Chapter 8

A Language with Class

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

The higher-order parametric mechanisms required by our theory are translated into a more user-friendly syntax, in which much of the formal model is hidden, but important aspects of polymorphism are revealed. Our language captures both the recursive type and recursive implementation of extensible classes. Formal equivalences are given for the concrete syntax. Finally, the treatment of polymorphism is extended to support objects with the same polymorphic type as self but having a different parameterisation. The language is illustrated with examples of single and multiple inheritance, homogeneous and heterogeneous polymorphism.

8.1 Class Designs

This chapter describes how to implement our theory of classification in a concrete language syntax. The main improvement over existing object-oriented languages is in distinguishing *simple types* from *type functions*, wherever they occur. The difference is indicated in our syntax by the presence or absence of brackets [ ] denoting *type application* wherever this occurs. Explicit F-bounded *type parameters* are provided for all polymorphic types, including the *self*-type.

In our language, new data types are typically introduced by defining classes. This is because a class is always amenable to extension through inheritance and this is expected to be the preferred design option, in accordance with Meyer's open/closed principle [Meye88, 23-25]. Furthermore, nothing is lost by introducing classes rather than types, since a monomorphic type may always be inferred from the generator bounding a class.

8.1.1 A Syntax for Classes

Figure 8.1 sketches a standard #OBJECT[ ] class, useful as the root of any inheritance hierarchy. One could imagine providing many more methods shadowing all basic system operations in this class.

The keyword **class** introduces a type function used to restrict a type parameter in the manner of an F-bound. As a textual convention, we write type functions in upper case, followed by brackets [ ] to distinguish them from simple types; and distinguish class generators with a preceding #. Thus, #OBJECT[ ] is the concrete syntax for a generator ΦOBJECT; and OBJECT is the corresponding simple type, understood to be the fixpoint ($\mathbf{Y}$ ΦOBJECT) of the generator. The shape of the type function #OBJECT[ ] is established by compiling the type signature of the class body.

```
class #OBJECT [O] uses BOOLEAN
(self : @O) is
{  public
   identity : @O { self }
   equal (other : @O) : BOOLEAN { self = other }
   identical (other : @O) : BOOLEAN { self == other }
   ...
}
```

Figure 8.1: Definition of an Object Class

The **class** clause also serves the dual purpose of introducing the F-bound quantifying over all types in the class. It should be immediately obvious how this concrete syntax translates into the theoretical model of chapter 5:

class #OBJECT [O] ... (self : @O) **is** {...}

$\Leftrightarrow$ $\forall(\tau \subseteq \Phi\text{OBJECT}\ [\tau]).\lambda(\text{self}: \tau).\{...\}$

The brackets introduce a parameter O, which is understood to range over the type family $\forall(\tau \subseteq \Phi\text{OBJECT}\ [\tau])$. The type parameter O is used to type the explicit class recursion variable (*self* : @O); and occurs freely in the class body to type other arguments in the *self*-type, such as *other* in the *equal()* method. The parametric typing expresses exactly the polymorphism allowed by these methods: O must always be instantiated uniformly, ensuring that *equal()* compares objects having the same type. As a textual convention, we write type parameters as single upper-case letters. As a matter of style, the initial letter of the class is often chosen to refer to the *self*-type.

The **uses** keyword introduces further types and classes on which the class definition #OBJECT[ ] depends. This information is useful to a compiler, which may then process information in a suitable order.

The keyword **is** introduces the body of the class definition, which is delimited using braces {}. The keyword **public** introduces a set of exported methods. Formally, **public** indicates that the rest of the class body is to be treated as a record of public functions, the body of a typed object generator. The types and implementations of methods are introduced together, though they may be considered apart for formal purposes. The type signature of a method is given in the Eiffel style [Meye92], with names and types of additional arguments (if any) in parentheses and the result type (if any) following a colon. The use of empty parentheses is not required for unary methods, nor is the UNIT type notated explicitly either as an argument or result. It is assumed that formal treatments can infer these. The body of each method is considered a compound expression and therefore delimited using braces {}.

The typing semantics offered by types and parameters mixes freely with the object semantics offered by value and reference variables. The methods accept alias arguments having the same type @O as *self*. This is partly for the sake of semantics - *identity()* should return the object itself, not a copy; *identical()* should compare references to objects, not copies - and partly for efficiency - *equal()* should take a reference to *self* and *other* before comparing their states, not wastefully copying *other* first. The existence of primitive state and reference comparison operations is assumed, denoted here by "=" and "==". The language could require a set of primitives to be supplied by each implementation [GR83], or else provide an open-ended external calling syntax [Meye92].

8.1.2 A Syntax for Inheritance

Class inheritance requires a little more subtlety in interpreting the concrete style of declaration. The main concern is to ensure that all type-elements of the definition are properly introduced; and then that all object-elements are appropriately typed and bound. Figure 8.2 illustrates a simple #POINT[ ] class.

```
class #POINT [P] uses INTEGER, BOOLEAN
(self : @P) is (super : #OBJECT [P]) with
{  private
    x, y : INTEGER;
   public
    x : INTEGER { x }
    y : INTEGER { y }
    equal (other : @P) : BOOLEAN
    {          x = other.x & y = other.y
    }
    move (nx, ny : INTEGER)
    {          x := nx;  y := ny;
    }
}
```

Figure 8.2: Definition of a Point Class

Again, the keyword **class** declares a type generator #POINT[ ] which is used to constrain a type parameter P; and again #POINT [P] is understood to introduce an F-bound $\forall(\tau \subseteq \Phi\text{POINT}\ [\tau])$ quantifying over all types in the class. In the context of the **class** ... **uses** type introduction clause, concrete syntax like #POINT [P] is always interpreted as *introducing* a new polymorphic type P. Elsewhere in the object definition, concrete syntax like #POINT [P] is interpreted differently as a *type application*. This is an important distinction.

The syntax: (*self* : @P) **is** (*super* : #OBJECT [P]) **with** {...} indicates that the new class is an extension of an existing class. The keyword **is** may be followed either by a complete record, or by an argument representing a *super* record to be combined with an extension record following the **with** keyword, which functions exactly like the record combination operator $\oplus$ defined in chapters 4 and 5. Although this concrete syntax bears a resemblance to the λ-calculus model, much of the explicit redistribution of parameters and recursion variables present in the formal model has in fact been removed, since this can be reconstructed automatically. The following expansion is assumed:

(self : @P) **is** (super : #OBJECT [P]) **with** {...} $\Leftrightarrow$

λ(self : τ).(λ(super : ΦOBJECT [τ]).super $\oplus$ {...}
(Φobject [τ] (self)))

in which *super* is automatically bound to the result of distributing P and *self* to the parent's typed object generator. The scope of identifiers such as *self* and P is of course local to a declaration; it is assumed that suitable α-conversion rules will avoid the unexpected aliasing of parameter names in type-function applications. A compiler might count repeated introductions of the same type parameter, automatically appending an integer index to distinguish the new occurrence.

Again, the **uses** keyword lists several types, separated by commas, on which the class depends. Technically, **uses** is only required to list the types on which the *extension record* depends, rather than the whole *class*. It is assumed that the full set of component types can be recovered by unrolling inheritance. Likewise, the eventual shape of the type function #POINT[ ] is only known after the compiler has assembled it from the interface of all ancestor classes.

8.1.3 A Syntax for Encapsulation

In the body of the #POINT[ ] class definition, there are **private** and **public** sections. The keyword **private** has the semantics of the **let** ... **in** construction that was used in chapter 7 to bind closures over state variables. The keyword **public** has the force of introducing the record of functions characterising the behaviour of the class. Although the concrete syntax includes state variables along with methods in the class body, for formal purposes we always assume the automatic lifting of **private** state declarations to *outside* of the scope of the recursion variable, *self*. This presumes the following automatic translation:

```
(self : @P) is (super : #OBJECT [P]) with
{  private
     x, y : INTEGER;
   public
     x : INTEGER { x }
     y : INTEGER { y }
... }                                        ⇔

let (state : {x : INTEGER, y : INTEGER}) = {x ↦ 0, y ↦ 0} in
    λ(self : τ).( λ(super : ΦOBJECT [τ] ).
                                  super ⊕ {x ↦ state.x, y ↦ state.y, ...}
                  (Φobject [τ] (self)) )
```

Public access to the state variables: *x* and *y* is provided through methods having the same names: *x()* and *y()*. The two are not confused in normal usage, since methods are always invoked against an object: *self.x*, whereas state variables are accessed by name: *x*, as in Smalltalk [GR83]. The psychological advantage of identifying named storage with its access function outweighs the burden of supplying a public access function for each visible value. We model state variables as fields of a *state*-record, whose type is parameterised by the *self*-type. Occurrences of state variable names: *x* are formally interpreted as private record access: *state.x* and therefore not confused with methods.

The state variables: *x* and *y* may only be modified by invoking the *move()* method. Formally, any object state declared using **private** is not visible externally. In any case, our assignment rules prohibit assignment to an expression: for some *p* : POINT, *p.x* := 3 is always illegal. It therefore serves no purpose to declare state variables in the **public** part of a class declaration. Methods like *move()* are always necessary to update the state of an object. This has certain efficiency considerations which are discussed in [SLN94] and touched on in chapter 10. A compiler should be permitted to inline calls of the form: *p.move(3,4)* automatically in all situations where safe inlining is possible.

8.1.4 Supporting Objects and Values

A bone of contention in existing languages is the mismatch between "object types" and "value types" (see chapter 7). Our language provides a cleaner integration between simple values and objects, since both value and reference semantics are supported for all kinds of object. An object may retain value-status by not supplying any update methods.

Figure 8.3 illustrates a #COMPLEX[ ] class whose methods have been designed carefully to preserve the state of *self*; yet still operate efficiently. State variables *real* and *imag* translate into aliases for storage in *self*. However, the signatures of the access methods *real()* and *imag()* specify a return-by-value semantics, causing the content of a state variable to be copied once at the call-site. Arguments to #COMPLEX[ ]'s other methods are passed in by reference,

since it is more efficient to copy one pointer than storage for two REALs for each #COMPLEX[ ] instance. However, the state of these arguments may not be modified remotely by brute force as a consequence of our aliasing rules.

```
class #COMPLEX [C] uses REAL
(self : @C)  is (super : #NUMBER [C]) with
{  private
    real, imag : REAL;                    ...both initialised to default 0.0
   public
    real : REAL { real }
    imag : REAL { imag }
    conjugate : C
    {           result : C (real, -imag)
    }
    plus (other : @C) : C
    {           result : C (real + other.real, imag + other.imag)
    }
    minus (other : @C) : C
    {           result : C (real - other.real, imag - other.imag)
    }
    times (other : @C) : C
    {           result : C (real * other.real - imag * other.imag,
                                imag * other.real + real * other.imag)
    }
    divide (other : @C) : C
    {           self.reciprocal.times(other)
    }
    reciprocal : C
    {           normal : REAL (real * real + imag * imag);
                result : C (real / normal, -imag / normal)
    }
}
```

Figure 8.3: Definition of a COMPLEX Class

In order to preserve the value-semantics of #COMPLEX[ ] numbers, this class design deliberately avoids providing any public update methods[1]. This means that the results of #COMPLEX[ ]'s computed methods must be *initialised* into the local storage reserved on the stack frame for the return value. For example, the result of *conjugate()* is *initialised* with its whole state:

```
result : C (real, -imag)              ...declaration with initialisation
```

which is preferred over the style which declares, then modifies:

```
result : C;                           ...default initialisation
result.complex(real, -imag)           ...public initialisation function
```

[1] The system method *copy()* is of course available to initialise new objects with a shallow copy of the state of another object. Here, the system method *init()* works fine.

The latter is possible, but less efficient, since *result* is initialised twice; and breaks with value-semantics, because of *complex()*. The formal model for declaration with whole-state initialisation is as described in chapter 6. In most cases, *explicit* local variable storage is reserved for the return value; *divide()* is an exception, reserving *implicit* storage to hold the results of two nested calls: *self.reciprocal* copies its result to the first return buffer, and this *self*-argument is passed by reference to *times(other)*, which copies its result to the second return buffer. All functions copy their results to the call-site, since they represent new numbers; in any case a function may not return an alias to a local buffer.

8.2 Inheritance Designs

Our language supports the full breadth of inheritance techniques. In contrast to the false divisions created in some languages between implementation inheritance and subtyping (see chapter 2), a parametric model for type inheritance is provided which is consistent with incremental additions to implementation. Multiple inheritance is supported by the type semantics of intersection types, characterised by the *merge* operator. In other ways, difficulties with the mixing of implementations is resolved through method combination techniques.

In chapter 6, the benefits of introducing common functionality at single points in the hierarchy were established. *Deferred classes* support this by declaring *deferred methods*, which generalise over families of methods which have different implementations in descendent classes. Deferred classes are not just convenient theoretical abstractions, they are practically necessary as base types in parametric polymorphic constructions.

8.2.1 Supporting Abstraction and Reification

Figure 8.4 illustrates the deferred class #NUMBER[ ], the ancestor of all numeric types with arithmetical operations. A deferred class is *partially abstract*, in the sense that it provides no implementation for some of its methods.

```
class #NUMBER [N]
(self : @N) is (super : #TOTAL_ORDER [N]) with
{  public
   plus (other : @N) : N {}
   minus (other : @N) : N {}
   times (other : @N) : N {}
   divide (other : @N) : N {}
   ...
}
```

Figure 8.4: Definition of a Number Class

Here, because each eventual numerical type has a different physical representation, implementations cannot be provided for *plus(), minus()* ... at this level of generality, but signatures may be declared for these methods, for the sake of their later polymorphic use in structured classes parameterised over all numerical types. The main reason for deferring a method is to provide common type information about a forthcoming family of descendent methods which necessarily have different implementations.

Method deferment is indicated by a type signature followed by the empty method body {}, corresponding to an *undefined method*. It is therefore an error to invoke a deferred method directly, which has the semantics of accessing an undefined value ⊥. Descendent classes are expected to override all deferred methods with suitable implementations. A compiler may check this statically where the propagation of type information allows. This may be accomplished by collecting the closed set of types which are known to be instantiated anywhere in an application and report if any of these attempt to invoke undefined methods.

```
class #INTEGER [J] values {minint .. maxint}
(self : @J) is (super : #NUMBER [J]) with
{  private
     value : J (0);
   public
     plus (other : @J) : J { self + other }
     minus (other : @J) : J { self - other }
     times (other : @J) : J { self * other }
     divide (other : @J) : J { self / other }
     ...
}
```

Figure 8.5: Definition of an Integer Class

Figure 8.5 illustrates a concrete #INTEGER[ ] class, inheriting from the deferred #NUMBER[ ] class. Defining #INTEGER[ ] as a class situates the primitive INTEGER type in its proper place in the class hierarchy. This opens up the possibility of constructing further classes parameterised by #NUMBER [N] at a later stage, which may have legal INTEGER instantiations. It is intended that all basic built-in numerical types such as REAL and CARDINAL will be provided in this way, with the expectation that further numerical types such as COMPLEX and FRACTION will be made available in class libraries.

In the type introduction section, the keyword **values** introduces an ordered set {*minint .. maxint*} representing the literal values of the class. The practical effect of the keyword is to notify the compiler that special symbols ..., -1, 0, 1, ... are to be treated as self-identifying objects of this class. This has the effect of fixing the concrete representation of the class, which must store a single primitive value-element as its *whole state*. In this way, a clean interface is provided with the underlying simple storage types supported by the hardware.

A simple-valued class, such as BOOLEAN, INTEGER or REAL, may only have one state variable in its **private** part, typically called *value*. The reason for this is to define the storage occupied by *self* to be equivalent to that necessary to store the primitive **values** of the type; and also to provide a default initialisation value for instances, here 0. The **values** declaration ensures that requests to access *self* are implicitly translated into *value* and functions returning primitive values ..., -1, 0, 1, ... are implicitly translated back into INTEGER instances. That self-identification is possible owes as much to the fact that *value* is never modified by any method.

A class *introducing* **values** may not inherit any storage from its ancestors. Note how #NUMBER[ ] and #OBJECT[ ] provided no concrete storage. It would be inappropriate to inherit storage, since this would interfere with the underlying physical representation of integers. The #INTEGER[ ] class provides primitive implementations for all methods that were deferred in #NUMBER[ ]. The operators "+" and "-" used in the bodies of these methods denote basic machine operations defined over the concrete integer representation.

A **values** set is understood to be the union of all values of all the types in the class family. It is possible to define subclasses of #INTEGER[ ], provided that the physical representation is unchanged. When defining a value subclass, the **values** set must be partitioned. It is not legal to extend a set to include more values. Although *value* has the physical representation of an integer, it is given the polymorphic type J. Polymorphism is retained for the sake of defining integer subranges, which may then be closed over their own types.

8.2.2 A Syntax for Method Combination

The availability of recursion variables standing for the inherited part of an object is more obviously useful in cases where *method combination* is desired. In figure 8.6, a #HOT_POINT[ ] class extends the state of the #POINT[ ] class by adding a *selected* variable; and so must redefine the *equal()* method.

```
class #HOT_POINT [H] uses BOOLEAN
(self : @H) is (super : #POINT [H]) with
{  private
    selected : BOOLEAN (false);
   public
    selected : BOOLEAN { selected }
    select { selected := true; }
    deselect { selected := false; }
    equal (other : @H) : BOOLEAN
    {          super.equal(other) & selected = other.selected
    }
}
```

Figure 8.6: Selectable Point using Method Combination

The naming of *super* in the **is**-clause gives a handle on #POINT[ ]'s original methods, so that the inherited *equal()* may be used inside the redefinition. The typing of *super* in the **is**-clause also propagates type information into the original methods of #POINT[ ], which evolve in type when they are inherited by the #HOT_POINT[ ] class. A compiler would be expected to inline many short *super*-method invocations, making this a common mode of expression.

An alternative approach is to make mouse-sensitive selection a *mixin* facility that may be inherited by many classes. Figure 8.7 illustrates a possibly contentious language design for a #SELECTION[ ] mixin class, and shows the style in which mixins are combined:

```
class #SELECTION [S] uses BOOLEAN, #EQUAL [E]
(self : @S; super : #E [S]) is                          ...defined as an extension generator
{  private
    selected : BOOLEAN (false);
   public
    selected : BOOLEAN { selected }
    select { selected := true; }
    deselect { selected := false; }
    equal (other : @S) : BOOLEAN                        ...expects to use method combination
    {          super.equal(other) & selected = other.selected
    }
}

class #HOT_POINT [H]
(self : @H) is (super : #POINT [H]) with (mixin : #SELECTION [H])
```

Figure 8.7: Selectable Point using Mixin Inheritance

The appearance of two recursion variables: (*self* : @S; *super* : #E [S]) indicate that #SELECTION[ ] is an *extension generator* that expects to be combined with some class possessing an *equal()* method. The typing of *super* is not especially attractive in this proposed concrete syntax, since it uses an implicit higher-order quantification #E[ ] over subclasses of $\forall(\tau \subseteq \Phi\text{EQUAL}\ [\tau])$. The problem is that E ranges over types and in order to type *super*, the generator #E[ ] for that type must be inferred retrospectively (with the attendant problems exposed by [AC95]). While this concrete syntax could always be given a dependent, second-order translation:

class #SELECTION [S] **uses** ... #EQUAL [E]
(self : @S; super : #E [S]) **is** ... $\Leftrightarrow$

$\forall(\varepsilon \subseteq \Delta\text{SELECTION}\ [\varepsilon]).\forall(\beta \subseteq \Phi\text{EQUAL}\ [\varepsilon]).\lambda(\text{self}: \varepsilon).\lambda(\text{super}: \beta).$ ...

there is no mechanism for expressing subtyping constraints directly in the concrete syntax, which hides this in the way it expresses F-bounds. Furthermore, the obvious translation of the higher-order typing apparently does not force *super* explicitly to have a supertype of *self*. This constraint could be handled by prohibiting circular inheritance in the underlying language

mechanism. Nonetheless, the subtlety of typing required for mixins may elude most programmers, making this a contentious area. *Mixin inheritance* (see chapter 5) is obtained by supplying a basic object after **is** and an extension object after **with**, in which case no class body should be supplied. The mixin inheritance construction has the translation:

class #HOT_POINT [H]
(self : @H) **is** (super : #POINT [H]) **with** (mixin : #SELECTION [H]) $\Leftrightarrow$

$\forall(\tau \subseteq \Phi$HOT_POINT [$\tau$]).$\lambda$(self: τ).
Σselection [τ, ΦPOINT [τ]] (self, Φpoint [τ] (self))

in which *super* is bound to a parent record and *mixin* to an extension record inside the class extension function Σ*selection*, which combines them internally. The *override* semantics of $\oplus$ makes sure that methods in *mixin* replace those in *super*, after appropriate method combination has taken place.

8.2.3 A Syntax for Multiple Inheritance

Perhaps a more straightforward approach is to use *multiple inheritance* (see chapter 6). Figure 8.8 illustrates an alternative design for #SELECTION[ ], which here is a normal class, rather than a freestanding mixin. The figure also shows a selectable #HOT_POINT[ ] class defined using multiple inheritance:

```
class #SELECTION [S] uses BOOLEAN
(self : @S) is (super : #OBJECT [S]) with                    ...defined normally
{  private
    selected : BOOLEAN (false);
   public
    selected : BOOLEAN { selected }
    select { selected := true; }
    deselect { selected := false; }
    equal (other : @S) : BOOLEAN                             ...simple version
    {          selected = other.selected
    }
}


class #HOT_POINT [H] uses BOOLEAN
(self : @H) is (father : #POINT [H]; mother : #SELECTION [H]) with
{  public
    equal (other : @H) : BOOLEAN
    {          father.equal(other) & mother.equal(other)
    }
}
```

Figure 8.8: Selectable Point using Multiple Inheritance

Multiple inheritance is obtained by supplying a *list* of parents after **is** and then resolving any outstanding inheritance conflicts in a record supplied after **with**. The best advantage offered by the explicit naming of superclass recursion

variables comes when using multiple inheritance, since this enables a resolution to the inheritance conflict over the *equal()* method, by allowing the invocation of an explicit combination of the parents' methods. The use of parentheses to collect superclass recursion variables deliberately mimics the syntax of an argument list, since there is an obvious translation of the above into our formal model:

(self : @H) **is** (father : #POINT [H]; mother : #SELECTION [H])
with {...} $\Leftrightarrow$

λ(self : τ).(λ(father : ΦPOINT [τ]).λ(mother : ΦSELECTION [τ]).
(father $\otimes$ mother) $\oplus$ {...}
(Φpoint [τ] (self))(Φselection [τ] (self)))

in which the objects bound to these internal arguments are made explicit. This translation also illustrates the semantic interpretation of multiple inheritance. The parent object records are understood to be combined using $\otimes$, the *merge* operator defined in chapter 6, before the record combination operator $\oplus$ adds or replaces methods. The implementation of *merge* builds the union of the parents' methods, merging duplicate methods that are identical in implementation. If duplicate methods have different implementations, then the compiler checks that they were declared in a single common ancestor. If not, an error is immediately reported, since this constitutes an accidental name-clash that must be resolved; otherwise the compiler inserts the undefined method {} and expects the method combination to be resolved in the child class.

It is also possible to define a new class simply by combining multiple parents. The class declaration would then have an **is**-clause and no **with**-clause:

(self : @T) **is** (father : #FATHER [T]; mother : #MOTHER [T])

In this case, the *father* and *mother* objects should not leave any method combination unresolved, since this would lead to the merged child inheriting an undefined method (see chapter 6). A compiler should provide warnings in the case that a child class inherits an undefined method and fails to provide an effective resolution. The notation {} for the undefined method is deliberately identical to the syntax for a deferred method, since the semantic consequences are the same.

8.4 Polymorphic Designs

Our language shows its strengths when defining more complex polymorphic structures, such as classes with internal polymorphic components. The main practical improvement over existing object-oriented languages is that all forms of polymorphism are notated in essentially the same way. The programmer is not burdened with separate mechanisms for run-time dynamic binding and statically-resolved generic parameterisation. To a large extent, the compiler

may be relied upon to sort this out (see chapter 10). An intuitive nested parametric scheme: S[T] is adopted, suggesting the idea of "type parameters with holes in", to reflect the fact that the *self*-type S may become dependent on a further type parameter T. The language supports both homogeneous and heterogeneous polymorphism by two different formal translations of this syntax, which are automatically determined from the context. Below, the nature of these two translations, and why they are necessary, is explained.

8.4.1 Extending Polymorphism

A polymorphic list is described as *homogeneous* if it contains elements all of the same type. Such lists may be generalised as classes, amenable to later extension through inheritance, by abstracting over the type of *self*:

$$\Phi\text{LIST} = \lambda\tau.\lambda\sigma.\{\text{add} : \tau \to \sigma, \text{head} : \tau, \text{tail} : \sigma\}$$

$$\Phi\text{list} : \forall(t \subseteq \Phi\text{TOP}\ [t]).\forall(s \subseteq \Phi\text{LIST}\ [t, s]).s \to \Phi\text{LIST}\ [t, s]$$

ΦLIST is a type function of two arguments, τ and σ, which are bound in that order. The type σ of the tail describes a recursion which is closed over the element type τ. Applying ΦLIST to a particular elment type ε yields a type generator ($\sigma \to \sigma$) whose fixpoint σ is a recursive type closed over ε:

$$(\mathbf{Y}\ (\Phi\text{LIST}\ [\varepsilon])) = \mu\sigma.\{\text{add} : \varepsilon \to \sigma, \text{head} : \varepsilon, \text{tail} : \sigma\}$$

Note how a list of this type may only *add()* new elements of type ε; its *head()* has type ε; and its *tail()* has the same type σ as itself, so will be bound recursively to have elements all of the same type. For this reason, such lists are called *homogeneous*. It is not possible to provide heterogeneous lists, containing different types of element, using this form of quantification.

The only way to break the recursive pattern of dependency is to move the element type parameter *inside* the scope of the recursion variable, binding the arguments σ and τ in the reverse order. This has two consequences.

Firstly, occurrences of σ in the body of the type function ΦLIST do not stand for a *self*-type that is closed over some fixed element type, but rather for a *type function* that must be applied to an element type in order to generate a simple list type. This was the reason why the order of dependency was reversed, for it suggests types with alternative parameterisations: $\sigma[p]$, $\sigma[q]$ embedded within a given type: $\sigma[\tau]$.

Secondly, a *higher-order* quantification for *s* must be provided in the typed object generator Φ*list*. This is because we do not have complete information about the element type *t* when we bind the parameter *s* to a constructor for lists. We can only say that we want all possible *s*[*t*] to be pointwise subtypes of Φ LIST [*s, t*]. This suggests a higher-order subtyping constraint **<:** having the particular form:

$$\forall s.(s \mathbf{<:} \Phi\text{LIST}\ [s]) \Leftrightarrow \forall s.\forall(u \subseteq \Phi\text{TOP}\ [u]).(s[u] \subseteq \Phi\text{LIST}\ [s, u])$$

The translation includes the F-bound on the element type, since in general it is not correct to test for pointwise subtyping across the full range $\forall t$. Restrictions on the element type, if ignored, lead to invalid comparisons which may fail unfairly. We assume that static type information about the list type function and the element type generator are available. This produces functions of the form:

$$\Phi\text{LIST} = \lambda\sigma.\lambda\tau.\{\text{add} : \tau \rightarrow \sigma[\tau], \text{head} : \tau, \text{tail} : \sigma[\tau]\}$$

$$\Phi\text{list} : \forall(s \mathbf{<:} \Phi\text{LIST}\ [s]).\forall(t \subseteq \Phi\text{TOP}\ [t]).s[t] \rightarrow \Phi\text{LIST}\ [s, t]$$

which are quite different from those given above. ΦLIST is no longer a type function from an element type τ to a generator ($\sigma \rightarrow \sigma$) for a recursive type, but rather a functional whose fixpoint is a *recursive type function*. It accepts a type constructor $\sigma = (t \rightarrow s[t])$ and yields another type constructor ($\tau \rightarrow \sigma[\tau]$). Taking the fixpoint makes these constructors identical. The significant change is that the first parameter to ΦLIST expects a *type function*, rather than a *simple type*, as its argument.

Higher-order polymorphism complicates type checking somewhat. Whereas before, parameters were bound in an order that provided full information on the constraints affecting each type before that type was checked, now parameters are bound to type constructors in advance of full knowledge about the particular types to which they will be applied. In practice, this problem may be solved by considering that the higher-order constraint:

$$\forall(s \mathbf{<:} \Phi\text{LIST}\ [s])$$

is a way of quantifying over all subclasses of a given class. It is possible to cache full information about classes and their descendants, or classes and their type instantiations, in a graph structure. Higher-order type checking is a lazy, dependent scheme with delayed proof obligations. Provided that *s* is consistent with ΦLIST [*s*], verified by checking in the graph, then it is safe to assume the correctness of any *s*[*t*], on the grounds that this can be discharged later.

8.4.2 A Syntax for Polymorphism

Polymorphism demands more subtlety in interpreting the concrete style of declaration. It is important not to overload the syntax, and yet ensure that all type-elements of the definition have well-founded translations. Figure 8.9 illustrates a parameterised #POINT[ ] class. The keyword **class** introduces a type function #POINT[ ] used to restrict a parametric structure P[N]. This indicates that the class has internal polymorphic components and describes the external parametric-polymorphic type interface to the class. The following **uses** clause introduces the bound #NUMBER[ ] for the parameter N, on which P depends. The **uses** clause therefore serves both a practical purpose in listing dependencies and a formal purpose in quantifying over component types.

```
class #POINT [P[N]] uses #NUMBER [N], BOOLEAN
(self : @P[N]) is (super : #OBJECT [P[N]]) with
{  private
    x, y : N;
   public
    x : N { x }
    y : N { y }
    equal (other : @P[N]) : BOOLEAN
    {          x = other.x & y = other.y
    }
    move (nx, ny : N)
    {          x := nx;  y := ny;
    }
}
```

Figure 8.9: Definition of a Polymorphic Point Class

The first semantic translation, introduced in chapter 6, binds N *outside* P, such that P[N] is understood to have the meaning: $N \rightarrow (P \rightarrow P)$:

class #POINT [P[N]] **uses** #NUMBER [N], ...
(self : @P[N]) **is** (super : #OBJECT [P[N]]) **with** ... $\Leftrightarrow$

$\forall(n \subseteq \Phi\text{NUMBER } [n]).\forall(p \subseteq \Phi\text{POINT } [n, p]).\lambda(\text{self} : p).$
$(\lambda(\text{super} : \Phi\text{OBJECT } [p]).\text{super} \oplus \{...\}$
$(\Phi\text{object } [p]\ (\text{self}))\ )$

This second-order translation may only be performed if all the component type parameters introduced by the **uses** clause, here just N, are also present in the structure P[N] introduced by the **class** clause. The parameters present inside P[...] have the status of generic type parameters, which may be replaced in full at the site of some type declaration, or partly during inheritance, using the partial instantiation approach described in chapter 6.

The second semantic translation, introduced above, binds N *inside* P, such that P[N] is understood to have the meaning: $(N \rightarrow P[N]) \rightarrow (N \rightarrow P[N])$:

class #POINT [P[N]] **uses** #NUMBER [N], ...
(self : @P[N]) **is** (super : #OBJECT [P[N]]) **with** ... $\Leftrightarrow$

$\forall(p \textbf{ <: } \Phi\text{POINT } [p]).\forall(n \subseteq \Phi\text{NUMBER } [n]).\lambda(\text{self} : p[n]).$
$(\lambda(\text{super} : \Phi\text{OBJECT } [p[n]]).\text{super} \oplus \{...\}$
$(\Phi\text{object } [p[n]]\ (\text{self}))\ )$

This higher-order translation is generally more useful, because it allows us to think about the type of *self* as a function P expecting a further parameter N. This means that #POINT[ ] is a generator whose fixpoint POINT[ ] is a recursive type function over N, in the manner of a Girard-Reynolds type constructor. This translation *must* be used if there are component type parameters introduced by **uses** which are *not* present in the structure introduced by **class**. Such

parameters have the status of unresolved polymorphic types which are not present in the interface of the class and may not be statically bound.

In both translations, the concrete syntax takes a small liberty by introducing type parameters in the style #POINT [P[N]] with the component parameter embedded, rather than use #POINT [N, P] or #POINT [P, N] to indicate that the class is a two-parameter type function. This is for reasons of style consistency and aims to foster a feeling in our programmers for the notion of "type parameters with holes in". In the concrete syntax, the type of *self* is always represented P[N], since P now stands in general for a type function, rather than for a recursive type. The whole of the *self*-type P[N] is passed to the parent class generator #OBJECT[ ] to type the super-record. Whenever a class like #POINT[ ] declares a new component type parameter, this introduces extra structure into the way the self-type is expressed that was implicit in the parent class.

The concrete syntax conventions are now reviewed. Our language now has a way of expressing class generators, recursive type constructors and recursive types:

- #POINT[ ] is the style of a class generator. Its purpose is always to describe an F-bound on some *self*-type. It introduces parameters in **class** ... **uses** type introductions and is applied in the **is**-clause of inheritance constructions, to give *super* recursion variables their adapted types.

- POINT[ ] is the style of a type constructor. Its purpose is to permit parameterised type declarations, in which it is applied to component types, in the style: *p* : POINT [INTEGER]. The recursive type function POINT[ ] is implicitly understood to be the fixpoint of the class generator #POINT[ ].

- POINT is the style of a monomorphic type. Its purpose is to permit simple type declarations. Simple type identifiers do not have the brackets [ ] used to indicate a type function. The type POINT is implicitly understood to be the result of fixing both the *self*-type and component-type, such that: *p* : POINT means the same as: *p* : POINT [NUMBER].

8.4.3 A Syntax for Higher-Order Inheritance

The concrete syntax readily adapts to inheritance of polymorphic components. Figure 8.9 showed the *introduction* of a component parameter that was not present in the parent class; figure 8.10 illustrates the *rebinding* and *transmission* of component parameters to a child class. The declaration **class** #HOT_POINT [H[J]] introduces a new structured *self*-type H[J] which matches the structure of the *self*-type P[N] in the parent #POINT[ ] class. Whereas P ranged over all POINT[ ] constructors, the new bound on H restricts this to range over HOT_POINT[ ] constructors; likewise this example chooses (for demonstration's sake) to restrict J to the #INTEGER[ ] family. This is to illustrate the adaptation of both the *self*-type and coordinate-type of #POINT[ ] when it is inherited.

```
class #HOT_POINT [H[J]] uses #INTEGER [J], BOOLEAN
(self : @H[J]) is (super : #POINT [H[J]]]) with
{  private
    selected : BOOLEAN (false);
   public
    selected : BOOLEAN { selected }
    select { selected := true; }
    deselect { selected := false; }
    equal (other : @H[J]) : BOOLEAN
    {          super.equal(other) & selected = other.selected
    }
}
```

Figure 8.10: A Polymorphic Selectable Point Class

The inheritance construction: **is** (super : #POINT [H[J]]]) distributes H to P and J to N in a way that can be thought of as matching one structure directly to another. The following kind of translation is intended:

class #HOT_POINT [H[J]] **uses** #INTEGER [J], ...
(self : @H[J]) **is** (super : #POINT [H[J]]) **with** ... $\Leftrightarrow$

$\forall$(h **<:** ΦHOT_POINT [h]).$\forall$(j $\subseteq$ ΦNUMBER [j]).λ(self : h[j]).
(λ(super : ΦPOINT [h, j]).super $\oplus$ {...}
(Φpoint [h, j] (self)))

in which it is more apparent how our concrete syntax preserves the nested structuring of parameters, while the formal model tends confusingly to string them out in linear fashion.

Another important aspect to introducing **class** #HOT_POINT [H[J]] as a structure containing J is that this preserves the component type interface of #POINT[ ] in the new class: J matches the structural position of N. The reason that J appears in the **uses** #INTEGER [J] clause is so that it may be constrained with a more restricted F-bound. The reason J appears in the inheritance construction **is** (super : #POINT [H[J]]]) is in order to distribute it to N and so restrict the inherited coordinate type to #INTEGER[ ]. There is no other reason for J to exist, since no new component is typed in J. This sheds light on the distinct roles played by the different clauses:

- the **class** clause defines which type parameters appear in the interface, expressing the dependency of the self-type on component types by appropriately nesting parameters;
- the **uses** clause defines F-bounds on component type parameters and also defines those component types on which the self-type depends;
- the **is** clause defines how parameters with more restricted F-bounds are to be distributed to superclass generators, when inheriting their methods.

Our syntax is well adapted to the *introduction* and *transmission* of type parameters. The syntax takes a small liberty in order to *eliminate* type parameters in one step during inheritance. The sketch below illustrates why:

class #INT_POINT [K]
(self : @K) **is** (super : #POINT [K [INTEGER]]) $\Leftrightarrow$

Λ(k $\subseteq$ ΦINT_POINT [k]).λ(self: k).
  (λ(super: ΦPOINT [INTEGER, k]).super
   (Φpoint [INTEGER, k] (self)))

Whereas this second-order translation may always be adopted to replace the component parameter N, the concrete syntax appears to use K both as a *self*-type and a type function. It is to be hoped that programmers will nonetheless find this style intuitive, since it pattern-matches the whole structure of the child class #INT_POINT[ ] with the structure of the parent #POINT[ ].

8.4.4 A Syntax for Unresolved Polymorphic Type

In chapter 5, a technique involving parameter substitution was used to create a simple POLY_POINT type and *poly_point* object having an unresolved polymorphic coordinate type:

ΦPOINT = $\Lambda\tau.\Lambda\sigma.${x: τ, y: τ, identity: σ, equal: $\sigma \rightarrow$ BOOLEAN}

Φpoint : $\forall$(t $\subseteq$ ΦNUMBER [t]).$\forall$(s $\subseteq$ ΦPOINT [t, s]).s $\rightarrow$ ΦPOINT [t, s]

Φpoint = Λ(t $\subseteq$ ΦNUMBER [t]).Λ(s $\subseteq$ ΦPOINT [t, s]).λ(self: s).
  {x $\mapsto \perp$, y $\mapsto \perp$, identity $\mapsto$ self,
    equal $\mapsto$ λ(other: s).(self.x = other.x $\wedge$ self.y = other.y)}

POLY_POINT = $\forall$(u $\subseteq$ ΦNUMBER [u]).($\mathbf{Y}$ (ΦPOINT [u]))

poly_point = $\forall$(v $\subseteq$ ΦNUMBER [v]).($\mathbf{Y}$ (Φpoint [v, POLY_POINT [v]]))

This makes POLY_POINT a recursive type function over its type argument *u*, in the manner of a type constructor:

POLY_POINT = Λu.$\mu\sigma.${x: u, y: u, identity: σ, equal: $\sigma \rightarrow$ BOOLEAN}

likewise the instance *poly_point* is a function from a type to a recursive object:

poly_point : $\forall$(v $\subseteq$ ΦNUMBER [v]).POLY_POINT [v]

poly_point = Λ(v $\subseteq$ ΦNUMBER [v]).μ(self: POLY_POINT [v]).
  {x $\mapsto \perp$, y $\mapsto \perp$, identity $\mapsto$ self,
    equal $\mapsto$ λ(other: s).(self.x = other.x $\wedge$ self.y = other.y)}

What does this mean? The fields of *poly_point* cannot be accessed directly, without first supplying a type for *v*. Whereas it is reasonable to think of

supplying a simple type INTEGER to a type function POLY_POINT in a static context, it is strange to think of propagating a type into a runtime object *poly_point*, when there are no grounds for assuming this type. However, it is legitimate to distribute *new* type parameters having *exactly the same bounds*, in order to release methods from objects protected by F-bounded type abstraction:

$$\forall(\tau \subseteq \Phi\text{NUMBER}\ [\tau]).\text{poly\_point}\ [\tau].\text{x} \Rightarrow \bot : \tau$$

An object containing *k* unresolved polymorphic components is therefore represented as a type function from *k* arguments to a recursive object; and this would require the distribution of *k* new parameters each time one of its methods were accessed. If instead the binding of component types is moved inside the recursion of *self*, new type parameters may then be introduced *inside* the object itself, which may be used to type unresolved polymorphic components.

The syntax is now extended to allow the declaration of variables having an unresolved polymorphic type. The **uses** clause may introduce further types:

uses ... #NUMBER [N]
{ ...
var : N; $\Leftrightarrow$ $\forall(\tau \subseteq \Phi\text{NUMBER}\ [\tau]).\text{var} : \tau$
... }

The dynamic type checking that results from such a declaration can be motivated in the following way. To access *var*, the system would have to distribute a type or a parameter satisfying $\forall(\tau \subseteq \Phi\text{NUMBER}\ [\tau])$ to the type abstraction protecting *var*. If it can determine a unique type from the surrounding context, the system may distribute this to τ and assume *var* has this type. If it cannot determine a unique type, then it must distribute the undefined type $\bot$ to τ and mark *var* as needing a dynamic type check when its value is accessed. From the context, the bound on the type of *var* is always available, which may also be useful.

8.4.5 A Syntax for Heterogeneous Polymorphism

A list is described as *heterogeneous* if it contains elements of different types. Heterogenous lists often contain elements of different, but related types which all satisfy the same F-bound, describing the least upper bound on heterogeneity. To provide such a list, a chain of LIST cells must be constructed, each of which has the same polymorphic type, but a different parameterisation. This may be expressed as:

$$\Phi\text{LIST} = \lambda\sigma.\lambda\tau.\{\text{add} : \lambda\upsilon.\upsilon \rightarrow \sigma[\upsilon], \text{head} : \tau, \text{tail} : \lambda\varpi.\sigma[\varpi]\}$$

Here, ΦLIST is a functional whose fixpoint is a recursive type function in the element type τ. However, applying (**Y** ΦLIST) to some element type ε results in an unusual recursive type:

$$(\mathbf{Y}\ \Phi\text{LIST})\ [\varepsilon] = \mu\sigma.\{\text{add} : \lambda\upsilon.\upsilon \rightarrow \sigma[\upsilon], \text{head} : \varepsilon, \text{tail} : \lambda\varpi.\sigma[\varpi]\}$$

which contains further polymorphic components. In particular, complete information is only known about the simple type of the *head()*.

Looking at the other fields of this type, it is clear that the *add()* method, which extends the list, is also a type function. It accepts a type υ and an element in this type, returning a new list in the type σ[υ]. Because the binding τ = ε is lost when σ[υ] is formed, we may only determine that an extended list has the type:

$$\mu\sigma.\{\text{add} : \lambda\alpha.\alpha \rightarrow \sigma[\alpha], \text{head} : \upsilon, \text{tail} : \lambda\beta.\sigma[\beta]\}$$

in other words, we may infer that the *head()* of the extended list has the type υ and no other simple type information. This is exactly how a heterogenous LIST should behave. It is reasonable that we should only know about the simple type of the *head()* in a context where that LIST link was created.

Looking now at the *tail()* of a heterogeneous list, it is clear that this must describe a list with the same overall shape as *self*, the current LIST cell, but no information is available about what type of element it contains. This is represented as λϖ.σ[ϖ], an application of the recursive type function σ[ ] to an unknown element type ϖ, which may or may not be available in the current context. This offers more information than simply saying the result has an unknown list type, since it insists that the recursive structure follows σ[ ], the shape of *self*. Again, this is quite reasonable for a heterogeneous list.

Our final example in figure 8.11 is a #LIST[ ] class, which inherits certain properties common to all sequences from a deferred #SEQUENCE[ ] class. This class design is used to illustrate the suggested style for heterogenous polymorphism. Here, the type parameter structure introduced by **class** takes on a particular significance in its role as the external polymorphic type interface. The keyword **class** introduces a type function #LIST[ ] constraining a *self*-type parameter L that is dependent on three element-type parameters O, P and Q, constrained by #OBJECT[ ] in the **uses** clause. Whereas the parameter O is bound in the type interface, indicated by its presence in L[O], the remaining parameters are considered bound inside the scope of *self*. A list-cell created from this class has a statically-determined *head()* type O, and unresolved polymorphic types for *add()* : Q → L[Q] and *tail()* : L[P]. It is reasonable to suppose that *add()* may bind a type to Q and this static information may be retained as the *head()* type of the resulting list, but *tail()* cannot presume to guess its type. Instead, elements are retrieved from the tail of the list by distributing ⊥ to the parameter P and then dynamically checking their type later.

```
class #LIST [L[O]] uses #OBJECT [O],
            #OBJECT [P], #OBJECT [Q]                ...distinct parameters
(self : @L[O]) is (super : #SEQUENCE [L[O]]) with
{  private
   head : O;
   tail : @L[P];
  public
   head : O { head }
   tail : @L[P] { tail }
   add (elem : Q) : @L[Q]
   {        cell : @L[Q];
            cell.create(elem, self)                 ...allocate and initialise
   }
}
```

Figure 8.11: A Heterogeneous List Class

The translation of this concrete syntax is a showcase for many of the formal techniques introduced so far. The aim is to provide a heterogeneous version of $\Phi\alpha$*list*, the extended typed object generator expecting a state initialisation argument. The initialisation record may be given the parameterised type:

$$\Psi\text{LIST} = \lambda\sigma.\lambda\tau.\{\text{hd: } \tau, \text{ tl: } \sigma[\bot]\}$$

reflecting the change of σ from a type to a type function and marking the type of the tail as partially unknown. The typed definition of a heterogeneous list may be considered to have the following structure.

$$\Phi\text{LIST} = \lambda\sigma.\lambda\tau.\{\text{add : } \lambda\upsilon.\upsilon \to \sigma[\upsilon], \text{ head : } \tau, \text{ tail : } \lambda\varpi.\sigma[\varpi]\}$$

$$\textbf{rec } \Phi\alpha\text{list} : \forall(\text{s } \textbf{<:} \ \Phi\text{LIST [s]}).\forall(\text{t} \subseteq \Phi\text{OBJECT [t]}).\ \Psi\text{LIST [s, t]} \to (\text{s[t]} \to \Phi\text{LIST [s, t]})$$

since it accepts an initialisation argument and returns a generator for a list, a function from the self-type *s*[*t*] to the list type itself. Notice how only *t* is present in the external type interface of the typed object generator, $\Phi\alpha$*list*, whose definition is given by:

$$\begin{aligned}&\textbf{rec } \Phi\alpha\text{list} = \Lambda(\text{s } \textbf{<:} \ \Phi\text{LIST [s]}).\Lambda(\text{t} \subseteq \Phi\text{OBJECT [t]}).\\ &\quad \lambda(\text{state : } \Psi\text{LIST [s, t]}).\lambda(\text{self: s[t]}).\\ &\quad \{\text{ head} \mapsto \text{state.hd, tail} \mapsto \Lambda(\text{u} \subseteq \Phi\text{OBJECT [u]}).\text{state.tl},\\ &\quad\ \text{add} \mapsto \Lambda(\text{v} \subseteq \Phi\text{OBJECT [v]}).\lambda(\text{elem: v}).\\ &\quad\quad (\mathbf{Y}\ (\Phi\alpha\text{list [s, v]})\ \{\text{hd} \mapsto \text{elem, tl} \mapsto \text{self}\})\ \}\end{aligned}$$

This function is recursive, which is marked using **rec**. A recursive occurrence of the object generator $\Phi\alpha$*list* [*s, v*] appears in the body of *add()*. The body of the generator has two further type abstractions, one for typing the argument of *add()* and one protecting access to the *tail()*, effectively hiding the exact type of all elements in the rest of the list.

To solve the recursion, a typed object definition $\Gamma\alpha$*list* may be provided that abstracts over its own object generator *and* accepts an initialisation argument. The type signature for this function is wonderfully complicated:

$\Gamma\alpha$list : $\forall$(s **<:** ΦLIST [s]).$\forall$(t $\subseteq$ ΦOBJECT [t]).
(ΨLIST [s, t] $\rightarrow$ (s[t] $\rightarrow$ s[t])) $\rightarrow$ (ΨLIST [s, t] $\rightarrow$ (s[t] $\rightarrow$ ΦLIST [s, t]))

$\Gamma\alpha$list = Λ(s **<:** ΦLIST [s]).Λ(t $\subseteq$ ΦOBJECT [t]).
λ(selfgen : ΨLIST [s, t] $\rightarrow$ (s[t] $\rightarrow$ s[t])).
λ(state : ΨLIST [s, t]).λ(self: s[t]).
{ head $\mapsto$ state.hd, tail $\mapsto$ Λ(u $\subseteq$ ΦOBJECT [u]).state.tl,
add $\mapsto$ Λ(v $\subseteq$ ΦTOP [v]).λ(elem: v).
($\mathbf{Y}$($\mathbf{Y}$ selfgen {hd $\mapsto$ elem, tl $\mapsto$ self})) }

but makes sense when you consider that *selfgen* must have a type similar to the object generator $\Phi\alpha$*list* [*s, v*] over which it abstracts, apart from the fact that it maps its result to an as-yet unfixed *self*-type *s*[*t*]. This mode of definition is used here merely to show that ($\mathbf{Y}$ *selfgen*) = $\Phi\alpha$*list* [*s, v*] is well-formed - henceforward, **rec** may be used to indicate recursively-defined functions. It would be wrong to use generator-abstraction to adapt *selfgen* through inheritance, because functions with initialisation arguments require modification to the initialisation argument in non-deterministic ways.

8.4.6 In Support of Classes

A set of concrete language features supporting the object-oriented notion of classification and inheritance has been presented. The language distinguishes the notions of *class* and *type*. Classes are polymorphic entities, typed using F-bounded parameters. The language encourages the declaration of open-ended classes, since the least fixed point types may be inferred automatically. Both reference and value semantics are supported, allowing the programmer to extend the universe of basic types. Encapsulation is supported. Single and multiple inheritance are both supported, with clear translations in the λ-calculus model. Inheritance conflict resolution is handled by naming the inherited parts of objects and invoking combinations of methods explicitly. Abstract classes are supported through the declaration of deferred methods. Classes with internal polymorphic components are supported using a nested parametric syntax, which has two alternative translations in the λ-calculus model. The choice of translation is determined by the context of use. Finally, both homogeneous and heterogeneous polymorphism are supported, with distinct translations in the λ-calculus model.

This completes the tour of language facilities supporting objects, classes, inheritance and polymorphism. It remains now to describe in some detail how control flow is handled and how type information is propagated. This will lead finally to a discussion of implementation issues.