Essentials of Programming Languages, Third Edition

by Daniel P. Friedman and Mitchell Wand

Chapter 9: Objects and Classes

Overview

Many programming tasks require the program to manage some piece of state through an interface. For example, a file system has internal state, but we access and modify that state only through the file system interface. Often, the piece of state spans several variables, and changes to those variables must be coordinated in order to maintain the consistency of the state. One therefore needs some technology to ensure that the various variables that constitute the state are updated in a coordinated manner. *Object-oriented programming* is a useful technology for accomplishing this task.

In object-oriented programming, each managed piece of state is called an *object*. An object consists of several stored quantities, called its *fields*, with associated procedures, called *methods*, that have access to the fields. The operation of calling a method is often viewed as sending the method name and arguments as a message to the object; this is sometimes called the *message-passing* view of object-oriented programming.

Procedures in stateful languages, like those in chapter 4 give another example of the power of programming with objects. A procedure is an object whose state is contained in its free variables. A closure has a single behavior: it may be invoked on some arguments. For example, the procedure g on page 105 controls the state of a counter, and the only thing one can do with this state is to increment it. More often, however, one wants an object to have several behaviors. Object-oriented programming languages provide support for this ability.

Often, one needs to manage several pieces of state with the same methods. For example, one might have several file systems or several queues in a program. To facilitate the sharing of methods, object-oriented programming systems typically provide *classes*, which are structures that specify the fields and methods of each such object. Each object is created as a class *instance*.

Similarly, one may often have several classes with fields and methods that are similar but not identical. To facilitate the sharing of implementation, object-oriented languages typically provide *inheritance*, which allows the programmer to define a new class as a small modification of an existing class by adding or changing the behavior of some methods, or by adding fields. In this case, we say the new class *inherits from* or *extends* the old class, since the rest of the class's behavior is inherited from the original class.

Whether program elements are modeling real-world objects or artificial aspects of a system's state, a program's structure is often clarified if it can be composed of objects that combine both behavior and state. It is also natural to associate behaviorally similar objects with the same class.

Real-world objects typically have some *state* and some *behavior* that either controls or is controlled by that state. For example, cats can eat, purr, jump, and lie down, and these activities are controlled by their current state, including how hungry and tired they are.

Objects and modules have many similarities, but they are very different. Both modules and classes provide a mechanism for defining opaque types. However, an object is a data structure with behavior; a module is just a set of bindings. One may have many objects of the same class; most module systems do not offer a similar capability. On the other hand, module systems such as PROC-MODULES allow a much more flexible way of controlling the visibility of names. Modules and classes can work fruitfully together.

9.1 Object-Oriented Programming

In this chapter, we study a simple object-oriented language that we call CLASSES. A CLASSES program consists of a sequence of class declarations followed by an expression that may make use of those classes.

Figure 9.1 shows a simple program in this language. It defines c1 as a class that inherits from object. Each object of class c1 will contain two fields named i and j. The fields are called *members* or *instance variables*. The class c1 supports three *methods*, sometimes called *member functions*, named initialize, countup, and getstate. Each method consists of its *method*

```
class c1 extends object
 field i
 field j
 method initialize (x)
  begin
   set i = x;
```

```
   set j = -(0,x)
  end
 method countup (d)
  begin
   set i = +(i,d);
   set j = -(j,d)
  end
 method getstate () list(i,j)
let t1 = 0
    t2 = 0
    o1 = new c1(3)
in begin
    set t1 = send o1 getstate();
    send o1 countup(2);
    set t2 = send o1 getstate();
    list(t1,t2)
   end
```

Figure 9.1: A simple object-oriented program

name, its *method vars* (also called *method parameters*), and its *method body*. The method names correspond to the kinds of *messages* to which instances of c1 can respond. We sometimes refer to “c1’s countup method.”

In this example, each of the methods of the class maintains the integrity constraint or *invariant* that $i = -j$. A real programming example would, of course, likely have far more complex integrity constraints.

The program in figure 9.1 first initializes three variables. t1 and t2 are initialized to zero. o1 is initialized to an object of the class c1. We say this object is an *instance* of class c1. An object is created using the new operation. This causes the class’s initialize method to be invoked, in this case setting the object’s field i to 3 and its field j to -3. The program then calls the getstate method of o1, returning the list (3 -3). Next, it calls o1’s countup method, changing the value of the two fields to 5 and -5. Then the getstate method is called again, returning the list (5 -5). Last, the value of list(t1,t2), which is ((3 -3) (5 -5)), is returned as the value of the entire program.

The program in figure 9.2 illustrates a key idea in object-oriented programming: *dynamic dispatch*. In this program we have trees with two kinds of nodes, interior-node and leaf-node. To find the sum of the leaves of a node, we send it the sum message. Generally, we do not know what kind of node we are sending the message to. Instead, each node accepts the sum message and uses its sum method to do the right thing. This is called *dynamic dispatch*. Here the expression builds a tree with two interior nodes and three leaf nodes. It sends a sum message to the node o1; o1 sends sum messages to its subtrees, and so on, returning 12 at the end. This program also shows that all methods are mutually recursive.

```
 class interior-node extends object
  field left
  field right
  method initialize (l, r)
   begin
    set left = l;
    set right = r
   end
  method sum () +(send left sum(),send right sum())
 class leaf-node extends object
  field value
  method initialize (v) set value = v
  method sum () value
let o1 = new interior-node(
          new interior-node(
           new leaf-node(3),
           new leaf-node(4)),
          new leaf-node(5))
in send o1 sum()
```

Figure 9.2: Object-oriented program for summing the leaves of a tree

A method body can invoke other methods of the same object by using the identifier self (sometimes called this), which is always bound to the object on which the method has been invoked. For example, in

```
class oddeven extends object
 method initialize () 1
```

```
 method even (n)
  if zero?(n) then 1 else send self odd(-(n,1))
 method odd (n)
  if zero?(n) then 0 else send self even(-(n,1))
let o1 = new oddeven()
in send o1 odd(13)
```

the methods even and odd invoke each other recursively, because when they are executed, self is bound to an object that contains them both. This is much like the dynamic-binding implementation of recursion in exercise 3.37.

9.2 Inheritance

Inheritance allows the programmer to define new classes by incremental modification of old ones. This is extremely useful in practice. For example, a colored point is like a point, except that it has additional methods to manipulate its color, as in the classic example in figure 9.3.

```
class point extends object
 field x
 field y
 method initialize (initx, inity)
  begin
   set x = initx;
   set y = inity
  end
 method move (dx, dy)
  begin
   set x = +(x,dx);
   set y = +(y,dy)
  end
 method get-location () list(x,y)
class colorpoint extends point
 field color
 method set-color (c) set color = c
 method get-color () color
let p = new point(3,4)
    cp = new colorpoint(10,20)
in begin
    send p move(3,4);
    send cp set-color(87);
    send cp move(10,20);
    list(send p get-location(),    % returns (6 8)
         send cp get-location(),   % returns (20 40)
         send cp get-color())      % returns 87
   end
```

Figure 9.3: Classic example of inheritance: `colorpoint`

If class c_2 extends class c_1, we say that c_1 is the *parent* or *superclass* of c_2 or that c_2 is a *child* of c_1. Since inheritance defines c_2 as an extension of c_1, c_1 must be defined before c_2. To get things started, the language includes a predefined class called object with no methods or fields. Since object has no initialize method, it is impossible to create an object of class object. Each class other than object has a single parent, but it may have many children. Thus the relation extends imposes a tree structure on the set of classes, with object at the root. Since each class has at most one immediate superclass, this is a *single-inheritance* language. Some languages allow classes to inherit from multiple superclasses. Such *multiple inheritance* is powerful, but it is also problematic; we consider some of the difficulties in the exercises.

The genealogical analogy is the source of the term *inheritance*. The analogy is often pursued so that we speak of the *ancestors* of a class (the chain from a class's parent to the root class object) or its *descendants*. If c_2 is a descendant of c_1, we sometimes say that c_2 is a *subclass* of c_1, and write $c_2 < c_1$.

If class c_2 inherits from class c_1, all the fields and methods of c_1 will be visible from the methods of c_2, unless they are redeclared in c_2. Since a class inherits all the methods and fields of its parent, an instance of a child class can be used anywhere an instance of its parent can be used. Similarly, any instance of any descendant of a class can be used anywhere an instance of the class can be used. This is sometimes called *subclass polymorphism*. This is the design we have chosen for our language; other object-oriented languages may have different visibility rules.

We next consider what happens when the fields or methods of a class are redeclared. If a field of c_1 is redeclared in one of its subclasses c_2, the new declaration *shadows* the old one, just as in lexical scoping. For example, consider figure 9.4. An object

of class c2 has two fields named y: the one declared in c1 and the one declared in c2. The methods declared in c1 see c1's fields x and y. In c2, the x in getx2 refers to c1's field x, but the y in gety2 refers to c2's field y.

```
class c1 extends object
 field x
 field y
 method initialize () 1
 method setx1 (v) set x = v
 method sety1 (v) set y = v
 method getx1 () x
 method gety1 () y
class c2 extends c1
 field y
 method sety2 (v) set y = v
 method getx2 () x
 method gety2 () y
let o2 = new c2()
in begin
    send o2 setx1(101);
    send o2 sety1(102);
    send o2 sety2(999);
    list(send o2 getx1(), % returns 101
         send o2 gety1(), % returns 102
         send o2 getx2(), % returns 101
         send o2 gety2()) % returns 999
   end
```

Figure 9.4: Example of field shadowing

If a method *m* of a class c_1 is redeclared in one of its subclasses c_2, we say that the new method *overrides* the old one. We call the class in which a method is declared that method's *host class*. Similarly, we define the host class of an expression to be the host class of the method (if any) in which the expression occurs. We also define the superclass of a method or expression as the parent class of its host class.

If an object of class c_2 is sent an *m* message, then the new method should be used. This rule is simple, but it has subtle consequences. Consider the following example:

```
class c1 extends object
 method initialize () 1
 method m1 () 11
 method m2 () send self m1()
class c2 extends c1
 method m1 () 22
let o1 = new c1() o2 = new c2()
in list(send o1 m1(), send o2 m1(), send o2 m2())
```

We expect send o1 m1() to return 11, since o1 is an instance of c1. Similarly, we expect send o2 m1() to return 22, since o2 is an instance of c2. Now what about send o2 m2()? Method m2 immediately calls method m1, but which one?

Dynamic dispatch tells us that we should look at the class of the object bound to self. The value of self is o2, which is of class c2. Hence the call send self m1() should return 22.

Our language has one more important feature, *super calls*. Consider the program in figure 9.5. There we have supplied the class colorpoint with an overly specialized initialize method that sets the field color as well as the fields x and y. However, the body of the new method duplicates the code of the overridden one. This might be acceptable in our small example, but in a large example this would clearly be bad practice. (Why?) Furthermore, if colorpoint declared a field x, there would be no way to initialize the field x of point, just as there is no way to initialize the first y in the example on page 331.

```
class point extends object
 field x
 field y
 method initialize (initx, inity)
  begin
   set x = initx;
   set y = inity
  end
 method move (dx, dy)
  begin
   set x = +(x,dx);
```

```
    set y = +(y,dy)
   end
  method get-location () list(x,y)
class colorpoint extends point
  field color
  method initialize (initx, inity, initcolor)
   begin
    set x = initx;
    set y = inity;
    set color = initcolor
   end
  method set-color (c) set color = c
  method get-color () color
let o1 = new colorpoint(3,4,172)
in send o1 get-color()
```

Figure 9.5: Example demonstrating a need for `super`

The solution is to replace the duplicated code in the body of colorpoint's initialize method with a *super call* of the form super initialize(). Then the initialize method in colorpoint would read

```
method initialize (initx, inity, initcolor)
 begin
  super initialize(initx, inity);
  set color = initcolor
 end
```

A super call super *n*(...) in the body of a method *m* invokes a method *n* of the parent of *m*'s host class. This is not necessarily the parent of the class of self. The class of self will always be a subclass of *m*'s host class, but it may not be the same, because *m* might have been declared in an ancestor of the target object.

To illustrate this distinction, consider figure 9.6. Sending an m3 message to an object o3 of class c3 finds c2's method for m3, which executes super m1(). The class of o3 is c3, whose parent is c2. But the host class is c2, and c2's superclass is c1. So c1's method for m1 is executed. This is an example of *static method dispatch*. Though the object of a super method call is self, method dispatch is static, because the specific method to be invoked can be determined from the text, independent of the class of self.

```
class c1 extends object
 method initialize () 1
 method m1 () send self m2()
 method m2 () 13
class c2 extends c1
 method m1 () 22
 method m2 () 23
 method m3 () super m1()
class c3 extends c2
 method m1 () 32
 method m2 () 33
let o3 = new c3()
in send o3 m3()
```

Figure 9.6: Example illustrating interaction of `super` call with `self`

In this example, c1's method for m1 calls o3's m2 method. This is an ordinary method call, so dynamic dispatch is used, so it is c3's m2 method that is found, returning 33.

9.3 The Language

For our language CLASSES, we extend the language IMPLICIT-REFS with the additional productions shown in figure 9.7. A program is a sequence of class declarations followed by an expression to be executed. A class declaration has a name, an immediate superclass name, zero or more field declarations, and zero or more method declarations. A method declaration, like a procedure declaration in a letrec, has a name, a list of formal parameters, and a body. We also extend the language with multiargument procedures, multideclaration let, letrec expressions, and some additional operations like addition and list. The operations on lists are as in exercise 3.9. Last, we add a begin expression, as in exercise 4.4, that evaluates its subexpressions from left to right and returns the value of the last one.

Program ::= {*ClassDecl*}* *Expression*

```
a-program (class-decls body)
```

ClassDecl ::= `class` *Identifier* `extends` *Identifier*
{`field` *Identifier*}* {*MethodDecl*}*

```
a-class-decl
 (class-name super-name
  field-names method-decls)
```

MethodDecl ::= `method` *Identifier* ({*Identifier*}*(,)) *Expression*

```
a-method-decl (method-name vars body)
```

Expression ::= `new` *Identifier* ({*Expression*}*(,))

```
new-object-exp (class-name rands)
```

Expression ::= `send` *Expression* *Identifier* ({*Expression*}*(,))

```
method-call-exp (obj-exp method-name rands)
```

Expression ::= `super` *Identifier* ({*Expression*}*(,))

```
super-call-exp (method-name rands)
```

Expression ::= `self`

```
self-exp ()
```

Figure 9.7: New productions for a simple object-oriented programming language

We add objects and lists as expressed values, so we have

$$\begin{aligned} ExpVal &= Int + Bool + Proc + Listof(ExpVal) + Obj \\ DenVal &= Ref(ExpVal) \end{aligned}$$

We write *Listof*(*ExpVal*) to indicate that the lists may contain any expressed value.

We will consider *Obj* in section 9.4.1. Classes are neither denotable nor expressible in our language: they may appear as part of objects but never as the binding of a variable or the value of an expression, but see exercise 9.29.

We have included four additional expressions. The new expression creates an object of the named class. The initialize method is then invoked to initialize the fields of the object. The rands are evaluated and passed to the initialize method. The value returned by this method call is thrown away and the new object is returned as the value of the new expression.

A self expression returns the object on which the current method is operating.

A send expression consists of an expression that should evaluate to an object, a method name, and zero or more operands. The named method is retrieved from the class of the object, and then is passed the arguments obtained by evaluating the operands. As in IMPLICIT-REFS, a new location is allocated for each of these arguments, and then the method body is

evaluated within the scope of lexical bindings associating the method's parameters with the references to the corresponding locations.

A super-call expression consists of a method name and zero or more arguments. It looks for a method of the given name, starting in the superclass of the expression's host class. The body of the method is then evaluated, with the current object as self.

9.4 The Interpreter

When a program is evaluated, all the class declarations are processed using initialize-class-env! and then the expression is evaluated. The procedure initialize-class-env! creates a global *class environment* that maps each class name to the methods of the class. Because this environment is global, we model it as a Scheme variable. We discuss the class environment in more detail in section 9.4.3.

```
value-of-program : Program → ExpVal
(define value-of-program
  (lambda (pgm)
    (initialize-store!)
    (cases program pgm
      (a-program (class-decls body)
        (initialize-class-env! class-decls)
        (value-of body (init-env))))))
```

The procedure value-of contains, as usual, a clause for each kind of expression in the language, including a clause for each of the four new productions.

We consider each new kind of expression in turn.

Usually, an expression is evaluated because it is part of a method that is operating on some object. In the environment, this object is bound to the pseudo-variable %self. We call this a *pseudo-variable* because it is bound lexically, like an ordinary variable, but it has somewhat different properties, which we explore below. Similarly, the name of the superclass of the host class of the current method is bound to the pseudo-variable %super.

When a self expression is evaluated, the value of %self is returned. The clause in value-of is

```
(self-exp ()
  (apply-env env '%self))
```

When a send expression is evaluated, the operands and the object expression are evaluated. We look in the object to find its class name. Then we find the method using find-method, which takes a class name and a method name and returns a method. That method is then applied to the current object and the method arguments.

```
(method-call-exp (obj-exp method-name rands)
  (let ((args (values-of-exps rands env))
        (obj (value-of obj-exp env)))
    (apply-method
      (find-method
        (object->class-name obj)
        method-name)
      obj
      args)))
```

Super method invocation is similar to ordinary method invocation except that the method is looked up in the superclass of the host class of the expression. The clause in value-of is

```
(super-call-exp (method-name rands)
  (let ((args (values-of-exps rands env))
        (obj (apply-env env '%self)))
    (apply-method
      (find-method (apply-env env '%super) method-name)
      obj
      args)))
```

Our last task is to create objects. When a new expression is evaluated, the operands are evaluated and a new object is created from the class name. Then its initialize method is called, but its value is ignored. Finally, the object is returned.

```
(new-object-exp (class-name rands)
  (let ((args (values-of-exps rands env))
        (obj (new-object class-name)))
```

```
    (apply-method
      (find-method class-name 'initialize)
      obj
      args)
    obj))
```

Next we determine how to represent objects, methods, and classes. To illustrate the representation, we use a running example, shown in figure 9.8.

```
class c1 extends object
 field x
 field y
 method initialize ()
  begin
   set x = 11;
   set y = 12
  end
 method m1 () ... x ... y ...
 method m2 () ... send self m3() ...
class c2 extends c1
 field y
 method initialize ()
  begin
   super initialize();
   set y = 22
  end
 method m1 (u,v) ... x ... y ...
 method m3 () ...
class c3 extends c2
 field x
 field z
 method initialize ()
  begin
   super initialize();
   set x = 31;
   set z = 32
  end
 method m3 () ... x ... y ... z ...
let o3 = new c3()
in send o3 m1(7,8)
```

Figure 9.8: Sample program for OOP implementation

9.4.1 Objects

We represent an object as a data type containing the object's class name and a list of references to its fields.

```
(define-datatype object object?
  (an-object
    (class-name identifier?)
    (fields (list-of reference?))))
```

We lay out the list with the fields from the "oldest" class first. Thus in figure 9.8, an object of class c1 would have its fields laid out as (x y); an object of class c2 would lay out its fields as (x y y), with the second y being the one belonging to c2, and an object of class c3 would be laid out as (x y y x z). The representation of object o3 from figure 9.8 is shown in figure 9.9. Of course, we want the methods in class c3 to refer to the field x declared in c3, not the one declared in c1. We take care of this when we set up the environment for evaluation of the method body.

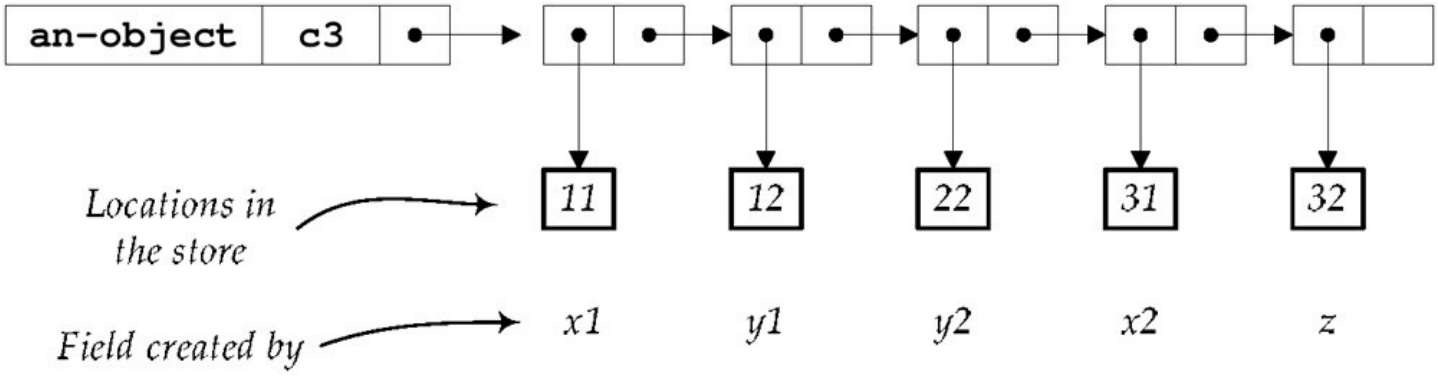

Figure 9.9: A simple object

This strategy has the useful property that any subclass of c3 will have these fields in the same positions in the list, because any fields added later will appear to the right of these fields. What is the position of x in a method that is defined in any subclass of c3? Assuming that x is not redefined, we know that the position of x must be 3 throughout all such methods. Thus, when a field variable is declared, the position of the corresponding value remains unchanged. This property allows field references to be determined statically, similarly to the way we handled variables in section 3.6.

Making a new object is easy. We simply create an an-object with a list of new references equal to the number of fields in the object. To determine that number, we get the list of field variables from the object's class. We initialize each location with an illegal value that will be recognizable in case the program dereferences the location without initializing it.

```
ClassName = Sym

new-object : ClassName → Obj
(define new-object
  (lambda (class-name)
    (an-object
      class-name
      (map
        (lambda (field-name)
          (newref (list 'uninitialized-field field-name)))
        (class->field-names (lookup-class class-name))))))
```

9.4.2 Methods

We next turn to methods. Methods are like procedures, except that they do not have a saved environment. Instead, they keep track of the names of the fields to which they refer. When a method is applied, it runs its body in an environment in which

- The method's formal parameters are bound to new references that are initialized to the values of the arguments. This is analogous to the behavior of apply-procedure in IMPLICIT-REFS.
- The pseudo-variables %self and %super are bound to the current object and the method's superclass, respectively.
- The visible field names are bound to the fields of the current object. To implement this, we define

```
(define-datatype method method?
  (a-method
    (vars (list-of identifier?))
    (body expression?)
    (super-name identifier?)
    (field-names (list-of identifier?))))

apply-method : Method × Obj × Listof(ExpVal) → ExpVal
(define apply-method
  (lambda (m self args)
    (cases method m
      (a-method (vars body super-name field-names)
        (value-of body
          (extend-env* vars (map newref args)
            (extend-env-with-self-and-super
              self super-name
              (extend-env field-names (object->fields self)
                (empty-env)))))))))
```

Here we use extend-env*from exercise 2.10, which extends an environment by binding a list of variables to a list of denoted

values. We have also added to our environment interface the procedure extend-env-with-self-and-super, which binds %self and %super to an object and a class name, respectively.

In order to make sure that each method sees the right fields, we need to be careful when constructing the field-names list. Each method should see only the last declaration of a field; all the others should be shadowed. So when we construct the field-names list, we will replace all but the rightmost occurrence of each name with a fresh identifier. For the program of figure 9.8, the resulting field-names fields look like

Class	Fields Defined	Fields	field-names
c1	x, y	(x y)	(x y)
c2	y	(x y y)	(x y%1 y)
c3	x, z	(x y y x z)	(x%1 y%1 y x z)

Since the method bodies do not know anything about x%1 or y%1, they can only see the rightmost field for each field variable, as desired.

Figure 9.10 shows the environment built for the evaluation of the method body in send o3 m1(7,8) in figure 9.8. This figure shows that the list of references may be longer than the list of variables: the list of variables is just (x y%1 y), since those are the only field variables visible from method m1 in c2, but the value of (object->fields self) is the list of all the fields of the object. However, since the values of the three visible field variables are in the first three elements of the list, and since we have renamed the first y to be y%1 (which the method knows nothing about) the method m1 will associate the variable y with the y declared in c2, as desired.

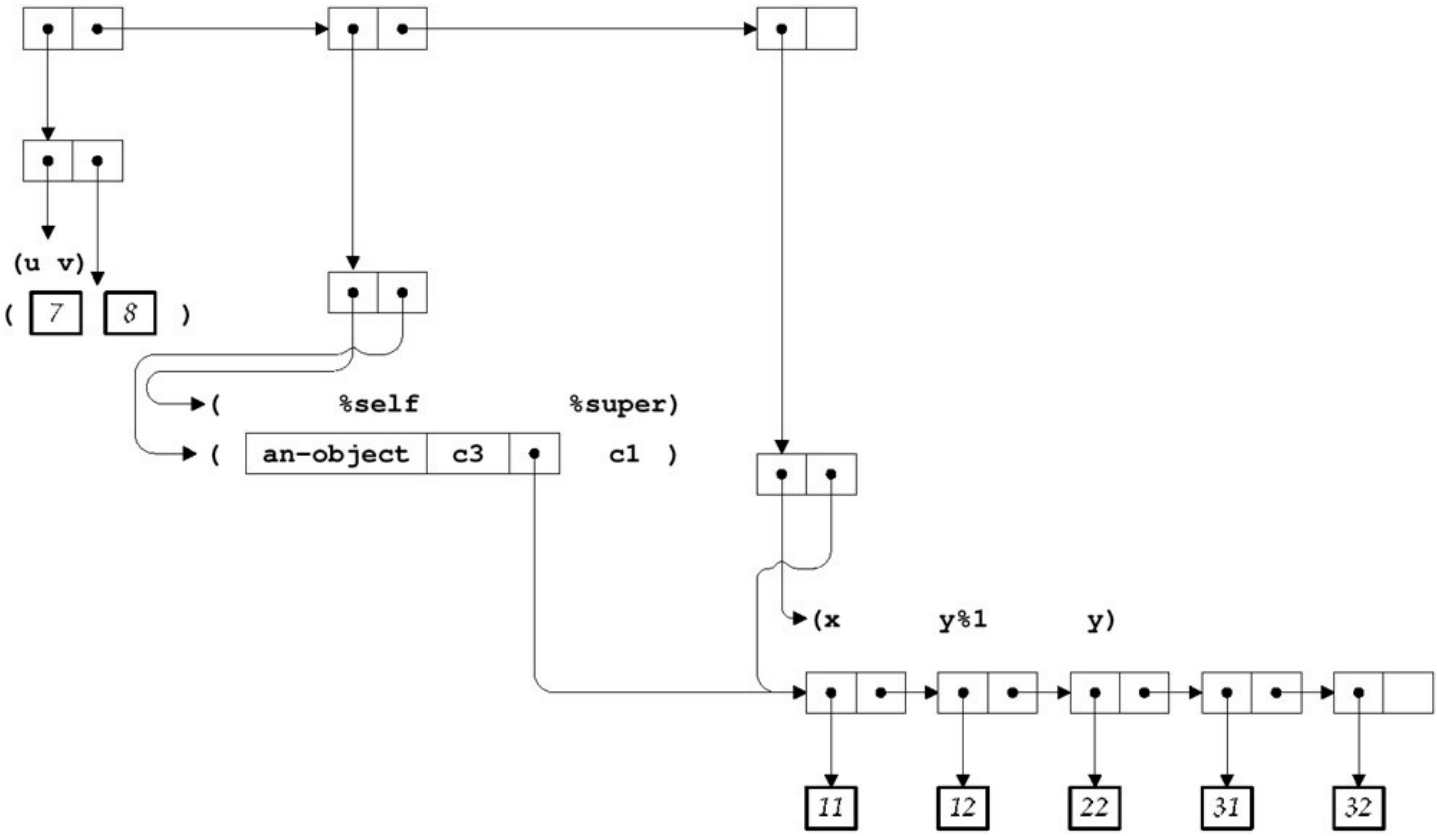

Figure 9.10: Environment for method application

When the host class and the class of self are the same, the list of variables is generally of the same length as the list of field locations. If the host class is higher up the class chain, then there may be more locations than field variables, but the values corresponding to the field variables will be at the beginning of the list, and the extra values will be inaccessible.

9.4.3 Classes and Class Environments

Our implementation so far has depended on the ability to get information about a class from its name. So we need a *class environment* to accomplish this task. The class environment will associate each class name with a data structure that describes the class.

The class environment is global: in our language, class declarations are grouped at the beginning of the program and are in force for the entire program. So, we represent the class environment as a global variable named the-class-env, which will contain a list of (class-name, class) lists, but we hide this representation behind the procedures add-to-class-env! and lookup-class.

```
ClassEnv = Listof(List(ClassName, Class))

the-class-env : ClassEnv
(define the-class-env '())

add-to-class-env! : ClassName × Class → Unspecified
(define add-to-class-env!
  (lambda (class-name class)
    (set! the-class-env
      (cons
        (list class-name class)
        the-class-env))))

lookup-class : ClassName → Class
(define lookup-class
  (lambda (name)
    (let ((maybe-pair (assq name the-class-env)))
      (if maybe-pair (cadr maybe-pair)
        (report-unknown-class name)))))
```

For each class, we need to keep track of three things: the name of its superclass, the list of its field variables, and an environment mapping its method names to its methods.

```
(define-datatype class class?
  (a-class
    (super-name (maybe identifier?))
    (field-names (list-of identifier?))
    (method-env method-environment?)))
```

Here we use the predicate (maybe identifier?) which is satisfied by any value that is either a symbol or is #f. The latter possibility is necessary for the class object, which has no superclass. The field-names will be the fields of the class, as seen by methods of that class, and methods will be an environment giving a definition to each method name that is defined for the class.

We will initialize the class environment with an entry for the class object. For each declaration, we add a new binding to the class environment, binding the name of the class to a class consisting of the name of the superclass, the field-names for the methods of that class, and the environment of methods for that class.

```
initialize-class-env! : Listof(ClassDecl) → Unspecified
(define initialize-class-env!
  (lambda (c-decls)
    (set! the-class-env
      (list
        (list 'object (a-class #f '() '())))))
    (for-each initialize-class-decl! c-decls)))

initialize-class-decl! : ClassDecl → Unspecified
(define initialize-class-decl!
  (lambda (c-decl)
    (cases class-decl c-decl
      (a-class-decl (c-name s-name f-names m-decls)
        (let ((f-names
                (append-field-names
                  (class->field-names (lookup-class s-name))
                  f-names)))
          (add-to-class-env!
            c-name
            (a-class s-name f-names
              (merge-method-envs
                (class->method-env (lookup-class s-name))
                (method-decls->method-env
                  m-decls s-name f-names)))))))))
```

The procedure append-field-names is used to create the field-names for the current class. It appends the fields of the superclass and the fields declared by the new class, except that any field of the superclass that is shadowed by a new field is replaced by a fresh identifier, as in the example on page 341.

```
append-field-names :
  Listof(FieldName) × Listof(FieldName) → Listof(FieldName)
(define append-field-names
  (lambda (super-fields new-fields)
    (cond
      ((null? super-fields) new-fields)
      (else
       (cons
         (if (memq (car super-fields) new-fields)
           (fresh-identifier (car super-fields))
           (car super-fields))
         (append-field-names
           (cdr super-fields) new-fields))))))
```

9.4.4 Method Environments

All that's left to do is to write find-method and merge-method-envs.

As we did for classes, we represent a method environment by a list of (method-name, method) lists. We look up a method using find-method.

```
MethodEnv = Listof(List(MethodName, Method))

find-method : Sym × Sym → Method
(define find-method
  (lambda (c-name name)
    (let ((m-env (class->method-env (lookup-class c-name))))
      (let ((maybe-pair (assq name m-env)))
        (if (pair? maybe-pair) (cadr maybe-pair)
          (report-method-not-found name))))))
```

With this information we can write method-decls->method-env. It takes the method declarations of a class and creates a method environment, recording for each method its bound variables, its body, the name of the superclass of the host class, and the field-names of the host class.

```
method-decls->method-env :
  Listof(MethodDecl) × ClassName × Listof(FieldName) → MethodEnv
(define method-decls->method-env
  (lambda (m-decls super-name field-names)
    (map
      (lambda (m-decl)
        (cases method-decl m-decl
          (a-method-decl (method-name vars body)
            (list method-name
              (a-method vars body super-name field-names)))))
      m-decls)))
```

Last, we write merge-method-envs. Since methods in the new class override those of the old class, we can simply append the environments, with the new methods first.

```
merge-method-envs : MethodEnv × MethodEnv → MethodEnv
(define merge-method-envs
  (lambda (super-m-env new-m-env)
    (append new-m-env super-m-env)))
```

There are ways of building method environments that will be more efficient for method lookup (exercise 9.18).

9.4.5 Exercises

Exercise 9.1 [*] Implement the following using the language of this section:

1. A queue class with methods empty?, enqueue, and dequeue.
2. Extend the queue class with a counter that counts the number of operations that have been performed on the current queue.
3. Extend the queue class with a counter that counts the total number of operations that have been performed on all the queues in the class. As a hint, remember that you can pass a shared counter object at initialization time.

Exercise 9.2 [*] Inheritance can be dangerous, because a child class can arbitrarily change the behavior of a method by

overriding it. Define a class bogus-oddeven that inherits from oddeven and overrides the method even so that let o1 = new bogus-oddeven() in send o1 odd (13) gives the wrong answer.

Exercise 9.3 [**] In figure 9.11, where are method environments shared? Where are the field-names lists shared?

```
((c3
  #(struct:a-class c2 (x%2 y%1 y x z)
    ((initialize #(struct:a-method ()
                   #(struct:begin-exp ...) c2 (x%2 y%1 y x z)))
     (m3 #(struct:a-method ()
            #(struct:diff-exp ...)) c2 (x%2 y%1 y x z))
     (initialize #(struct:a-method ...))
     (m1 #(struct:a-method (u v)
            #(struct:diff-exp ...) c1 (x y%1 y)))
     (m3 #(struct:a-method ...))
     (initialize  #(struct:a-method ...))
     (m1 #(struct:a-method ...))
     (m2 #(struct:a-method ()
            #(struct:method-call-exp #(struct:self-exp) m3 ())
            object (x y))))))
 (c2
  #(struct:a-class c1 (x y%1 y)
    ((initialize #(struct:a-method ()
                   #(struct:begin-exp ...) c1 (x y%1 y)))
     (m1 #(struct:a-method (u v)
            #(struct:diff-exp ...) c1 (x y%1 y)))
     (m3 #(struct:a-method ()
            #(struct:const-exp 23) c1 (x y%1 y)))
     (initialize #(struct:a-method ...))
     (m1 #(struct:a-method ...))
     (m2 #(struct:a-method ()
            #(struct:method-call-exp #(struct:self-exp) m3 ())
            object (x y))))))
 (c1
  #(struct:a-class object (x y)
    ((initialize #(struct:a-method ()
                   #(struct:begin-exp ...) object (x y)))
     (m1 #(struct:a-method ()
            #(struct:diff-exp ...) object (x y)))
     (m2 #(struct:a-method ()
            #(struct:method-call-exp #(struct:self-exp) m3 ())
            object (x y))))))
 (object
   #(struct:a-class #f () ())))
```

Figure 9.11: The class environment for figure 9.8

Exercise 9.4 [*] Change the representation of objects so that an *Obj* contains the class of which the object is an instance, rather than its name. What are the advantages and disadvantages of this representation compared to the one in the text?

Exercise 9.5 [*] The interpreter of section 9.4 stores the superclass name of a method's host class in the lexical environment. Change the implementation so that the method stores the host class name, and retrieves the superclass name from the host name.

Exercise 9.6 [*] Add to our language the expression instanceof *exp class-name*. The value of this expression should be true if and only if the object obtained by evaluating *exp* is an instance of *class-name* or of one of its subclasses.

Exercise 9.7 [*] In our language, the environment for a method includes bindings for the field variables declared in the host class *and* its superclasses. Limit them to just the host class.

Exercise 9.8 [*] Add to our language a new expression,

```
fieldref obj field-name
```

that retrieves the contents of the given field of the object. Add also

```
fieldset obj field-name = exp
```

which sets the given field to the value of *exp*.

Exercise 9.9 [*] Add expressions superfieldref *field-name* and superfieldset *field-name* = *exp* that manipulate the fields of self that would otherwise be shadowed. Remember super is static, and always refers to the superclass of the host class.

Exercise 9.10 [**] Some object-oriented languages include facilities for named-class method invocation and field references. In a named-class method invocation, one might write named-send c1 o m1(). This would invoke c1's m1 method on o, so long as o was an instance of c1 or of one of its subclasses, even if m1 were overridden in o's actual class. This is a form of static method dispatch. Named-class field reference provides a similar facility for field reference. Add named-class method invocation, field reference, and field setting to the language of this section.

Exercise 9.11 [**] Add to CLASSES the ability to specify that each method is either *private* and only accessible from within the host class, *protected* and only accessible from the host class and its descendants, or *public* and accessible from anywhere. Many object-oriented languages include some version of this feature.

Exercise 9.12 [**] Add to CLASSES the ability to specify that each field is either private, protected, or public as in exercise 9.11.

Exercise 9.13 [**] To defend against malicious subclasses like bogus-oddeven in exercise 9.2, many object-oriented languages have a facility for *final* methods, which may not be overridden. Add such a facility to CLASSES, so that we could write

```
class oddeven extends object
 method initialize () 1
 final method even (n)
  if zero?(n) then 1 else send self odd(-(n,1))
 final method odd (n)
  if zero?(n) then 0 else send self even(-(n,1))
```

Exercise 9.14 [**] Another way to defend against malicious subclasses is to use some form of *static dispatch*. Modify CLASSES so that method calls to *self* always use the method in the host class, rather than the method in the class of the target object.

Exercise 9.15 [**] Many object-oriented languages include a provision for *static* or *class* variables. Static variables associate some state with a class; all the instances of the class share this state. For example, one might write:

```
class c1 extends object
 static next-serial-number = 1
 field my-serial-number
 method get-serial-number () my-serial-number
 method initialize ()
  begin
   set my-serial-number = next-serial-number;
   set next-serial-number = +(next-serial-number,1)
  end
let o1 = new c1()
    o2 = new c1()
in list(send o1 get-serial-number(),
        send o2 get-serial-number())
```

Each new object of class c1 receives a new consecutive serial number.

Add static variables to our language. Since static variables can appear in a method body, apply-method must add additional bindings in the environment it constructs. What environment should be used for the evaluation of the initializing expression for a static variable (1 in the example above)?

Exercise 9.16 [**] Object-oriented languages frequently allow *overloading* of methods. This feature allows a class to have multiple methods of the same name, provided they have distinct *signatures*. A method's signature is typically the method name plus the types of its parameters. Since we do not have types in CLASSES, we might overload based simply on the method name and number of parameters. For example, a class might have two initialize methods, one with no parameters for use when initialization with a default field value is desired, and another with one parameter for use when a particular field value is desired. Extend our interpreter to allow overloading based on the number of method parameters.

Exercise 9.17 [**] As it stands, the classes in our language are defined globally. Add to CLASSES a facility for local classes, so one can write something like letclass *c* = ...in *e*. As a hint, consider adding the class environment as a parameter to the interpreter.

Exercise 9.18 [**] The method environments produced by merge-method-envs can be long. Write a new version of merge-method-envs with the property that each method name occurs exactly once, and furthermore, it appears in the same place as its earliest declaration. For example, in figure 9.8, method m2 should appear in the same place in the method environments of c1, c2, c3, and any descendant of c3.

Exercise 9.19 [**] Implement lexical addressing for CLASSES. First, write a lexical-address calculator like that of section 3.7.1 for the language of this section. Then modify the implementation of environments to make them nameless, and modify value-of so that apply-env takes a lexical address instead of a symbol, as in section 3.7.2.

Exercise 9.20 [***] Can anything equivalent to the optimizations of the exercise 9.19 be done for method invocations? Discuss why or why not.

Exercise 9.21 [**] If there are many methods in a class, linear search down a list of methods can be slow. Replace it by some faster implementation. How much improvement does your implementation provide? Account for your results, either positive or negative.

Exercise 9.22 [**] In exercise 9.16, we added overloading to the language by extending the interpreter. Another way to support overloading is not to modify the interpreter, but to use a syntactic preprocessor. Write a preprocessor that changes the name of every method *m* to one of the form *m*:@*n*, where *n* is the number of parameters in the method declaration. It must similarly change the name in every method call, based on the number of operands. We assume that :@ is not used by programmers in method names, but is accepted by the interpreter in method names. Compilers frequently use such a technique to implement method overloading. This is an instance of a general trick called *name mangling.*

Exercise 9.23 [***] We have treated super calls as if they were lexically bound. But we can do better: we can determine super calls *statically*. Since a super call refers to a method in a class's parent, and the parent, along with its methods, is known prior to the start of execution, we can determine the exact method to which any super call refers at the same time we do lexical-addressing and other analyses. Write a translator that takes each super call and replaces it with an abstract syntax tree node containing the actual method to be invoked.

Exercise 9.24 [***] Write a translator that replaces method names in named method calls as in exercise 9.10 with numbers indicating the offset of the named method in the run-time method table of the named class. Implement an interpreter for the translated code in which named method access is constant time.

Exercise 9.25 [***] Using the first example of inheritance from figure 9.5, we include a method in the class point that determines if two points have the same x- and y-coordinates. We add the method similarpoints to the point class as follows:

```
method similarpoints (pt)
 if equal?(send pt getx(), x)
 then equal?(send pt gety(), y)
 else zero?(1)
```

This works for both kinds of points. Since getx, gety, and similarpoints are defined in class point, by inheritance, they are defined in colorpoint. Test similarpoints to compare points with points, points with color points, color points with points, and color points with color points.

Next consider a small extension. We add a new similarpoints method to the colorpoint class. We expect it to return true if both points have the same x- and y- coordinates and further, in case both are color points, they have the same color. Otherwise it returns false. Here is an incorrect solution.

```
method similarpoints (pt)
 if super similarpoints(pt)
 then equal?(send pt getcolor(),color)
 else zero?(1)
```

Test this extension. Determine why it does not work on all the cases. Fix it so that all the tests return the correct values.

The difficulty of writing a procedure that relies on more than one object is known as the *binary method problem*. It demonstrates that the class-centric model of object-oriented programming, which this chapter explores, leaves something to be desired when there are multiple objects. It is called the *binary* method problem because the problem shows up with just two objects, but it gets progressively worse as the number of objects increases.

Exercise 9.26 [***] Multiple inheritance, in which a class can have more than one parent, can be useful, but may introduce serious complications. What if two inherited classes both have methods of the same name? This can be disallowed, or resolved by enumerating the methods in the class by some arbitrary rule, such as depth-first left-to-right, or by requiring that the ambiguity be resolved at the point such a method is called. The situation for fields is even worse. Consider the following situation, in which class c4 is to inherit from c2 and c3, both of which inherit from c1:

```
class c1 extends object
 field x
class c2 extends c1
```

```
class c3 extends c1
class c4 extends c2, c3
```

Does an instance of c4 have one instance of field x shared by c2 and c3, or does c4 have two x fields: one inherited from c2 and one inherited from c3? Some languages opt for sharing, some not, and some provide a choice, at least in some cases. The complexity of this problem has led to a design trend favoring single inheritance of classes, but multiple inheritance only for interfaces (section 9.5), which avoids most of these difficulties.

Add multiple inheritance to CLASSES. Extend the syntax as necessary. Indicate what issues arise when resolving method and field name conflicts. Characterize the sharing issue and its resolution.

Exercise 9.27 [***] Implement the following design for an object language without classes. An object will be a set of closures, indexed by method names, that share an environment (and hence some state). Classes will be replaced by procedures that return an object. So instead of writing send o1 m1(11,22,33), we would write an ordinary procedure call (getmethod(o1,m1) 11 22 33), and instead of writing

```
class oddeven extends object
 method initialize () 1
 method even (n)
  if zero?(n) then 1 else send self odd(-(n,1))
 method odd (n)
  if zero?(n) then 0 else send self even(-(n,1))
let o1 = new oddeven()
in send o1 odd(13)
```

we might write something like

```
let make-oddeven
 = proc ()
    newobject
     even = proc (n) if zero?(n) then 1
                     else (getmethod(self,odd) -(n,1))
     odd = proc (n) if zero?(n) then 0
                    else (getmethod(self,even) -(n,1))
    endnewobject
in let o1 = (make-oddeven) in (getmethod(o1,odd) 13)
```

Exercise 9.28 [***] Add inheritance to the language of exercise 9.27.

Exercise 9.29 [***] Design and implement an object-oriented language without explicit classes, by having each object contain its own method environment. Such an object is called a *prototype*. Replace the class object by a prototype object with no methods or fields. Extend a class by adding methods and fields to its prototype, yielding a new prototype. Thus we might write let c2 = extend c1 ... instead of class c2 extends c1 .... Replace the new operation with an operation clone that takes an object and simply copies its methods and fields. Methods in this language occur inside a lexical scope, so they should have access to lexically visible variables, as usual, as well as field variables. What shadowing relation should hold when a field variable of a superprototype has the same name as a variable in a containing lexical scope?

9.5 A Typed Language

In chapter 7, we showed how a type system could inspect a program to guarantee that it would never execute an inappropriate operation. No program that passes the checker will ever attempt to apply a nonprocedure to an argument, or to apply a procedure or other operator to the wrong number of arguments or to an argument of the wrong type.

In this section, we apply this technology to an object-oriented language that we call TYPED-OO. This language has all the safety properties listed above, and in addition, no program that passes our checker will ever send a message to an object for which there is no corresponding method, or send a message to an object with the wrong number of arguments or with arguments of the wrong type.

A sample program in TYPED-OO language is shown in figure 9.12. This program defines a class tree, which has a sum method that finds the sum of the values in the leaves, as in figure 9.2, and an equal method, which takes another tree and recursively descends through the trees to determine if they are equal.

```
interface tree
 method int sum ()
 method bool equal (t : tree)

class interior-node extends object implements tree
```

```
  field tree left
  field tree right
  method void initialize(l : tree, r : tree)
   begin
    set left = l; set right = r
   end
  method tree getleft () left
  method tree getright () right
  method int sum () +(send left sum(), send right sum())
  method bool equal (t : tree)
   if instanceof t interior-node
   then if send left equal(send
                           cast t interior-node
                           getleft())
        then send right equal(send
                              cast t interior-node
                              getright())
        else zero?(1)
   else zero?(1)

class leaf-node extends object implements tree
 field int value
 method void initialize (v : int) set value = v
 method int sum () value
 method int getvalue () value
 method bool equal (t : tree)
  if instanceof t leaf-node
  then zero?(-(value, send cast t leaf-node getvalue()))
  else zero?(1)

let o1 = new interior-node (
           new interior-node (
            new leaf-node(3),
            new leaf-node(4)),
          new leaf-node(5))
in list(send o1 sum(),
        if send o1 equal(o1) then 100 else 200)
```

Figure 9.12: A sample program in TYPED-OO

The major new features of the language are:

- Fields and methods are specified with their types, using a syntax similar to that used in chapter 7.
- The concept of an *interface* is introduced in an object-oriented setting.
- The concept of *subtype polymorphism* is added to the language.
- The concept of *casting* is introduced, and the instanceof test from exercise 9.6 is incorporated into the language.

We consider each of these items in turn.

The new productions for TYPED-OO are shown in figure 9.13. We add a void type as the type of a set operation, and list types as in exercise 7.9; as in exercise 7.9 we require that calls to list have at least one argument. We add identifiers to the set of type expressions, but for this chapter, an identifier used as a type is associated with the class or interface of the same name. We consider this correspondence in more detail below. Methods require their result type to be specified, along with the types of their arguments, using a syntax similar to that used for letrec in chapter 7. Last, two new expressions are added, cast and instanceof.

ClassDecl ::= `class` *Identifier* `extends` *Identifier*
`{implements` *Identifier*`}`*
`{field` *Type Identifier*`}`*
{*MethodDecl*}*

```
a-class-decl (c-name s-name i-names
                f-types f-names m-decls)
```

ClassDecl ::= `interface` *Identifier* {*AbstractMethodDecl*}*

```
an-interface-decl (i-name abs-m-decls)
```

MethodDecl ::= `method` *Type Identifier* ({*Identifier* : *Type*}*(,)) *Expression*

```
a-method-decl
  (res-type m-name vars var-types body)
```

AbstractMethodDecl ::= `method` *Type Identifier* ({*Identifier* :*Type*}*(,))

```
an-abstract-method-decl
  (result-type m-name m-var-types m-vars)
```

Expression ::= `cast` *Expression Identifier*

```
cast-exp (exp c-name)
```

Expression ::= `instanceof` *Expression Identifier*

```
instanceof-exp (exp name)
```

Type ::= `void`

```
void-type ()
```

Type ::= *Identifier*

```
class-type (class-name)
```

Type ::= `listof` *Type*

```
list-type (type1)
```

Figure 9.13: New productions for TYPED-OO

In order to understand the new features of this language, we must define the types of the language, as we did in definition 7.1.1.

Definition 9.5.1 *The property of an expressed value v being of type t is defined as follows:*

- *If c is a class, then a value is of type c if and only if it is an object, and it is an instance of the class c or one of its descendants.*
- *If I is an interface, then a value is of type I if and only if it is an object that is an instance of a class that implements I. A class implements I if and only if it has an implements I declaration or if one of its ancestors implements I.*
- *If t is some other type, then the rules of definition 7.1.1 apply.*

An object is an instance of exactly one class, but it can have many types.

- It has the type of the class that created it.
- It has the type of that class's superclass and of all classes above it in the inheritance hierarchy. In particular, every object has type object.
- It has the type of any interfaces that its creating class implements.

The second property is called *subclass polymorphism*. The third property could be called *interface polymorphism*.

An interface represents the set of all objects that implement a particular set of methods, regardless of how those objects were constructed. Our typing system will allow a class *c* to declare that it implements interface *I* only if *c* provides all the methods, with all the right types, that are required by *I*. A class may implement several different interfaces, although we have only used one in our example.

In figure 9.12, the classes interior-node and leaf-node both implement the interface tree. The typechecker allows this, because they both implement the sum and equal methods that are required for tree.

The expression instanceof *e c* returns a true value whenever the object obtained by evaluating *e* is an instance of the class *c* or of one of its descendants. Casting complements instanceof. The value of a cast expression cast *e c* is the same as the value of *e* if that value is an object that is an instance of the class *c* or one of its descendants. Otherwise the cast expression reports an error. The type of cast *e c* will always be *c*, since its value, if it returns, is guaranteed to be of type *c*.

For example, our sample program includes the method

```
method bool equal(t : tree)
 if instanceof t interior-node
 then if send left
          equal(send cast t interior-node getleft())
      then send right
            equal(send cast t interior-node getright())
      else false
 else false
```

The expression cast t interior-node checks to see if the value of t is an instance of interior-node (or one of its descendants, if interior-node had descendants). If it is, the value of t is returned; otherwise, an error is reported. An instanceof expression returns a true value if and only if the corresponding cast would succeed. Hence in this example the cast is guaranteed to succeed, since it is guarded by the instanceof. The cast, in turn, guards the use of send ... getleft(). The cast expression is guaranteed to return a value of class interior-node, and therefore it will be safe to send this value a getleft message.

For our implementation, we begin with the interpreter of section 9.4.1. We add two new clauses to value-of to evaluate instanceof and cast expressions:

```
(cast-exp (exp c-name)
  (let ((obj (value-of exp env)))
    (if (is-subclass? (object->class-name obj) c-name)
      obj
      (report-cast-error c-name obj))))

(instanceof-exp (exp c-name)
  (let ((obj (value-of exp env)))
    (if (is-subclass? (object->class-name obj) c-name)
      (bool-val #t)
      (bool-val #f))))
```

The procedure is-subclass? traces the parent link of the first class structure until it either finds the second one or stops when the parent link is #f. Since interfaces are only used as types, they are ignored in this process.

```
is-subclass? : ClassName × ClassName → Bool
(define is-subclass?
  (lambda (c-name1 c-name2)
    (cond
      ((eqv? c-name1 c-name2) #t)
      (else
        (let ((s-name (class->super-name
                        (lookup-class c-name1))))
          (if s-name (is-subclass? s-name c-name2) #f))))))
```

This completes the modification of the interpreter for the language of this section.

Exercise 9.30 [*] Create an interface summable:

```
interface summable
 method int sum ()
```

Now define classes for summable lists, summable binary trees (as in figure 9.12) and summable general trees (in which each node contains a summable list of children).

Then do the same thing for an interface

```
interface stringable
 method string to-string ()
```

Exercise 9.31 [*] In figure 9.12, would it have worked to make tree a class and have the two node classes inherit from tree? In what circumstances is this a better method than using an interface like summable? In what circumstances is it inferior?

Exercise 9.32 [**] Write an equality predicate for the class tree that does not use instanceof or cast. What is needed here is a *double dispatch*, in place of the single dispatch provided by the usual methods. This can be simulated as follows: Instead of using instanceof to find the class of the argument t, the current tree should send back to t a message that encodes its own class, along with parameters containing the values of the appropriate fields.

9.6 The Type Checker

We now turn to the checker for this language. The goal of the checker is to guarantee a set of safety properties. For our language, these properties are those of the underlying procedural language, plus the following properties of the object-oriented portion of the language: no program that passes our type checker will ever

- send a message to a non-object,
- send a message to an object for which there is no corresponding method,
- send a message to an object with the wrong number of arguments or with arguments of the wrong type.

We make no attempt to verify that the initialize methods actually initialize all the fields, so it will still be possible for a program to reference an uninitialized field. Similarly, because it is in general impossible to predict the type of an initialize method, our checker will not prevent the explicit invocation of an initialize method with the wrong number of arguments or arguments of the wrong type, but the implicit invocation of initialize by new will always be correct.

The checker begins with the implementation of type-of-program. Since all the methods of all the classes are mutually recursive, we proceed much as we do for letrec. For a letrec, we first built tenv-for-letrec-body by collecting the declared type of the procedure (figure 7.3). We then checked each procedure body against its declared result type. Finally, we checked the body of the letrec in tenv-for-letrec-body.

Here, we first call initialize-static-class-env!, which walks through the class declarations, collecting all the types into a static class environment. Since this environment is global and never changes, we keep it in a Scheme variable rather than passing it as a parameter. Then we check each class declaration, using check-class-decl!. Finally, we find the type of the body of the program.

```
type-of-program : Program → Type
(define type-of-program
  (lambda (pgm)
    (cases program pgm
      (a-program (class-decls exp1)
        (initialize-static-class-env! class-decls)
        (for-each check-class-decl! class-decls)
        (type-of exp1 (init-tenv))))))
```

The static class environment will map each class name to a static class containing the name of its parent, the names and types of its fields, and the names and types of its methods. In our language, interfaces have no parent and no fields, so they will be represented by a data structure containing only the names and types of its required methods (but see exercise 9.36).

```
(define-datatype static-class static-class?
  (a-static-class
    (super-name (maybe identifier?))
    (interface-names (list-of identifier?))
    (field-names (list-of identifier?))
    (field-types (list-of type?))
```

```
    (method-tenv method-tenv?))
  (an-interface
    (method-tenv method-tenv?)))
```

Before considering how the static class environment is built, we consider how to extend type-of to check the types of the six kinds of object-oriented expressions: self, instanceof, cast, method calls, super calls, and new.

For a self expression, we look up the type of self using the pseudo-variable %self, which we will be sure to bind to the type of the current host class, just as in the interpreter we bound it to the current host object.

If an instanceof expression returns, it always returns a bool. The expression cast *e c* returns the value of *e* provided that the value is an object that is an instance of *c* or one of its descendants. Hence, if cast *e c* returns a value, then that value is of type *c*. So we can always assign cast *e c* the type *c*. For both instanceof and cast expressions, the interpreter evaluates the argument and runs object->class-name on it, so we we must of course check that the operand is well-typed and returns a value that is an object. The code for these three cases is shown in figure 9.14.

```
(self-exp ()
  (apply-tenv tenv '%self))

(instanceof-exp (exp class-name)
  (let ((obj-type (type-of exp tenv)))
    (if (class-type? obj-type)
      (bool-type)
      (report-bad-type-to-instanceof obj-type exp))))

(cast-exp (exp class-name)
  (let ((obj-type (type-of exp tenv)))
    (if (class-type? obj-type)
      (class-type class-name)
      (report-bad-type-to-cast obj-type exp))))
```

Figure 9.14: `type-of` clauses for object-oriented expressions, part 1

We next consider method calls. We now have three different kinds of calls in our language: procedure calls, method calls, and super calls. We abstract the process of checking these into a single procedure.

```
type-of-call : Type × Listof(Type) × Listof(Exp) → Type
(define type-of-call
  (lambda (rator-type rand-types rands exp)
    (cases type rator-type
      (proc-type (arg-types result-type)
        (if (not (= (length arg-types) (length rand-types)))
          (report-wrong-number-of-arguments
            (map type-to-external-form arg-types)
            (map type-to-external-form rand-types)
            exp))
        (for-each check-is-subtype! rand-types arg-types rands)
        result-type)
      (else
        (report-rator-not-of-proc-type
          (type-to-external-form rator-type)
          exp)))))
```

This procedure is equivalent to the line for call-exp in CHECKED (figure 7.2) with two notable additions. First, because our procedures now take multiple arguments, we check to see that the call has the right number of arguments, and in the for-each line we check the type of each operand against the type of the corresponding argument in the procedure's type. Second, and more interestingly, we have replaced check-equal-type! of figure 7.2 by check-is-subtype!.

Why is this necessary? The principle of subclass polymorphism says that if class c_2 extends c_1, then an object of class c_2 can be used in any context in which an object of class c_1 can appear. If we wrote a procedure proc (o : c_1) ..., that procedure should be able to take an actual parameter of type c_2.

In general, we can extend the notion of subclass polymorphism to *subtype polymorphism*, as we did with <: in chapter 8. We say that t_1 is a subtype of t_2 if and only if

- t_1 and t_2 are classes, and t_1 is a subclass of t_2, or

- t_1 is a class and t_2 is an interface, and t_1 or one of its superclasses implements t_2, or
- t_1 and t_2 are procedure types, and the argument types of t_2 are subtypes of the argument types of t_1, and the result type of t_1 is a subtype of t_2.

To understand the last rule, let t_1 be (c1 -> d1), let t_2 be (c2 -> d2), with c2 < c1 and d1 < d2. Let f be a procedure of type t_1. We claim that f also has type t_2. Why? Imagine that we give f an argument of type c2. Since c2 < c1, the argument is also a c1. Therefore it is an acceptable argument to f. f then returns a value of type d1. But since d1 < d2, this result is also of type d2. So, if f is given an argument of type c2, it returns a value of type d2. Hence f has type (c2 -> d2). We say that subtyping is *covariant* in the result type and *contravariant* in the argument type. See figure 9.15. This is similar to the definition of <:-iface in section 8.3.2.

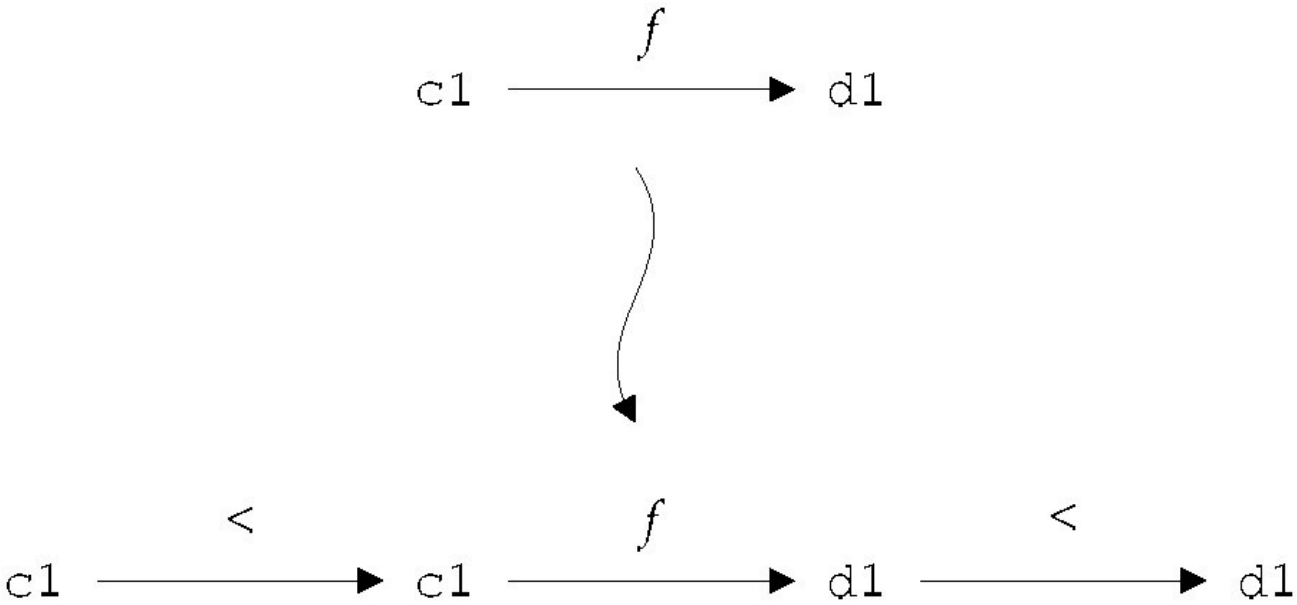

Figure 9.15: Subtyping a procedure type

The code for this is shown in figure 9.16. The code uses every2?, an extension of the procedure every? from exercise 1.24 that takes a two-argument predicate and two lists, and returns #t if the lists are of the same length and corresponding elements satisfy the predicate, or returns #f otherwise.

```
check-is-subtype! : Type × Type × Exp → Unspecified
(define check-is-subtype!
  (lambda (ty1 ty2 exp)
    (if (is-subtype? ty1 ty2)
      #t
      (report-subtype-failure
        (type-to-external-form ty1)
        (type-to-external-form ty2)
        exp))))

is-subtype? : Type × Type → Bool
(define is-subtype?
  (lambda (ty1 ty2)
    (cases type ty1
      (class-type (name1)
        (cases type ty2
          (class-type (name2)
            (statically-is-subclass? name1 name2))
          (else #f)))
        (proc-type (args1 res1)
          (cases type ty2
            (proc-type (args2 res2)
              (and
                (every2? is-subtype? args2 args1)
                (is-subtype? res1 res2)))
            (else #f)))
        (else (equal? ty1 ty2)))))

statically-is-subclass? : ClassName × ClassName → Bool
```

```
(define statically-is-subclass?
  (lambda (name1 name2)
    (or
      (eqv? name1 name2)
      (let ((super-name
              (static-class->super-name
                (lookup-static-class name1))))
        (if super-name
          (statically-is-subclass? super-name name2)
          #f))
      (let ((interface-names
              (static-class->interface-names
                (lookup-static-class name1))))
        (memv name2 interface-names)))))
```

Figure 9.16: Subtyping in TYPED-OO

We can now consider each of the three kinds of calls (figure 9.17). For a method call, we first find the types of the target object and of the operands, as usual. We use find-method-type, analogous to find-method, to find the type of the method. If the type of the target is not a class or interface, then type->class-name will report an error. If there is no corresponding method, then find-method-type will report an error. We then call type-of-call to verify that the types of the operands are compatible with the types expected by the method, and to return the type of the result.

```
(method-call-exp (obj-exp method-name rands)
  (let ((arg-types (types-of-exps rands tenv))
        (obj-type (type-of obj-exp tenv)))
    (type-of-call
      (find-method-type
        (type->class-name obj-type)
        method-name)
      arg-types
      rands
      exp)))

(super-call-exp (method-name rands)
  (let ((arg-types (types-of-exps rands tenv))
        (obj-type (apply-tenv tenv '%self)))
    (type-of-call
      (find-method-type
        (apply-tenv tenv '%super)
        method-name)
      arg-types
      rands
      exp)))

 (new-object-exp (class-name rands)
   (let ((arg-types (types-of-exps rands tenv))
         (c (lookup-static-class class-name)))
     (cases static-class c
       (an-interface (method-tenv)
         (report-cant-instantiate-interface class-name))
       (a-static-class (super-name i-names
                        field-names field-types
                        method-tenv)
         (type-of-call
           (find-method-type
             class-name
             'initialize)
           arg-types
           rands
           exp)
         (class-type class-name)))))
```

Figure 9.17: type-of clauses for object-oriented expressions, part 2

For a new expression, we first retrieve the class information for the class name. If there is no class associated with the name, a type error is reported. Last, we call type-of-call with the types of the operands to see if the call to initialize is safe. If these checks succeed, then the execution of the expression is safe. Since the new expression returns a new object of the specified class, the type of the result is the corresponding type of the class.

We have now completed our discussion of checking expressions in TYPED-OO, so we now return to constructing the static class environment.

To build the static class environment, initialize-static-class-env! first sets the static class environment to empty, and then adds a binding for the class object. It then goes through each class or instance declaration and adds an appropriate entry to the static class environment.

```
initialize-static-class-env! : Listof(ClassDecl) → Unspecified
(define initialize-static-class-env!
  (lambda (c-decls)
    (empty-the-static-class-env!)
    (add-static-class-binding!
      'object (a-static-class #f '() '() '() '()))
    (for-each add-class-decl-to-static-class-env! c-decls)))
```

The procedure add-class-decl-to-static-class-env! (fig. 9.18) does the bulk of the work of creating the static classes. For each class, we must collect all its interfaces, fields, and methods:

- A class implements any interfaces that its parent implements, plus the interfaces that it claims to implement.
- A class has the fields that its parent has, plus its own, except that its parent's fields are shadowed by the locally declared fields. So the field-names are calculated with append-field-names, just as in initialize-class-env! (page 344).
- The types of the class's fields are the types of its parent's fields, plus the types of its locally declared fields.
- The methods of the class are those of its parent plus its own, with their declared types. We keep the type of a method as a proc-type. We put the locally declared methods first, since they override the parent's methods.
- We check that there are no duplicates among the local method names, the interface names, and the field names. We also make sure that there is an initialize method available in the class.

```
add-class-decl-to-static-class-env! : ClassDecl → Unspecified
(define add-class-decl-to-static-class-env!
  (lambda (c-decl)
    (cases class-decl c-decl
      (an-interface-decl (i-name abs-m-decls)
        (let ((m-tenv
                (abs-method-decls->method-tenv abs-m-decls)))
          (check-no-dups! (map car m-tenv) i-name)
          (add-static-class-binding!
            i-name (an-interface m-tenv))))
      (a-class-decl (c-name s-name i-names
                     f-types f-names m-decls)
        (let ((i-names
                (append
                  (static-class->interface-names
                    (lookup-static-class s-name))
                  i-names))
              (f-names
                (append-field-names
                  (static-class->field-names
                    (lookup-static-class s-name))
                  f-names))
              (f-types
                (append
                  (static-class->field-types
                    (lookup-static-class s-name))
                  f-types))
              (method-tenv
                (let ((local-method-tenv
                        (method-decls->method-tenv m-decls)))
                  (check-no-dups!
                    (map car local-method-tenv) c-name)
                  (merge-method-tenvs
                    (static-class->method-tenv
                      (lookup-static-class s-name))
                    local-method-tenv))))
          (check-no-dups! i-names c-name)
          (check-no-dups! f-names c-name)
          (check-for-initialize! method-tenv c-name)
          (add-static-class-binding! c-name
            (a-static-class
```

```
            s-name i-names f-names f-types method-tenv)))))))
```

Figure 9.18: add-class-decl-to-static-class-env!

For an interface declaration, we need only process the method names and their types.

Once the static class environment has been built, we can check each class declaration. This is done by check-class-decl! (figure 9.19). For an interface, there is nothing to check. For a class declaration, we check each method, passing along information collected from the static class environment. Finally, we check to see that the class actually implements each of the interfaces that it claims to implement.

```
check-class-decl! : ClassDecl → Unspecified
(define check-class-decl!
  (lambda (c-decl)
    (cases class-decl c-decl
      (an-interface-decl (i-name abs-method-decls)
        #t)
      (a-class-decl (class-name super-name i-names
                     field-types field-names method-decls)
        (let ((sc (lookup-static-class class-name)))
          (for-each
            (lambda (method-decl)
              (check-method-decl! method-decl
                class-name super-name
                (static-class->field-names sc)
                (static-class->field-types sc)))
            method-decls))
        (for-each
          (lambda (i-name)
            (check-if-implements! class-name i-name))
          i-names)))))
```

Figure 9.19: check-class-decl!

To check a method declaration, we first check to see whether its body matches its declared type. To do this, we build a type environment that matches the environment in which the body will be evaluated. We then check to see that the result type of the body is a subtype of the declared result type.

We are not done, however: we have to make sure that if this method is overriding some method in the superclass, then it has a type that is compatible with the superclass method's type. We have to do this because this method might be called from a method that knows only about the super-type. The only exception to this rule is initialize, which is only called at the current class, and which needs to change its type under inheritance (see figure 9.12). To do this, it calls maybe-find-method-type, which returns either the type of the method if it is found, or #f otherwise. See figure 9.20.

```
check-method-decl! :
 MethodDecl × ClassName × ClassName × Listof(FieldName) × Listof(Type)
   → Unspecified
(define check-method-decl!
  (lambda (m-decl self-name s-name f-names f-types)
    (cases method-decl m-decl
      (a-method-decl (res-type m-name vars var-types body)
        (let ((tenv
                (extend-tenv
                  vars var-types
                  (extend-tenv-with-self-and-super
                    (class-type self-name)
                    s-name
                    (extend-tenv f-names f-types
                      (init-tenv))))))
          (let ((body-type (type-of body tenv)))
            (check-is-subtype! body-type res-type m-decl)
            (if (eqv? m-name 'initialize) #t
              (let ((maybe-super-type
                      (maybe-find-method-type
                        (static-class->method-tenv
                          (lookup-static-class s-name))
                        m-name)))
                (if maybe-super-type
```

```
                  (check-is-subtype!
                    (proc-type var-types res-type)
                      maybe-super-type body)
                  #t)))))))))
```

Figure 9.20: `check-method-decl!`

The procedure check-if-implements?, shown in figure 9.21, takes two symbols, which should be a class name and an interface name. It first checks to see that each symbol names what it should name. It then goes through each method in the interface and checks to see that the class provides a method with the same name and a compatible type.

```
check-if-implements! : ClassName × InterfaceName → Bool
(define check-if-implements!
  (lambda (c-name i-name)
    (cases static-class (lookup-static-class i-name)
      (a-static-class (s-name i-names f-names f-types
                        m-tenv)
        (report-cant-implement-non-interface
          c-name i-name))
      (an-interface (method-tenv)
        (let ((class-method-tenv
                (static-class->method-tenv
                  (lookup-static-class c-name))))
          (for-each
            (lambda (method-binding)
              (let ((m-name (car method-binding))
                    (m-type (cadr method-binding)))
                (let ((c-method-type
                        (maybe-find-method-type
                          class-method-tenv
                          m-name)))
                  (if c-method-type
                    (check-is-subtype!
                      c-method-type m-type c-name)
                    (report-missing-method
                      c-name i-name m-name)))))
            method-tenv))))))
```

Figure 9.21: check-if-implements

The static class environment built for the sample program of figure 9.12 is shown in figure 9.22. The static classes are in reverse order, reflecting the order in which the class environment is built. Each of the three classes has its methods in the same order, with the same type, as desired.

```
((leaf-node
   #(struct:a-static-class
      object
      (tree)
      (value)
      (#(struct:int-type))
      ((initialize #(struct:proc-type
                      (#(struct:int-type))
                      #(struct:void-type)))
       (sum #(struct:proc-type () #(struct:int-type)))
       (getvalue #(struct:proc-type () #(struct:int-type)))
       (equal #(struct:proc-type
                 (#(struct:class-type tree))
                 #(struct:bool-type))))))
 (interior-node
   #(struct:a-static-class
      object
      (tree)
      (left right)
      (#(struct:class-type tree) #(struct:class-type tree))
      ((initialize #(struct:proc-type
                      (#(struct:class-type tree)
                      #(struct:class-type tree))
                    #(struct:void-type)))
       (getleft #(struct:proc-type ()
                   #(struct:class-type tree)))
       (getright #(struct:proc-type ()
```

```
                    #(struct:class-type tree)))
       (sum #(struct:proc-type () #(struct:int-type)))
       (equal #(struct:proc-type
                 (#(struct:class-type tree))
                 #(struct:bool-type))))))
 (tree
   #(struct:an-interface
      ((sum #(struct:proc-type () #(struct:int-type)))
       (equal #(struct:proc-type
                 (#(struct:class-type tree))
                 #(struct:bool-type))))))
 (object
   #(struct:a-static-class #f () () () ())))
```

Figure 9.22: Static class environment built for the sample program

This completes the presentation of the checker.

Exercise 9.33 [*] Extend the type checker to enforce the safety property that no instanceof or cast expression is ever performed on a value that is not an object, or on a type that is not a class.

Exercise 9.34 [*] The expression cast *e c* can not succeed unless the type of *e* is either a descendant or an ancestor of *c*. (Why?) Extend the type checker to guarantee that the program never evaluates a cast expression unless this property holds. Extend the checker for instanceof to match.

Exercise 9.35 [*] Extend the type checker to enforce the safety property that an initialize method is called only from within a new-object-exp.

Exercise 9.36 [*] Extend the language to allow interfaces to inherit from other interfaces. An interface should require all the methods required by all of its parents.

Exercise 9.37 [**] Our language TYPED-OO uses dynamic dispatch. An alternative design is *static dispatch*. In static dispatch, the choice of method depends on an object's type rather than its class. Consider the example

```
class c1 extends object
 method int initialize () 1
 method int m1 () 11
 staticmethod int m2 () 21
class c2 extends c1
 method void m1 () 12
 staticmethod int m2 () 22

let f = proc (x : c1) send x m1()
    g = proc (x : c1) send x m2()
    o = new c2()
in list((f o), (g o))
```

When f and g are called, x will have type c1, but it is bound to an object of class c2. The method m1 uses dynamic dispatch, so c2's method for m1 is invoked, returning 12. The method m2 uses static dispatch, so sending an m2 message to x invokes the method associated with the type of x, in this case c1, so 21 is returned.

Modify the interpreter of section 9.5 to handle static methods. As a hint, think about keeping type information in the environment so that the interpreter can figure out the type of the target expression in a send.

Exercise 9.38 [**] Why must the class information be added to the static class environment before the methods are checked? As a hint, consider what happens if a method body invokes a method on self?)

Exercise 9.39 [**] Make the typechecker prevent calls to initialize other than the implicit call inside new.

Exercise 9.40 [*] Modify the design of the language so that every field declaration contains an expression that is used to initialize the field. Such a design has the advantage that a checked program will never refer to an uninitialized value.

Exercise 9.41 [**] Extend the typechecker to handle fieldref and fieldset, as in exercise 9.8.

Exercise 9.42 [**] In the type checker, static methods are treated in the same way as ordinary methods, except that a static method may not be overridden by a dynamic one, or vice versa. Extend the checker to handle static methods.