$(\text{Unit}, \rho, s) \downarrow (\text{Unit}, s),$$

$$(\text{NIL}, \rho, s) \downarrow (\text{errorLocation}, \text{s}),$$

$$(x, \rho, s) \downarrow (\rho(x), s), \text{ where x is a variable}$$

$$\frac{(x, \rho, s) \downarrow (Loc, s'), \quad s'(Loc) = V}{(val\ x, \rho, s) \downarrow (V, s')},$$

$$(function(v_1, \ldots, v_n)B, \rho, s) \downarrow (\langle function(v_1, \ldots, v_n)B, \rho\rangle, s),$$

$$\frac{\begin{array}{c}(M, \rho, s^0) \downarrow (\langle function(v_1, \ldots, v_n)B, \rho_f\rangle, s^1), \quad (N_i, \rho, s^i) \downarrow (V_i, s^{i+1}), \\ (B, \rho_f[V_i/v_i], s^{n+1}) \downarrow (R, s^{n+2})\end{array}}{(M(N_1, \ldots, N_n), \rho, s^0) \downarrow (R, s^{n+2})},$$

$$\frac{(e_i, \rho, s^i) \downarrow (v_i, s^{i+1}), \quad 1 \leq i \leq n}{(\{m_1 = e_1, \ldots, m_n = e_n\}, \rho, s^1) \downarrow (\{m_1 = v_1, \ldots, m_n = v_n\}, s^{n+1})},$$

$$\frac{(e, \rho, s) \downarrow (\{m_1 = v_1, \ldots, m_n = v_n\}, s')}{(e.m_i, \rho, s) \downarrow (v_i, s')},$$

$$\frac{(e', \rho, s) \downarrow (v', s'), \quad (e'', \rho, s') \downarrow (v'', s'')}{(class(e', e''), \rho, s) \downarrow (class(v', v''), s'')} ,$$

$$\frac{\begin{array}{c}(e, \rho, s) \downarrow (class(\{iv_1 = v_1, \ldots, iv_n = v_n\}, \{m_1 = \langle f_1, \rho_1 \rangle, \ldots, m_r = \langle f_r, \rho_r \rangle\}), s^0), \\ (newLoc_i, s^i) = (GetNewLoc\ s^{i-1}), 1 \leq i \leq (n+1)\end{array}}{\begin{array}{c}(new(e), \rho, s) \downarrow (newLoc_{n+1}, s^i[newLoc_1 \mapsto v_1, \ldots, newLoc_n \mapsto v_n, \\ newLoc_{n+1} \mapsto obj(newIV, newMethods)])\end{array}} ,$$

where $newIV = \{iv_1 = newLoc_1, \ldots, iv_n = newLoc_n\}$,
$o = obj(newIV, \{m_1 = \langle f_1, \rho_1 \rangle, \ldots, m_r = \langle f_r, \rho_r \rangle\})$,
$newMethods = \{m_1 = \langle f_1, \rho_1[o/self] \rangle, \ldots, m_r = \langle f_r, \rho_r[o/self] \rangle\}$

$$\frac{\begin{array}{c}(e, \rho, s) \downarrow (obj(e', \{m_1 = \langle f_1, \rho_1 \rangle, \ldots, m_n = \langle f_n, \rho_n \rangle\}), s'), \\ (newLoc, s'') = (GetNewLoc\ s')\end{array}}{(close(e), \rho, s) \downarrow (newLoc, s''[newLoc \mapsto obj(e', newMethods)])} ,$$

where $o = obj(e', \{m_1 = \langle f_1, \rho_1 \rangle, \ldots, m_n = \langle f_n, \rho_n \rangle\})$,
$newMethods = \{m_1 = \langle f_1, \rho_1[o/self] \rangle, \ldots, m_n = \langle f_n, \rho_n[o/self] \rangle\}$

$$\frac{(e, \rho, s) \downarrow (Loc, s'), \quad s'(Loc) = obj(e', \{m_1 = v_1, \ldots, m_n = v_n\})}{(e \Leftarrow m_i, \rho, s) \downarrow (v_i, s')} ,$$

$$\frac{(e, \rho, s) \downarrow (obj(\{iv_1 = v_1, \ldots, iv_n = v_n\}, e''), s')}{(e.iv_i, \rho, s) \downarrow (v_i, s')} ,$$

$$\frac{(P, \rho, s) \downarrow (class(\{iv_1 = v_1, \ldots, iv_{upd} = v_{old}, \ldots, iv_n = v_n\}, e''), s'), \quad (u, \rho, s') \downarrow (v_{new}, s'')}{\begin{array}{c}(update\ P\ by\ (\{iv_{upd} = u\}, \{\}), \rho, s) \downarrow \\ (class(\{iv_1 = v_1, \ldots, iv_{upd} = v_{new}, \ldots, iv_n = v_n\}, e''), s'')\end{array}} ,$$

$$\frac{(P, \rho, s) \downarrow (class(e', \{m_1 = v_1, \ldots, m_{upd} = v_{old}, \ldots, m_n = v_n\}), s'), \quad (u, \rho', s') \downarrow (v_{new}, s'')}{\begin{array}{c}(update\ P\ by\ (\{\}, \{m_{upd} = u\}), \rho, s) \downarrow \\ (class(e', \{m_1 = v_1, \ldots, m_{upd} = v_{new}, \ldots, m_n = v_n\}), s'')\end{array}} ,$$

where $\rho' = \rho[(\lambda self.\{m_1 = v_1, \ldots, m_{upd} = v_{old}, \ldots, m_n = v_n\})/super]$

$$\frac{(P,\rho,s)\downarrow(class(\{iv_1=v_1,\ldots,iv_n=v_n\},e''),s'),\quad (u,\rho,s')\downarrow(v_{n+1},s'')}{\begin{array}{c}(extend\ p\ with\ (\{iv_{n+1}=u\},\{\}),\rho,s)\downarrow\\ (class(\{iv_1=v_1,\ldots,iv_n=v_n,iv_{n+1}=v_{n+1}\},e''),s'')\end{array}},$$

$$\frac{(P,\rho,s)\downarrow(class(e',\{m_1=v_1,\ldots,m_n=v_n\}),s'),\quad (u,\rho',s')\downarrow(v_{n+1},s'')}{\begin{array}{c}(extend\ p\ with\ (\{\},\{m_{n+1}=u\}),\rho,s)\downarrow\\ (class(e',\{m_1=v_1,\ldots,m_n=v_n,m_{n+1}=v_{n+1}\}),s'')\end{array}},$$

where $\rho' = \rho[(\lambda self.\{m_1=v_1,\ldots,m_n=v_n\})/super]$

Chapter 7

Denotational semantics

7.1 The basics

Type-checking rules represent the algorithm used by a type checker, while natural semantics correspond to the actions of an interpreter. Denotational semantics, however, are a much more abstract and mathematical representation of the meanings of constructs in a language. It can be used to provide a mathematical basis for formal reasoning about the language. For example, at the end of this chapter we include the full denotational semantics for TOIL, and then in Chapter 8 go on to prove that they are consistent with our type-checking rules. Thus, the language as defined by the denotational semantics can be proved to be completely type-safe. The semantics themselves tend to be very complicated. It is more important to get a general sense of what is going on than to understand the intricacies of each rule fully. For a general introduction to denotational semantics, see [Sch86].

Our denotational semantics are based on a model of the F-bounded second-order polymorphic lambda calculus. In the rest of this chapter, it is essential to distinguish between syntactic values and semantic values. Syntactic values are just the kinds of values we have introduced in the last few chapters. For example, $Func(\sigma_1, \ldots, \sigma_n)\colon \tau$ is a syntactic type. Similarly, the functions we can define in TOIL are only syntactic entities. The whole point of the denotational semantics is to map these syntactic values to corresponding semantic values from a model. Thus, semantic meaning for function types will assign $Func(\sigma_1, \ldots, \sigma_n)\colon \tau$ to a semantic type in the model. This type will correspond to a collection of mathematical functions of the appropriate functionality. Also, the semantics for a function definition will map it onto an equivalent mathematical function from the corresponding type in the model. In the rest of the chapter, we will use many of the concepts introduced in previous chapters, including types, the environment ρ, and our definitions of C and E. We also need some more definitions:

- Let $\mathcal{A}$ be a model of the F-bounded second-order polymorphic lambda calculus with recursive types and elements. This is sufficient to assure that all type and normal expressions have interpretations in $\mathcal{A}$. If ξ is the interpretation of a type, σ, then $\mathcal{A}^{\xi}$ should represent the set of elements of that type. Throughout this thesis, however, we shall often abbreviate by omitting the A and simply writing the superscript, so ξ would represent the elements of semantic type ξ.

- Let *Type* be the collection of meanings of types in A. Then we require that StoreType, UnitType, ProgramType, CommandType, $\bot$, and $\top$ are elements of *Type*.

- We define UnitType such that $UnitType = \{Unit\}$, and we define ProgramType and CommandType in a similar fashion.

- The result of a non-terminating computation is $\bot_{Command}$. We shall write $(A \times B)_{\bot}$ to indicate a pair type which also includes $\bot_{Command}$.

- We write $\leq_{\mathcal{A}}$ to denote the subtyping relation for semantic types.

- We write $(\tau_1 \times \ldots \times \tau_n)$ to denote a product of types. Elements of the product type are grouped together using the notation $\langle V_1, \ldots, V_n \rangle$. Thus, $\langle 2, 4, 6 \rangle \in (Num \times Num \times Num)$.

- We write $\sigma \rightarrow \tau$ to denote a collection of mathematical functions from elements of type σ to elements of type τ. Functions which take multiple parameters are typically presented in Curried form. See [Wat91] for an explanation of Curried functions in the lambda calculus.

- $\prod_{i \in S} T_i$ denotes the set of all functions, f, with domain S such that for all $i \in S$, $f(i) \in T_i$.

- A memory store consists of a mapping between locations and values which are organized by the semantic type of the data stored there. We write LOC_{τ} to represent the set of locations which contain elements of type τ. A store s is said to be consistent if for any location $l \in LOC_{\tau}, s(l) \in \tau$. Since *errorLocation* is not a member of any LOC_{τ}, $s(errorLocation)$ (which results in a run-time warning) is always considered consistent. However, the initial value for all memory locations, *errorFlag*, is considered to be a member of all types, and so an initialized memory location results in a consistent store. For any consistent store s, $s \in StoreType$.

- We again use the environment ρ to hold bindings for variables. ρ is modelled as a function from identifiers to values. To introduce a binding V for an identifier x, we write $\rho[V/x]$. For constants, the value will be a non-location irreducible value of some

kind. For variables, the value will be a location. Thus, for a variable x, $s(\rho(x))$ returns the value stored in that variable.

- We have stated before that the redefinition of instance variables in a subclass is required to maintain the same type for the instance variable. The redefinition of methods, on the other hand, requires only the subtype relation. We introduce a semantic function **ext** to make the tighter requirement for instance variables explicit. We are allowed to add new fields to the record of instance variables, but not to change any of the types of the already defined ones. Thus,
  $\{v_1 : \sigma_1; \ldots; v_k : \sigma_k; \ldots; v_n : \sigma_n\}$ **ext** $\{v_1 : \sigma_1; \ldots; v_k : \sigma_k\}$ for $1 \leq k \leq n$.

7.2 Meanings of Types

We will first present the meanings of types in TOIL, followed by the meanings of terms. The notation $[\![T]\!]\rho$ for the meanings of types should be interpreted as "The meaning of type T in the environment ρ". Similarly, the notation $[\![M]\!]\rho\ s$ for terms should be interpreted as "The meaning of M in the environment ρ and memory store s". Our intention, which we shall prove is achieved in the next chapter, is that:

$[\![\tau]\!]\rho \in Type$, for a type expression τ.

$[\![C, E \vdash e : \tau]\!]\rho\ s = ([\![\tau]\!]\rho \times StoreType)_\perp$, for a command or expression e.

Thus, the meaning of a syntactic type in ρ is just a semantic type, and the meaning of a command or expression of type τ is a pair containing a semantic element of type τ and a new memory store. We now present the semantics of types, with a more readable explanation following. Refer to the definitions below while reading the intuition behind them.

Definition 7.2.1 *If ρ is a type environment mapping type variables to types in the model, $\mathcal{A}$, then we define the meanings of types as follows:*

$[\![c]\!]\rho = \mathcal{A}_c$ for type constants c.

$[\![t]\!]\rho = \rho(t)$ (type variables).

$[\![ref\ \tau]\!]\rho = LOC_{[\![\tau]\!]\rho}$.

[[COMMAND]]$\rho = CommandType$.

[[UNIT]]$\rho = UnitType$.

[[PROGRAM]]$\rho = ProgramType$.

$[\![\bot]\!]\rho = \{errorLocation\} \leq_{\mathcal{A}} [\![TObj(MT)\tau]\!]\rho$ *for all* τ.

$[\![Func(A_1; ...; A_n)\colon B]\!]\rho = StoreType \to ([\![A_1]\!]\rho \times \ldots \times [\![A_n]\!]\rho) \to ([\![B]\!]\rho \times StoreType)_\bot$.

$[\![\{m_1\colon \tau_1, \ldots, m_n\colon \tau_n\}]\!]\rho = \prod_{m_i \in \{m_1,\ldots,m_n\}} [\![\tau_i]\!]\rho$.

$[\![(\sigma, \tau)]\!]\rho = [\![\sigma]\!]\rho \times [\![\tau]\!]\rho$

$[\![Tclass(MT)(\sigma, \tau)]\!]\rho = \prod_{\mu \leq_{\mathcal{A}} [\![\tau]\!]\rho'(\mu)} \prod_{\nu \textbf{ ext } [\![\sigma]\!]\rho'(\mu)} ([\![\sigma]\!]\rho'(\mu) \times (\xi^S(\nu, \mu) \to [\![\tau]\!]\rho'(\mu)))$

where $\xi^S(\nu, \mu) = RecToMem(\nu) \times \mu$,
$\xi^P(\mu) = LOC_{\exists v.(v \times \mu)}$,
$\rho'(\mu) = \rho[\xi^P(\mu)/MT]$.

$[\![Tobj(MT)\tau]\!]\rho = FIX(\lambda\xi^P.LOC_{\exists v.[\![(t,\tau)]\!]\rho[\xi^P/MT, v/t]} \cup \{errorLocation\})$

As usual, the last definition can be rewritten as:

$$\begin{aligned} [\![Tobj(MT)\tau]\!]\rho &= LOC_{\exists v.([\![(t,\tau)]\!]\rho[[\![Tobj\ (MT)\tau]\!]\rho/MT, v/t])} \cup \{errorLocation\} \\ &= LOC_{\exists v.(v \times [\![\tau]\!]\rho[[\![Tobj(MT)\tau]\!]\rho/MT])} \cup \{errorLocation\}. \end{aligned}$$

As we would expect, the meaning of a type constant such as *Bool* is simply its interpretation in $\mathcal{A}$. In the case of *Bool*, the interpretation is $\{true, false\}$. The meaning of a type variable is simply its bound value in ρ. The meaning of a reference type to τ is simply the set of locations in the store which hold values which have the semantic type corresponding to τ. COMMAND, UNIT, and PROGRAM all map onto semantic types which we have defined to be sets with only one element each. *UnitType* contains the single value *Unit*, and so on. Finally, the meaning of a record type is simply a function which takes as a parameter the name of one of the record's fields and returns a value of the correct type.

To understand the meaning of a function type, we must think ahead to how we are going to write the meaning of a function definition. We will have to map it onto a semantic function. As we saw in Chapter 5, we want to store the current value of the environment in this semantic function, instead of using the environment available when the function is applied. However, we do not want to freeze the current memory store in a similar fashion. Instead, we should use the store available when the function is applied. We can achieve this goal by explicitly parameterizing the store in the semantic function we are constructing. Therefore, the meaning of a function type is the type of a semantic function which takes an element of *StoreType* and a tuple of values corresponding to the original syntactic function's parameters. It returns an element of the return type of the function, along with a new store to capture any side effects.

The remaining types correspond to TOIL's object-oriented features and are a little more complicated. Once we wade through all of the notation, however, their meanings are not too

hard to understand. The notation used includes existential quantifiers, which are commonly used in the lambda calculus to model data abstraction. [CW85] includes a good introduction to existential types. However, the intuition behind our definitions of the meanings of these types can be understood even without a full background in the notation used.

The meaning of a pair type (σ, τ) is very simple. Remember that the pair type represents the type of an object from the inside, so that all instance variables and methods are fully visible. Therefore, the meaning of a pair type is simply the product of the meanings of the record types for instance variables and methods. Thus, an element of this type can simply be a pair containing the semantic equivalent of the instance variables and the semantic equivalent of the methods.

Since instance variables are abstracted out of the object type, we are forced to use existential quantifiers to model them. To avoid committing ourselves to any particular set of instance variables (maintaining the abstraction), we quantify the type of the instance variables out as v. As we have discussed before, an object type is actually a location in memory, However, we need to classify what sorts of data will be stored in that location. Well, it should be obvious that the value stored will be a pair type. The instance variable part of the pair is just the v which we have quantified out. Furthermore, to get the full meaning of the method types, we need to evaluate their meaning in an environment where MT is replaced by the current object type. Thus,

$$[\![TObj(MT)\tau]\!]\rho = LOC_{\exists v.(v \times [\![\tau]\!]\rho[[\![Tobj(MT)\tau]\!]\rho/MT])} \cup \{errorLocation\}$$

It can be shown that this definition is equivalent to the alternative one used in the definitions of semantics of types which uses fixed points. Both definitions say the same thing, so the reader should think of object types in terms of the one which seems most intuitive. When we move on to define the meaning of objects themselves, we will see that an object is formed by taking an element of a pair type (σ, τ), somehow hiding its instance variables, and then storing it in a location.

We want classes to work so that we can generate semantic objects from them. Furthermore, a class needs to be able to generate the meanings of methods and instance variables not only for the current class, but for all subclasses as well. For example, the Point class needs to have enough information to generate methods for a ColorPoint class since some of the methods will be inherited and therefore not redefined. If the type of the methods we want to make is called μ, and the types of the instance variables is called ν, then we are going to generate an object with semantic type $LOC_{\exists v.(v \times \mu)}$. Again, we were forced to quantify out the instance variable part in order to preserve the abstraction. In order to get these two types μ and ν, we model the class as a function which takes two parameters. The first corresponds to the type of the methods created from any subclass. More explic-

itly, we take a parameter μ such that $\mu \leq_{\mathcal{A}} [\![\tau]\!]\rho'(\mu)$, where $\rho'(\mu)$ has MT bound to the type of the object we will generate. The second parameter corresponds to the type of the instance variables generated by any subclass. Since the types of instance variables cannot be changed during inheritance, we use the **ext** relation instead of subtyping. Thus, our second parameter is in ν such that ν **ext** $[\![\sigma]\!]\rho'(\mu)$.

Now, our factory class should return a pair type with instance variables and methods. However, one final complication remains. Namely, we are allowed to use *self* in the object's methods. Therefore, instead of returning the object's methods directly, we model them as a function which takes an element of $\xi^S(\nu, \mu)$. $\xi^S(\nu, \mu)$ corresponds to the inner type of the object, in other words the type ST. Since *self* is of type ST, the function for methods essentially takes a value for *self* and returns the meaning of the methods.

7.3 Meanings of terms

7.3.1 Operations on the store

Let s be a memory store. Then we define the following operations on s:

$UpdateMem_T$: $LOC_T \rightarrow T \rightarrow StoreType \rightarrow StoreType$,

$LookupMem_T$: $LOC_T \rightarrow StoreType \rightarrow T$,

$PlaceInMem_T$: $T \rightarrow StoreType \rightarrow (LOC_T \times StoreType)$,

$UpdateEnv$: $id \rightarrow value \rightarrow Env \rightarrow Env$, and

$GetLoc_T$: $StoreType \rightarrow (LOC_T \times StoreType)$.

UpdateMem takes a typed location, an element of that type, and a store, and returns a new store where the given value is at the given location. LookupMem takes a typed location and a store, and returns an element of the type found at that location. PlaceInMem takes a value and a store, allocates a new place in memory for a value of that type, stores the value in that location, and then returns the newly allocated location and the updated store. It is used to store objects in memory. UpdateEnv takes an identifier, a value for the identifer, and the current environment. It returns a new environment which is the same as the old one except that the identifier is now associated with the given value, i.e. $\rho(\text{id}) = \text{value}$. GetLoc takes a store and returns a new location where something can be stored, as well as a new store. The new location is initialized with an *errorFlag* which results in a run-time error if it is accessed. Thus, variables cannot be used before they are initialized in a program. Note that by definition, $errorFlag \in [\![\tau]\!]\rho$ for all τ and ρ. GetLoc is used to store regular variables. Note that in a real implementation, GetLoc might allocate stack memory, where

the amount of space needed is known at compile time, whereas PlaceInMem might allocate heap memory, since it deals with object variables, whose size cannot be known at compile time.

7.3.2 Operations on semantic types

We also provide several functions which operate on semantic types:

- **Convert** is a function which coerces one semantic type to another. As discussed in Chapter 5, such a function is guaranteed to exist if the two types are subtypes. The notation used for the coercion of a value $V \in \sigma$ to type τ (if $\sigma \leq_{\mathcal{A}} \tau$), is:

  convert$[\tau]$ $[\sigma]$ V.

  In other words, **convert** [final type] [original type] (element of original type to be converted).

- *Pack* and *open* are used when dealing with existential types. *Pack* hides the parts of the existential type which have been quantified out, while *open* makes them visible again.

7.3.3 Differences from TOOPL

The meanings of terms are even more complicated than the meanings of types. Since a full explanation would take more space than is available in this thesis, we direct the reader to [Bru93] for a step-by-step introduction to the denotational semantics of TOOPL. Here we will describe some of the different techniques used in TOIL.

The most fundamental changes are due to the addition of a store and side effects. TOOPL is purely functional, and therefore does not need to model a store at all. TOIL adds imperative features such as variables and assignment which in and of themselves are not too complicated, but introduce side effects which affect all of our other constructs. The meaning of every term in TOIL is dependent not only on the current environment ρ, but on the current status of the store as well. This dependency is made explicit by listing s along with ρ on the left hand side of each semantic rule. Each time we recursively find the meaning of a term, we need to keep track of side effects introduced by the term. For example, if we encounter a `WHILE` loop with environment ρ and store s and find that the boolean condition is false, we still have to make sure that we capture any side effect the evaluation of the condition might have had on the store. Thus, instead of returning s unchanged, we must return s', where s' is the store returned by the evaluation of the condition. Every rule must keep track of side effects by passing around a store in this manner.

It should be noted that the way in which this is done implicitly imposes a specific order on the computation. In TOOPL, the order in which constructs were evaluated did not

matter at all, but here side effects from one computation may affect later ones, so order does matter.

The only other major difference from the semantics of TOOPL is due to the representation of objects and instance variables as locations. While we dealt directly with an object in TOOPL, we need to be aware of the fact that in TOIL the object is actually a reference to its value. Thus, if X is a variable of object type, then $\rho(X)$ is a memory location which holds a reference to the actual value of the object. Again, see Figure 4.4 for an illustration of object variables. In TOIL, for the `NEW` operation we do all the same kind of operations we did in TOOPL, and then use *PlaceInMem* to stick it in a location. The same kind of change is needed to model the *close* function. *Close* returns a value of type MT, which is bound to an object type, and thus should be a location as well. In the message-sending rule, we have to go in the other direction. Before we can send the object a message we need to look up the value stored in its location.

Furthermore, we have to be careful with instance variables. The initial values for instance variables are held by the class, but when we generate an object we want to store these initial values somewhere in memory. Thus, during a `NEW` we need to use *PlaceInMem* to allocate some memory in the store and place the correct values there. Now, in an actual object the instance values are stored as references. If the value of an instance variable is available through *self*, then we can assign directly to it since it is a reference.

Finally, our representation of *super* in the definition of `update` and `extend` reflects our discussion in Chapter 6. Instead of being simply a record of methods, *super* is modelled as a function which takes a parameter called *self* and returns the record of methods.

7.3.4 Formal semantics

Each clause in our definition of the semantics of terms corresponds to a type assignment axiom or rule from Chapter 5. All of the semantics are made under the condition that ρ is consistent with C,E. Consistency is formally defined in Section 7.4.

Program Semantics:

$[\![program\ p\ ;\ Block\ .\ :PROGRAM]\!]\rho_0\ s_0 =$

let $\langle Command, s' \rangle = [\![Block\colon COMMAND]\!]\rho_0\ s_0$

in $\langle Program, s' \rangle$.

Block and Declaration Semantics:

$[\![C, E \vdash const\ id = E; restBlock]\!]\rho\ s =$

let $\langle v_E, s' \rangle = [\![C, E \vdash E\colon \tau]\!]\rho\ s$

in $[\![restBlock]\!]\rho'\ s'$,

where $\rho' = UpdateEnv\ id\ v_E\ \rho$.

$[\![C, E \vdash var\ id\colon \tau; restBlock]\!]\rho\ s =$

let $\langle l, s' \rangle = GetLoc_{[\![\tau]\!]\rho}\ s$

in $[\![restBlock]\!]\rho'\ s'$,

where $\rho' = UpdateEnv\ id\ l\ \rho$.

$[\![begin\ S\ return\ R\ ;\ end\colon \sigma]\!]\rho\ s =$

let $\langle Command, s' \rangle = [\![C, E \vdash S\colon \mathrm{COMMAND}]\!]\rho\ s$

in $[\![C, E \vdash R\colon \sigma]\!]\rho\ s'$.

Command Semantics:

$[\![C, E \vdash X :=\ M : \mathrm{COMMAND}]\!]\rho\ s =$

let $\langle v_X, s' \rangle = [\![C, E \vdash X\colon ref\ \tau]\!]\rho\ s$

in let $\langle v_M, s'' \rangle = [\![C, E \vdash M\colon \tau]\!]\rho\ s'$

in $\langle Command, UpdateMem_{[\![\tau]\!]\rho}\ v_X\ v_M\ s'' \rangle$.

$[\![C, E \vdash if\ B\ then\ S\ else\ T\ end\colon \mathrm{COMMAND}]\!]\rho\ s =$

let $\langle v_B, s' \rangle = [\![C, E \vdash B\colon Bool]\!]\rho\ s$

in $\begin{cases} [\![C, E \vdash S\colon \mathrm{COMMAND}]\!]\rho\ s', & \text{if } v_B, \\ [\![C, E \vdash T\colon \mathrm{COMMAND}]\!]\rho\ s', & \text{if not } v_B. \end{cases}$

$[\![C, E \vdash while\ B\ do\ S\ end\colon \mathrm{COMMAND}]\!]\rho\ s =$

let $\langle v_B, s' \rangle = [\![C, E \vdash B\colon Bool]\!]\rho\ s$

in $\begin{cases} [\![C, E \vdash while\ B\ do\ S\ end\colon \mathrm{COMMAND}]\!]\rho\ s'', & \text{if } v_B, \\ \langle Command, s' \rangle, & \text{if not } v_B. \end{cases}$

where $\langle Command, s'' \rangle = [\![C, E \vdash S\colon \mathrm{COMMAND}]\!]\rho\ s'$.

$[\![C, E \vdash S;\ T\colon \mathrm{COMMAND}]\!]\rho\ s =$

let $\langle Command, s'\rangle = [\![C, E \vdash S\colon \mathrm{COMMAND}]\!]\rho\ s$

in $[\![C, E \vdash T\colon \mathrm{COMMAND}]\!]\rho\ s'$.

Expression Semantics:

$[\![C, E \vdash NIL\colon \bot]\!]\rho\ s = \langle errorLocation,\ s\rangle$.

$[\![C, E \vdash x\colon \tau]\!]\rho\ s = \langle \rho(x),\ s\rangle$.

$[\![C, E \vdash val\ x\colon \tau]\!]\rho\ s =$

let $\langle v_x, s'\rangle = [\![C, E \vdash x\colon ref\ \tau]\!]\rho\ s$

in $\langle LookupMem_{[\![\tau]\!]\rho}\ v_x\ s',\ s'\rangle$.

$[\![C, E \vdash function\ (v_1\colon \sigma_1; \ldots; v_n\colon \sigma_n)\ Block\colon (func(\sigma_1, \ldots, \sigma_n)\colon \tau)]\!]\rho\ s = \langle f,\ s\rangle$
where $f = \lambda s'\colon StoreType.\lambda\langle d_1, \ldots, d_n\rangle \in ([\![\sigma_1]\!]\rho \times \ldots \times [\![\sigma_n]\!]\rho.$
$[\![C, E \vdash Block\colon \tau]\!]\rho[d_1/v_1 \ldots d_n/v_n]\ s'.$

$[\![C, E \vdash \{m_1 = e_1, \ldots, m_n = e_n\}\colon \{m_1\colon \tau_1; \ldots; m_n\colon \tau_n\}]\!]\rho\ s^0 = \langle f,\ s^n\rangle$,
where $\langle v_i, s^i\rangle = [\![C, E \vdash e_i\colon \tau_i]\!]\rho\ s^{i-1}$, $1 \le i \le n$,
dom(f) $= \{m_1, \ldots, m_n\}$ and $f(m_i) = v_i$.

$[\![C, E \vdash e.m_i\colon \tau_i]\!]\rho\ s =$

let $\langle e', s'\rangle = [\![C, E \vdash e\colon \{m_1\colon \tau_1; \ldots; m_n\colon \tau_n\}]\!]\rho\ s$

in $\langle e'\ (m_i), s'\rangle$.

$[\![C, E \vdash M(N_1 \ldots N_n)\colon \tau]\!]\rho\ s =$

let $\langle v_M, s^1\rangle = [\![C, E \vdash M\colon func(\sigma_1 \ldots \sigma_n)\colon \tau]\!]\rho\ s$

in let $\langle v_{N_i}, s^{i+1}\rangle = [\![C, E \vdash N_i\colon \sigma_i]\!]\rho\ s^i,\ 1 \le i \le n$

in $(v_M)(s^{n+1})\langle v_{N_1}, \ldots, v_{N_n}\rangle$.

$[\![C, E \vdash o \Leftarrow m\colon \tau[\gamma/MT]]\!]\rho\ s =$

let $\langle o', s'\rangle = [\![C, E \vdash o\colon \gamma]\!]\rho\ s$

in let $o'' = \mathbf{convert}[\exists v.(v \times [\![\{m\colon \tau\}]\!]\rho[[\![\gamma]\!]\rho/MT])][\gamma']$
$(LookupMem_{\gamma'}\ o'\ s')$

in $\langle open\ o''\ as\ ob\ in\ (ob)_2\ (m), s'\rangle$.

where $[\![\gamma]\!]\rho = LOC_{\gamma'}$ for some γ'.

$[\![C, E \vdash class(e_1, e_2)\colon Tclass(MT)(\sigma,\tau)]\!]\rho\ s =$
$\langle \lambda\mu \leq_{\mathcal{A}} [\![\tau]\!]\rho'.\lambda\nu\ \mathbf{ext}\ [\![\sigma]\!]\rho'.\langle e_1', \lambda o \in \xi^S(\nu,\mu).e_2'\rangle,\ s''\rangle$

where $C^{IV} = C \cup \{MT \leq_{meth} Tobj(MT)\tau\}$,
$C^{METH} = C^{IV} \cup \{ST \leq (RecToMem(\sigma), \tau)\}$,
$E^{METH} = E \cup \{self\colon ST, close\colon func(ST)\colon MT\}$,
$\xi^S(\nu,\mu) = RecToMem(\nu) \times \mu$,
$\xi^P(\mu) = LOC_{\exists v.(v\times\mu)}$
$cl(\nu,\mu) = \lambda s \in StoreType.\lambda o \in A^{\xi^S(\nu,\mu)}.$
$\quad PlaceInMem_{\exists v.(v\times\mu)}\ (pack[v = RecToMem(\nu)\ in\ v \times \mu]\ o)\ s.$
$\rho' = \rho[\xi^P(\mu)/MT]$
$\rho'' = \rho'[\xi^S(\nu,\mu)/ST, o/self, cl(\nu,\mu)/close]$
$(e_1', s') = [\![C^{IV}, E \vdash e_1\colon\sigma]\!]\rho'\ s,$
$(e_2', s'') = [\![C^{METH}, E^{METH} \vdash e_2\colon\tau]\!]\rho''\ s',$

$[\![C, E \vdash new\ c\colon Tobj\ (MT)\tau]\!]\rho\ s = PlaceInMem_{\exists v.(v\times\mu)}\ hiddenObj\ s^{n+1},$

where $hiddenObj = pack[v = RecToMem(\nu)\ in\ v \times \mu]$
$\langle\{v_1 = Vloc_1, \ldots, v_n = Vloc_n\}, methfun\rangle,$

and $\xi^P = [\![Tobj\ (MT)\tau]\!]\rho,$
$\xi^S = RecToMem(\nu) \times \mu,$
$\rho' = \rho[\xi^P/MT],$
$\nu = [\![\sigma]\!]\rho',$
$\mu = [\![\tau]\!]\rho',$
$\langle c', s'\rangle = [\![C, E \vdash c\colon Tclass(MT)(\sigma,\tau)]\!]\rho\ s,$
$\langle initvar, meth_{approx}\rangle = (c')(\mu)(\nu)$
$\langle Vloc_i, s^{i+1}\rangle = PlaceInMem_{[\![\sigma_i]\!]\rho}(initvar(v_i))\ s^i$
$methfun = FIX(\lambda f \in \mu.meth_{approx}(\{v_1 = Vloc_1, \ldots, v_n = Vloc_n\}, f))$

$[\![C, E \vdash o.v_i\colon\sigma_i]\!]\rho\ s =$

let $\langle o', s'\rangle = [\![C, E \vdash o\colon\gamma]\!]\rho\ s$

in $\langle (o')_1(v_i), s'\rangle.$

$[\![C, E \vdash \mathit{update}\ c\ \mathit{by}\ (\{v_1 = a'_1\}, \{m_1 = e'_1\}):$
$\qquad TClass(MT)(\{v_1\colon \sigma_1; \ldots; v_m\colon \sigma_m\}, \{m_1\colon \tau'_1; m_2\colon \tau_2; \ldots; m_n\colon \tau_n\})]\!]\rho\ s =$
$\langle \lambda\mu \leq_{\mathcal{A}} [\![\{m_1\colon \tau'_1; m_2\colon \tau_2; \ldots; m_n\colon \tau_n\}]\!]\rho'.\lambda\nu\ \mathbf{ext}\ [\![\{v_1\colon \sigma_1; \ldots; v_m\colon \sigma_m\}]\!]\rho'.$
$\qquad \langle inst, \lambda o \in \xi^S(\nu,\mu).meth\rangle, s'''\rangle,$

where $C^{IV} = C \cup \{MT \leq_{meth} TObj(MT)\{m_1\colon \tau'_1; m_2\colon \tau_2; \ldots; m_n\colon \tau_n\}\}$,
$C^{METH} = C^{IV} \cup \{ST \leq (RecToMem(\{v_1\colon \sigma_1; \ldots; v_m\colon \sigma_m\}),$
$\quad \{m_1\colon \tau'_1; m_2\colon \tau_2; \ldots; m_n\colon \tau_n\})\}$,
$E^{METH} = E \cup \{self\colon ST, close\colon func(ST)\colon MT,$
$\quad super\colon \xi^S(\nu,\mu) \to \{m_1\colon \tau_1; \ldots; m_n\colon \tau_n\}\}$,
$\xi^P(\mu) = LOC_{\exists v.(v\times\mu)}$,
$\xi^S(\nu,\mu) = RecToMem(\nu) \times \mu$,
$\rho' = \rho[\xi^P(\mu)/MT]$,
$\rho'' = \rho'[\xi^S(\nu,\mu)/ST, o/self, cl(\nu,\mu)/close, (sup)_2/super]$,
$cl(\nu,\mu) = \lambda s \in StoreType.\lambda o \in A^{\xi^S(\nu,\mu)}.$
$\qquad PlaceInMem_{\exists v.(v\times\mu)}\ (pack[v = RecToMem(\nu)\ in\ v \times \mu]\ o)\ s.$
$\langle V_c, s'\rangle = [\![C, E \vdash c\colon TClass(MT)(\{v_1\colon \sigma_1; \ldots; v_m\colon \sigma_m\}, \{m_1\colon \tau_1; \ldots; m_n\colon \tau_n\})]\!]\rho\ s,$
$sup = (V_c)(\mu)(\nu)$,
$supmeth = (sup)_2\ (o)$,
$supvar = (sup)_1$,
$\langle V_{a'_1}, s''\rangle = [\![C^{IV}, E \vdash a'_1\colon \sigma_1]\!]\rho'\ s'$
$\langle V_{e'_1}, s'''\rangle = [\![C^{METH}, E^{METH} \vdash e'_1\colon \tau'_1]\!]\rho''\ s''$
$inst = supvar[V_{a'_1}/v_1]$,
$dom(meth) = \{m_1, \ldots, m_n\}$,
$meth(m_1) = V_{e'_1}$,
and for $2 \leq j \leq n$,
$\quad meth(m_j) = supmeth(m_j)$.

$[\![C, E \vdash \mathit{extend}\ c\ \mathit{with}\ (\{v_{m+1} = a_{m+1}\}, \{m_{n+1} = e_{n+1}\}):$
$\qquad TClass(MT)(\{v_1\colon \sigma_1; \ldots; v_{m+1}\colon \sigma_{m+1}\}, \{m_1\colon \tau_1; \ldots; m_{n+1}\colon \tau_{n+1}\})]\!]\rho\ s =$
$\langle \lambda\mu \leq_{\mathcal{A}} [\![\{m_1\colon \tau_1; \ldots; m_{n+1}\colon \tau_{n+1}\}]\!]\rho'.\lambda\nu\ \mathbf{ext}\ [\![\{v_1\colon \sigma_1; \ldots; v_{m+1}\colon \sigma_{m+1}\}]\!]\rho'.$
$\qquad \langle inst, \lambda o \in \xi^S(\nu,\mu).meth\rangle, s'''\rangle,$

where $C^{IV} = C \cup \{MT \leq_{meth} TObj(MT)\{m_1\colon \tau_1; \ldots; m_{n+1}\colon \tau_{n+1}\}\}$,
$C^{METH} = C^{IV} \cup$
$\quad \{ST \leq (RecToMem(\{v_1\colon \sigma_1; \ldots; v_{m+1}\colon \sigma_{m+1}\})), \{m_1\colon \tau_1; \ldots; m_{n+1}\colon \tau_{n+1}\})\}$,
$E^{METH} = E \cup \{self\colon ST, close\colon func(ST)\colon MT,$
$\quad super\colon \xi^S(\nu,\mu) \to \{m_1\colon \tau_1; \ldots; m_n\colon \tau_n\})\}$,
$\xi^P(\mu) = LOC_{\exists v.(v\times\mu)}$,
$\xi^S(\nu,\mu) = RecToMem(\nu) \times \mu$,

$\rho' = \rho[\xi^P(\mu)/MT]$,
$\rho'' = \rho'[\xi^S(\nu,\mu)/ST, o/self, cl(\nu,\mu)/close, (sup)_2/super]$
$cl(\nu,\mu) = \lambda s \in StoreType.\lambda o \in A^{\xi^S(\nu,\mu)}.$
$\quad PlaceInMem_{\exists v.(v\times\mu)}\ (pack[v = RecToMem(\nu)\ in\ v\times\mu]\ o)\ s.$
$\langle V_c, s'\rangle = [\![C, E \vdash c{:}\, TClass(MT)(\{v_1{:}\,\sigma_1;\ldots;v_m{:}\,\sigma_m\},\{m_1{:}\,\tau_1;\ldots;m_n{:}\,\tau_n\})]\!]\rho\ s,$
$sup = (V_c)(\mu)(\nu),$
$supmeth = (sup)_2\ (o),$
$supvar = (sup)_1,$
$\langle V_{a_{m+1}}, s''\rangle = [\![C^{IV}, E \vdash a_{m+1}{:}\,\sigma_{m+1}]\!]\rho'\ s'$
$\langle V_{e_{n+1}}, s'''\rangle = [\![C^{METH}, E^{METH} \vdash e_{n+1}{:}\,\tau_{n+1}]\!]\rho''\ s''$
$inst = supvar[V_{a_{m+1}}/v_{m+1}],$
$dom(meth) = \{m_1,\ldots,m_{n+1}\},$
$meth(m_{n+1}) = V_{e_{n+1}}$
and for $1 \le j \le n$,
$meth(m_j) = supmeth(m_j).$

$[\![C, E \vdash e{:}\,\sigma]\!]\rho\ s = \langle \mathbf{convert}[[\![\sigma]\!]\rho][[\![\tau]\!]\rho]\ V_e, s'\rangle,$

where $\langle V_e, s'\rangle = [\![C, E \vdash e{:}\,\tau]\!]\rho\ s$, if $C \vdash \tau \le \sigma$ and $C, E \vdash e{:}\,\tau$.

7.4 Definitions and lemmas on consistency

This section is highly technical and presents some formal definitions and lemmas relating to consistency which will be used in the proof in the next chapter.

Definition 7.4.1 *The following inductive definition determines when a type environment ρ is* consistent *with a type constraint system C:*

1. *If C is empty then ρ is consistent with C.*
2. *Suppose $C \cup \{t \le \sigma\}$ is a type constraint system. If ρ is consistent with C and $\rho(t) \le_{\mathcal{A}} [\![\sigma]\!]\rho$ then ρ is* consistent *with $C \cup \{t \le \sigma\}$.*
3. *Suppose $C \cup \{t \le_{meth} TObj(MT)\tau\}$ is a type constraint system. If ρ is consistent with C and $\rho(t) = LOC_\gamma$, where $\gamma \le_{\mathcal{A}} \exists v.(v \times [\![\tau]\!]\rho[\rho(t)/MT])$ then ρ is* consistent *with $C \cup \{t \le_{meth} TObj(MT)\tau\}$.*

Intuitively, ρ is consistent with C if the values bound to type variables in ρ do not violate any of the constraints in C. Thus, in rule 2, a constraint $\{t \le \sigma\}$ can be added to C if the current meaning of t in ρ is a subtype of the meaning of σ. The rule for the $\le_{meth}$ relation on object types is similar except that it takes into account the fact that the meaning of an object type is really a location.

Definition 7.4.2 *Let ρ be a semantic environment mapping type variables to types and regular variables to elements of $\mathcal{A}$. We say ρ is* consistent *with respect to C, E, if ρ is consistent with C and, for all x in the domain of ρ, if $E(x) = \sigma$ then $\rho(x) \in [\![\sigma]\!]\rho$.*

Lemma 7.4.3 *Let ρ be a standard type environment. Suppose σ is freely substitutable in τ for y (i.e. no variable in σ is captured by $\exists$, TObj, TClass). Then $[\![\tau]\!]\rho[[\![\sigma]\!]\rho/y] = [\![\tau[\sigma/y]]\!]\rho$.*

Proof. Left to the reader.

Lemma 7.4.4 *Let the type environment ρ be consistent with C.*

1. *If $C \vdash \sigma \leq \tau$ then $[\![\sigma]\!]\rho \leq_{\mathcal{A}} [\![\tau]\!]\rho$.*
2. *If $C \vdash \gamma \leq_{meth} TObj(MT)\tau$ then $[\![\gamma]\!]\rho = LOC_{\gamma'}$, where $\gamma' \leq_{\mathcal{A}} \exists v.(v \times [\![\tau]\!]\rho[[\![\gamma]\!]\rho/MT])$*

Proof. The proof is done by induction on the length of derivations. The proof for (1) is similar to that for Lemma 4.3 of [Bru93]. The assumption for (2) can only be proved through the $(Var(\leq_{meth}))$ axiom, the $(Refl(\leq_{meth}))$ axiom, or the $(Trans(\leq_{meth}))$ rule.

Case $(Var(\leq_{meth}))$: γ is some type variable t. Then $C \cup \{t \leq_{meth} \tau\} \vdash t \leq_{meth} \tau$. Since ρ is consistent with $C \cup \{t \leq_{meth} \tau\}$, by our definition of consistency we can show that $\rho(t) = LOC_{\gamma'}$, where $\gamma' \leq_{\mathcal{A}} \exists v.(v \times [\![\tau]\!]\rho[\rho(t)/MT])$. Since $[\![\gamma]\!]\rho = \rho(t)$ from our definition of the meanings of types, we can show that $[\![\gamma]\!]\rho = LOC_{\gamma'}$, where $\gamma' \leq_{\mathcal{A}} \exists v.(v \times [\![\tau]\!]\rho[[\![\gamma]\!]\rho/MT])$.

Case $(Refl(\leq_{meth}))$: γ is an object type exactly equal to $TObj(MT)\tau$. Then $C \vdash TObj(MT)\tau \leq_{meth} TObj(MT)\tau$, so $C \vdash \gamma \leq_{meth} TObj(MT)\tau$. From our definition of types, $[\![\gamma]\!]\rho = LOC_{\gamma'}$ where $\gamma' = \exists v.(v \times [\![\tau]\!]\rho[[\![Tobj(MT)\tau]\!]\rho/MT]) = \exists v.(v \times [\![\tau]\!]\rho[[\![\gamma]\!]\rho/MT])$. Since subtyping is reflexive, $\gamma' \leq_{\mathcal{A}} \exists v.(v \times [\![\tau]\!]\rho[[\![\gamma]\!]\rho/MT])$.

Case $(Trans(\leq_{meth}))$: Suppose that $C \vdash \gamma \leq_{meth} TObj(MT)\tau$ and $C \vdash \tau \leq \tau'$ prove that $C \vdash \gamma \leq_{meth} TObj(MT)\tau'$. By induction, we can assume that

$$[\![\gamma]\!]\rho = LOC_{\gamma'}, \text{ where } \gamma' \leq_{\mathcal{A}} \exists v.(v \times [\![\tau]\!]\rho[[\![\gamma]\!]\rho/MT]).$$

Furthermore, by (1) the subtype relation for τ and τ' holds over any environment consistent with C, so:

$$[\![\tau]\!]\rho[[\![\gamma]\!]\rho/MT] \leq_{\mathcal{A}} [\![\tau']\!]\rho[[\![\gamma]\!]\rho/MT].$$

Therefore, $[\![\gamma]\!]\rho = LOC_{\gamma'}$, where

$\gamma' \leq_{\mathcal{A}} \exists v.(v \times [\![\tau]\!]\rho[[\![\gamma]\!]\rho/MT])$

$\quad \leq_{\mathcal{A}} \exists v.(v \times [\![\tau']\!]\rho[[\![\gamma]\!]\rho/MT])$

and we are done.

Chapter 8

Proof of Type Safety

8.1 Significance of the proof

In the last chapter, we presented a formal mathematical model for TOIL. Each syntactic construct of the language is mapped onto a semantic value from the typed lambda calculus. Also, Chapter 5 contains a set of type assignment rules and axioms which can be used to prove that a syntactic construct has a certain syntactic type. We would like to show that our semantics are consistent with the typing rules. In other words, consider an expression M which we can show using the type assignment rules to have syntactic type τ, so $C, E \vdash M: \tau$. Then we should be able to show that the meaning of M according to our denotational semantics is an element of the meaning of type τ.

The significance of this proof is that it shows that TOIL, as defined by the denotational semantics presented in Chapter 7, is type-safe. The ability to formally prove type safety is in fact a major reason for the development of the mathematical model in the first place. The proof proceeds by induction on the length of type derivations, and considers each type assignment rule in turn.

To avoid complicating the proof, we often ignore the case where the value of a computation is $\perp$. However, all of these cases are trivial.

8.2 Theorem

Theorem 8.2.1 *(Soundness of semantics with respect to the typing rules) Suppose there is a model for the F-bounded second order lambda calculus with fixed points at both the type and term levels. For all typing derivations, $C, E \vdash M: \tau$, if ρ is consistent with C, E, and s is consistent, then:*

$[\![C, E \vdash M: \tau]\!]\rho\ s \in ([\![\tau]\!]\rho \times StoreType)_\perp$.

Proof. By induction on the length of derivations:

Program:

$C_0, E_0 \vdash program\ p\ ;\ Block.\ : PROGRAM$ follows from $C_0, E_0 \vdash Block$: COMMAND.
We can show using induction that:

$[\![program\ p\ ;\ Block\ .\ : PROGRAM]\!]\rho_0\ s_0 =$

let $\langle Command, s' \rangle = [\![Block\text{: COMMAND}]\!]\rho_0\ s_0 \in ([\![\text{COMMAND}]\!]\rho \times StoreType)_\perp$

in $\langle Program, s' \rangle \in ([\![PROGRAM]\!]\rho \times StoreType)$.
$[\![Block\text{: } PROGRAM]\!]\rho_0\ s_0 \in ([\![PROGRAM]\!]\rho \times StoreType)_\perp$.

For the next case of blocks, we need the following lemma:

Lemma 8.2.2

1. *Consider C, E consistent with ρ, and a consistent store s, and the ρ' and s' which result from a constant declaration $id = E$. Then if $C, E \vdash E\text{:}\tau$, then ρ' is consistent with $C, E \cup \{id\text{:}\tau\}$, and s' is consistent.*

2. *Consider C, E consistent with ρ, and a consistent store s, and the ρ' and s' which result from a variable declaration $id\text{:}\tau$. Then ρ' is consistent with $C, E \cup \{id\text{:}\ ref\ \tau\}$, and s' is consistent.*

Proof:

(Const) Case:

$[\![C, E \vdash const\ id = E; restD]\!]\rho\ s =$

let $\langle v_E, s' \rangle = [\![C, E \vdash E\text{:}\tau]\!]\rho\ s$

in $[\![restD]\!]\rho'\ s'$,

where $\rho' = UpdateEnv\ id\ v_E\ \rho$.

Now, $(E \cup \{id\text{:}\tau\})id = \tau$ and $\rho'(id) = v_E \in [\![\tau]\!]\rho$. Since C remains unchanged throughout the declarations, we can use Definition 7.4.2 to show that $C, E \cup \{id\text{:}\tau\}$ is consistent with ρ'. Furthermore, s' is still consistent.

(Var) Case:

$[\![C, E \vdash var\ id\text{:}\tau; restD]\!]\rho\ s =$

let $\langle l, s' \rangle = GetLoc_{[\![\tau]\!]\rho}\ s$

in $[\![restD]\!]\rho'\ s'$,
where $\rho' = UpdateEnv\ id\ l\ \rho$.

The proof is similar to case for constant declarations, except that we are adding the assumption $\{id\colon ref\ \tau\}$ for each declaration, and using *UpdateEnv* to bind identifiers to locations. In fact, for *id*, a location is reserved which is guaranteed to hold elements of type $[\![\tau]\!]\rho$. The initial value will be *errorFlag*, which by definition is a member of $[\![\tau]\!]\rho$. By the meaning of types, we know that $[\![ref\ \tau]\!]\rho = LOC_{[\![\tau]\!]\rho}$. Thus we can show that $(E \cup \{id\colon ref\ \tau\})id = ref\ \tau$ and $\rho(id) \in [\![ref\ \tau]\!]\rho$. Again, we can use Definition 7.4.2 to prove that $C, E \cup \{id\colon ref\ \tau\}$ is consistent with ρ'. Furthermore, $s'(l) \in [\![\tau]\!]\rho$, and therefore, s' is still consistent.

We can use Lemma 8.2.2 and induction to show that all of the declarations at the start of a block result in a consistent store s and a new ρ which is consistent with $C, E \cup \{all\ the\ declarations\}$. Now, back to the proof of Theorem 8.2.1.

Blocks:

$C, E \vdash begin\ S\ return\ R :\ \sigma;\ end\ :\sigma$ follows from $C, E \vdash S\colon$ COMMAND and $C, E \vdash R\colon\sigma$. Then:

$[\![begin\ S\ return\ R\ ;\ end\colon\sigma]\!]\rho\ s =$

let $\langle Command, s'\rangle = [\![C, E \vdash S\colon \text{COMMAND}]\!]\rho\ s \in ([\![\text{COMMAND} \times StoreType]\!]\rho)_\perp$

in $[\![C, E \vdash R\colon\sigma]\!]\rho\ s'$.

By induction, S evaluates to an element of type $[\![\text{COMMAND}]\!]\rho$. Note that the type-checking rules are written so that C,E, and ρ already include all of the constant and variable declarations from the block, and from Lemma 8.2.2 we know that ρ is consistent with C and E. Thus, the block returns an element of $([\![\sigma]\!]\rho \times StoreType)_\perp$, and we are done.

Commands:

(Assn) Case:

$C, E \vdash X := M\colon$ COMMAND follows from $C, E \vdash X\colon ref\ \tau,\ C, E \vdash M\colon\tau$. Then:

$[\![C, E \vdash X :=\ M : \text{COMMAND}]\!]\rho\ s =$

let $\langle v_X, s'\rangle = [\![C, E \vdash X\colon ref\ \tau]\!]\rho\ s \in (LOC_{[\![\tau]\!]\rho} \times StoreType)_\perp$

in let $\langle v_M, s''\rangle = [\![C, E \vdash M\colon\tau]\!]\rho\ s' \in ([\![\tau]\!]\rho \times StoreType)_\perp$

in $\langle Command, UpdateMem_{[\![\tau]\!]\rho}\ v_X\ v_M\ s''\rangle$.

$\in (CommandType \times StoreType)_\perp$.

By definition, $UpdateMem_T\colon LOC_T \to T \to StoreType \to StoreType$. Above, $v_X \in LOC_{[\![\tau]\!]\rho}$, and $v_M \in [\![\tau]\!]\rho$, both by induction. Thus, the $UpdateMem_{[\![\tau]\!]\rho}$ function is well typed, and the result is a consistent state.

(Cond) Case:

$(C, E \vdash \mathit{if}\ B\ \mathit{then}\ S\ \mathit{else}\ T\ \mathit{end}\colon \mathrm{COMMAND})$ follows from $(C, E \vdash B\colon \mathit{Bool})$, $(C, E \vdash S\colon \mathrm{COMMAND})$, $(C, E \vdash T\colon \mathrm{COMMAND})$. Then:

$[\![C, E \vdash \mathit{if}\ B\ \mathit{then}\ S\ \mathit{else}\ T\ \mathit{end}\colon \mathrm{COMMAND}]\!]\rho\ s =$

$$\text{let}\ \ \langle v_B, s'\rangle = [\![C, E \vdash B\colon \mathit{Bool}]\!]\rho\ s \in (\{\mathit{true}, \mathit{false}\} \times \mathit{StoreType})_\perp$$

$$\text{in}\ \begin{cases} [\![C, E \vdash S\colon \mathrm{COMMAND}]\!]\rho\ s', & \text{if } v_B, \\ [\![C, E \vdash T\colon \mathrm{COMMAND}]\!]\rho\ s', & \text{if not } v_B. \end{cases}$$

By induction, both cases return an element of $(\mathit{CommandType} \times \mathit{StoreType})_\perp$, so our final result is an element of $(\mathit{CommandType} \times \mathit{StoreType})_\perp$.

(While) Case:

$C, E \vdash \mathit{while}\ B\ \mathit{do}\ S\ \mathit{end}\colon \mathrm{COMMAND}$ follows from
$C, E \vdash B\colon \mathit{Bool},\ C, E \vdash S\colon \mathrm{COMMAND}$. Then:
$[\![C, E \vdash \mathit{while}\ B\ \mathit{do}\ S\ \mathit{end}\colon \mathrm{COMMAND}]\!]\rho\ s =$

$$\text{let}\ \ \langle v_B, s'\rangle = [\![C, E \vdash B\colon \mathit{Bool}]\!]\rho\ s \in (\{\mathit{true}, \mathit{false}\} \times \mathit{StoreType})_\perp$$

$$\text{in}\ \begin{cases} [\![C, E \vdash \mathit{while}\ B\ \mathit{do}\ S\ \mathit{end}\colon \mathrm{COMMAND}]\!]\rho\ s'', & \text{if } v_B, \\ \langle \mathit{Command}, s'\rangle, & \text{if not } v_B. \end{cases}$$

$$\text{where}\ \langle \mathit{Command}, s''\rangle = [\![C, E \vdash S\colon \mathrm{COMMAND}]\!]\rho\ s' \in (\mathit{CommandType} \times \mathit{StoreType})_\perp.$$

If v_B is false, then $(\mathit{Command}, s') \in (\mathit{CommandType} \times \mathit{StoreType})_\perp$. If v_B is true, then we return the result of a computation, which is essentially the recursive unrolling of the while loop. If the loop does not terminate, then our result will be $\perp_{\mathit{Command}} \in (\mathit{CommandType} \times \mathit{StoreType})_\perp$. If the loop does terminate, however, we have a finite chain of computations upon which we can use induction (with the terminating case where v_B is false as the base case) to show that the result of each computation $\in (\mathit{CommandType} \times \mathit{StoreType})_\perp$.

(StmtList) Case:

$C, E \vdash S;\ T\colon \mathrm{COMMAND}$ follows from
$(C, E \vdash S\colon \mathrm{COMMAND}),\ (C, E \vdash T\colon \mathrm{COMMAND})$. Then:
$[\![C, E \vdash S;\ T\colon \mathrm{COMMAND}]\!]\rho\ s =$

$$\text{let}\ \ \langle \mathit{Command}, s'\rangle = [\![C, E \vdash S\colon \mathrm{COMMAND}]\!]\rho\ s$$

$$\text{in}\ \ [\![C, E \vdash T\colon \mathrm{COMMAND}]\!]\rho\ s'.$$

Easy using induction.

Expressions:

(Nil) Case:
From our definition of types, we have $[\![\bot]\!]\rho = \{errorLocation\}$. Then:
$[\![C, E \vdash NIL\colon \bot]\!]\rho\ s = \langle errorLocation,\ s\rangle \in ([\![\bot]\!]\rho \times StoreType)_\bot$.

(Var) Case:
$C, E \vdash X\colon \tau$ follows from $E(X) = \tau$. Then $[\![C, E \vdash X\colon \tau]\!]\rho\ s = \langle \rho(X),\ s\rangle$. Since ρ is consistent with C, E, it follows from our definition of consistency that $\rho(X) \in [\![\tau]\!]\rho$. Furthermore, $s \in StoreType$, so $(\rho(X),\ s)$ is an element of $([\![\tau]\!]\rho \times StoreType)_\bot$.

(Ref) Case:
$C, E \vdash val\ X\colon \tau$ follows from $C, E \vdash X\colon ref\ \tau$. Then
$[\![C, E \vdash val\ X\colon \tau]\!]\rho\ s =$

let $\langle v_X, s'\rangle = [\![C, E \vdash X\colon ref\ \tau]\!]\rho\ s \in (LOC_{[\![\tau]\!]\rho}, StoreType)_\bot$

in $\langle LookupMem_{[\![\tau]\!]\rho}\ v_X\ s',\ s'\rangle$.
$\in ([\![\tau]\!]\rho, StoreType)_\bot$.
LookupMem is well typed and returns an element of $[\![\tau]\!]\rho$ since $v_X \in LOC_{[\![\tau]\!]\rho}$.
(Appl) Case:

$C, E \vdash M(N_1, \ldots, N_n)\colon \tau$
follows from $C, E \vdash M\colon func(\sigma_1, \ldots, \sigma_n)\colon \tau$, $\quad C, E \vdash N_i\colon \sigma_i$, $\quad for \forall\ 1 \le i \le n$.
$[\![C, E \vdash M(N_1 \ldots N_n)\colon \tau]\!]\rho\ s =$

let $\langle v_M, s^1\rangle = [\![C, E \vdash M\colon func(\sigma_1 \ldots \sigma_n)\colon \tau]\!]\rho\ s \in ([\![func(\sigma_1 \ldots \sigma_n)\colon \tau]\!]\rho \times StoreType)_\bot$

in let $\langle v_{N_i}, s^{i+1}\rangle = [\![C, E \vdash N_i\colon \sigma_i]\!]\rho\ s^i \in ([\![\sigma_i]\!]\rho \times StoreType)_\bot,\ 1 \le i \le n$

in $(v_M)(s^{n+1})\langle v_{N_1}, \ldots, v_{N_n}\rangle$.
By our definition of the meanings of types,
$v_M \in [\![func(\sigma_1 \ldots \sigma_n)\colon \tau]\!]\rho = StoreType \to ([\![A_1]\!]\rho \times \ldots \times [\![A_n]\!]\rho) \to ([\![B]\!]\rho \times StoreType)$.
Thus, we have:
$[\![C, E \vdash M(N_1 \ldots N_n)\colon \tau]\!]\rho\ s$
$= (v_M)(s^{n+1})\langle v_{N_1}, \ldots, v_{N_n}\rangle$
$\in ([\![\tau]\!]\rho \times StoreType)_\bot$.

(Abs) Case:
$C, E \vdash function\ (v_1\colon \sigma_1; \ldots; v_n\colon \sigma_n)\ Block\colon\ (func(\sigma_1, \ldots, \sigma_n)\colon \tau)$ follows from
$C, E \cup \{v_1\colon \sigma_1, \ldots, v_n\colon \sigma_n\} \vdash Block\colon \tau$.

From our definition of the meaning of types:

$[\![func(\sigma_1, \ldots, \sigma_n)\colon \tau]\!]\rho = StoreType \to ([\![\sigma_1]\!]\rho \times \ldots \times [\![\sigma_n]\!]\rho) \to ([\![\tau]\!]\rho \times StoreType)$

Then:

$[\![C, E \vdash function\ (v_1\colon \sigma_1; \ldots; v_n\colon \sigma_n)\ Block\colon (func(\sigma_1, \ldots, \sigma_n)\colon \tau)]\!]\rho\ s = \langle f,\ s\rangle$

where $f = \lambda s' \in StoreType.\lambda(d_1, \ldots, d_n) \in ([\![\sigma_1]\!]\rho \times \ldots \times [\![\sigma_n]\!]\rho).$

$[\![C, E \vdash Block\colon \tau]\!]\rho[d_1/v_1 \ldots d_n/v_n]\ s'.$

Since $d_i \in [\![\sigma_i]\!]\rho$, $\rho[d_1/v_1 \ldots d_n/v_n]$ is consistent with $C, E \cup \{v_1\colon \sigma_1, \ldots, v_n\colon \sigma_n\}$ by our definition of consistency.

By induction, then:

$[\![C, E \vdash function\ (v_1\colon \sigma_1; \ldots; v_n\colon \sigma_n)\ Block\colon func(\sigma_1, \ldots, \sigma_n)\colon \tau]\!]\rho\ s$

$= \langle \lambda s' \in StoreType.\lambda(d_1, \ldots, d_n) \in ([\![\sigma_1]\!]\rho \times \ldots \times [\![\sigma_n]\!]\rho).$

$[\![C, E \vdash Block\colon \tau]\!]\rho[d_1/v_1 \ldots d_n/v_n]\ s',\ s\rangle$

$\in ([\![func(\sigma_1, \ldots, \sigma_n)\colon \tau]\!]\rho \times StoreType)_\perp$ -Meanings of types

(Rec) Case:

$C, E \vdash \{m_1 = e_1, \ldots, m_n = e_n\}\colon \{m_1\colon \tau_1; \ldots; m_n\colon \tau_n\}$ follows from $C, E \vdash e_i\colon \tau_i, \quad for \forall\ 1 \leq i \leq n.$

From our definition of the meanings of types:

$[\![\{m_1\colon \tau_1, \ldots, m_n\colon \tau_n\}]\!]\rho = \prod_{m_i \in \{m_1, \ldots, m_n\}} [\![\tau_i]\!]\rho.$

Then:

$[\![C, E \vdash \{m_1 = e_1, \ldots, m_n = e_n\}\colon \{m_1\colon \tau_1; \ldots; m_n\colon \tau_n\}]\!]\rho\ s^0 = \langle f,\ s^n\rangle,$

where $\langle v_i, s^i\rangle = [\![C, E \vdash e_i\colon \tau_i]\!]\rho\ s^{i-1} \in ([\![\tau_i]\!]\rho \times StoreType)_\perp, \quad 1 \leq i \leq n,$

$\mathrm{dom(f)} = \{m_1, \ldots, m_n\}$ and $f(m_i) = v_i.$

$\in ((\prod_{m_i \in \{m_1, \ldots, m_n\}} [\![\tau_i]\!]\rho) \times StoreType)_\perp$

$\in (([\![\{m_1\colon \tau_1, \ldots, m_n\colon \tau_n\}]\!]\rho) \times StoreType)_\perp.$

(Proj) Case:

$C, E \vdash e.m_i\colon \tau_i$ follows from $C, E \vdash e\colon \{m_1\colon \tau_1; \ldots; m_n\colon \tau_n\}.$

$[\![C, E \vdash e.m_i\colon \tau_i]\!]\rho\ s =$

let $\langle e', s'\rangle = [\![C, E \vdash e\colon \{m_1\colon \tau_1; \ldots; m_n\colon \tau_n\}]\!]\rho\ s$

$\in ([\![\{m_1\colon \tau_1; \ldots; m_n\colon \tau_n\}]\!]\rho \times StoreType)_\perp$

$= ((\prod_{m_i \in \{m_1, \ldots, m_n\}} [\![\tau_i]\!]\rho) \times StoreType)_\perp$

in $\langle e'\ (m_i), s'\rangle.$

$\in ([\![\tau_i]\!]\rho \times StoreType)_\perp.$

(Class) Case:

$C, E \vdash class(e_1, e_2)\colon Tclass(MT)(\sigma, \tau)$ follows from

$C^{IV} = C \cup \{MT \leq_{meth} Tobj(MT)\tau\}, E \vdash e_1 : \sigma,$
$C^{METH} = C^{IV} \cup \{ST \leq (RecToMem(\sigma), \tau)\},$
$E^{METH} = E \cup \{self : ST, close : func(ST) : MT\} \vdash e_2 : \tau$

The corresponding semantic rule is:

$[\![C, E \vdash class(e_1, e_2) : Tclass(MT)(\sigma, \tau)]\!]\rho\ s =$
$\langle \lambda\mu \leq_{\mathcal{A}} [\![\tau]\!]\rho'.\lambda\nu\ \mathbf{ext}\ [\![\sigma]\!]\rho'.(e_1', \lambda o \in \xi^S(\nu, \mu).e_2''),\ s'' \rangle$

where $C^{IV} = C \cup \{MT \leq_{meth} Tobj(MT)\tau\},$
$C^{METH} = C^{IV} \cup \{ST \leq (RecToMem(\sigma), \tau)\},$
$E^{METH} = E \cup \{self : ST, close : func(ST) : MT\},$
$(e_1', s') = [\![C^{IV}, E \vdash e_1 : \sigma]\!]\rho'\ s,$
$(e_2', s'') = [\![C^{METH}, E^{METH} \vdash e_2 : \tau]\!]\rho''\ s',$
$\xi^S(\nu, \mu) = RecToMem(\nu) \times \mu,$
$\xi^P(\mu) = LOC_{\exists v.(v \times \mu)}$
$cl(\nu, \mu) = \lambda s \in StoreType.\lambda o \in A^{\xi^S(\nu,\mu)}.$
$\quad PlaceInMem_{\exists v.(v \times \mu)}\ (pack[v = \nu\ in\ v \times \mu]\ o)\ s.$
$\rho' = \rho[\xi^P(\mu)/MT]$
$\rho'' = \rho'[\xi^S(\nu, \mu)/ST, o/self, cl(\nu, \mu)/close]$

To be able to use induction, we must show that C^{IV} and E are consistent with ρ', and that C^{METH} and E^{METH} are consistent with ρ'', since those are the environments in which the instance variables and methods, respectively, are evaluated.

$\rho'(MT) = \xi^P(\mu)$
$= LOC_\gamma$, where $\gamma = \exists v.(v \times \mu)$ - substitution
$\gamma \leq_{\mathcal{A}} \exists v.(v \times [\![\tau]\!]\rho')$ since $\mu \leq_{\mathcal{A}} [\![\tau]\!]\rho'.$
$= \exists v.(v \times [\![\tau]\!]\rho[\xi^P(\mu)/MT])$ -substitution
$= \exists v.(v \times [\![\tau]\!]\rho[\rho'(MT)/MT]).$ -substitution

Thus, by our definition of consistency, ρ' is consistent with C^{IV}. E is unchanged, and ρ' is the same as ρ on regular variables, so ρ' is consistent with C^{IV} and E. Similarly:

$\rho''(ST) = \xi^S(\nu, \mu)$
$= RecToMem(\nu) \times \mu$ - substitution
$\leq_{\mathcal{A}} RecToMem([\![\sigma]\!]\rho') \times [\![\tau]\!]\rho'$ - substitution,subtyping
$= [\![(RecToMem(\sigma), \tau)]\!]\rho'.$ - Meanings of types

Therefore, ρ'' is consistent with C^{METH}. Furthermore,

$E^{METH}(self) = ST$, and $\rho''(self) = o \in \xi^S(\nu, \mu)$ and $\xi^S(\nu, \mu) = \rho''(ST) = [\![ST]\!]\rho''.$
$E^{METH}(close) = func(ST) : MT$, and
$\rho''(close) = cl(\nu, \mu) = \lambda s \in StoreType.\lambda o \in \xi^S(\nu, \mu).$
$PlaceInMem_{\exists v.(v \times \mu)}\ (pack[v = RecToMem(\nu)\ in\ v \times \mu]\ o)\ s$
$\in StoreType \to \xi^S(\nu, \mu) \to (LOC_{\exists v.(v \times \mu)} \times StoreType)_\perp$
$\in StoreType \to [\![ST]\!]\rho'' \to ([\![MT]\!]\rho'' \times StoreType)_\perp$

$\in [\![func(ST)\colon MT]\!]\rho''$.

Therefore, ρ'' is consistent with C^{METH}, E^{METH} and we can use induction. Now:

$\langle \lambda\mu \leq_{\mathcal{A}} [\![\tau]\!]\rho'.\lambda\nu$ **ext** $[\![\sigma]\!]\rho'.(e_1', \lambda o \in \xi^S(\nu,\mu).e_2''),\ s''\rangle$

$\in ((\prod_{\mu \leq_{\mathcal{A}} [\![\tau]\!]\rho'} \prod_{\nu \textbf{ ext } [\![\sigma]\!]\rho'} ([\![\sigma]\!]\rho' \times (\xi^S(\nu,\mu) \to [\![\tau]\!]\rho'))) \times StoreType)_\perp$ -substitution

$\in ([\![TClass(MT)(\sigma,\tau)]\!]\rho \times StoreType)_\perp$. Meaning of types

(Object) Case:

$C, E \vdash new\ c\colon Tobj(MT)\tau$ follows from $C, E \vdash c\colon Tclass(MT)(\sigma,\tau)$. Then:

$[\![C, E \vdash new\ c\colon Tobj\ (MT)\tau]\!]\rho\ s = PlaceInMem_{\exists v.(v \times \mu)}\ hiddenObj\ s^{n+1}$,

where $hiddenObj = pack[v = RecToMem(\nu)\ in\ v \times \mu]$
$(\{v_1 = Vloc_1, \ldots, v_n = Vloc_n\}, methfun)$,

and $\xi^P = [\![Tobj\ (MT)\tau]\!]\rho$,
$\xi^S = RecToMem(\nu) \times \mu$,
$\rho' = \rho[\xi^P/MT]$,
$\nu = [\![\sigma]\!]\rho'$,
$\mu = [\![\tau]\!]\rho'$,
$(c, s') = [\![C, E \vdash c\colon Tclass(MT)(\sigma,\tau)]\!]\rho\ s \in ([\![Tclass(MT)(\sigma,\tau)]\!]\rho \times StoreType)$,
$(initvar, meth_{approx}) = (c)(\mu)(\nu) \in ([\![\sigma]\!]\rho' \times (\xi^S(\nu,\mu) \to [\![\tau]\!]\rho'))$
$\langle Vloc_i, s^{i+1}\rangle = PlaceInMem_{[\![\sigma_i]\!]\rho}(initvar)(v_i)\ s^i$
$methfun = FIX(\lambda f \in \mu.meth_{approx}(\{v_1 = Vloc_1, \ldots, v_n = Vloc_n\}, f))$

Note that initvar can be applied to v_i because σ is guaranteed to be a record type by the syntax. Furthermore, $meth_{approx} \in (\xi^S(\nu,\mu) \to [\![\tau]\!]\rho')$, and $\{v_1 = Vloc_1, \ldots, v_n = Vloc_n\} \in \xi^S$. Thus, if $f \in \mu$:

$(meth_{approx})(\{v_1 = Vloc_1, \ldots, v_n = Vloc_n\}) \in [\![\tau]\!]\rho' = \mu$

Then we have:

$methfun = FIX(\lambda f \in \mu.meth_{approx}(\{v_1 = Vloc_1, \ldots, v_n = Vloc_n\}, f)) \in \mu = [\![\tau]\!]\rho'$.

Thus, $(\{v_1 = Vloc_1, \ldots, v_n = Vloc_n\}, methfun) \in (RecToMem([\![\sigma]\!]\rho') \times [\![\tau]\!]\rho')$, which after being packed as $hiddenObj$ is an element of $\exists v.(v \times [\![\tau]\!]\rho')$, and

$PlaceInMem_{\exists v.(v \times \mu)}\ hiddenObj\ s^{n+1}$

$\in (LOC_{\exists v.(v \times [\![\tau]\!]\rho')} \times StoreType)$ -application

$= (LOC_{\exists v.(v \times [\![\tau]\!]\rho[\xi^P/MT])} \times StoreType)$ -substitution

$= (LOC_{\exists v.(v \times [\![\tau]\!]\rho[[\![Tobj\ (MT)\tau]\!]\rho/MT])} \times StoreType)$ -substitution

$= ([\![Tobj(MT)\tau]\!]\rho \times StoreType)_\perp$. -Meaning of types

(InstVar) Case:

$C, E \vdash o.v_i\colon \sigma_i$ follows from $C \vdash \gamma \leq_{\mathcal{A}} (\{v_1\colon \sigma_1; \ldots; v_n\colon \sigma_n\}, \tau)$, $C, E \vdash o\colon \gamma$. Then:

$[\![C, E \vdash o.v_i\colon \sigma_i]\!]\rho\ s =$

let $\langle o', s' \rangle = [\![C, E \vdash o\colon\gamma]\!]\rho\ s$

in $\langle (o')_1(v_i), s' \rangle$.

Let $\nu = \{v_1\colon\sigma_1; \ldots; v_n\colon\sigma_n\}$. $C \vdash \gamma \leq (\nu, \tau)$, so γ must be of the form (ν', τ'), where $\nu' \leq \nu$. Thus, $o' \in [\![(\nu', \tau')]\!]\rho$, and $(o')_1 \in [\![\nu']\!]\rho$. From this we can get that $(o')_1(v_i) \in [\![\sigma_i]\!]\rho$ since all of the σ_i are reference types and therefore cannot be changed by subtyping. Now,

$[\![C, E \vdash o.v_i\colon\sigma_i]\!]\rho\ s$
$= \langle (o')_1(v_i), s' \rangle$
$\in ([\![\sigma_i]\!]\rho \times StoreType)_\perp$.

(Msg) Case:
$C, E \vdash o \Leftarrow m\colon\tau[\gamma/MT]$ follows from $C \vdash \gamma \leq_{meth} Tobj(MT)\{m\colon\tau\}$, $C, E \vdash o\colon\gamma$. Then:

$[\![C, E \vdash o \Leftarrow m\colon\tau[\gamma/MT]]\!]\rho\ s =$

let $\langle o', s' \rangle = [\![C, E \vdash o\colon\gamma]\!]\rho\ s \in ([\![\gamma]\!]\rho \times StoreType)_\perp$

in let $o'' = \mathbf{convert}[\exists v.(v \times [\![\{m\colon\tau\}]\!]\rho[[\![\gamma]\!]\rho/MT])][\gamma']$
$(LookupMem_{\gamma'}\ o'\ s')$

in $\langle open\ o''\ as\ ob\ in\ (ob)_2\ (m), s' \rangle$.

where $[\![\gamma]\!]\rho = LOC_{\gamma'}$ for some γ'
Now our LookupMem works correctly:

$LookupMem_{\gamma'}\ o'\ s' \in \gamma'$. -application

Lemma 7.4.4, $C \vdash \gamma \leq_{meth} Tobj(MT)\{m\colon\tau\}$ implies that:

$\gamma' \leq_{\mathcal{A}} \exists v.(v \times [\![\{m\colon\tau\}]\!]\rho[[\![\gamma]\!]\rho/MT])$.

Therefore, we can use **convert** to coerce γ' to that type. Then since

$o'' \in \exists v.(v \times [\![\{m\colon\tau\}]\!]\rho[[\![\gamma]\!]\rho/MT])$

we can show that:

$\langle open\ o''\ as\ ob\ in\ (ob)_2\ (m), s' \rangle \in ([\![\tau]\!]\rho[[\![\gamma]\!]\rho/MT] \times StoreType)$

which by Lemma 7.4.3 is equivalent to

$([\![\tau[\gamma/MT]]\!]\rho \times StoreType)_\perp$.

(Update) Case:
Let $\tau = \{m_1\colon\tau_1; \ldots; m_n\colon\tau_n\}$, $\tau' = \{m_1\colon\tau_1'; m_2\colon\tau_2; \ldots; m_n\colon\tau_n\}$, and $\sigma = \{v_1\colon\sigma_1; \ldots; v_m\colon\sigma_m\}$ in the following.

$C, E \vdash$ *update c by* $(\{v_1 = a_1'\}, \{m_1 = e_1'\})$: $Tclass(MT)(\sigma, \tau')$ follows from

$C, E \vdash c$: $Tclass(MT)(\{v_1: \sigma_1; \ldots; v_m: \sigma_m\}, \{m_1: \tau_1; \ldots; m_n: \tau_n\})$,

$C \vdash \tau_1' \leq \tau_1$,

$C^{IV}, E \vdash a_1'$: σ_1,

$C^{METH}, E^{METH} \vdash e_1'$: τ_1'

where $C^{IV} = C \cup \{MT \leq_{meth} Tobj\ (MT)\{m_1: \tau_1'; m_2: \tau_2; \ldots; m_n: \tau_n\}\}$

$C^{METH} = C^{IV} \cup \{ST \leq (RecToMem(\{v_1: \sigma_1; \ldots; v_m: \sigma_m\}),$

$\{m_1: \tau_1'; m_2: \tau_2; \ldots; m_n: \tau_n\})\}$

$E^{METH} = E \cup \{self: ST, close: func(ST): MT, super: ST \to \{m_1: \tau_1; \ldots; m_n: \tau_n\}\}$

The corresponding semantic rule is:

$[\![C, E \vdash$ *update c by* $(\{v_1 = a_1'\}, \{m_1 = e_1'\})$: $TClass(MT)(\sigma, \tau')]\!]\rho\ s =$

$\langle \lambda\mu \leq_{\mathcal{A}} [\![\tau']\!]\rho'.\lambda\nu$ **ext** $[\![\sigma]\!]\rho'.(inst, \lambda o \in \xi^S(\nu, \mu).meth), s'''\rangle$,

where $\xi^P(\mu) = LOC_{\exists v.(v \times \mu)}$,

$\xi^S(\nu, \mu) = RecToMem(\nu) \times \mu$,

$\rho' = \rho[\xi^P(\mu)/MT]$,

$\rho'' = \rho'[\xi^S(\nu, \mu)/ST, o/self, cl(\nu, \mu)/close, (sup)_2/super]$,

$cl(\nu, \mu) = \lambda s \in StoreType.\lambda o \in A^{\xi^S(\nu,\mu)}.$

$PlaceInMem_{\exists v.(v \times \mu)}\ (pack[v = RecToMem(\nu)\ in\ v \times \mu]\ o)\ s.$

$\langle V_c, s' \rangle = [\![C, E \vdash c$: $TClass(MT)(\sigma, \tau)]\!]\rho\ s$

$\in ([\![TClass(MT)(\sigma, \tau)]\!]\rho \times StoreType)$,

$sup = (V_c)(\mu)(\nu)$,

$supmeth = (sup)_2\ (o)$,

$supvar = (sup)_1$,

$\langle V_{a_1'}, s'' \rangle = [\![C^{IV}, E \vdash a_1'$: $\sigma_1]\!]\rho'\ s' \in ([\![\sigma_1]\!]\rho' \times StoreType)_\perp$

$\langle V_{e_1'}, s''' \rangle = [\![C^{METH}, E^{METH} \vdash e_1'$: $\tau_1']\!]\rho''\ s'' \in ([\![\tau_1']\!]\rho' \times StoreType)_\perp$

$inst = supvar[V_{a_1'}/v_1]$,

$dom(meth) = \{m_1, \ldots, m_n\}$,

$meth(m_1) = V_{e_1'}$,

and for $2 \leq j \leq n$,

$meth(m_j) = supmeth(m_j)$.

Notice that C^{IV}, C^{METH}, and E^{METH} are very similar to the ones used in the (Class) case. The only difference is that one of the methods has a new type, but since this type is provably a subtype of the old one, it can be converted up to the larger one and the record types will therefore be subtypes as well. The only new case is that of "super", which is a function taking $ST = \xi^S(\nu, \mu)$ and returning a record of methods. Its type is in fact consistent with its definition as we shall show shortly. Therefore, by the same argument as we used in the (Class) case, C^{IV} is consistent with ρ' and C^{METH}, and E^{METH} are consistent with ρ''. Now we can use induction, and:

$sup = (V_c)(\mu)(\nu)$ where $V_c \in (\prod_{\mu \leq_{\mathcal{A}} [\![\tau]\!]\rho'} \prod_{\nu \textbf{ ext } [\![\sigma]\!]\rho'} ([\![\sigma]\!]\rho' \times (\xi^S(\nu,\mu) \to [\![\tau]\!]\rho')))$. We know that $\mu \leq_{\mathcal{A}} [\![\tau']\!]\rho'$. Since $\tau'_1 \leq \tau_1$, we can also show that the new record of methods is a subtype of the old one, so $\tau' \leq \tau$. Thus, $[\![\tau']\!]\rho' \leq_{\mathcal{A}} [\![\tau]\!]\rho'$ and by transitivity of subtyping, $\mu \leq_{\mathcal{A}} [\![\tau]\!]\rho'$. Therefore, μ is a legal argument to V_c. Furthermore, we also know that ν **ext** $[\![\sigma]\!]\rho'$, so ν is a legal argument as well. Thus,

$sup = (V_c)(\mu)(\nu) \in ([\![\sigma]\!]\rho' \times (\xi^S(\nu,\mu) \to [\![\tau]\!]\rho'))$

Thus,

$supvar = (sup)_1 \in [\![\sigma]\!]\rho'$

and

$supmeth = (sup)_2(o) \in [\![\tau]\!]\rho'$ since $o \in \xi^S(\nu,\mu)$.

Also notice here that the type of $(sup)_2$ is what we wanted for "super", so our use of induction is warranted. Now we can continue with:

$inst = supvar[V_{a'_1}/v_1] \in [\![\sigma]\!]\rho'$ since $V_{a'_1} \in [\![\sigma_1]\!]\rho'$

and

$dom(meth) = \{m_1, \ldots, m_n\}$,

$meth(m_1) = V_{e'_1} \in [\![\tau'_1]\!]\rho'$,

and for $2 \leq j \leq n$,

$meth(m_j) = supmeth(m_j) \in [\![\tau_j]\!]\rho'$

so

$meth \in [\![\tau']\!]\rho'$.

Finally, $\langle \lambda\mu \leq_{\mathcal{A}} [\![\tau']\!]\rho'.\lambda\nu \textbf{ ext } [\![\sigma]\!]\rho'.(inst, \lambda o \in \xi^S(\nu,\mu).meth), s'''\rangle$,

$\in (\prod_{\mu \leq_{\mathcal{A}} [\![\tau']\!]\rho'} \prod_{\nu \textbf{ ext } [\![\sigma]\!]\rho'} ([\![\sigma]\!]\rho' \times (\xi^S(\nu,\mu) \to [\![\tau']\!]\rho')) \times StoreType)$

$= ([\![TClass(MT)(\sigma,\tau')]\!]\rho \times StoreType)_\perp$.

(Extend) Case:

The (Extend) case is very similar to the (Update) case except that we show that the new set of records for instance and variables are subtypes of the old ones because they contain all of the same constructors (with exactly the same types) plus one extra one. After this step to justify induction the proof is essentially the same as for the update case.

(Subsumption) Case:

$C, E \vdash e\colon \tau$ follows from $C \vdash \sigma \leq \tau$, $C, E \vdash e\colon \sigma$. Then:

$[\![C, E \vdash e\colon \tau]\!]\rho\ s = \langle \textbf{convert}[[\![\tau]\!]\rho][[\![\sigma]\!]\rho]\ V_e, s'\rangle$,

where $\langle V_e, s'\rangle = [\![C, E \vdash e\colon \sigma]\!]\rho\ s$, if $C \vdash \sigma \leq \tau$ and $C, E \vdash e\colon \sigma$.

Well, we know that C does in fact show that $\sigma \leq \tau$, and $C, E \vdash e\colon \sigma$, so:

$\langle V_e, s'\rangle \in ([\![\sigma]\!]\rho \times StoreType)_\perp$

and $[\![\sigma]\!]\rho \leq_{\mathcal{A}} [\![\tau]\!]\rho$. Thus:

$[\![C, E \vdash e\colon \tau]\!]\rho\ s = \langle \textbf{convert}[[\![\tau]\!]\rho][[\![\sigma]\!]\rho]\ V_e, s'\rangle \in ([\![\tau]\!]\rho \times StoreType)_\perp$. □

Chapter 9

The Implementation

9.1 Overview of the prototype interpreter

The current implementation of TOIL is written in Standard ML ([Mil85],[MTH90]). It is complete in that it supports the full TOIL language. However, as a prototype interpreter it avoids many of the real difficulties that a real implementation for a compiler would run into.

The data structures used in the current implementation are fairly simple. The interpreter proceeds in three distinct steps. First, the program is parsed using ML-Lex and ML-Yacc, which are provided with the ML language. The result is a parse tree, where each syntactic construct in the language corresponds to a certain kind of node. See Figure 9.1 for a graphical example for the parse tree for an IF-THEN-ELSE command.

After the parse tree has been generated, we go on to type-check it. The type checker is a recursive algorithm which runs over the parse tree. So to type check the IF-THEN-ELSE from Figure 9.1, we would just type check the COND subtree to ensure that it is a boolean, and then type check the THEN and ELSE subtrees to make sure they are commands. The algorithm follows naturally from the type assignment rules presented in Chapter 5. During this phase, we need to keep track of our assumptions in C and E. Both are stored in binary search trees.

Finally, we move on to evaluate the program. Error-free programs will result in a value of "Exit OK" for the program node at the root of the tree. However, one of the side effects of the computation could be I/O which is printed to the screen. The interpreter acts recursively and is based on the natural semantics of TOIL presented in Chapter 6. For our IF-THEN-ELSE example, it would evaluate the conditional and then evaluate either the THEN expression or the ELSE expression depending on its result. The algorithm has to be careful and keep track of the updated store in the same way that the natural semantics do.

We use a binary tree again to hold the bindings in the environment, but we use an array to model the store s. An array was used largely for efficiency reasons, since it can get fairly large and we want to be able to access elements of the store keyed numerically as quickly as possible. In fact, changing the data structure for memory from a simple association list into an array sped up a sample execution of the maze program in Appendix A from about 5 minutes to 25 seconds. The array starts out fairly small, and doubles in size if it ever runs out of space. Thus, we avoid statically allocating a large array for memory, and only use a large data structure if it becomes necessary.

As we discussed in Chapter 6, TOIL's support for functions as first-class values requires that frames be allocated in a heap instead of being organized in a stack. A real implementation would therefore have to include a heap data structure, along with a garbage collector to keep it clean. This implies a significant complication for the compiler. However, our prototype interpreter does not have to deal with these issues since bindings are held in data structures supported by ML, as opposed to in memory directly. Of course, we pay for this simplicity by losing efficiency, not only in access time for the data structures, but also in translation time to ML.

The prototype interpreter supports numbers, booleans, and strings as base types. It also includes some common functions on those types, such as elementary arithmetic, boolean functions, and I/O. Support for other base types could be added easily and naturally.

9.2 Syntactic sugar

The type assignment rules from Chapter 5 describe an abstract syntax for TOIL. However, the actual grammar in the next section is substantially different. The differences are due to syntactic sugars which are intended to ease the burden on the programmer of some of the requirements made in the abstract syntax.

- A section for abbreviations for types is included at the beginning of each program. Thus, the programmer can define PointObjectType to be some object type and then use the abbreviation in the rest of the code, to make it more readable.

- A procedure construct is added to the syntax. The parser can simply transform a procedure into a function which returns the *Command* value.

- The transformation of *super* to a function which takes *self* as a parameter discussed in Chapter 6 is performed by the parser. Thus, the programmer can simply use *super* as one would expect and remain blissfully unaware of the added complexity in the semantics and implementation.

- We can use the shorthand `F()` to stand for `F(Unit)`. In fact, neither the UNIT type nor the Unit value are directly expressible from the grammar, since they can always be inferred by the parser.
- Similarly, the COMMAND type and Command value can always be inferred and thus are not expressible in the grammar.
- The parser can determine the way a variable is being used from its context. Thus, the VAL command which transforms a reference to the value stored at that location is inserted automatically. Again, the programmer need not be aware of how variables are being modelled at all.
- The reference type is not included in the grammar. This omission prevents the declaration of doubly referential types such as $ref\ (ref\ \tau)$. When variables and instance variables are declared, the parser interprets their type as a reference automatically.
- The grammar includes extra type information in many places, which is used to overspecify the program for added reliability and error-checking.
- Finally, the syntax for class definitions, updates, and extends is made somewhat more readable.

9.3 Formal grammar for TOIL

The grammar is presented in BNF form. For an introduction to formal grammars, see [Wat91]. See Appendix A for an example of a fairly long TOIL program which implements arrays and stacks using objects.

Program ::= `PROGRAM id;` AbbrevList UnitBlock .

AbbrevList ::= `ABBREV` AbbrevDecls |
::= ϵ

AbbrevDecls ::= `id =` Type`;` AbbrevDecls |
::= `id =` Type

Block ::= :Type ConstList VarList `BEGIN` StmtList `RETURN` Expr `;` `END`

UnitBlock ::= ConstList VarList `BEGIN` StmtList `;` `END`

ConstList ::= `CONST` InitializedVarDecls |
::= ϵ

InitializedVarDecls ::= `id =` Expr : Type; InitializedVarDecls |
::= `id =` Expr : Type

VarList ::= `VAR` VarDecls |
::= ϵ

VarDecls ::= `id :` Type; VarDecls |
::= `id :` Type

InitializedVarList ::= `VAR` InitializedVarDecls |
::= ϵ

MethodsList ::= `METHODS` Methods |
::= ϵ

Methods ::= `id =` FunctionExpr : Type; Methods |
::= `id =` FunctionExpr : Type

StmtList ::= Stmt StmtList |
::= ϵ

TypeList ::= NotEmptyTypeList |
::= ϵ

NotEmptyTypeList ::= Type |
::= Type `,` NotEmptyTypeList

LabelledTypeList ::= { } |
::= { VarDecls }

Type ::= `BOOL` |
::= `NUM` |
::= `STRING` |
::= `MT` |
::= `ST` |
::= `id` |
::= `FUNC(` TypeList `) :` Type |
::= `PROC(` TypeList `)` |
::= `TCLASS(MT)(` LabelledTypeList `,` LabelledTypeList `)` |
::= `TOBJ(MT)` LabelledTypeList |
::= `(` Type `)`

Stmt ::= Expr `:=` Expr `;` |
::= `WHILE` Expr `DO` StmtList `END;` |
::= `IF` Expr `THEN` StmtList `END;` |
::= `IF` Expr `THEN` StmtList `ELSE` StmtList `END;` |
::= Expr `(` ExprList `) ;` |

ExprList ::= NotEmptyExprList |
::= ϵ

NotEmptyExprList ::= Expr |
::= Expr `,` NotEmptyExprList

FunctionExpr ::= `FUNCTION (` VarDecls `)` Block |
::= `FUNCTION ()` Block |
::= `PROCEDURE (` VarDecls `)` UnitBlock |
::= `PROCEDURE ()` UnitBlock

Expr ::= `number` |
::= `boolean` |
::= `string` |
::= `id` |
::= `NIL` |
::= `SELF` |
::= Expr `(` ExprList `)` |
::= `(` Expr `INFIXOP` Expr `)` |
::= Expr ⇐ `id` |
::= `SELF.id` |
::= `SUPER . id` |
::= FunctionExpr |
::= `NEW (` Expr `)` |
::= `CLASS` InitializedVarList MethodsList `END` |
::= `UPDATE` Expr `BY` InitializedVarList MethodsList `END` |
::= `EXTEND` Expr `WITH` InitializedVarList MethodsList `END`

Built-in operators currently include:

1. Arithmetic operators +, -, /, * on numbers. Thus, +: $func(num, num)$: num.
2. Comparison operators =NUM=, >, <, =<, =>, and != on numbers.
   Thus, =NUM= : $func(num, num)$: $bool$.
3. Comparison operators =OBJ=, =STR= on objects and strings, respectively.
   Thus, =OBJ= : $func(\top, \top)$: $bool$, and =STR= : $func(string, string)$: $bool$. $\top$ is the top type, so that all object types are a subtype of $\top$. Thus, any two objects can be used as legal parameters to =OBJ=.
4. Logical operators &&, ||, **not** on boolean values.
   Thus, && : $func(bool, bool)$: $bool$, and **not** : $func(bool)$: $bool$.
5. $\wedge$, the concatenation operator on strings. Thus, $\wedge$: $func(string, string)$: $string$.
6. The output procedure primitives printNum(x:num), printBool(b:bool), printString(s:string), and newline(num-lines:num); and
7. The input function primitives readNum():num, readString():string, and readChar():string. Input is taken from the file *input.toil*. Output is printed to the screen. ReadString() reads from the input file until a space or carriage return is encountered, while readChar() only reads one character. Note, however, that both return values of type *String*; there is no separate type for characters.

9.4 How to use the interpreter

To run the interpreter on the Williams College SUN network:

1. Write a TOIL program called *filename.toil*.
2. Create a file called *input.toil* which contains all of the input to the program. This file needs to exist even if the program does not take any input.
3. Type *sml* to run the Standard ML interpreter. It is located in */home/bull/93rpg/mldist/src*. It is a good idea to be on a machine with plenty of memory and a fast processor to speed things up.
4. Type *use "/home/bull/93rpg/thesis/TOIL/implementation/loader.sml";* to load the TOIL compiler. It will take about 5 minutes to load.
5. Type *eval "filename.toil";* to run the program. Debugging facilities and error messages are not very good. It should also be noted that the current version of ML-Yacc has a bug and thus sometimes returns the wrong line numbers for parse errors.
6. The program should terminate with "Exit OK."

Chapter 10

Complexity of Type Checking

10.1 About complexity

Our goal in this chapter is to demonstrate that type checking in TOIL is decidable, and to find an upper bound for the complexity of the algorithm. Pierce recently showed in [Pie91] that the type-checking problem for the language $F_{\leq}$ is undecidable. The undecidability results from an inability to determine if one type is a subtype of another. This result implies that perhaps $F_{\leq}$ is too strong to serve as an underpinning for real object-oriented languages. However, TOOPLE and TOIL are more conservative in their approaches. In fact, [DM92] have recently proved that the subtyping algorithm in TOOPLE is decidable. Then, using techniques from [CG92],[BCD$^+$92] show that the type checking algorithm is decidable as well.

First we present a proof that the subtyping algorithm in TOIL is decidable and claim an upper bound for the complexity. Then, using this result, we find the complexity for the type checking algorithm in TOIL as well. It turns out that subtyping is exponential on the size of the types being subtyped, leading to an exponential upper bound for type checking as well. The complexity arises because assumptions of the form $ST \leq \tau$ are added to C during type checking, and so the subtyping algorithm must allow the transitive replacement of identifiers by arbitrary types to which they are related. For example, assume we are trying to show that the identifier ST is a subtype of some type τ'. If we have an assumption of the form $ST \leq \tau$, then we can use transitivity to show that $ST \leq \tau'$ if we can show that $\tau \leq \tau'$. Thus we must call the subtyping algorithm recursively with a larger type! It turns out that allowing such lookups in the subtyping algorithm balloons its complexity to exponential on the number of $id \leq \tau$ relations in C. Since such constraints can be added during type checking, the complexity of the type checking algorithm can only be prevented from leaping to exponential on itself, $O(n^n)$, by disallowing nested classes and therefore

necessitating only one $id \leq \tau$ relation involving ST at a time. The omission of nested classes does not present a major loss in programming power; in fact, there do not seem to be many instances in which such a construct would be useful. Furthermore, if classes can be nested then it can be shown that the type of an expression may be exponential in the size of the expression, which would automatically balloon the complexity of the type checking algorithm even more. However, without nested classes, and with initial assumptions only about base types, the type checking algorithm runs in a slow but manageable polynomial time.

10.2 The Subtyping Algorithm

An algorithm for subtyping can be derived from the formal rules presented in Section 5.3.1. When presented with two types, the algorithm can simply match the structure of the types with the rules in the order they are presented. If a match is found, it can be determined whether the two types are subtypes, although the result may depend on one or more recursive calls to the subtyping algorithm. If no match is found, then the types are dissimilar and are not subtypes.

The only rule which presents us with some difficulty is the transitivity rule. Any two types σ and τ can be proven subtypes if a "middle" type γ can be found which is a supertype of the first and a subtype of the second. Thus, $C \vdash \sigma \leq \tau$ if $C \vdash \sigma \leq \gamma, \gamma \leq \tau$. Any subtyping query will match this rule, making our algorithm non-deterministic. Luckily, it has been shown in [BCD+92] that, given certain reasonable restrictions on the initial assumptions in C, the only possible applications of the transitivity rule are on type variables. Thus, we can replace the transitivity rule in our algorithm with a rule for transitivity on type variables only. Since this rule can be matched deterministically on the structure of the types in the subtyping query, our algorithm is deterministic. In our analysis of the subtyping algorithm, each case represents a possible structural match on the types being subtyped, and shows that the solution to the query can be found in a bounded time.

10.3 Complexity of Subtyping Algorithm

Complexity of the subtyping algorithm in TOIL

The size of a type is defined as the number of characters needed to write it out. Let SZ be a function which given a type returns its size. Thus, $SZ(func(\sigma)\colon \tau) = SZ(\sigma) + SZ(\tau) + 7$.

The size of a subtyping query $\sigma \leq \tau$ is defined as $SZ(\sigma) + SZ(\tau)$.

For a subtyping query $\sigma \leq \tau$, let:

- k= max{2,number of (id $\leq$ Type) constraints in C, where Type is not a type variable},

- s=$SZ(\sigma) + SZ(\tau)$, and
- c=$\sum_{(v \leq Type) \in C}(SZ(v) + SZ(Type)) + s$.

The measure (k, c, s) will be used to describe the complexity of a subtyping query. Then: $(k, c, s) < (k', c', s')$ iff

1. $k < k'$ *or*
2. $k = k'$ *and* $s < s'$.

The size of the query, s, can increase when an identifier is looked up in C and replaced by some type. In fact, this occurs in Case 2b of our proof. We have banned recursive subtyping contraints such as $(id \leq Type)$, where *Type* contains *id*. Thus, no identifier will have to be looked up more than once. Without this restriction, it would be impossible to put a bound on the subtyping algorithm since the size of any type could balloon infinitely as we replaced *id* with *Type* over and over again. However, in our case, once an identifier has been looked up once, we can remove the constraint which involves it in C for further recursive calls since it may not be used again. Therefore, k decreases. Each time a type identifier is replaced with some Type related to it through a constraint in C, s may grow, but s can only grow in this way while $k > 0$. Thus we say that as long as we either decrease k or decrease s, the complexity of the query is decreasing. Eventually, there will be no more constraints in C, and s will be forced to continuously decrease as long as recursive calls are made only on smaller constructs. Also, note that c can never increase because it includes all types which can be found either in C or in the types being subtyped.

Theorem 10.3.1 *A subtyping query with complexity measure* (k, c, s) *as defined above involves only less complex recursive subtyping calls and takes time* $O(sc^k)$. *Thus,* $TIME(k, c, s) = O(sc^k)$.

Proof. By induction:

Case 1: $Refl(\leq)$
Equality of two types can be checked in O(s) time.

Case 2: $Var(\leq)$

Case 2a: $id_1 \leq id_2$
Note that the use of transitivity is impossible here since each identifier is only used in one constraint in C. In particular, there are no looping constraints such as $s \leq t, t \leq s$. Furthermore, we will never have to use transitivity on identifiers because of our restrictions

on C in Definition 5.2.1. Thus, we will never run into the case where $s \leq t$ and $t \leq u$ are both in C. C must be checked to see if the relation $id_1 \leq id_2$ is in it, which takes linear time. $id_1 \leq id_2$ spawns no recursive calls.

TIME: $c \leq sc^k$.

Case 2b: $id \leq T$

We will end up having to try to prove the query using transitivity. Thus, we look up the identifier in C, hopefully find an assumption of the form $(id \leq \tau)$, and then recursively try the query $\tau \leq T$. The subtyping algorithm involves a linear search on C, and a recursive subtyping call with smaller C and a larger s. The recursive call is less complex since k decreases. Let $b = SZ(\tau)$.

TIME: (time for lookup=size of C) + (time for recursive call)

$= (size\ of\ C) + TIME(k-1, c, s+b-1)$

$\leq c + (s+b-1)c^{k-1}$

$\leq c + cc^{k-1}$ -Since $c \geq s+b-1$

$\leq c^k + c^k$ -Since $k \geq 1$

$\leq sc^k$. -Since $s \geq 2$

Case 3: $func(\leq)$

Consider a subtyping query on two function types $func(\sigma_1; \ldots; \sigma_n)\colon \tau$ and $func(\sigma'_1; \ldots; \sigma'_n)\colon \tau'$. Then let:

$A_i = SZ(\sigma_i) + SZ(\sigma'_i)$ for $1 \leq i \leq n$,

$A_{sum} = \sum_{i=1}^{n} A_i$, and

$B = SZ(\tau) + SZ(\tau')$.

The algorithm involves a series of subtyping calls. C remains unchanged in all of them, but s is smaller since $A_{sum} + B < s$.

TIME: (time for subtyping each pair of parameters) + (time for subtyping return type)

$= TIME(k, c-B, A_1) + \ldots + TIME(k, c-B, A_n) + TIME(k, c-A_{sum}, B)$

$= A_1(c-B)^k + \ldots + A_n(c-B)^k + B(c-A_{sum})^k$

$= A_{sum}(c-B)^k + B(c-A_{sum})^k$

$\leq A_{sum}c^k + Bc^k$

$\leq sc^k$.

Case 4: $Rec(\leq)$

Consider a subtyping query on two record types $\{l_1\colon \tau_1; \ldots; l_m\colon \tau_m\}$ and $\{v_1\colon \sigma_1; \ldots; v_n\colon \sigma_n\}$.

Subtyping of a record type involves, for each component $1 \ldots n$ in the second record, looking it up in the list of m components for the first record and then making a recursive subtyping call on the types of each pair of matched components. If a match for a component

is not found, we can fail immediately. For each of these subtyping calls, C remains the same but the size of the constructs get smaller.

For the ith component in the second record, let f(i) be the position of the corresponding component in the first record. Also, let A_i be the sum of the types of matched components, so:

$A_i = SZ(\tau_i) + SZ(\sigma_{f(i)})$.

Note that $(\sum_{i=1}^{n} A_i) + 2 \leq s$.

TIME: (time for lookup followed by subtyping for each matching pair)

$= \sum_{i=1}^{n}(m + TIME(k, c - s + A_i, A_i))$

$\leq mn + TIME(k, c, A_1) + \ldots + TIME(k, c, A_n)$

$\leq s^2 + A_1(c)^k + \ldots + A_n(c)^k$ -Since $m, n \leq s$

$\leq s^k + (s-2)c^k$ -Since $k \geq 2, (\sum_{i=1}^{n} A_i) + 2 \leq s$

$\leq c^k + (s-2)c^k$ -Since $c \geq s$

$\leq sc^k$

Case 5: $Pair(\leq)$

Consider a subtyping query on two pair types (σ, τ) and (σ', τ').

Subtyping of the pair types involves checking to see if $\sigma \leq \sigma'$ and if $\tau \leq \tau'$. C remains unchanged for the recursive calls, but s gets smaller.

Let $A = SZ(\sigma) + SZ(\sigma')$, and let $B = SZ(\tau) + SZ(\tau')$, so $A + B + 2 = s$.

TIME: (time for first recursive call) + (time for second recursive call)

$= TIME(k, c - B, A) + TIME(k, c - A, B)$

$= A(c - B)^k + B(c - A)^k$

$\leq Ac^k + Bc^k$

$\leq sc^k$.

Case 6: $Obj(\leq)$

Consider a subtyping query on two object types $TObj(MT)\tau$ and $TObj(MT')\tau'$.

Let $A = SZ(\tau)$ and $B = SZ(\tau')$, so $s = A + B + 16$.

Subtyping of objects involves a check to see if the bodies are subtypes. An assumption of the form $s \leq t$ is added to C. The recursive call is still less complex, however, since the assumption relates two identifiers, not an identifier and a type, and therefore k remains the same. Furthermore, $A + B < s$.

c' (in the recursive call) is equal to the original c, less the size of the subtyping query on the two objects, plus the size of the recursively spawned subtyping query, plus 2 for the size of the new assumption. Thus:

$c' = (c) - (s) + (A + B) + 2 = (c) - (A + B + 2) + (A + B) + (2) = c$

TIME = (time for subtyping call)

$= TIME(k, c, A + B)$
$= (A + B)c^k$
$\leq sc^k$. □

10.4 Complexity of Type Checking

Complexity of the Typechecking algorithm in TOIL

The size of an expression is defined similarly to the size of a type, except that all type information provided by the programmer in an expression counts three times as much. Thus,

SZ(function():τ begin return 3; end) = $3(SZ(\tau)) + 28$.

The reason for this will be made clear in the discussion which follows. Furthermore, for an expression X, we often write x as shorthand for $SZ(X)$.

We begin with the following definitions, some of which are similar to the ones used in the previous theorem for subtyping:

- k= max{2,number of (id $\leq$ Type) constraints in C, where Type is not a type variable},
- s = size of the expression to be typechecked,
- $c = \sum_{(v \leq Type) \in C}(SZ(v) + SZ(Type)) + \sum_{(v \leq_{meth} Type) \in C}(SZ(v) + SZ(Type)) + s$,
- e = max{s, largest type expression in the range of E}.

Note that we have changed c to include the sum of the sizes of all of the $\leq_{meth}$ relations in C as well. This change is necessary because in the (Msg) rule we will be looking up a type identifiers (MT) and replacing it with a type related to it through a $\leq_{meth}$ constraint. However, this new definition of c will always be larger than the old one, and so sc^k, where c is our new definition, can still be used as an upper bound on the subtyping algorithm.

We assume that the largest possible size of the type of an expression of size s is es. No proof for this assertion is provided here, but it seems to be a reasonable assumption. Note that this assumption depends on the fact that we are not allowed to nest classes, since the type of nested class expressions can be made to grow exponentially. An expression consisting of only a variable can have a type as large as the largest type stored in E.

Since we do not allow nested classes, we can use ST as a keyword. We will start our processing with a dummy relation (ST $\leq$ ({},{})) in C. Whenever a `class`, `update`, or `extend` expression is evaluated, an assumption of the form (ST $\leq \tau$) is added to C, but the new assumption will simply overwrite our dummy relation. Thus, k (the number of assumptions in C) will remain unchanged. A similar dummy relation (MT $\leq_{meth}$ TObj(MT){}) is included for the $\leq_{meth}$ relation.

It is essential to our proof that c never increase during recursive calls. It may seem obvious that c should increase whenever we add an assumption to C. However, whenever we add an assumption, the type we are using in the assumption has been enumerated by the programmer in the expression being type checked. For example, consider the type checking rule for classes:

$$\frac{\begin{array}{c} C^{IV} = C \cup \{MT \leq_{meth} Tobj(MT)\tau\}, E \vdash e_1 : \sigma \\ C^{METH} = C^{IV} \cup \{ST \leq (RecToMem(\sigma), \tau)\}, \\ E^{METH} = E \cup \{self : ST, close : (func(ST) : MT)\} \vdash e_2 : \tau \end{array}}{C, E \vdash class(e_1, e_2) : Tclass(MT)(\sigma, \tau)}$$

When we add the (MT $\leq_{meth}$ $TObj(MT)\tau$) assumption during the type checking of a class, the type $TObj(MT)\tau$ has been written down by the programmer in the class expression. Hence, it is included in the size of the expression itself – in fact, it counts three times in the size of the expression. Thus, c (the sum of the constraints in $C + s$) never increases in recursive calls because the new s will decrease to match the increase in the size of C. For example, both constraints added in the class rule are dependent on type information included in the class expression. However, this information is not included in the expressions which describes the methods and instance variables. Therefore, the increase in c due to the new constraints is matched by the decrease in s due to the removal of type information. It is for this reason that we counted type information three times in the size of expressions. In the class case, we add at most (SZ(τ) + 3*SZ(σ) + SZ(τ)) to c. We must count the size of σ three times because each type in σ will be augmented by a *ref*, which is three letters long. On the other hand, we remove 3*(SZ(τ) + SZ(σ)) worth of type information from the size of the expression, s. Thus, the removal of type information will reduce s enough to counterbalance the increase in c.

We assume that the data structures for C and E allow for instantaneous lookup. Although this is probably untrue, lookup times for C and E will add only small factors to the complexity for typechecking. In the interests of clarity, we omit this complication.

Theorem 10.4.1 *Given the above definitions of variables, the typechecking of an expression $C, E \vdash M : \tau$ of size s takes time s^2ec^k, where $s, c, e,$ and k are defined as above.*

Proof.

Case 1: *Assn*: $X := V$

We simply need to typecheck the variable, typecheck the value being assigned, and then do a subtype query to ensure that the assignment is a legal one. Recall that the size of the type returned for an expression of size s is bounded by es.

Remember, we will use x to denote the size of the expression X, and v to denote the size of the expression V. Thus, $s = x + v + 1$.

TIME: (typecheck variable) + (typecheck value) + (subtype call)
$= (x^2 ec^k) + (v^2 ec^k) + (ex + ev)c^k$ -By inductive assumption and Theorem 10.3.1
$\leq (x^2 + v^2 + x + v)ec^k$
$\leq s^2 ec^k$

Case 2,3,4: Other commands

The cases for IF-THEN-ELSE statements, WHILE loops, blocks, and statement lists are all straightforward using induction.

Case 5,6,7: Easy expressions

The cases for NIL, variables, NEW expressions, and VAL expressions are all straightforward as well.

Case 8: *Function*: $function(v_1{:}\,\sigma_1, \ldots, v_n{:}\,\sigma_n)B$

We need to add n assumptions to E, then typecheck the block. Recall that e is defined as the maximum of s and the largest type expression in the range of E. The type used in each assumption is included (three times) in the expression for the function. Thus, s is larger than the type used in any assumption. Therefore e will not be increased for recursive calls by adding the assumptions.

TIME: (typecheck block B)
$= b^2 ec^k$
$\leq s^2 ec^k$.

Case 9: *FuncAppl*: $F(P_1, \ldots P_n)$

We need to typecheck the function, typecheck all of the parameters, and then ensure that all of the parameters are subtypes of the formal parameter types for the function. Note that the types of the formal parameters must be enumerated in the type of the function F. Therefore, the sum of the sizes of the types for the formal parameters (not the actual parameters) must be bounded by the size of the type of F=ef.

TIME: (typecheck F) + (typecheck $P_1 \ldots P_n$) + (n subtype calls)
$= (f^2 ec^k) + (p_1^2 ec^k) + \ldots + (p_n^2 ec^k) + (ef + \sum_{i=1}^{n} ep_i)c^k$
$\leq (f^2 + p_1^2 + \ldots + p_n^2 + (f + p_1 + \ldots + p_n))ec^k$
$\leq s^2 ec^k$

Case 10: *Rec*: $\{R_1 = V_1, \ldots, R_n = V_n\}$

We need to typecheck each component expression separately.

TIME: (typecheck V_1) + ... + (typecheck V_n)
$= (v_1^2 ec^k) + \ldots + (v_n^2 ec^k)$
$\leq s^2 ec^k$

Case 11: $Proj$: $R.V_i$

We need to typecheck the record expression and then look up the correct component, which is linear on the size of the record type.

TIME: (typecheck record) + (look up type of component)
$= (r^2 ec^k) + er$
$\leq (r^2 + r) ec^k$
$\leq s^2 ec^k$ -Since $s \geq r + 1$, so $s^2 \geq r^2 + 2r + 1$

Case 12: $Class$: $class(V, M)$

We need to add ($ST \leq Type$) and ($MT \leq_{meth} Type$) relations to C. Remember that k does not change, since the new relations overwrite ones that are already there. We also add assumptions about *self* and *close* to E. Both of these assumptions are rather small (*self*:ST and *close*:func(ST):MT), so the largest addition to the types in the range of E is 11. The size of the class expression s is guaranteed to be larger than 11 by the syntax, so e= max{s,largest type expression in the range of E} will not increase. We can then typecheck the instance variables of the new class, the methods of the new class, and finally check to make sure that the actual types we prove are subtypes of those the programmer promised.

TIME: (typecheck instance variables V) + (typecheck methods F) + (two subtype checks)
$= (v^2 ec^k) + (m^2 ec^k) + (evc^k) + (emc^k)$
$= (v^2 + m^2 + v + m) ec^k$
$\leq s^2 ec^k$.

Case 13: $Message$: $O \Leftarrow M$

Case 13a: O's type is a type variable, probably MT

After typechecking O, we find that its type is a type variable. Therefore, we need to find our best approximation for O's type from our $\leq_{meth}$ assumptions in C. Then, look up the appropriate method type and substitute the type of O in for MT everywhere. We shall assume that the substitution can be done in linear time on the size of the method type.

TIME : (typecheck O) + (look up method in approximate type) + (substitute for MT)
$\leq (o^2 ec^k) + c + c$
$\leq (o^2 + 2) ec^k$
$\leq s^2 ec^k$.

Case 13b: O's type is an actual object type
Similar to (13a), except we don't have to look up an approximation for the type.
TIME : (typecheck O) + (look up method in type) + (substitute for MT)
$\leq (o^2ec^k) + oe + oe$
$\leq (o^2 + 2o)ec^k$
$\leq s^2ec^k$.

Case 14: *InstVar*: $O.V$
The proof for instance variables is similar to that for messages. However, if O is a type variable, we use the subtype relations in C to find an approximation for its type. Also, we do not have to perform the substitution for MT in instance variables.

Case 15: *Update*: *update P by* $(\{\}, \{V = M\})$
We need to

1. Typecheck the parent class;
2. Add assumptions about MT, ST, *self*, *super*, and *close* to C and E;
3. Typecheck the new method;
4. Make sure that the new declared method type is a subtype of the old method type;
5. Make sure that the new method type is a subtype of what the programmer said it would be; and
6. Insert the new method into the parent class.

TIME: (typecheck parent) + (typecheck method) + (subtype new declared versus old) + (subtype new versus declared) + (insert)
$\leq (p^2ec^k) + (m^2ec^k) + (em + ep)c^k + (em + m)c^k + ep$
$\leq (p^2 + m^2 + 2p + 3m)ec^k$
$\leq (p^2 + m^2 + 2p + 2m + pm)ec^k$ -Since $p \geq 1$
$\leq s^2ec^k$ -Since $s = p + m + 2$, so $s^2 = p^2 + m^2 + 4 + 2pm + 2p + 2m$

Case 16: *Extend*
Similar to case for `Update`. □

Chapter 11

Summary and Future Work

11.1 Summary

Object-oriented programming languages are attractive because they provide a great deal of abstraction. Once an implementation for a kind of object has been designed, the data and the operations associated with each object of that kind can be used without any knowledge about the details of implementation. Furthermore, the implementation of objects can be changed without affecting the way they are used as long as their interfaces remain the same. Because of their powerful support for abstraction, object-oriented languages are especially appealing for large-scale programming projects which may employ dozens of programmers working on various parts of the problem. Various kinds of objects may be used to solve different subproblems of the original large problem, and the abstraction can be used to ensure that as long as the interfaces of the objects remain consistent, changes in the implementation of one kind of object will not affect the behavior of all the rest.

The use of inheritance to simplify code-reuse in object-oriented languages also makes them attractive for large-scale projects. Even in small-scale applications, however, inheritance can be very useful. For example, many languages provide a library of common functions, often pre-compiled and without the source code. In some cases, one of the functions provided will be close to the one desired, but may be missing some key attributes. Instead of having to write the entire function from scratch, in object-oriented languages it may be possible to create a new kind of object which includes all of the attributes of the library class, but adds some customized ones.

Unfortunately, as we saw in Chapter 3, most of the popular object-oriented languages in use today fall short in one way or another. Some are not type-safe, meaning that while a program is running, an object may be requested to perform an operation that it is not equipped to handle. This kind of error presents a severe problem for the reliability of large-

scale problems. Some languages, such as C and C++, do not even attempt to be type-safe, and place the burden of making sure that all of the types in a program work out on the shoulders of the programmer. Others make more of an effort, but fall victim to holes in their type systems for various reasons, many of which have been presented throughout this thesis. In general, type checking can be very useful in debugging and increasing the reliability of programs. However, in order to take full advantage of the power of type checking, it is necessary to ensure that the language is type-safe. It is much more useful to know that a program will NOT have any run-time type errors than to know that it will probably not have any. Thus, any hole in the type system seriously compromises the effectiveness of the whole approach.

Some other languages preserve type-safety but are forced to place tight restrictions on some of the features, such as inheritance, which made object-oriented languages attractive in the first place. TOIL, however, preserves type safety while placing a minimum amount of restrictions on the use of its object-oriented features. Many languages run into trouble by relating the use of inheritance in the implementation of objects with the use of subtyping. Subtyping, where one type of object can be used in place of another, depends on the interface the objects present to the exterior world, not on how they are implemented. In fact, we showed in Chapter 3 that the inheritance relation is not necessarily the same as the subtyping relation. Thus, subtypes need not be determined by inheritance, and inheritance does not necessarily result in subtypes. Unlike many other languages, TOIL explicitly separates the two concepts, thereby avoiding tight restrictions on subtyping or inheritance while still remaining type-safe.

TOIL also provides added flexibility by supporting references within an object's methods not only to the object itself, but also to the object's type. For a single class, this may seem unimportant, but it greatly increases the power and flexibility of inheritance. It allows the parameters for the methods of a class to be automatically updated when a subclass is formed using inheritance. For example, any methods for a Point object which takes Points as parameters can be written so that they automatically take ColorPoints as parameters when the ColorPoint class is generated using inheritance.

The properties mentioned above are the main ones which make TOIL stand out. However, Bruce's language TOOPLE has all of these properties as well. What makes TOIL more attractive than TOOPLE? Well, TOOPLE was designed as a functional language largely to demonstrate the semantic techniques involved in modelling an object-oriented language with the desirable attributes mentioned above. TOIL, on the other hand, is an imperative version of TOOPLE. It is much easier to attack many common kinds of problems in an imperative paradigm than in a functional one, making TOIL much more useful as a real-life language designed to be programmed in. Unfortunately, this gain in programming power is matched by an increase in the complexity of the semantic model.

The denotational semantics for TOIL were presented in Chapter 7. They map each

construct in the language onto a corresponding value from a model of the typed lambda calculus. We were then able to prove that the semantics as given are consistent with our type checking rules, presented in Chapter 5. In other words, TOIL is provably type-safe.

We have presented several justifications for the usefulness of TOIL as a real-life programming language. Besides the advantages of object-oriented languages due to abstraction, TOIL is provably type-safe, has minimal restrictions on object-oriented features, and is imperative. However, so far we have ignored the issue of speed, which is vital for real-life applications. In this area as well, TOIL has a huge advantage over TOOPLE because imperative languages are generally much faster than functional ones. The current implementation of TOIL is only a prototype interpreter, and therefore not geared toward speed at all. However, in Chapter 10 we were able to prove that the type checking algorithm for TOIL is decidable, and in fact runs in polynomial time under reasonable assumptions. This result is important because type checking for another object-oriented language, $F_{\leq}$, has been shown by [Pie92] to be undecidable due to looping behavior in the subtyping algorithm.

In this thesis we have presented both an implementation and a formal semantics for a type-safe object-oriented language, TOIL. The capabilities presently supported by TOIL, along with some of the features proved herein, make it an appealing language for real-life programming. However, there is plenty of work to be done before such a goal is realistic.

11.2 Future Work

In Chapter 8, we proved that the denotational semantics for TOIL were consistent with the type rules. However, the implementation is more easily based on the natural semantics presented in Chapter 6. Thus, a proof which ensures that the natural semantics are consistent with the type rules (subject-reduction) is necessary. Furthermore, a proof which shows that the natural semantics are consistent with the denotational semantics would be desirable as well. Both of these kinds of proofs have already been provided for TOOPLE in [BCK93], and so it seems likely that they will go through without difficulty for TOIL as well, at worst finding minor errors, but they have not been written out yet.

TOIL could also be easily expanded in several ways. Because it is based on a model of the second-order lambda calculus, extensions to support recursion and polymorphism should be relatively painless. However, these extensions may conflict with another direction for future research, the development of a real compiler for TOIL. As we have mentioned, some of TOIL's features which are fairly straightforward in the semantics and in the current prototype implementations will present serious complications to a real implementation. Further extensions such as polymorphism and recursion which can be naturally added to the semantics without much increase in complexity may lead to further complications. Thus,

a true implementation is outside the scope of this thesis. However, we hope that TOIL has succeeded in demonstrating the potential of its approach towards the development of a realistically useful programming language for the future.

Appendix A

Sample TOIL program

A.1 Code for a maze program

```
program mazer;

abbrev  stack_node_class_type = TClass(mt)({value:num;next:mt},
                                           {get_value:func():num;
                                            get_next:func():mt;
                                            attach_right:proc(mt)});
        stack_node_type = Tobj(mt){get_value:func():num;
                                   get_next:func():mt;
                                   attach_right:proc(mt)};
        stack_class_type = TClass(mt)({top:stack_node_type},
                                      {push:proc(num);
                                       pop:func():num;
                                       is_empty:func():bool});
        stack_type = Tobj(mt){push:proc(num);
                              pop:func():num;
                              is_empty:func():bool};
        elt_type = string;
        array_node_class_type = TClass(mt)({value:elt_type;
                                            right:mt;
                                            down:mt},
                                           {find:func(num,num):mt;
                                            set_right:proc(mt);
                                            set_down:proc(mt);
```

```
                                        get_value:func():elt_type;
                                        set_value:proc(elt_type)});
        array_node_type = Tobj(mt) {find:func(num,num):mt;
                                    set_right:proc(mt);
                                    set_down:proc(mt);
                                    get_value:func():elt_type;
                                    set_value:proc(elt_type)};
        array_class_type = TClass(mt)
                      ({top_left:array_node_type;
                        x_dim:num;
                        y_dim:num},
                       {create:proc(num,num,elt_type);
                        create_down:
                           func(num,num,elt_type):array_node_type;
                        create_right:func(num,elt_type):array_node_type;
                        get_height:func():num;
                        get_width:func():num;
                        get:func(num,num):elt_type;
                        set:proc(num,num,elt_type)});
        array_type = Tobj(mt){create:proc(num,num,elt_type);
                         create_down:
                              func(num,num,elt_type):array_node_type;
                         create_right:func(num,elt_type):array_node_type;
                         get_width:func():num;
                         get_height:func():num;
                         get:func(num,num):elt_type;
                         set:proc(num,num,elt_type)}

const   generate_stack_node_class = function(n:num):stack_node_class_type
         begin
           return
              class
              var
                 value=n : num;
                 next = nil : mt
              methods
                 get_value = function():num
                             begin
                                return self.value;
```

```
                    end : func():num;
         get_next = function():mt
                    begin
                       return self.next;
                    end : func():mt;
         attach_right =  procedure(n:mt)
                    begin
                       self.next:=n;
                    end : proc(mt)
      end;
 end : func(num):stack_node_class_type;
stack_class =
      class
      var
         top=nil : stack_node_type
      methods
         push =  procedure(n:num)
                 var
                    temp:stack_node_type
                 begin
                    temp:=self.top;
                    self.top := new (generate_stack_node_class(n));
                    self.top<=attach_right(temp);
                 end : proc(num);
         pop =   function():num
                 var
                    answer : num
                 begin
                    answer:=self.top <= get_value();
                    self.top := self.top<=get_next();
                    return answer;
                 end : func():num;
         is_empty =  function():bool
                     var
                        answer : bool
                     begin
                        if (self.top =OBJ= nil)
                           then  answer:=true;
                           else  answer:=false;
```

```
                          end;
                        return answer;
                      end : func():bool
 end : stack_class_type;
errorFlag = "illegalarray" : elt_type;
generate_array_node_class =
function(init_value:elt_type):array_node_class_type
  begin
  return
   class
   var
      value=init_value:elt_type;
      right=nil:mt;
      down=nil:mt
   methods
      find =  function(m:num;n:num):mt
              var
                 result:mt
              begin
                 if (n =NUM= 1)
                   then if (m =NUM= 1)
                        then result:=close(self);
                          else result:=self.right<=find((m-1),n);
                        end;
                   else  result:=self.down<=find(m,(n-1));
                   end;
                 return result;
               end : func(num,num):mt;
      set_right =   procedure(new_r:mt)
                    begin self.right:=new_r; end : proc(mt);
      set_down =    procedure(new_d:mt)
                    begin self.down:=new_d; end : proc(mt);
      get_value =   function():elt_type
                    begin return self.value; end : func():elt_type;
      set_value =   procedure(new_v:elt_type)
                    begin self.value:=new_v; end : proc(elt_type)
   end;
end : func(elt_type):array_node_class_type;
array_class =
```

```
class
var
   top_left=nil:array_node_type;
   x_dim=0:num;
   y_dim=0:num
methods
   get =   function(m:num;n:num):elt_type
           var
              result:elt_type
           begin
             if ((m =< self.x_dim) && (n =< self.y_dim))
               then result:=(self.top_left<=find(m,n))
                             <=get_value();
               else result:=errorFlag;
               end;
             return result;
           end : func(num,num):elt_type;
   set =   procedure(m:num;n:num;new_v:elt_type)
           begin
              if ((m =< self.x_dim) && (n =< self.y_dim))
                 then  (self.top_left<=find(m,n))
                               <=set_value(new_v);
                 end;
              end : proc(num,num,elt_type);
   get_width = function():num
               begin return self.x_dim; end : func():num;
   get_height =function():num
               begin return self.y_dim; end : func():num;
   create= procedure(m:num;n:num;i:elt_type)
           begin
              if ((m>0) && (n>0))
                 then  self.x_dim:=m;
                       self.y_dim:=n;
                 else  self.x_dim:=0;
                       self.y_dim:=0;
                 end;
              self.top_left:= close(self)<=create_down
                                (self.x_dim,self.y_dim,i);
           end : proc(num,num,elt_type);
```

```
      create_down =
        function(m:num;n:num;i:elt_type):array_node_type
        var
           result:array_node_type
        begin
           if (n =NUM= 0)
              then result:=nil;
              else result:=new(generate_array_node_class(i));
                   result<=set_right
                    (close(self)<=create_right(m,i));
                   result<=set_down
                    (close(self)<=create_down(m,(n-1),i));
              end;
           return result;
        end : func(num,num,elt_type):array_node_type;
      create_right=
        function(m:num;i:elt_type):array_node_type
        var
           result:array_node_type
        begin
           if (m =NUM= 0)
              then  result:=nil;
              else  result:=new(generate_array_node_class(i));
                    result<=set_down(nil);
                    result<=set_right
                       (close(self)<=create_right((m-1),i));
              end;
           return result;
        end : func(num,elt_type):array_node_type
    end : array_class_type;
  read_maze =  function():array_type
               var
                  xSize:num;
                  ySize:num;
                  x:num;
                  y:num;
                  maze:array_type
               begin
                  xSize:=readNum();
```

```
                ySize:=readNum();
                maze:=new(array_class);
                maze<=create(xSize,ySize,"X");
                y:=1;
                while (y =< ySize) do
                   x:=1;
                   maze<=set(1,y,readChar());
                   while (x < xSize) do
                      x:=(x+1);
                      maze<=set(x,y,readChar());
                   end;
                   y:=(y+1);
                end;
                return maze;
             end : func():array_type;
print_maze = procedure(maze:array_type)
             var
                x:num;
                y:num;
                symbol:string
             begin
                y:=1;
                while (y =< maze<=get_height()) do
                   x:=1;
                   symbol:=maze<=get(1,y);
                   while (x < maze<=get_width()) do
                      if (symbol =STR= "B")
                         then  symbol:=" ";
                         end;
                      printString(symbol);
                      x:=(x+1);
                      symbol:=maze<=get(x,y);
                   end;
                   if (symbol =STR= "B")
                      then symbol:=" ";
                      end;
                   printString(symbol);
                   y:=(y+1);
                   newline(1);
```

```
                    end;
                end : proc(array_type);
        record_path =
          procedure(maze:array_type;x_path:stack_type;y_path:stack_type)
          var
             x:num;
             y:num
          begin
             printString("Solved Maze: ");
             newline(1);
             while (not (x_path<=is_empty())) do
                x:=x_path<=pop();
                y:=y_path<=pop();
                maze<=set(x,y,".");
             end;
             print_maze(maze);
          end : proc(array_type,stack_type,stack_type)

var   maze:array_type;
      end_x:num;
      end_y:num;
      path_x:stack_type;
      path_y:stack_type;
      path_dir:stack_type;
      cur_x:num;
      cur_y:num;
      new_x:num;
      new_y:num;
      success:bool;
      cur_dir:num   -- 0=up, 1=right, 2=down, 3=left

begin
  maze:=read_maze();
  printString("Original Maze: ");
  newline(1);
  print_maze(maze);
  newline(2);
  end_x:=readNum();
  end_y:=readNum();
```

```
path_x:=new(stack_class);
path_y:=new(stack_class);
path_dir:=new(stack_class);
path_x<=push(2);
path_y<=push(2);
path_dir<=push(0);
success:=false;

while (not (path_x<=is_empty())) do
   cur_x:=path_x<=pop();
   cur_y:=path_y<=pop();
   cur_dir:=path_dir<=pop();
   while (cur_dir < 4) do
      if (cur_dir =NUM= 0)
         then  new_x:=cur_x;
               new_y:=(cur_y-1);
         end;
      if (cur_dir =NUM= 1)
         then  new_x:=(cur_x+1);
               new_y:=cur_y;
         end;
      if (cur_dir =NUM= 2)
         then  new_x:=cur_x;
               new_y:=(cur_y+1);
         end;
      if (cur_dir =NUM= 3)
         then  new_x:=(cur_x-1);
               new_y:=cur_y;
         end;
      cur_dir:=(cur_dir+1);
      if ((new_x =NUM= end_x) && (new_y =NUM= end_y))
         then  path_x<=push(cur_x);
               path_y<=push(cur_y);
               path_x<=push(new_x);
               path_y<=push(new_y);
               record_path(maze,path_x,path_y);
               success:=true;
         else  if (maze<=get(new_x,new_y) =STR= " ")
                  then  maze<=set(new_x,new_y,"B");
```

```
                        path_x<=push(cur_x);
                        path_y<=push(cur_y);
                        path_dir<=push(cur_dir);
                        cur_x:=new_x;
                        cur_y:=new_y;
                        cur_dir:=0;
                  end;
            end;
      end;
   end;
   if (not (success))
      then  printString("Impossible Maze!");
            newline(1);
      end;
end.
```

A.2 The maze program in action

For an input file containing:

```
19 10 @@@@@@@@@@@@@@@@@@@
      @     @           @
      @@@@@ @@@@@@ @@ @@@
      @   @       @ @    @
      @@@ @ @@@@ @ @@@@ @
      @   @ @  @ @    @ @
      @ @ @ @ @@ @@@@ @ @
      @ @@@ @ @       @ @
      @     @ @ @@@@@@@
      @@@@@@@@@@@@@@@@@@@ 19 9
```

We get the following output:

```
Original Maze:
@@@@@@@@@@@@@@@@@@@
@     @           @
@@@@@ @@@@@@ @@ @@@
@   @       @ @    @
@@@ @ @@@@ @ @@@@ @
@   @ @  @ @    @ @
@ @ @ @ @@ @@@@ @ @
@ @@@ @ @       @ @
@     @ @ @@@@@@@
@@@@@@@@@@@@@@@@@@@
Solved Maze:
@@@@@@@@@@@@@@@@@@@
@.....@     ....  @
@@@@@.@@@@@@.@@.@@@
@   @......@.@ ...@
@@@ @ @@@@.@.@@@@.@
@   @ @  @.@....@.@
@ @ @ @ @@.@@@@.@.@
@ @@@ @ @ ......@.@
@     @ @ @@@@@@@..
@@@@@@@@@@@@@@@@@@@
Exit OK.
```

Bibliography

[Aba92] Martín Abadi. Baby Modula-3 and a Theory of Objects. Technical Report 0, Systems Research Center, DEC, October 1992.

[AC90] Roberto Amadio and Luca Cardelli. Subtyping Recursive Types. Technical Report 62, Digital Systems Research Center, 1990.

[BCD+92] Kim Bruce, Jon Crabtree, Allyn Dimock, Robert Muller, Tom Murtagh, and Robert van Gent. Type-checking in TOOPLE is Decidable. Unpublished manuscript, 1992.

[BCK93] Kim Bruce, Jon Crabtree, and Gerald Kanapathy. An operational semantics for TOOPLE: A statically-typed object-oriented programming language. To appear in Proceedings of MFPS IX, 1993.

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

[Bru93] Kim Bruce. Safe Type Checking in a Statically Typed Object-Oriented Programming Language. In *Proc. ACM Symp. on Principles of Programming Languages*, pages 285–298, 1993.

[BTCGS91] V. Breazu-Tannen, T. Coquand, C.A. Gunter, and A. Scedrov. Inheritance and implicit coercion. *Information and Computation*, 93(1):172–221, 1991.

[CDJ+89] Luca Cardelli, Jim Donahue, Mick Jordan, Bill Kalsow, and Greg Nelson. The Modula-3 Type System. In *Proceedings: Sixteenth Annual ACM Symposium of Principles of Programming Languages*, pages 202–212, January 1989.

[CG92] P.L. Curien and G. Ghelli. Coherence of subsumption, minimum typing and type-checking in $F_{\leq}$. *Mathematical Structures in Computer Science*, 2:55–91, 1992.

[CW85] Luca Cardelli and Peter Wegner. On Understanding Types, Data Abstraction, and Polymorphism. *Computing Surveys*, 17(4):471–522, 1985.

[DM92] Allyn Dimock and Robert Muller. Private Communication, 1992.

[GJ87] Carlo Ghezzi and Mehdi Jazayeri. *Programming Language Concepts.* John Wiley and Sons, New York, second edition, 1987.

[GR83] A. Goldberg and D. Robson. *Smalltalk–80: The Language and its Implementation.* Addison Wesley, 1983.

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

[Kir89] Bjorn Kirkerud. *Object-Oriented Programming with Simula.* Addison-Wesley, Wokingham, England, 1989.

[Mey88] Bertrand Meyer. *Object-Oriented Software Construction.* Prentice-Hall, New York, 1988.

[Mey92] Bertrand Meyer. *Eiffel: The Language.* Prentice-Hall, 1992.

[Mil85] Robin Milner. The Standard ML Core Language. *Polymorphism*, 2(2), 1985. 28 pages. An earlier version appeared in Proc. 1984 ACM Symp. on Lisp and Functional Programming.

[MTH90] Robin Milner, Mads Tofte, and Robert Harper. *The Definition of Standard ML.* MIT Press, 1990.

[NN92] Hanne Riis Nielson and Flemming Nielson. *Semantics With Applications: A Formal Introduction.* John Wiley and Sons, West Sussex, England, 1992.

[Pie91] Benjamin C. Pierce. Bounded Quantification is Undecidable. Technical Report 91-161, Carnegie Mellon University, July 1991.

[Pie92] Benjamin C. Pierce. Type-Theoretic Foundations for Object-Oriented Programming. Technical report, University of Edinburgh, May 1992.

[Rey81] J.C. Reynolds. The Essence of Algol. In de Bakker and van Vliet, editors, *Algorithmic Languages*, pages 345–372. IFIP, North Holland, 1981.

[SCB+86] C. Schaffert, T. Cooper, B. Bullis, M. Kilian, and C. Wilpolt. An Introduction to Trellis/Owl. In *OOPSLA '86 Proceedings*, pages 9–16. ACM SIGPLAN Notices,21(11), November 1986.

[Sch86] David A. Schmidt. *Denotational Semantics: A Methodology For Language Development.* Allyn and Bacon, Inc., Boston, MA, 1986.

[Str86] Bjarne Stroustrop. *The C^{++} Programming Language.* Addison-Wesley, Inc., Reading, MA, 1986.

[Tan76] A.S. Tanenbaum. A Tutorial on Algol 68. *Computing Surveys*, 8(2):155–190, June 1976.

[Tea92] Stanford/TRW Prototech Team. Types and Modules in Rapide 1.0. Technical report, Stanford University and TRW, October 1992.

[TRea92] Thomas Turba, John Reagan, and Pete Boton et. al. Object-Oriented Extensions to Pascal. Technical report, UniSys Corp., December 1992.

[Wat91] David A. Watt. *Programming Language Syntax and Semantics.* Prentice Hall, New York, 1991.