Chapter 7

A Language with Objects

This chapter introduces a more concrete syntax for a "language with objects".

Apart from meeting the challenge of describing inherited object behaviour, our model must incorporate more basic mechanisms for handling the important properties of object identity and state. Here, object creation is described, using constructors to initialise object state, which is protected using closure techniques. The complex issue of constructor encapsulation and inheritance is discussed. Object identity must also be preserved. The first part of a concrete programming language is presented. Here, variables and assignment are introduced, then simple and compound expressions and finally basic memory management.

7.1 Object State

The theory of classification has been developed so far chiefly to account for the observable *behaviour* of objects, described in terms of their evolving type under polymorphic inheritance. Objects have other important properties, such as *identity* and *state* (see chapter 2). It is possible, though long-winded, to model an object-oriented program as a set of functions on a global environment which is passed from function to function. The environment is a map from identifiers to objects and the mutation of objects is modelled by replacing individual maplets in the environment. This is described in more detail in Appendix 1.

7.1.1 Mutable State

A more direct approach is taken here that assumes an implicit translation into the above form. In order to model the notion of object *state*, the pure functional calculus is augmented with assignment, to reflect the fact that state can be

updated. As soon as the evaluation of expressions becomes dependent on mutable state, the property of referential transparency is lost. Two similar-looking expressions may yield different results. Equivalence up to isomorphism under β-reduction is therefore lost and expressions acquire a distinct *identity*.

Chapter 2 described a technique for enclosing state values inside functions. A *closure* was introduced as a function defined within the lexical scope of certain free variables. The effect of this is to give those free variables an indefinite extent, but a scope local to invocations of the closure function. With the addition of assignment, updates to state values may be modelled as modifications to the bindings of hidden variables, whenever the closure is invoked. For example, a simple *point* object may be defined:

let xstate = 3 **in**
  let ystate = 4 **in**
  μself.{x $\mapsto$ xstate, y $\mapsto$ ystate, identity $\mapsto$ self,
    equal $\mapsto$ λother.(xstate = other.x $\wedge$ ystate = other.y),
    move $\mapsto$ λnx.λny.(xstate := nx; ystate := ny; self)}

which is closed over the variables *xstate* and *ystate* when it is defined. These variables are not externally visible, since the *point* has the type:

$\mu\sigma$.{x: INTEGER, y: INTEGER,
  identity: σ, equal: $\sigma \rightarrow$ BOOLEAN,
  move: INTEGER $\times$ INTEGER $\rightarrow \sigma$}

Access to the hidden variables *xstate* and *ystate* is handled through the public methods *x()*, *y()* and modification is handled through the procedure *move()* which uses assignment to update them. Note how, in the body of *equal()*, it is possible to access *self*'s *xstate* directly (since it is in scope) but that the *other*'s *xstate* must be obtained through the access method, since *other* is a separate closure whose hidden variables are not seen from *self*. This provides a natural encapsulation in the style of *Smalltalk*, in which all access to variables must be controlled through methods [GR83], and *Eiffel*, in which access to the *Current* object's public attributes has the semantics of read-only functions [Meye88]. The encapsulation rules of *C++* presume to reveal *other*'s hidden state at the same time as *self*'s; this defies any simple analysis.

7.1.2 Operations on Variables

Here, a more concrete syntax for a minimal typed object-oriented language is gradually introduced. The language provides variables, which are amenable to assignment and inspection. When making variable declarations, identifiers precede type labels, which are prefixed by a colon in the Pascal style. Multiple identifiers of the same type may be separated using a comma, which is just considered a syntactic shorthand for the repeated longer form:

x, y : INTEGER; $\Leftrightarrow$ x : INTEGER; y : INTEGER;

All declarations are delimited with a semicolon, which has the force of the "dot" denoting the start of the body of a λ-abstraction:

```
( λ(x : INTEGER).λ(y : INTEGER).f(x, y)
    0 0)
```

in which variables are bound to default values, here expressed by the application to zero constants. The types of variables are bound throughout the block *f(x, y)* in which they occur; however their values are subject to rebinding through assignment.

Simple expressions are delimited with a semicolon and compound expressions are delimited using braces, one to indicate the start of a compound and another to act as a terminator:

```
{   x := 3;
    y := 4;          ⇔      (λa.λb.λc.c (x := 3) (y := 4) unit)
}
```

The semicolon is technically an expression-terminator rather than a separator. This comes from explaining a compound expression as a projection function of the form: λ*a*.λ*b*. ... λ*n.n*, an *n*-place expression binder that returns the last bound value, here represented by the empty element of the trivial UNIT type. The use of the semicolon is consistent with declaration, in that it introduces another λ-abstraction which consumes another value in sequence. All expressions have a value, including assignment, which returns the value assigned. Values may be used in nested expressions, or ignored by terminating with a semicolon. Compound expressions also have a value. This is either the empty value or the final expression in a sequence that is *not* terminated with a semicolon:

```
{   x := 3;
    y := 4           ⇔      (λa.λb.b (x := 3) (y := 4))
}
```

This semantics avoids having to introduce any special syntax for *return*-values from method expressions; witness the body of the *move()* method above which returns *self* and has the type of *self*.

7.1.3 Object Constructors

A rationale for the **let** ... **in** sugared syntax used above for binding state variables inside closures may be provided by observing that it is equivalent to the following λ-abstraction:

```
(λxstate.λystate.(Y λself.{x ↦ xstate, y ↦ ystate, identity ↦ self,
            equal ↦ λother.(xstate = other.x ∧ ystate = other.y),
            move ↦ λnx.λny.(xstate := nx; ystate := ny; self)}
    3 4))
```

which is also the form of *object construction*. To generate a particular *point* object, the free variables in the body of a generator must first be bound and then the fixpoint taken. A natural way to do this is to define *object constructor functions* which accept additional initialisation arguments and return the fixpoint of a generator. The initialisation variables must occur free in the body of the generator, as *xstate* and *ystate* do here. One way to achieve this is to extend a typed object generator to abstract over its initialisation arguments:

$\Phi\alpha$point : $\forall$(t $\subseteq$ ΦPOINT [t]).INTEGER $\times$ INTEGER $\rightarrow$ (t $\rightarrow$ ΦPOINT [t])

$\Phi\alpha$point = Λ(t $\subseteq$ ΦPOINT [t]).
  λ(xstate: INTEGER).λ(ystate: INTEGER).λ(self: t).
    {x $\mapsto$ xstate, y $\mapsto$ ystate, identity $\mapsto$ self,
      equal $\mapsto$ λ(other: t).
        (xstate = other.x $\wedge$ ystate = other.y),
      move $\mapsto$ λ(nx: INTEGER).λ(ny: INTEGER).
        (xstate := nx; ystate := ny; self)}

because this will allow the derivation of subclasses which also expect initialisation arguments. The constructor is then formed by taking a fixpoint internally:

newPoint : INTEGER $\times$ INTEGER $\rightarrow$ POINT

newPoint = λa.λb.(Υ ($\Phi\alpha$point [POINT] (a, b)))

  = λa.λb.μself.{x $\mapsto$ a, y $\mapsto$ b, identity $\mapsto$ self,
      equal $\mapsto$ λ(other: POINT).
        (a = other.x $\wedge$ b = other.y),
      move $\mapsto$ λ(nx: INTEGER).λ(ny: INTEGER).
        (a := nx; b := ny; self)}

Many constructors may be defined for each object type, accepting different numbers of arguments, as is deemed appropriate for initialisation.

7.2 Object Creation

Cook *et al.* note that object constructors and inheritance do not easily work together [CHC90, Harr91a]. These authors suppose each class has a single constructor, a *new()* method in the spirit of Smalltalk, which is strongly-typed in a fixed set of initialisation arguments. We might imagine the following types for constructors for OBJECTs and POINTs:

newObject : OBJECT

newPoint : INTEGER $\times$ INTEGER $\rightarrow$ POINT

This clearly interferes with the polymorphic operation of *new()*, for it is illegal to replace a function defined over one set of initialisation arguments by a function

defined over different (typically more) arguments. A polymorphic *new()* method is nonetheless desirable, since it is especially useful in expressions which clone objects, or which otherwise generate objects having the same dynamic type as self, such as functions which map from a collection to a collection.

7.2.1 Initialising Object State

There are two aspects to the problem. The first is to find a style of homogenous typing for initialisation arguments. [Harr91a] proposes to lump all initialisation arguments into a single record, whose type is parameterised by the type of *self*. If we define a type function describing the initialisation argument of a 2D *point*:

ΨPOINT = $\Lambda\sigma$.{x: INTEGER, y: INTEGER}

then the extended $\Phi\alpha$*point* generator now accepts a single record argument standing for the *state* to be encapsulated inside a *point* object:

$\Phi\alpha$point : $\forall$(t $\subseteq$ ΦPOINT [t]).ΨPOINT [t] $\rightarrow$ (t $\rightarrow$ ΦPOINT [t])

$\Phi\alpha$point = Λ(t $\subseteq$ ΦPOINT [t]). λ(state: ΨPOINT [t]).λ(self: t).
{x $\mapsto$ state.x, y $\mapsto$ state.y, identity $\mapsto$ self,
equal $\mapsto$ λ(other: t).
(state.x = other.x $\wedge$ state.y = other.y),
move $\mapsto$ λ(nx: INTEGER).λ(ny: INTEGER).
(state.x := nx; state.y := ny; self)}

This *state* argument is now amenable to adaptation through inheritance, since it is parameterised by *self*'s type (this approach may be extended to include other type parameters). We may define a type function describing a selectable *hot point*'s initialisation argument:

ΨHOTPOINT = $\Lambda\sigma$.{x: INTEGER, y: INTEGER, s: BOOLEAN}

and derive the extended $\Phi\alpha$*hotpoint* generator from the $\Phi\alpha$*point* generator:

$\Phi\alpha$hotpoint : $\forall$(t $\subseteq$ ΦHOTPOINT [t]).
ΨHOTPOINT [t] $\rightarrow$ (t $\rightarrow$ ΦHOTPOINT [t])

$\Phi\alpha$hotpoint = Λ(t $\subseteq$ ΦHOTPOINT [t]).λ(state: ΨHOTPOINT [t]).λ(self: t).
(λ(super: ΦPOINT [t]).
super $\oplus$ {selected $\mapsto$ state.s, select $\mapsto$ (state.s := true),
deselect $\mapsto$ (state.s := false),
equal $\mapsto$ λ(other: t).
(super.equal(other) $\wedge$ state.s = other.selected)}
($\Phi\alpha$point [t] (state, self)))

Here, the *super* record is constructed by applying the parent's typed generator to the child's type, *state* and *self*. For example, we might apply:

$\Phi\alpha$point [HOTPOINT] ({x $\mapsto$ 3, y $\mapsto$ 4, s $\mapsto$ false}, self)

This is legitimate, since the child's *state* is a subtype of that expected in the parent's typed generator:

$\forall$(t $\subseteq$ ΦHOTPOINT [t]).ΨHOTPOINT [t] $\subseteq$ ΨPOINT [t] $\Leftrightarrow$

{x: INTEGER, y: INTEGER, s: BOOLEAN} $\subseteq$ {x: INTEGER, y: INTEGER}

However, there is one outstanding problem with this approach. The taking of fixpoints interacts with the rebinding of *state* in the *super*-record. The *state* of a *hot point* is a different variable from the *state* of a *point*. If we understand a fixpoint to be equivalent to its infinite expansion:

a_point = μself.$\Phi\alpha$point [POINT] (state, self)

= ... $\Phi\alpha$point [POINT] (state, $\Phi\alpha$point [POINT] (state, ...))

then the construction of a *super*-record inside the recursion of *self* will lead to repeated copies of *super*'s *state* being built [Harr91a, p25-26]. To solve this, most of the *super*-record's instantiation may be pulled outside the recursion of *self*:

```
( λ(supergen: t → ΦPOINT [t]).λ(self: t).
    ( λ(super: ΦPOINT [t]).super ⊕ { ... }
    supergen (self))
Φαpoint [t] (state))
```

at the expense of abstracting over a type function *supergen*: t $\rightarrow$ ΦPOINT [t]. This constructs the *state* of the *super*-record once, such that subsequent unrollings of the inheritance construction only rebind *self* harmlessly.

This technique solves the problem of translating child initialisation arguments to the form expected by the parent; and therefore supports the derivation of subclass generators with initialisation arguments. However, it is also necessary to translate parent initialisation arguments to the form expected by the child, when parent constructors are applied polymorphically. [CHC90] supposes the existence of a translation function which would convert a polymorphic call of the form: *p := newPoint (3, 4)* into a call of the form: *p := newHotPoint (3, 4, false)*. The idea is that a child class will specify what default assumptions to make when its parent constructors are called. This imposes another layer of transformations upon the model which seem hard to justify. The possibility of general transformations would allow the arbitrary inference of missing arguments under polymorphic applications. For example, we might translate a *move()* in 2D space into an arbitrary *move()* in 3D space. Whereas applying a 2D translation to a 3D point seems reasonable, inferring an arbitrary z-displacement seems an unreasonably powerful mechanism.

7.2.2 State Templates

Instead, our model supports polymorphic object creation with default *state templates* for each class. It respects two kinds of object generator: one that accepts initialisation arguments, but may only be used in a monomorphic context, and one that accepts no initialisation arguments, but may be used polymorphically.

A generator can be given a *state template* by providing default initial values for each state variable; and an *init()* method may later be used to modify these:

ΦCIRCLE = $\Lambda\sigma$.{init1: REAL $\rightarrow \sigma$, radius: REAL, diameter: REAL,
circumference: REAL, area: REAL}

Φcircle : $\forall(t \subseteq \Phi\text{CIRCLE}[t]).t \rightarrow \Phi\text{CIRCLE}[t]$

Φcircle = **let** pi = 3.1415926 **in**
$\Lambda(t \subseteq \Phi\text{CIRCLE}[t])$.(**let** rad = 0.0 **in** λ(self: t).
{init1 $\mapsto \lambda$(r: REAL).(rad := r; self), radius $\mapsto$ rad,
diameter $\mapsto$ (2*r), circumference $\mapsto$ (2*pi*r),
area $\mapsto$ (pi*r*r)})

Here, *pi* is a class variable, visible to all instances of CIRCLE and its subclasses, because it is bound outside the class $\forall(t \subseteq \Phi\text{CIRCLE}[t])$. When the type closure ($\mathbf{Y}$ ΦCIRCLE) is formed, an environment containing *pi* is built. The state variable *rad* is an instance variable, since it is bound inside the class but outside the recursion of *self*. Each time the closure ($\mathbf{Y}$ Φcircle [CIRCLE]) is formed, a new state environment is constructed for each object.

We may inherit such descriptions without any need for special treatments of initialisation arguments:

ΦCYLINDER = $\Lambda\sigma$.{init1: REAL $\rightarrow \sigma$, init2: REAL $\times$ REAL $\rightarrow \sigma$,
radius: REAL, height : REAL, diameter: REAL,
circumference: REAL, area: REAL, volume: REAL}

Φcylinder : $\forall(t \subseteq \Phi\text{CYLINDER}[t]).t \rightarrow \Phi\text{CYLINDER}[t]$

Φcylinder = $\Lambda(t \subseteq \Phi\text{CYLINDER}[t])$.(**let** hgt = 0.0 **in** λ(self: t).
(λ(super: ΦCIRCLE [t]).
super $\oplus$ {height $\mapsto$ hgt, volume $\mapsto$ (super.area*hgt),
area $\mapsto$ (super.circumference*hgt + super.area*2),
init2 $\mapsto \lambda$(r: REAL).λ(h: REAL).(hgt := h; super.init1(r))}
(Φcircle [t] (self))))

because this time, the state of the *super*-record is only bound once and subsequent unrollings of the inheritance construction can only rebind *self*.

This style supports a natural encapsulation of state. Whereas the variable *pi* is bound outside the class $\forall(t \subseteq \Phi\text{CIRCLE}[t])$ and is therefore visible to all

instances of all types in this family, the variable *rad* is only visible within the object generator Φ*circle* and *hgt* likewise only within Φ*cylinder*. Note especially how inherited instance variables are not directly visible to subclasses, modelling the *private* declarations of *C*++ [Stro91]. This is because the *super*-record is formed inside the recursion of *self* in the child class. Inherited instance variables are only visible indirectly, within the scope of *super*-methods invoked in the combined record.

The model allows any number of user-defined initialisation functions, here simply called *init1()* and *init2()*, which operate on different numbers of parameters. There is no theoretical problem in invoking *init1()* to initialise the *circle*-part of *cylinder* objects, either from within the *init2()* method for *cylinders*, or simply to reset the radius of the *cylinder*. It might be more appropriate to choose standard names for these initialisation functions, such as *circle()* and *cylinder()*, to denote which parts of the object they initialise. We do not call these *constructor* functions, like those of *C*++, since they do not create objects; instead they re-initialise objects that already exist.

7.2.3 Polymorphic Object Creation

Separating object creation from initialisation effectively removes any difficulty associated with incompatible parameter lists. The second aspect of the problem has to do with the extra level of recursion introduced by allowing objects to contain their own constructors (see the discussion in chapter 3).

Polymorphic object creation is explained in [CHC90] in terms of the flexible use of fixpoints when constructing object generators. In this model, objects are deemed to encompass their own creation-functions. As well as abstracting over *self* and the *self*-type, they abstract over the generator as well. The need for this is made clearer by a flawed attempt to generalise a *clone()* function:

$$\Delta\text{CLONER} = \Lambda\sigma.\{\text{clone: } \sigma\}$$

$$\Delta\text{cloner} : \forall(t \subseteq \Delta\text{CLONER}\ [t]).t \rightarrow \Delta\text{CLONER}\ [t]$$

$$\Delta\text{cloner} = \Lambda(t \subseteq \Delta\text{CLONER}\ [t]).\lambda(\text{self: } t).\ \{\text{clone} \mapsto \text{newCloner}\}$$

In the Δ*cloner* abstract class, the *clone()* method is implemented using an object constructor in the style we have been using elsewhere; but it is immediately apparent that *newCloner()* involves unresolved recursion in the definition of the generator Δ*cloner* itself:

$$\text{newCloner} = (\mathbf{Y}\ (\Delta\text{cloner}\ [\text{CLONER}]))$$

Furthermore, when the *clone()* method is inherited, it will always create an object of exactly the type CLONER, rather than some inheriting type. The latter

problem could be fixed by parameterising *newCloner()* over the type *t*, but this would not solve the recursive definition of Δ*cloner*. So, we abstract at the point of recursion, which happens to be over the object generator itself:

$$\Gamma\text{cloner} : \forall(t \subseteq \Delta\text{CLONER}\ [t]).(t \to t) \to (t \to \Delta\text{CLONER}\ [t])$$

$$\Gamma\text{cloner} = \Lambda(t \subseteq \Delta\text{CLONER}\ [t]).\lambda(\text{selfgen}: t \to t).\lambda(\text{self}: t).$$
$$\{\text{clone} \mapsto (\mathbf{Y}\ (\mathbf{Y}\ \text{selfgen}))\}$$

Here, *selfgen* stands for some generator for a recursive object having the polymorphic type $\forall(t \subseteq \Delta\text{CLONER}\ [t]).t \to t$. In the body of the *clone()* method, the fixpoint of *selfgen* is used without complete knowledge of the final binding of *t*. This allows us to abstract over many different generators, whose fixpoints may be taken when it is known what type of recursive object is desired. The recursion of *selfgen* is independent of the recursion of *self*; and two applications of $\mathbf{Y}$ are needed to establish the object. ($\mathbf{Y}$ *selfgen*) fixes the generator-recursion in Γ*cloner* and establishes a typed object generator:

$$(\mathbf{Y}\ (\Gamma\text{cloner}\ [t])) : t \to \Delta\text{CLONER}\ [t]$$

$$= \lambda(\text{self}: t).\{\text{clone} \mapsto (\mathbf{Y}\ (\Delta\text{cloner}\ [t]))\}$$

and ($\mathbf{Y}$ (Δcloner [CLONER])) fixes the recursion in *self*, establishing an instance of precisely the type CLONER. The flexibility of this arrangement is demonstrated through the inheritance of class definitions which abstract over their generators (the technique is called *constructor inheritance* in [Harr91a], and *class inheritance* in [CHC90], in which *selfgen* is known as *myclass*):

$$\Phi\text{POINT} = \Lambda\sigma.\{\text{clone}: \sigma, \text{x}: \text{INTEGER}, \text{y}: \text{INTEGER}\}$$

$$\Gamma\text{point} : \forall(t \subseteq \Phi\text{POINT}\ [t]).(t \to t) \to (t \to \Phi\text{POINT}\ [t])$$

$$\Gamma\text{point} = \Lambda(t \subseteq \Phi\text{POINT}\ [t]).\lambda(\text{selfgen}: t \to t).\lambda(\text{self}: t).$$
$$(\Gamma\text{cloner}\ [t]\ (\text{selfgen})\ (\text{self}))$$
$$\oplus\ \{\text{x} \mapsto 0, \text{y} \mapsto 0\}$$

Here, Γ*point* is a definition for a class of *self-cloning points* with *x* and *y* fields. The inheritance construction distributes the new generator argument *selfgen* to the old class definition, along with *self* and the *self*-type, such that the inherited *clone()* method is automatically redirected to create instances of the subclass. This technique can be mixed with the default state template and initialisation argument ideas by binding a *state* record outside *self*. The resulting *selfgen* has the type $(\alpha \to (t \to t))$ in the latter case, since we abstract over a generator expecting an initialisation argument of type α.

7.2.4 Sharing Responsibility for Creation

Every variable of type OBJ created by fixing a generator OBJ = (Υ ΦOBJ) has a default initialisation value *obj* = (Υ (Φ*obj* [OBJ])) created by fixing the typed object generator. It is therefore tempting to think of object creation as an external activity performed by a compiler having knowledge of the constructors for objects of different types. Polymorphic functions of the form:

identity: $\forall(\tau \subseteq \Phi\text{POINT}\ [\tau]).\tau \rightarrow (\tau)$
move: $\forall(\tau \subseteq \Phi\text{POINT}\ [\tau]).\tau \rightarrow (\text{INTEGER} \times \text{INTEGER} \rightarrow \tau)$

have their concrete result-types resolved by a global process of parameter instantiation. A compiler which knows how to resolve types may also give unambiguous initialisation values to variables with static types:

p : POINT; $\Leftrightarrow$ (λ(p: POINT).f(p)
f(p); point)

where point = (Υ Φpoint [POINT])

For this model of creation to work, we must assume a complete global map from fixpoint types to fixpoint values. In cases of dynamic typing and binding, the compiler creates a table of initialisation values to use, dependent on the type tag received at the site at run-time. This is no more difficult or unusual than selecting a dynamically-bound function.

Whereas in a pure functional model, it is natural to think of objects invoking their own constructors (recursively), the closer we come to implementation, the less obvious this appears. *Smalltalk* supports the *new()* method, not in instance-objects, but in class-objects whose protocols are described in metaclasses [GR83]. So, polymorphic *new()* is not in the object interface. *Eiffel* avoids having to deal with the dynamic semantics of expressions such as *Smalltalk*'s *self class new* by having declarations of the form: *x : like Current;* which capture the same dynamic intent, but are resolved externally by the type system. *Eiffel*'s *Create()* function [Meye88] always was a misnomer, performing only *initialisation* after the compiler had allocated space for the object concerned.

Many operations that would require the creation of a new object in a functional model simply update and return *self*, such as the *move()* method invoked on a *point* instance. In other cases, a method will return one or other of its input arguments, or a sub-object of one of these. Whether a new object is actually created or not depends on the particular *value* or *reference semantics* adopted. While all instances of a class behave in the same way at a certain level of abstraction, the general system mechanisms of memory-allocation, assignment and parameter binding may cause two instances to have quite different semantics. It is not that allocation, assignment or binding are operations owned by a particular class; nor is it that every class should have duplicate sets of operations for value and reference arguments. Rather, these global

mechanisms are orthogonal to the behavioural model for objects; and they are pervasive in their effect.

It is impossible to dismiss all object creation, since there are cases when a method *must* return a new object. However, we view object creation and management as a shared responsibility between the system and the object model. To handle this collaboration, we presume that any implementation will supply basic system operations. Candidates for these primitives are illustrated in figure 7.1. Assignment, aliasing and memory-management are dealt with later. As a first requirement, all variables should be properly initialised. Both assignment and initialisation copy the contents of one object to another. This we imagine being handled at a primitive level by the system, using *state()* to extract hidden state and *init()* to install it elsewhere.

init(s : State)	initialise hidden state from a supplied record
state() : State	reveal hidden state, for copying purposes only
copy(o : Object)	copy hidden state from a supplied object
assign(o : Object)	assign hidden state from a supplied object
alias(o : Object)	make *self* refer to a supplied object
create()	allocate dynamic memory, make *self* refer to it
destroy()	deallocate dynamic memory, make *self* void
scope() : Scope	reveal manner of storage used for *self*
type() : Type	reveal type of *self*

Figure 7.1: Language Primitives

Primitive initialisation may be given a rationale in the object model. Henceforward, it is assumed that each object generator Φobj is created by fixing a typed object definition that abstracts over its own generator Γobj. Such generators may inherit cloning expressions ($\mathbf{Y}$ ($\mathbf{Y}$ selfgen)) in methods which resolve to expressions ($\mathbf{Y}$ (Φobj [t])) to create new objects like *self*. In the context of simple and polymorphic type declarations:

o : OBJ; p : $\forall(\tau \subseteq \Phi$OBJ [$\tau$]);

object initialisation may be modelled as creating a new default instance of the required type, using the state template hidden in the closure Φobj. In the case of polymorphic initialisation, taking fixpoints is delayed until the precise object generator is known and therefore the appropriate default state will be used. It is further assumed that each extended object generator that expects an initialisation argument $\Phi\alpha obj$ is created by fixing a typed object definition $\Gamma\alpha obj$ that abstracts over its own generator and a state argument. Such generators inherit cloning expressions ($\mathbf{Y}$ ($\mathbf{Y}$ selfgen (state))) in methods, which resolve to expressions ($\mathbf{Y}$ ($\Phi\alpha obj$ [t]) (state)) to create new objects like *self*, but having a different state. In the context of simple type declarations:

o : OBJ (a, b); $\Leftrightarrow$ (λ(o: OBJ).f(o)
f(o); ($\mathbf{Y}$ ($\Phi\alpha obj$ [OBJ]) {v1 $\mapsto$ a, v2 $\mapsto$ b}))

initialising an object by supplying its state may be modelled as creating a new object with a record standing for its state. This strategy cannot be used safely in a polymorphic context, because our model does not allow the inference of arbitrary missing initialisation arguments.

Primitive state access may be given a rationale in the object model. A primitive method:

state : ΨOBJ [τ] state $\mapsto$ objstate

is inherited by all objects. It is polymorphic in the type of *self*, allowing overriding definitions. It returns object state by value, such that copy initialisation:

o : OBJ (p); $\Leftrightarrow$ (λ(o: OBJ).f(o)
f(o); ($\mathbf{Y}$ ($\Phi\alpha$*obj* [OBJ]) p.state))

and assignment do not affect the state of the copied object. In practice, this theoretical model is transformed into the more prosaic: *o.init(p.state)*, since it is easier to think in terms of system primitives acting on the underlying storage.

7.3 Object Identity

The notion of object identity stems from the unique states of free variables when a lexical closure is formed. A function *f()* may be defined in different binding environments and so behave differently to the extent that its result depends on the values of free variables. It then becomes a salient concern *which* function is being manipulated. Outwardly, the two closures seem identical, yet they are distinct. This is even more relevant when assignment is added to the functional model. The same closure may behave differently over time, as a result of changes in its encapsulated state. Two closures defined in the *same* binding environment must now be considered distinct, since their states may diverge.

Value and Reference Semantics

The concept of object identity complicates the semantics of assignment and argument passing. In value-oriented languages, it is immaterial whether a reference to an object, or a copy of the object is taken, since computation does not depend on identity. However, when computation depends on an interaction between a group of specific objects, it is important to affect the objects concerned (and not copies). To achieve this, objects must be passed by reference. This is typically handled by the copying of pointers. Dereferencing a copied pointer inside a method accesses the same object as that obtained through the original pointer outside the method. Basic types, such as integers, often masquerade as objects for the sake of uniformity; however, they are usually passed by value, since this is more efficient. The minor deception is

safe, since it is never the case that you want to update the state of an integer (in the sense that 2 can never become 3).

The policy on value and reference semantics is neither uniform nor especially clear in existing object-oriented languages. In *Smalltalk* and *Eiffel* version 2.x [GR83, Meye88], all values are assumed by default to be references to objects and the separate treatment of simple types is either implicit (*Smalltalk*) or given a short semantic gloss (*Eiffel*). This means that assignment is, in general, pointer copying with a reference semantics; although pointers are not explicit in the syntax of these two languages which hide such details. In order to obtain a true copy of an object, a copy function must be invoked explicitly for all non-basic types. The function may be external and global, like *Eiffel*'s *Clone()*, or internal, relying on metaclass behaviour to request allocation, like *Smalltalk*'s *copy* which calls *self class new*. The resulting copy may be shallow or deep. Against this, a shallow copy of a basic type may be obtained by simple assignment, which in this context has a value semantics.

Hiding the semantics of assignment and binding reduces syntactic complexity, but may lead to unwanted surprises. The result of an *Eiffel* access function looks the same whether it is a value or a reference; however manipulating a returned reference may accidentally break the encapsulation of the object from which it was accessed. To avoid such unintended confusions, *C*++ has distinct syntactic styles for value and reference arguments, supporting copy and reference assignment [ES90, Stro91]. In addition, explicit pointer manipulation is available at a lower level. By default, arguments are passed and returned by value. Ordinary global functions take and yield copies. Methods[1] are an exception, invoked with *self*[2] passed by reference and other arguments passed by value. This is so that updates to *self*'s state variables are not lost when the method terminates. To achieve true inter-object communication, other arguments should also be passed by reference, or else pointers may be used. By default, the system takes a shallow copy whenever pass-by-value is mandated. For non-basic types, a user-defined copy function may be supplied, which may choose to take a deep copy, or alternatively, augment the shallow copy mechanism with reference counting for a deallocator.

Reference and value semantics also affect the layout of data structures in memory. The components of a *C*++ object are either objects or pointers to objects. An object that is wholly contained by value is always distinct; it may be the target of copy assignment, but not reference assignment. To implement object sharing a reference or pointer must be used. Reference assignment is typically handled by copying pointers; or else a pointer may take the address of a value allocated elsewhere. *Eiffel* version 3.x [Meye92] provides optional inline expansion for non-basic types (*ie* to override the default reference semantics) and optional wrapped basic types (*ie* to override the default value semantics), to give the fullest possible range of containment options.

---

[1] Methods are called *member functions* in *C*++.

[2] *Self* is known as **this* in *C*++; *this* is a pointer to *self*.

7.3.2 Constraints on Binding

Various concerns compete for attention when considering how to design value and reference semantics into a language. Should an object-oriented language have exclusively one or the other, or a mixture? Should the distinctions be explicit or implicit? We would prefer an economical syntax like *Smalltalk* and *Eiffel*, but cannot contemplate unpredictable semantics. The overt style of *C++* is initially attractive, but ultimately confusing with both pointers and reference variables and no constraints on their combination. We should like especially to eliminate the notion of explicit pointers as first-class values.

It is worth examining boundary conditions on possible designs. First and foremost, it is essential to preserve the identity of objects in constructions where identity matters. So, for example, a method should always bind *self* by reference, because it may modify that object's internal data. Again, where a collaboration is set up through mutual message-passing between a group of specific objects, the secondary participants should normally appear as reference arguments to the method, which provides the context for the collaboration.

A less critical situation is where an object temporarily inspects another object's state, without wishing to retain this information. Here, the second object may feasibly be passed by value, since the outcome of the inspection does not depend on its identity, only on its state. In certain circumstances, the semantic distinction is lost: a large class of basic values are self-identifying, in the sense that they are uniquely identified by their immutable state. These include the integers, the reals and we may choose to extend uniqueness to complex numbers and fractions. Self-identifying objects may only be passed by value without loss of semantics because they cannot be updated.

A second set of considerations concern computability and efficiency. While it would be possible (as in *Smalltalk*) to build all structures using pointers and bind all arguments by reference, this brings space and time penalties. Structures consisting of pointers must still allocate their data elsewhere; this also requires extra dereferencing operations to access the primary data. At the other extreme, the overhead of frequently copying large objects by value onto the execution stack during method invocation should also be avoided. An optimum design for minimising storage and maximising the speed of stack frame copying might then be to have objects contain their structural components by value and bind all method arguments by reference, taking the addresses of offsets into objects. This naïve strategy breaks down in the face of dynamic data structures and local variables. Flexible data structures defeat a unilateral policy of containment by value, since they require dynamically allocated storage. Methods may not return the address of local variables, since this is tantamount to passing a dangling pointer to storage that has gone out of scope.

7.3.3 Alias Types

An acceptable compromise is to allow both values and references in the language. However, it should seek to restrict the use of each style such that clarity, integrity and efficiency are preserved, according to the above considerations.

An explicit *alias* mechanism is introduced, whereby variables may refer to storage allocated elsewhere. Our scheme intends to hide some of the complexity of pointers in conventional languages and at the same time prohibit unsafe use of aliases. It is also important, from a type-theoretic viewpoint, that although aliasing affects the language semantics at the level of individual objects, it should not affect the language semantics at the level of types and classes; thus *ordinary types* and *alias types* are considered behaviourally equivalent, except in the way their objects are created or assigned.

Syntactically, an alias variable is indicated by prepending @ to its type identifier in a declaration:

```
p1, p2 : POINT;        ...actual point objects
p3, p4 : @POINT;       ...aliases for points
```

Ordinary variables have := copy assignment and alias variables have @= reference assignment; otherwise they are both accessed using the same notation. In addition, alias variables may allocate and deallocate storage on the heap using the primitive instructions *create* and *destroy*. These are styled to look like methods, although they are really basic system operations. The following gives a flavour:

```
p1.move(3, 4);
p2 := p1;              ...copy assignment
p3 @= p1;              ...reference assignment

p2.move(1, 3);         ...p1 unchanged
p3.move(2, 5);         ...p1 also changed

p4.create;             ...dynamic allocation
p4.move(1, 0);
p4.destroy;            ...deallocated
```

Note how this style avoids explicit pointer manipulation. The reference assignment operator @= must have an alias variable as its target, but may otherwise accept alias and ordinary variables at the source:

```
p3 @= p1;              ...address of p1 taken
p4 @= p3;              ...pointer p3 copied
```

No confusing syntax is needed for address extraction or pointer dereferencing. The latter is also illustrated by the homogenous method invocation style, which ensures that *self* is passed by reference:

```
p1.move(3, 4);          ...self bound to address of p1
p3.move(2, 5);          ...pointer p3 copied into self
```

Other method arguments may be passed by value or reference. Formal alias arguments may bind to ordinary or alias variables; in the former case an address is extracted and in the latter case a pointer is copied. Ordinary formal arguments force copies to be taken, even of alias variables, whose dereferenced contents are copied, completing the symmetry.

Variables must be initialised to sensible values, since non-initialised variables are undefined and may contain garbage, leading to unpredictable system behaviour. The syntax of declaration is extended to include copy and reference initialisation to the *whole state* of another object:

```
p1 : POINT;             ...p1 initialised to default point
p2 : POINT (p1);        ...p2 initialised to copy of p1

p3 : @POINT;            ...p3 initialised to void reference
p4 : @POINT (p1);       ...p4 initialised to alias p1
```

which is deliberately distinct from the syntax of assignment. This style of declaration has a natural interpretation in the λ-calculus:

```
(λ(p1: POINT).
    (λ(p2 : POINT).f(p1, p2)
     (Y (Φα obj [POINT]) p1.state))     ...copy p1's state
 (Y (Φobj [POINT])))                    ...default state
```

The rules governing the initialisation of variables are:

- ordinary variables are initialised to the default object template, or to a copy of another object that is in scope, or to a completely specified state template;
- alias variables are initialised to the void reference, or to the address of a static object that is in scope, or to the address of a dynamic object allocated on the heap.

Once initialised, variables are protected from unauthorised kinds of modification. It is an error to apply @= to a non-alias variable, since this has no meaning; likewise it is an error to apply := to an alias variable, since this would permit the remote copying of objects through aliases. The rules governing assignment concern only the target:

- only variables, not expressions, are legal targets of assignment;
- ordinary variables have value semantics with copy assignment;
- alias variables have reference semantics with reference assignment.

Apart from this, it is legal to have an ordinary variable, an alias variable and even an expression as the source of an assignment. Expressions returning a value have unnamed storage reserved for them at the call-site by the compiler; this storage has the same status as a local variable in rules governing scope and aliasing. The scoping rules for variables are:

- storage for an ordinary variable is reclaimed when it goes out of scope;
- no primary storage is reclaimed when an alias variable goes out of scope;
- no alias variable may be passed outside the scope of any object it aliases.

The second rule is appropriate since an alias variable may refer to an object which remains in scope when the alias goes out of scope. Dynamically allocated data must be explicitly deallocated; however, there is a mechanism for making this semi-automatic, described below.

Assignment, initialisation and memory management are considered primitive system operations. Here it is even more apparent that creation is controlled as much by the caller as by the provider of the service, since the client code often decides how to allocate storage, not the object itself. The system model described here has nine operations (assign, alias, create, destroy, copy, init, state, scope, type), but we can conceive of extensions to handle *futures* for parallel objects and *proxies* for distributed or persistent objects. Like reference and value semantics, these properties are orthogonal to type classification.

7.3.4 Aliasing and Protection

Within the scope of a method, the assignment rules offer a degree of protection to arguments passed by reference. In particular, it is impossible to change the state of such a variable using := (but it may be possible to make it alias a different object using @=). The only thing that can be done with alias method arguments is to invoke further methods on them.

It is clear that aliased objects cannot be updated remotely by brute force:

```
modify1(p, q : @POINT) : @POINT
{   p := q                          ...error:  := applied to alias
}
```

and the reference assignment permitted on alias variables simply results in a transfer of the alias:

```
modify2(p, q : @POINT) : @POINT
{   p @= q                          ...lose handle on p object and
}                                   ...gain extra handle on q object
```

The rule prohibiting an expression from being on the left-hand side of assignment ensures that objects may not be updated remotely:

```
modify3(p, q : @POINT) : @POINT
{   p.x := q.x;                    ...error:  invalid LHS
    p.y := q.y;                    ...error:  invalid LHS
    p
}
```

and attempting to get around this using local alias variables to shadow components of objects also fails when values are transferred:

```
modify4(p, q : @POINT) : @POINT
{   x : @INTEGER (p.x);            ...x alias for p.x
    y : @INTEGER (p.y);            ...y alias for p.y
    x := q.x;                      ...error:  := applied to alias
    y := q.y;                      ...error:  := applied to alias
    p
}
```

In fact, the only way to modify the point *p* is to invoke one of its own updating methods:

```
modify5(p, q : @POINT) : @POINT
{   p.move(q.x, q.y)               ...move() returns self
}
```

In the calling context of a method, our rules also ensure that the result has adequate protection. Where an access method aims to return a specific object, rather than a copy, this should be returned by reference. Where a method creates a new object, this may be returned by value if its allocation is static, or by reference if its allocation is dynamic.

The simplest way to regulate encapsulation is by a contract between the participating objects. The *supplier* object provides access services and the *client* object uses these [Meye88]. If the supplier wishes to expose one of its internal components and the client wishes to create an alias for this component, then it may be manipulated outside the supplier. Consider a moveable CIRCLE object whose method *centre()* exposes its origin-point:

```
centre : @POINT                    ...result is an alias for the
{   centre                         ...centre instance variable
}
```

This origin-point can either be copied or aliased in any client code:

```
{   p : POINT;                     ...new point object
    c : CIRCLE;
    p := c.centre;                 ...copy taken using :=
    p.move(3, 4);                  ...old centre unaffected
}
```

```
{   p : @POINT;                    ...alias for existing point
    c : CIRCLE;
    p @= c.centre;                 ...centre aliased using @=
    p.move(3, 4);                  ...old centre also affected
}
```

This style provides a clear indication of the programmer's intentions in the client code. It improves on the surprise-factor present in *Smalltalk* and *Eiffel*, since it forces the programmer to think whether the *centre* point itself or a copy is to be extracted. It has the possible disadvantage that some decisions about protection rest with the client code. As an alternative, the supplier could enforce full protection by offering only copies of its internal components:

```
centre : POINT                     ...result is a copy of the
{   centre                         ...centre instance variable
}
```

This directs the compiler to create a return buffer for the method *centre()* at the call site. When *centre()* is executed, a copy of the instance variable *centre* is placed in the return buffer. In this case, an expression of the form:

```
p : POINT;
p := c.centre;                     ...p takes a copy of return buffer
```

will result in a twofold copy of the *centre* instance variable - once into the call-site buffer as the function returns and once into the variable *p* as a result of copy assignment. To prevent duplicate copying, it is legal to use an alias variable in the client code:

```
p : @POINT;
p @= c.centre;                     ...p is alias for return buffer
```

This makes *p* an alias for the return buffer. A compiler may treat the return buffer in exactly the same way as a local variable defined in the calling context of *centre()*. The restriction imposed by the third scoping rule ensures that neither *p* nor any alias for *p* may be passed beyond the scope in which the buffer exists. An example which violates this rule is the method:

```
centre : @POINT                    ...error:  result is an alias for
{   p : POINT;                     ...local variable p, which
    p := centre                    ...takes a copy of the centre
}
```

since this eliminates the return buffer and seeks to pass an alias for local variable storage which has gone out of scope. By the same token, alias variables which are bound to value-expressions may only be used in the same scope as the buffers holding the results of the expressions.

It is often the case that both protected and unprotected access to object components is desired. In this case, it is not appropriate to provide two sets of

access methods, since exposing components in one set cancels the protection offered by the second set. Here, it is more economic to define just one set of access methods and leave the responsibility for protection with the client. This is no less secure than the *C++* policy of offering duplicate sets of functions with and without *const* protection [Meys92, 73-78]. Since functions are dispatched on the types of their arguments, *const* functions will execute only if the target object has a *const* type. This decision rests with the client code.

7.3.5 Memory Management

Object creation and destruction is viewed as a process managed jointly by the system model and the object model. The collaboration is a two-way affair. So far, it has been emphasised how a method may generate a new object; yet it is the calling context that determines how the object is allocated. The allocation at the call site may be static, dynamic or an alias for storage held elsewhere. Now, the opposite case is considered. When the system wishes to initialise or reclaim an object in accordance with the scoping rules defined above, it must make certain requests of the object model.

It is common for objects to contain references to dynamic data. In such cases, it is clear that the system must be informed explicitly how to handle creation, copying and destruction. This is because some objects, such as the LIST cells that build linked lists, are naturally created with a void reference to the list tail, whereas other objects, such as a growing and shrinking STACK, require their dynamic data to be created and initialised at the same time as themselves. When such objects are copied by assignment, their dynamic data is typically copied as well; in this case their dynamic data should also be reclaimed when they go out of scope. To handle these special cases, the system operations are made available as methods with names like *assign()*, *alias(), create()* and *destroy()*, which may be redefined appropriately in any class. The system will use redefined methods in preference to the default operations; however it is most usual to augment the default operations through method combination.

A simple terminating list made up of LIST cells does not require any special *create()* method, since the default system *create()* will initialise the tail to a void reference. When a static variable *m* : LIST goes out of scope, or when an alias *n* : @LIST for a dynamically allocated list is explicitly deallocated with *n.destroy*, this calls LIST's own deallocation method:

```
destroy
{   if (tail.scope)                    ...if tail is not a void reference
    { heap  :  tail.destroy;           ...recursive dynamic deallocation
    }
    super.destroy;                     ...system supplied deallocation
}
```

to clean up the tail of the list. Similarly, the following method will correctly handle recursive cell copying arising through copy assignment or initialisation:

```
copy (other : @LIST)                    ...alias argument, to avoid recursion
{   super.copy(other);                  ...system supplied shallow copy
    if (other.tail.scope)
    { stack, heap  :  tail.create.copy(other.tail);
    }                                   ...recursive allocate and initialise
}
```

on the assumption that *assign()* checks for self-assignment, then calls *copy()*, which by default takes a shallow copy. The *copy()* method should be used carefully, since it replaces *self*'s state. Our preferred strategy is to define *copy()* methods but let the system decide when to use them. Clearly, there are many possible memory management strategies. Although we tend to favour styles which preserve the default deep copy semantics of := it is nonetheless possible to provide a reference counting version of *assign()* which increments the count while taking a shallow copy and a variant of *destroy()* which decrements the count and deallocates the object when this reaches zero.

7.3.6 In Support of Objects

A large number of theoretical and practical issues have been covered relating to object creation, identity and state. Initially, the focus was on providing a theoretical model of object state and state initialisation. Subsequently, the fact that objects might encapsulate their own constructor functions led to a discussion of constructor inheritance. Object creation has an impact on the underlying storage and memory management strategy used by a practical programming system. Progressively, elements of a concrete language syntax have been introduced and related to the theoretical model. A key new aspect supported by the concrete language is the correct handling of object identity and state through a set of aliasing rules. In the following chapter, the syntax of our object-oriented language is developed further to include type and class definitions.