Chapter 6

Multidimensional Classification

Here, the theory presented earlier is extended to allow classification in more than one dimension.

Multiple classification brings an ability to view a type from more than one perspective. It allows classes to be subsumed by more than one superclass, which may describe orthogonal or intersecting properties. Higher classification brings an ability to abstract over classes. It allows the definition of classes which describe families of classes. These two ideas are the principal mathematical constructions underlying such operational object-oriented concepts as multiple inheritance, classes of type constructors and classes with polymorphic components.

6.1 The Dimensions of Classification

In the *simple theory of classification*, classes are constructed by abstracting over the type of *self*. A class is a simple generalisation of a recursive abstract type whose fields are function types. Classes are ordered in a hierarchy, structured according to its perceived usefulness. *Multiple classification* offers the possibility of defining a lattice connecting the space of all recursive types, bringing an ability to view a type simultaneously from more than one perspective. *Higher classification* abstracts over further internal parts of a type, describing higher-order classes with polymorphic components.

6.1.1 Simple Classification

Simple classification groups sets of recursive abstract types under a *single hierarchy* of classes satisfying the partial order: $\forall(\tau \subseteq \Phi G[\tau]).\Phi G[\tau] \subseteq \Phi F[\tau]$.

This order allows us to visualise the notion of a class hierarchy as depicted in figure 6.1. Here, classes are shown to be a stacking series of cones, each corresponding to the space of recursive abstract types satisfying the class bound. The point at the apex of each cone is the fixpoint of the class, the least type that is also a member of the class. The volume under each cone describes the space of possible types that satisfy the bound. These include recursive types with strictly more functions lying on the conical surface area and also many pre-fixpoints occupying the central volume.

The root of this hierarchy is the class $\forall(\tau \subseteq \Phi TOP\ [\tau])$ bounded by the vacuous generator: $\Phi TOP = \lambda\sigma.\{\}$, which encompasses all possible recursive types. This class is the outermost cone in figure 6.1. Subsequent classes are defined by adding functions to ΦTOP to give extended generators ΦF_i which therefore describe subclasses: $\forall(\tau \subseteq \Phi F_i\ [\tau]).\Phi F_i\ [\tau] \subseteq \Phi TOP[\tau]$.

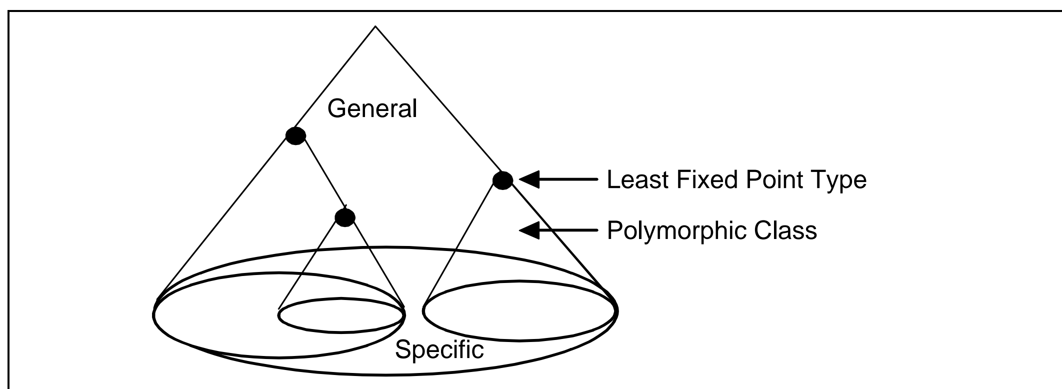

Figure 6.1: Simple Classification

A single hierarchy commits us to introducing functions in a particular order. There is no *a priori* correct hierarchy: several sensible class orders may be chosen to introduce the same *useful* set of recursive types. In general, a given class owning *m* functions can be derived in *m!* different ways, since the order in which its functions are acquired is not significant. However, having only a single dimension of classification usually leads to the repeated introduction of identical functions in disjoint parts of the hierarchy; this is an unwanted redundancy.

6.1.2 Multiple Classification

The space of all recursive abstract types is "quantal" in single functions[1] - this is the minimum unit of granularity distinguishing one class from another. According to this observation, it would be more accurate to describe the space of recursive abstract types as a *lattice*, or *heterarchy*. This space is mapped out and defined below.

---

[1] and axioms - consider the effect of adding the LIFO and FIFO axiom to further qualify an *add()* function to discriminate STACKs from QUEUEs [Simo94b].

Immediately under the class $\forall(\tau \subseteq \Phi TOP\ [\tau])$ there exists a flat layer of *n* classes, $\forall(\tau \subseteq \Phi F_i\ [\tau])$ each bounded by a generator ΦF_i which introduces a *single, distinct function* from the set of all *n* functions over ordinary values. Each of these classes will trivially satisfy $\forall(\tau \subseteq \Phi F_i\ [\tau]).\Phi F_i\ [\tau] \subseteq \Phi TOP\ [\tau]$. Directly below this layer, there exists a layer of classes $\forall(\tau \subseteq \Phi G_k\ [\tau])$ each bounded by a generator ΦG_k having a distinct *pair* from the set of all *n* functions. For each *2*-tuple generator ΦG_k there will be exactly two generators ΦF_i from the first layer satisfying $\forall(\tau \subseteq \Phi G_k\ [\tau]).\Phi G_k\ [\tau] \subseteq \Phi F_i\ [\tau]$, being those *1*-tuple generators which introduced each function singly; and therefore there will be *n(n-1)/2* classes in the second layer. Following the law of combinations, the *pth* layer will have *n!/(p!(n-p)!)* classes, each *p*-tuple class having *p* immediate parents, being all those *(p-1)*-tuples that omit one function in turn. If it can be assumed that each function is semantically independent and may combine with any other set of functions, the lattice will expand until the layer describing *(n/2)*-tuples, after which it will contract again, terminating in a single *n*-tuple class with *n* parents, describing the single type having all *n* functions over simple values.

In practice, a class lattice will not converge to a single type, because functions gradually acquire a dependent semantics. The STACK and QUEUE classes derived by axiomatising *add()* and *remove()* may not subsequently be recombined. Nonetheless, multiple classification is practical for many useful types since it avoids the redundancy of introducing identical functions in disjoint parts of a single hierarchy.

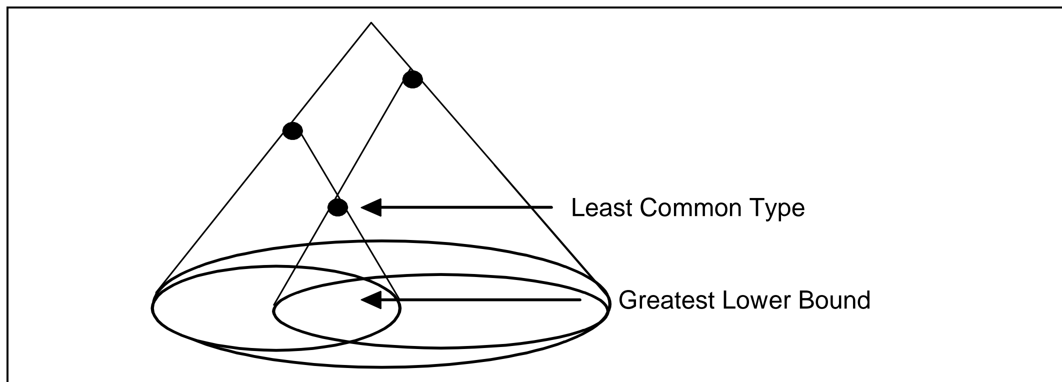

Figure 6.2: Multiple Classification

Figure 6.2 illustrates this. Here, classes are visualised as a stacking series of intersecting cones corresponding to overlapping spaces of recursive abstract types. Where two cones intersect, the volume of intersection describes the space of types satisfying both bounds. This is the *greatest lower bound*, the class which is also a subclass of both original classes. The apex of this intersection corresponds to the least type which is a member of both classes; this is also the fixpoint of the common subclass, the least type offering exactly the functions of both classes. Types with strictly more functions lie on the curved surface area bounding the volume of intersection, which also contains many other pre-fixpoints.

Our sketch uses the three dimensions of solid geometry to represent the many more directions in which a class with *m* functions can be developed. A planar cut through a cone that describes a circle on its conical surface just one "quantum" below its apex represents all types with *m+1* functions. Each point on the circle represents a type that has extended the apical type by one function, a different one in each case. If there are *n* distinct functions, a class may develop in *n-m* ways around its boundary. Where cones intersect, this is because they meet other cones which have developed the same functions.

6.1.3 Higher Classification

The theory presented so far describes families of recursive *types*. Another large and useful group which the theory might usefully be adapted to describe are the *type constructors*. Programming languages typically provide type constructors which generate typed collections, such as LIST, STACK and QUEUE. In contrast to a class generator, which abstracts over its *self*-type, a type constructor is a recursive function which abstracts over an *element*-type. However, it is possible to imagine the family of types generated from a type constructor as an F-bounded class:

$$\text{LIST} = \forall(\tau \subseteq \Phi\text{TOP}\ [\tau]).\mu\sigma.\{\text{add} : \tau \rightarrow \sigma,\ \text{head} : \tau,\ \text{tail} : \sigma\}$$

This is simply the F-bounded generalisation of universal quantification. The reasonableness of this becomes apparent when seeking to describe constructors which depend on their element-type satisfying certain criteria:

$$\text{SORTED_LIST} = \forall(\tau \subseteq \Phi\text{COMPARABLE}\ [\tau]).\\ \mu\sigma.\{\text{add} : \tau \rightarrow \sigma,\ \text{head} : \tau,\ \text{tail} : \sigma\}$$

F-bounded quantification is more useful in this context than bounded quantification, since it preserves the recursion of more specialised element-types (see chapter 4). The fact that F-bounds are useful to type elements and *self* suggests that they might form the basis for unifying descriptions of classes and constructors.

Intuitively, the theory may be extended to support generalisations over type constructors. A *class of list constructors* includes the LIST and SORTED_LIST given above, describing a family of list constructors that have *at least* the behaviour of $\forall(\tau \subseteq \Phi\text{TOP}\ [\tau]).\text{LIST}\ [\tau]$. Since further lists in this family may possess extra functions, such as *length()* or *rank()*, we need to consider abstracting over the *self*-type in addition to the element-type. A suitable order of quantification for this class is: $\forall(\tau \subseteq \Phi\text{TOP}\ [\tau]).\forall(\sigma \subseteq \Phi\text{LIST}\ [\tau, \sigma])$, where:

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

because the *self*-type σ of any list is dependent on its element-type τ. ΦLIST is a function from types to generators, since it maps τ to a generator for a recursive list $\Phi\text{LIST}\ [\tau]$, whose *self*-type is not fixed. For some particular element-type *e* we may visualise the cone of types included in the class given

by $\forall(\sigma \subseteq \Phi\text{LIST } [e, \sigma])$. The fact that we abstract over the element-type as well makes it difficult to visualise the class in three dimensions.

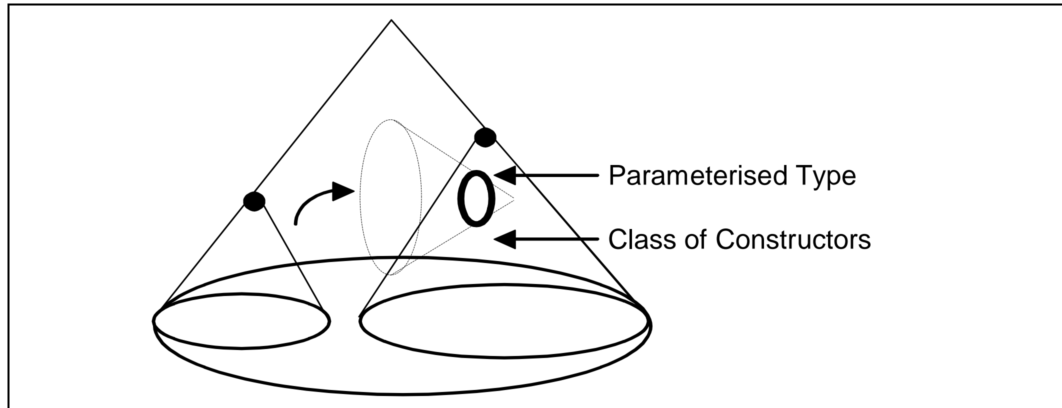

Figure 6.3: Higher Classification

Figure 6.3 attempts to depict the space of types and type constructors. Here, fourth-dimensional "holes" occur in the sides of the cone, each standing for a space of alternate list-type extensions resulting from adding a single function parameterised by the element-type. The holes are elastic and they may be mapped onto the cones representing the class of element required. By taking an element-cone and pushing it apex-first through a hole, the ellipse widens to accommodate, contacting it around a ring representing increasingly more developed element types. The widening ellipse on the list-cone's surface, which touches the element-types on the embedded cone, represents those alternative extended list-types resulting from different instantiations of the parameterised function. A hole which contracts to a point represents the list-type having the function instantiated by the element class's least fixed point. The apex of the list-cone represents the least list type instantiated with the least element type. A minimal list having a specific element type is visualised as one point on the circumference of a hole opened up exactly at the apex of the list-cone.

We call this *higher classification*, because in general it quantifies over type constructors as well as types. In general, we can imagine an infinite regression of classes whose parameterised components are other parameterised classes. At this point, higher-order quantification seems more appropriate. Since type constructors themselves describe classes of types, higher classification describes classes recursively.

6.1.4 Higher-Order Record Combination

So far, the simple theory of classification allows the field-types of objects to be overridden by simple subtype fields. This latitude comes from the override constraint Ω which types polymorphic record combination:

$$\varepsilon \; \Omega \; \beta \equiv \forall(a \in \text{dom}(\varepsilon) \cap \text{dom}(\beta)).\ \varepsilon.a \subseteq \beta.a$$

This says that an extension record type ε overrides a base record type β if the fields it has in common are in a simple subtype relationship. Because of dynamic binding, subtyping leads to a rudimentary kind of polymorphism, albeit not a very useful kind (see chapter 4). Since F-bounds provide a better mechanism for handling general polymorphism, the rudimentary subtyping kind is now removed, eliminating an unwanted degree of freedom in the type system. A language is desired in which variables are typed unequivocally either with a *monomorphic* type $\mu\sigma.F(\sigma)$ or a *polymorphic* type $\forall(\tau \subseteq \Phi F[\tau])$.

Henceforward, we shall allow simply-typed record fields to be replaced only by fields having the same type, *cf Emerald* [RL89, BHJL86]. The base case for overriding Ω says that a monomorphic type overrides itself.

$$\frac{\Gamma \vdash \sigma = \tau}{\Gamma \vdash \sigma \, \Omega \, \tau} \qquad \Omega\text{TYPE}$$

Now, because record combination must be generalised to handle fields with arbitrary parameterised polymorphic types, it is necessary to specify an override condition between F-bounded type parameters.

Trivially, a parameter may be replaced by any parameter. This is because parameters are not types, but abstractions over types. However, the effect of substituting one F-bounded parameter for another is to change the family of types that may legally instantiate the parameter. If we consider parameter-substitution as *identifying* one parameter with the other, then the effect of substituting $\forall(\tau \subseteq \Phi F[\tau]).\tau$ by $\forall(\sigma \subseteq \Phi G[\sigma]).\sigma$ is to *unify* [Robi65] the two parameters τ and σ. The only types which may instantiate the result are those which obey both F-bounds: $\forall(\tau \mid \tau \subseteq \Phi F[\tau] \wedge \tau \subseteq \Phi G[\tau])$, which is expressed as a greatest lower bound condition: $\forall(\tau \subseteq (\Phi F[\tau] \cap \Phi G[\tau]))$. Depending on the bounds, there may or may not exist any types in this family.

Greatest lower bounds will be considered again later. For the moment, it is important to ensure that where a record-field parameterised by τ is replaced by a field parameterised by σ, then a family of types exists for $f(\sigma)$ that is a subset of the family for $f(\tau)$. This may be expressed as an override condition Ω for parameterised methods:

$$\frac{\Gamma \vdash \forall(\tau \subseteq \Phi G[\tau]).\Phi G[\tau] \subseteq \Phi F[\tau]}{\Gamma \vdash \forall(\sigma \subseteq \Phi G[\sigma]).f(\sigma) \, \Omega \, \forall(\tau \subseteq \Phi F[\tau]).f(\tau)} \qquad \Omega\text{PARM}$$

which essentially allows a method that is polymorphically typed over a given class to be replaced by a method that is retyped over a subclass. This rule allows specialisation in other polymorphic types in the same way that inheritance specialises the *self*-type.

To complete the definition of Ω, the general case for records is specified, which says that an extension record type may override a base record type if the fields which it has in common are subject to the same override criterion:

$$\frac{\Gamma \vdash \forall (a \in dom(\varepsilon) \cap dom(\beta)).\ \varepsilon.a\ \Omega\ \beta.a}{\Gamma \vdash \varepsilon\ \Omega\ \beta} \qquad \Omega RECD$$

The definition of Ω is now recursive; therefore the override criterion is now expressed in a higher-order way. The extension to handle arbitrarily-nested polymorphic parameters allows us to type *higher-order record combination.* Higher-order inheritance is different from higher-order subtyping [SP94] in the same way that type inheritance is different from subtyping.

6.2 Multiple Classification

Many object-oriented languages express the idea of multiple classification by allowing inheritance from more than one parent class [Meye88, Keen89, Stro91]. Others provide independent *protocols* expressing the type-compatibility of components created in disjoint parts of a simple hierarchy [GJ90, NeXT93]. Protocols make up for the lack of expressiveness offered by single inheritance.

Multiple inheritance requires two or more parent classes to be combined in some fashion, then possibly augmented by additional methods. The resulting child class is a subclass of each of its parents and describes a subset of the objects that may be described by each of its parents. During multiple inheritance, a child class will acquire at least the union of the methods of its parents, since it must provide at least the services of each parent. This may be a *simple union* or a *disjoint union*, depending on how inheritance conflicts are to be handled. If there is no limit on the number of parents, these are both *distributed* unions.

6.2.1 Inheritance Conflict Resolution

An *inheritance conflict* arises where a class obtains the same named method from more than one parent - the resolution of this conflict decides which of these methods (perhaps both) are incorporated in the child. We distinguish *accidental* and *recombinant* inheritance conflicts. An *accidental* conflict arises from inheriting two semantically distinct methods, which accidentally have the same name and which were introduced in disjoint parts of the class heterarchy. A *recombinant* conflict arises from inheriting two semantically related methods, which are different specialisations of a single method that was originally introduced at a single point in the class heterarchy.

Object-oriented languages which support the *disjoint union* of methods [Stro91, Meye92] usually do so on the grounds that method names were ill-chosen and therefore both inherited versions of the method are required in the child.

Disjoint unions support the resolution of accidental conflicts. We consider it inappropriate for a language model to have to rectify poor nomenclature. Accidental conflicts can be resolved syntactically by renaming one or other method throughout. By removing accidental conflicts, our language model only has to provide for the *simple union* of inherited methods, in a context where any conflicts are due solely to method recombination.

Ideally, a child class should be a deterministic synthesis of the most specific aspects of its parent classes, without undue prejudice to either parent. It is expected in the majority of cases that multiple inheritance will lead to the straightforward concatenation of the parents' fields. Where a conflict occurs, it cannot easily be assumed that one or other method should automatically be chosen. If the method has been retyped in just one branch, it may be possible to select the method with the most specific type. Otherwise, there are no simple grounds for selecting one method over another, especially if they have the same type.

Automatic conflict-resolution on the basis of class pre-order [Moon86, BS83], or using sophisticated algorithms to linearise the heterarchy [BDGK88, Keen89], delivers incorrect results for certain lattices [DHHM94]. The way a child class inherits from its parents should not depend on unexpected interactions between the ordering of its more distant ancestors [Meye88, 246-250]. Since inheritance is only a short-hand for full local definition, a class is unaware of the original point of introduction of methods, so it is inappropriate to compute a recency-based class ordering along the lines of Touretzky's *inferential distance* [Tour86].

6.2.2 Merging and Intersection Types

A compromise is proposed, on the assumption that recombination is for the most part a benign merging of identical methods, inherited along two different paths. A new typed record combination operator $\otimes$ will concatenate two parents:

$$\text{result} = \text{father} \otimes \text{mother}$$

such that the *result* obtains the union of methods in the parents. The operator $\otimes$ will be typed to rule out combinations where methods in a conflict-pair have *incompatible* types. For methods with *compatible* types, those that are identical in implementation will be merged and in all other cases the programmer will be required to specify an implementation. What constitutes type compatibility between a method pair and, in general, between two parent classes? Working from first principles, it is important to ensure that a child class has a type which is a pointwise subtype of each of its parents:

$$\forall(t \subseteq \Phi\text{CHILD}\ [t]).$$
$$(\Phi\text{CHILD}\ [t] \subseteq \Phi\text{FATHER}\ [t]) \wedge (\Phi\text{CHILD}\ [t] \subseteq \Phi\text{MOTHER}\ [t])$$

Put another way, the child must have a type which is *at most* the greatest lower bound, or pointwise intersection of the parent-types; and may be a pointwise subtype of this intersection type, by virtue of having extra methods:

$$\forall(t \subseteq \Phi\text{CHILD}[t]).\Phi\text{CHILD}[t] \subseteq (\Phi\text{FATHER}[t] \cap \Phi\text{MOTHER}[t])$$

To handle this properly, *intersection types*, or *meet types* [Pier92a, CP93] must be considered in more detail. A general condition M is sought, expressing whether two types *meet*, analogous to our earlier override condition.

As a first step, arbitrary intersections $\sigma \cap \tau$ of different simple types σ and τ are to be disallowed, since admitting such intersections would entail the automatic generation of large numbers of subrange and subset types. The base case for the *meet* condition is therefore type equality:

$$\frac{\Gamma \vdash \sigma = \tau}{\Gamma \vdash \sigma \, \mathrm{M} \, \tau} \quad \text{MTYPE}$$

which says that a monomorphic type meets itself. This condition will permit the merging of methods that have identical simple types.

The generalisation of this to handle pairs of parameterised method signatures, where the parameters in each case are subject to different F-bounds, is derived from our earlier discussion on greatest lower bounds:

$$\frac{\Gamma \vdash \exists\tau \subseteq (\Phi G[\tau] \cap \Phi F[\tau])}{\Gamma \vdash \forall(\sigma \subseteq \Phi G[\sigma]).f(\sigma) \, \mathrm{M} \, \forall(\tau \subseteq \Phi F[\tau]).f(\tau)} \quad \text{MPARM}$$

Two parameterised methods have an intersection type if there exists at least one type that satisfies the bounds on each parameter. This in turn can be shown by proving that a class $\forall(\tau \subseteq \Phi H[\tau]).(\Phi H[\tau] = \Phi G[\tau] \cap \Phi F[\tau])$ exists and can be computed. The class $\forall(\tau \subseteq \Phi H[\tau])$ is that bounded by the generator ΦH obtained from merging the two generators ΦG and ΦF. The merging function is described below. Essentially, this rule permits the merging of parameterised methods on condition that an intersection type exists for the parameters, which may be calculated in turn by combining the two F-bounds using a merge function.

Finally, the *meet* condition for two record types says that two record types intersect only if their common fields have an intersection type:

$$\frac{\Gamma \vdash \forall(a \in \text{dom}(\varepsilon) \cap \text{dom}(\beta)).\ \varepsilon.a \, \mathrm{M} \, \beta.a}{\Gamma \vdash \varepsilon \, \mathrm{M} \, \beta} \quad \text{MRECD}$$

The *meet* condition M is symmetrical, unlike the *override* condition Ω given earlier. It is more appropriate when programming with multiple inheritance,

since it allows the types of a child's fields to be equally influenced by its father or mother. Given the condition M, the higher-order typed record combination function *merge* may be constructed:

$$\text{merge} : \forall\alpha.\forall(\beta \mid \beta \mathrm{M} \alpha)\ \alpha \to \beta \to \alpha\cap\beta$$

$$\begin{aligned}
&\text{merge} = \Lambda\alpha.\Lambda(\beta \mathrm{M} \alpha).\lambda(\text{father: }\alpha).\lambda(\text{mother: }\beta).\\
&\quad \{\ \text{label} \mapsto \text{value} \mid (\text{label} \in \text{dom(father)} \cup \text{dom(mother)})\\
&\qquad\quad \wedge (\text{if label} \in \text{dom(father)} \cap \text{dom(mother)}\\
&\qquad\qquad\quad \text{then if father.label} = \text{mother.label}\\
&\qquad\qquad\qquad\quad \text{then value} = \text{father.label} : \alpha.\text{label}\cap\beta.\text{label}\\
&\qquad\qquad\qquad\quad \text{else value} = \bot : \alpha.\text{label}\cap\beta.\text{label}\\
&\qquad\qquad\quad \text{else if label} \in \text{dom(father)}\\
&\qquad\qquad\qquad\quad \text{then value} = \text{father.label} : \alpha.\text{label}\\
&\qquad\qquad\qquad\quad \text{else value} = \text{mother.label} : \beta.\text{label})\ \}
\end{aligned}$$

which yields an intersection record type $\alpha\cap\beta$. Henceforward $\otimes$ will be used as an infix abbreviation for *merge*, and is understood to have the meaning:

$$\begin{aligned}
\otimes = \{\ \otimes_{\alpha,\beta} \mid\ & \forall\alpha.\forall(\beta \mid \beta \mathrm{M} \alpha).\\
& \otimes_{\alpha,\beta} : \alpha \to \beta \to \alpha\cap\beta\ \wedge\ \otimes_{\alpha,\beta} = \text{merge}\ [\alpha\ \beta]\ \}
\end{aligned}$$

The definition of *merge* introduces a typed value $\bot$. This is a placeholder corresponding to the *undefined*, or *empty* value. The *empty* value is inserted into the result of $\otimes$ when it cannot be determined automatically which of two implementations to choose. The empty value is also used later to stand for a *deferred* function implementation.

6.2.3 Multiple Inheritance

A relatively simple modification to the inheritance construction used in chapter 5 for mixins will now allow the typing of multiple inheritance. The essential difference is in replacing the unilateral override operator $\oplus$ by the symmetrical operator $\otimes$.

To illustrate various forms of conflict resolution during multiple inheritance, the existence of an *object* class and a 2D *point* class are assumed, whose definitions are repeated here:

$$\Phi\text{OBJECT} = \Lambda\sigma.\{\text{identity: }\sigma,\ \text{equal: }\sigma \to \text{BOOLEAN}\}$$

$$\Phi\text{object} : \forall(t \subseteq \Phi\text{OBJECT}\ [t]).t \to \Phi\text{OBJECT}\ [t]$$

$$\begin{aligned}
&\Phi\text{object} = \Lambda(t \subseteq \Phi\text{OBJECT}\ [t]).\lambda(\text{self: }t).\\
&\qquad \{\text{identity} \mapsto \text{self},\ \text{equal} \mapsto \lambda(\text{other: }t).(\text{self} = \text{other})\}
\end{aligned}$$

$$\begin{aligned}
\Phi\text{POINT} = \Lambda\sigma.\{&\text{x: INTEGER, y: INTEGER,}\\
&\text{identity: }\sigma,\ \text{equal: }\sigma \to \text{BOOLEAN}\}
\end{aligned}$$

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

$$\Phi point = \Lambda(t \subseteq \Phi POINT\ [t]).\lambda(self{:}\ t).$$
$$(\Phi object\ [t]\ (self)\ \oplus\ \{x \mapsto 0,\ y \mapsto 0,\ equal \mapsto \lambda(other{:}\ t).(self.x = other.x \wedge self.y = other.y)\})$$

and a *z-coordinate* class is introduced, representing the family of all those objects having a scalar quantity in the third dimension. This is different from the *z-coordinate* mixin from chapter 5, which was a class extension function; here the *z-coordinate* is a proper class that inherits all the behaviour of the *object* class:

$$\Phi ZCOORD = \Lambda\sigma.\{z{:}\ INTEGER,\ identity{:}\ \sigma,\ equal{:}\ \sigma \rightarrow BOOLEAN\}$$

$$\Phi zcoord : \forall(t \subseteq \Phi ZCOORD\ [t]).t \rightarrow \Phi ZCOORD\ [t]$$

$$\Phi zcoord = \Lambda(t \subseteq \Phi ZCOORD\ [t]).\lambda(self{:}\ t).$$
$$(\Phi object\ [t]\ (self)\ \oplus\ \{z \mapsto 0,\ equal \mapsto \lambda(other{:}\ t).(self.z = other.z)\})$$

A 3D *point* class may now be derived by multiple inheritance from the 2D *point* and *z-coordinate* classes. Conceptually, these classes describe overlapping spaces of objects with *at least* the services of a 2D *point* and a *z-coordinate*, respectively. It is useful to think of a 3D *point* class as the greatest lower bound, or intersection, of the two spaces.

The methods to be inherited are distributed in the following way: from Φ*point* we may obtain *x* and *y* uniquely; from Φ*zcoord* we may obtain *z* uniquely. Both classes offer versions of *identity* and *equal()*, which require resolution. In the case of *identity*, neither class has redefined this method since it was inherited from Φ*object*; but in the case of *equal()*, both classes have redefined the method; it is impossible to determine which version to inherit. In fact, it would be incorrect to choose either: the class Φ*3dpoint* must determine how to combine the inherited methods in this case.

First, the behaviour of the $\otimes$ operator is illustrated. The Φ*point* and Φ*zcoord* generators must both be specialised to generators having the same *self* and *self*-type. As before, this may be achieved by distributing the *self* and *self*-type of the new class to both generators:

$$\forall(t \subseteq \Phi 3DPOINT\ [t]).\forall(self{:}\ t).$$
$$\Phi point\ [t]\ (self)\ \otimes\ \Phi zcoord\ [t]\ (self)$$

$$= \{x \mapsto 0,\ y \mapsto 0,\ identity \mapsto self,\ equal \mapsto \lambda(other{:}\ t).(self.x = other.x \wedge self.y = other.y)\}$$
$$\otimes\ \{z \mapsto 0,\ identity \mapsto self,\ equal \mapsto \lambda(other{:}\ t).(self.z = other.z)\}$$

$$= \{x \mapsto 0,\ y \mapsto 0,\ z \mapsto 0,\ identity \mapsto self,\ equal \mapsto \bot\}$$

Where $\otimes$ encounters the two methods *identity*, they both have the same type: *identity: t* and the same implementation: *identity* $\mapsto$ *self*. When two identical methods are encountered, $\otimes$ arbitrarily selects one copy from the left-hand record argument, effectively merging the methods. Where $\otimes$ encounters the

two methods *equal()*, these have the same type: *equal:* $t \rightarrow$ BOOLEAN, but have different implementations. In this case, $\otimes$ refuses to choose an implementation, but inserts an undefined value $\perp$ having the type *equal:* $t \rightarrow$ BOOLEAN.

The inheriting class must specify how the two conflicting implementations are to be combined. This requires a construction similar to that for resolving *super* references. We abstract over each parent class internally in the multiple inheritance construction for 3D points. The names of the parents can be anything - here, *father* and *mother* are the names chosen:

$$\Phi\text{3DPOINT} = \Lambda\sigma.\{\text{x: INTEGER, y: INTEGER, z: INTEGER, identity: } \sigma, \text{ equal: } \sigma \rightarrow \text{BOOLEAN}\}$$

$$\Phi\text{3dpoint} : \forall(t \subseteq \Phi\text{3DPOINT}\ [t]).t \rightarrow \Phi\text{3DPOINT}\ [t]$$

$$\begin{aligned}&\Phi\text{3dpoint} = \Lambda(t \subseteq \Phi\text{3DPOINT}\ [t]).\lambda(\text{self: } t).\\&\quad(\lambda(\text{father: } \Phi\text{POINT}\ [t]).\lambda(\text{mother: } \Phi\text{ZCOORD}\ [t]).\\&\qquad(\text{father} \otimes \text{mother}) \oplus\\&\qquad\quad\{\text{equal} \mapsto \lambda(\text{other: } t).\\&\qquad\qquad(\text{father.equal(other)} \wedge \text{mother.equal(other)})\}\\&\quad(\Phi\text{point}\ [t]\ (\text{self}))\ (\Phi\text{zcoord}\ [t]\ (\text{self})))\end{aligned}$$

An extension record for 3D *points* determines how the inherited *equal()* methods are to be combined - obviously, the desired combination is the logical *and* of the two. Inside the extension record, free reference is made to the recursion variables *father* and *mother*, which are bound in the result to objects of appropriate types.

This example shows the resolution of a difficult case for multiple inheritance; the solution is most elegant, making maximum reuse of inherited methods. Clearly, other kinds of resolution are possible, depending on the semantics of the methods to be combined - a child class could simply invoke one or other parent method. It is a strength of this approach that the child may determine how resolution is performed at the *join-point* in the inheritance heterarchy. For comparison, *C++* forces this decision at the *fork-point* [Stro91], sometimes only retrospectively, entailing the revision of previously finished code.

6.2.4 Multiple Mixin Inheritance

The most difficult case for multiple inheritance is where two parent methods themselves invoke *super* methods. Since recombination only arises where two methods redefine a common original method, there is always the possibility that both the father and mother implementations are wrappers specialising a

common ancestor method; and the danger then exists that this method may be invoked twice if the father and mother methods are subsequently recombined. In our scheme, there is no way to prevent this directly. This is because we insist that the *super* mechanism is equivalent to its inline expansion. An important property of inheritance is that all shorthand constructions should be internally reducible.

Certain languages like Flavors and CLOS [Moon86, Keen89] adopt a finer-grained labelling: *primary*, *before* and *after*, for methods which have the same basic name. This allows a further semantic categorisation of methods into main, pre- and post-processing parts. To solve the above problem, the common ancestor method would be labelled *primary* and the mother and father extensions would be *after* methods. The method combination rule would ensure that, in the child class, the *primary* method was invoked once, followed by all *after* methods. We could simulate this approach by expanding the method name-space (essentially, this is what the finer-grained labelling achieves). The main method *m* could invoke *m-pre* and *m-post*, which have empty definitions. The father and mother classes could redefine *m-post* and the child class could happily recombine these two using the normal scheme. This means that *m* inherited from the common ancestor would invoke *self.m-post* in the child, performing all post-processing parts.

However, admitting that a method can be split into main and post-processing parts indicates a weakness in the posing of the problem. If *father* and *mother* only add post-processing, they should be defined as mixins rather than as full classes. To illustrate this, a Φ*load* class is defined which has a basic *equal()* method which should always be invoked (once) in all eventual subclasses which specialise the method:

$$\Phi\text{LOAD} = \Lambda\sigma.\{\text{kg: INTEGER, identity: } \sigma, \text{ equal: } \sigma \rightarrow \text{BOOLEAN}\}$$

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

$$\Phi\text{load} = \Lambda(t \subseteq \Phi\text{LOAD}\ [t]).\lambda(\text{self: } t).\ \{\text{kg} \mapsto 0, \text{ identity} \mapsto \text{self, equal} \mapsto \lambda(\text{other: } t).(\text{self.kg} = \text{other.kg})\}$$

Let us assume that various specialised classes are required to move in one-, two- and three-dimensional space. For this, an *xy-coordinate* bound mixin may be used in combination with a *z-coordinate* bound mixin:

$$\Sigma\text{xycoord} : \forall(\varepsilon \subseteq \Delta\text{XYCOORD}\ [\varepsilon]).\forall(\beta \mid \varepsilon \subset \beta \subseteq \Phi\text{EQUAL}\ [\varepsilon]).\ \varepsilon \rightarrow \beta \rightarrow \beta \cap \varepsilon$$

$$\Sigma\text{xycoord} = \Lambda(\varepsilon \subseteq \Delta\text{XYCOORD}\ [\varepsilon]).\Lambda(\beta \mid \varepsilon \subset \beta \subseteq \Phi\text{EQUAL}\ [\varepsilon]).\ \lambda(\text{self: } \varepsilon).\lambda(\text{super: } \beta).\ \text{super} \oplus \{x \mapsto 0, y \mapsto 0, \text{equal} \mapsto \lambda(\text{other: } \varepsilon).(\text{super.equal(other)} \wedge \text{self.x} = \text{other.x} \wedge \text{self.y} = \text{other.y})\}$$

$$\Sigma zcoord : \forall(\varepsilon \subseteq \Delta ZCOORD\ [\varepsilon]).\forall(\beta \mid \varepsilon \subset \beta \subseteq \Phi EQUAL\ [\varepsilon]).\ \varepsilon \rightarrow \beta \rightarrow \beta \cap \varepsilon$$

$$\Sigma zcoord = \Lambda(\varepsilon \subseteq \Delta ZCOORD\ [\varepsilon]).\Lambda(\beta \mid \varepsilon \subset \beta \subseteq \Phi EQUAL\ [\varepsilon]).\ \lambda(self{:}\ \varepsilon).\lambda(super{:}\ \beta).\ super \oplus \{z \mapsto 0,\ equal \mapsto \lambda(other{:}\ \varepsilon).\ (super.equal(other) \wedge self.z = other.z)\}$$

The salient fact about these mixins is that they both specialise *equal()* independently. Now, any combination of Φ*load* and these mixins may be provided. For example, a 2D *load* is given by applying the Σ*xycoord* mixin function:

$$\Phi 2dload : \forall(t \subseteq \Phi 2DLOAD\ [t]).t \rightarrow \Phi 2DLOAD\ [t]$$

$$\Phi 2dload = \Lambda(t \subseteq \Phi 2DLOAD\ [t]).\lambda(self{:}\ t).\ \Sigma xycoord\ [t, \Phi LOAD\ [t]]\ (self, \Phi load\ [t]\ (self))$$

and a 1D *load* is given by a similar application of the Σ*zcoord* mixin function. The more interesting 3D *load* is given by stacking up the two mixins. The Σ*zcoord* mixin is applied to the result generated by applying the Σ*xycoord* mixin:

$$\Phi 3dload : \forall(t \subseteq \Phi 3DLOAD\ [t]).t \rightarrow \Phi 3DLOAD\ [t]$$

$$\Phi 3dload = \Lambda(t \subseteq \Phi 3DLOAD\ [t]).\lambda(self{:}\ t).\ \Sigma zcoord\ [t, \Phi 2DLOAD\ [t]]\ (self, \Phi 2dload\ [t]\ (self))$$

Arguably, this expresses in a more natural way that *equal()* in the 3D *load* class is a combination of a main method and postprocessing. The order of mixins is in general significant, although in this example logical *and* commutes. In any case, the Φ*3dload* class is nonetheless a subclass of both Φ*2dload* and Φ*1dload*, no matter in which order mixins are combined. By expanding inline all *super* method invocations, it may be demonstrated that each *equal()* method provided in the classes Φ*1dload*, Φ*2dload* and Φ*3dload* incorporates Φ*load*'s basic *equal()* method only once.

6.3 Higher Classification

A *higher class* is defined as one whose member objects have the types of type constructors, in addition to various simple types. The hallmark of a type constructor is that it abstracts over some internal component type. An object with the type of a type constructor is therefore one whose field-types are not all fixed; in other words, the members of a higher class are those objects with polymorphic components.

Polymorphism appears in many guises in object-oriented languages; yet the theoretical treatment of polymorphism is often incorrect and unsatisfying. We

see in the notion of *higher classification* the possibility of uniting all treatments of polymorphism under a single scheme.

6.3.1 Polymorphism and Overloading

Polymorphism is sometimes confused with *dynamic binding*, the invocation of an object-specific response to a generic message, enabled through the multiple overloading of method names. Here the issues of redefinition, typing and binding are distinguished. A polymorphic type is technically a family of types and a polymorphic function is one which may apply to objects of different types (see chapter 2). Mathematically, polymorphism is always indicated by the presence of one or more type parameters.

A class is an inherently polymorphic construction. Since σ, the type of *self*, is not fixed, it may range over a whole family of types $\forall(\sigma \subseteq \Phi F[\sigma])$. The *identity* method above is therefore considered polymorphic, since even though it has a single implementation, its type changes when it is inherited. For OBJECTs and POINTs, *identity* has the types:

$$\text{identity} : \forall(\sigma \subseteq \Phi\text{OBJECT}\ [\sigma]).\sigma \rightarrow (\sigma)$$
$$\text{identity} : \forall(\sigma \subseteq \Phi\text{POINT}\ [\sigma]).\sigma \rightarrow (\sigma)$$

This is similar in spirit to Strachey's *parametric polymorphism*, since it can be shown that the method *identity* works uniformly over all types in the family bounded by $\forall(\sigma \subseteq \Phi\text{OBJECT}\ [\sigma])$. In contrast, many different *equal()* methods have been defined and introduced at many points in the heterarchy. Superficially, this is similar to Strachey's *ad hoc* polymorphism. However, in a regime with multiple classification, it is always possible to relate variants of the *equal()* function back to a common ancestor, perhaps that defined in an abstract class $\forall(\sigma \subseteq \Delta\text{EQUAL}\ [\sigma])$, which has the type:

$$\text{equal} : \forall(\sigma \subseteq \Delta\text{EQUAL}\ [\sigma]).\sigma \rightarrow (\sigma \rightarrow \text{BOOLEAN})$$

Even though there are many different implementations of *equal()*, it can be argued that they all behave in a semantically uniform way over the whole family of types $\forall(\sigma \subseteq \Delta\text{EQUAL}\ [\sigma])$. In practice, functions like *equal()* are not introduced singly, but in general, useful classes like $\forall(\sigma \subseteq \Phi\text{OBJECT}\ [\sigma])$, which introduce groups of functions, like *equal()* and *identity*, in one place. Semantic uniformity may then be enforced by insisting that all other classes with these methods are subclasses of $\forall(\sigma \subseteq \Phi\text{OBJECT}\ [\sigma])$.

Separating the binding and typing issues allows the treatment of quite diverse kinds of polymorphism under the same scheme. Redefinition that is subject to F-bounded inclusion is the special kind of overloading that occurs in object-oriented languages. We consider it inappropriate for our language model to handle other kinds of overloading, such as arbitrarily-extensible global functions whose variants are selected by the types of their arguments [Stro91]. This is a

purely syntactic mechanism for increasing the size of the name-space; and could be simulated using longer names. Overloading the function name-space *within* a class also interferes with the polymorphic selection mechanism for binding [Meys92, p183, 192-3; TW95, p352].

6.3.2 Polymorphism and Genericity

Here, we seek to understand the issues surrounding polymorphism in some other component type τ than the *self*-type σ. The obvious differences between classical universal polymorphism [Gira72, Reyn74, Reyn83] and class-based F-bounded polymorphism are in the scope and bounds of the parameters:

- τ abstracts over part of a type, whereas σ abstracts over the whole type;
- τ is unconstrained, whereas σ is constrained by an F-bound.

Some object-oriented languages offer a parametric polymorphism similar to the classical kind. *Eiffel* introduced a *generic parameter* mechanism based on *Ada*'s generic packages [IBHK79, Meye88], which influenced the *C++ template* mechanism [Stro91], allowing abstraction over internal parts of a type. It was demonstrated above how the universal polymorphism in languages like *ML* [Miln78, MTH90] and *Ada* is just a special case of F-bounded polymorphism, which may be shown using the translation:

$$\forall\tau.f(\tau) \;\Leftrightarrow\; \forall(\tau \subseteq \Phi TOP[\tau]).f(\tau)$$

Eiffel has subsequently introduced *constrained generic parameters* [Meye92], which are explained completely by the F-bounded generalisation. In *Eiffel*, a generic parameter T constrained using syntax of the form [T -> OBJECT] may only be replaced by some type inheriting from OBJECT, which is exactly the constraint: $\forall(\tau \subseteq \Phi OBJECT[\tau])$. F-bounds are therefore sufficient to describe polymorphism both in component types τ and the *self*-type σ.

6.3.3 Polymorphism and Conformance

Unfortunately, the mixing of classes and parametric polymorphism is not well-achieved in current languages. Explicit type parameters are most appropriate if all other types are strictly monomorphic. However, many object-oriented languages are traditionally lax in their treatment of monomorphic types. The notion of *conformance* allows monomorphic variables to behave in quasi-polymorphic ways: child class objects may be substituted where parent class objects were expected. This is often misunderstood as a kind of *subtyping*, when in fact it is a *subclassing* constraint, which requires the full machinery of F-bounds to explain completely.

The difference between conformance and genuine polymorphism is illustrated in figure 6.4 with an *Eiffel* class POINT which abstracts over the precise numerical type to be given to its coordinates:

class POINT **creation** make **feature** {ANY} x, y : NUMERIC; make (nx, ny : NUMERIC) **is** **do** x := nx; y := ny **end** **end** -- *POINT*	**class** POINT [T -> NUMERIC] **creation** make **feature** {ANY} x, y : T; make (nx, ny : T) **is** **do** x := nx; y := ny **end** **end** -- *POINT*

Figure 6.4: Variations on a Polymorphic Theme

The left-hand POINT class is not parameterised; polymorphism of a rudimentary kind is based on conformance. Some *p* : POINT may be initialised using *p.make(3,4)*; or else using *p.make(3.2, 4.1)*. Any type of object that conforms to NUMERIC may be assigned to *x* and *y*, even one each of INTEGER and REAL! Whatever exact coordinate types *x* and *y* contain, we may only infer that they have the quasi-polymorphic type NUMERIC. The right-hand POINT class is parameterised. Some *p* : POINT [INTEGER] can be initialised using *p.make(3,4)*; and some *p* : POINT [REAL] using *p.make(3.2, 4.1)*. In principle, parameterisation forces the objects assigned to *x* and *y* to have the same type, which must be some T conforming to NUMERIC. In principle, we may always infer the exact types of *x* and *y* as a result of parameter instantiation.

For reasons given before (see chapter 4), the *conformance* approach to polymorphism is suspect. Conformance is not type-sound where it pretends to be subtyping; furthermore, as the example above suggests, it also leads to type-loss. The only really indispensible use for conformance-based polymorphism is to type a dynamically-bound call site, something that may be preserved by other means. Perhaps the most damning indictment against conformance is that it subverts entirely the strong typing constraint offered by parametric polymorphism. The Eiffel variable *p* : POINT [INTEGER] is not monomorphic as one would hope, but may still receive objects that are more specific than POINT (*eg* HOT_POINT) and which contain coordinates with more specific types than INTEGER (*eg* SMALL_INTEGER). Finally, the redundancy of having both kinds of polymorphism is illustrated by comparing the left-hand POINT and the right-hand instantiation POINT [NUMERIC], which is exactly the same type.

6.3.4 A Single Polymorphic Mechanism

Our goal is a single mechanism to describe polymorphism in object-oriented languages. Ordinary simple types are to be strictly monomorphic. Polymorphic types will be expressed everywhere using F-bounded parameters. Mechanisms will eventually allow type parameters to be replaced either at compile-time or run-time, to obtain maximum static type information on the one hand while retaining the possibility of dynamic binding on the other.

A class containing polymorphic parts must be constructed carefully to reflect the dependency of one type on another. This is illustrated with a generalisation of

our earlier *point* class, over whose coordinate type we now abstract. The type function ΦPOINT binds the coordinate-type τ and the *self*-type σ:

$$\Phi\text{POINT} = \Lambda\tau.\Lambda\sigma.\{x\text{: } \tau,\ y\text{: } \tau,\ \text{identity: } \sigma,\ \text{equal: } \sigma \to \text{BOOLEAN}\}$$

To make *points* contain homogenous coordinates, *x* and *y* must both be given the same type τ. To close *point*'s *self*-type over the coordinate type, τ must be bound before σ, expressing the dependency of σ upon τ. The constructor for typed *points* puts F-bounds on valid types for σ and τ:

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

$$\begin{aligned}&\Phi\text{point} = \Lambda(t \subseteq \Phi\text{NUMERIC } [t]).\Lambda(s \subseteq \Phi\text{POINT } [t, s]).\lambda(\text{self: } s).\\ &\quad \{x \mapsto \bot,\ y \mapsto \bot,\ \text{identity} \mapsto \text{self},\\ &\qquad\quad \text{equal} \mapsto \lambda(\text{other: } s).(\text{self.x} = \text{other.x} \wedge \text{self.y} = \text{other.y})\}\end{aligned}$$

This function accepts a type $t \subseteq \Phi$NUMERIC [t] and produces a typed generator for *point* objects. This may be demonstrated by applying Φ*point* to some suitable type, such as INTEGER, and inspecting the type of the resulting generator:

$$\begin{aligned}&\Phi\text{point [INTEGER]} : \forall(s \subseteq \Phi\text{POINT [INTEGER, s]}).\\ &\qquad\qquad\qquad\qquad s \to \Phi\text{POINT [INTEGER, s]}\end{aligned}$$

As a consequence of abstracting over their type, the *x* and *y* fields may only be given an *undefined* value $\bot$, which initially has the type $\forall(t \subseteq \Phi$NUMERIC [t]). By considering the application Φpoint [INTEGER], it should be obvious that this distributes the type INTEGER internally to the type function ΦPOINT, resulting in an adapted type generator for *integer points*:

$$\begin{aligned}&\Phi\text{POINT [INTEGER]} = \Lambda\sigma.\{x\text{: INTEGER},\ y\text{: INTEGER},\\ &\qquad\qquad\qquad\qquad \text{identity: } \sigma,\ \text{equal: } \sigma \to \text{BOOLEAN}\}\end{aligned}$$

in which the *x* and *y* fields are retyped as we would expect. The result of applying Φpoint [INTEGER] therefore has the form of a *point* generator:

$$\begin{aligned}&\Phi\text{point [INTEGER]} = \Lambda(s \subseteq \Phi\text{POINT [INTEGER, s]}).\lambda(\text{self: } s).\\ &\quad \{x \mapsto \bot,\ y \mapsto \bot,\ \text{identity} \mapsto \text{self},\\ &\qquad\quad \text{equal} \mapsto \lambda(\text{other: } s).(\text{self.x} = \text{other.x} \wedge \text{self.y} = \text{other.y})\}\end{aligned}$$

in which *x* and *y*, though their values are still undefined, have acquired the type INTEGER. It is pleasing that the quantification for this *integer point* class is now: $\forall(s \subseteq \Phi$POINT [INTEGER, s]). Essentially this restricts the class to those *point* types whose coordinates are INTEGERs. There may of course be types in this class with strictly more functions, such as a 3D *integer point*.

Binding type parameters in the order τ, σ yields first the class of *integer points* and then by taking the fixpoint we obtain the exact type INTEGER_POINT:

$$\text{INTEGER_POINT} = (\Upsilon\ (\Phi\text{POINT [INTEGER]}))$$

$$\text{integer_point} = (\Upsilon\ (\Phi\text{point}\ [\text{INTEGER, INTEGER_POINT}]))$$

and this more or less reflects the *generic parameter* style of binding, in which complete type information is supplied at compile time. However, object-oriented languages are also characterised by their ability to handle unresolved polymorphic types at run-time. To obtain a polymorphic point whose coordinate type is unfixed requires a simple trick of parameter redistribution in the λ-calculus, in order to fix the type of *self*, while leaving the coordinate type open:

$$\text{POLY_POINT} = \Lambda(u \subseteq \Phi\text{NUMERIC}\ [u]).(\Upsilon\ (\Phi\text{POINT}\ [u]))$$

$$\text{poly_point} = \Lambda(v \subseteq \Phi\text{NUMERIC}\ [v]).(\Upsilon\ (\Phi\text{point}\ [v, \text{POLY_POINT}\ [v]]))$$

POLY_POINT is a type constructor in the Girard-Reynolds style, generalised to accept F-bounded type arguments. The instance *poly_point* is a point whose coordinates are of some unresolved type $\forall(v \subseteq \Phi\text{NUMERIC}\ [v])$. Clearly, this last example demonstrates that it is possible to capture late type binding in the style of *conformance*, but in a sounder mathematical framework.

6.3.5 Higher-Order Inheritance

Generalising inheritance to handle classes with polymorphic components requires a higher-order override constraint Ω to type the record combination operator $\oplus$. This is because Ω must compare fields having simple and polymorphic types; and type parameters may range over classes with further embedded polymorphic components.

A suitably complicated example of higher-order inheritance is constructed by inheriting from our parameterised class of 2D *points* in order to define a more extended parameterised class of 3D *points*, while at the same time restricting the bound on the polymorphic coordinate type:

$$\Phi\text{3DPOINT} = \Lambda\tau.\Lambda\sigma.\{x: \tau,\ y: \tau,\ z: \tau,\ \text{identity}: \sigma,\ \text{equal}: \sigma \rightarrow \text{BOOLEAN}\}$$

$$\begin{aligned}\Phi\text{3dpoint} : \forall(t \subseteq \Phi\text{INTEGER}\ [t]).\forall(s \subseteq \Phi\text{3DPOINT}\ [t, s]).\\ s \rightarrow \Phi\text{3DPOINT}\ [t, s]\end{aligned}$$

$$\begin{aligned}&\Phi\text{3dpoint} = \Lambda(t \subseteq \Phi\text{INTEGER}\ [t]).\Lambda(s \subseteq \Phi\text{3DPOINT}\ [t, s]).\lambda(\text{self}: s).\\ &\quad(\lambda(\text{super}: \Phi\text{POINT}\ [t, s]).\\ &\qquad \text{super} \oplus \{x \mapsto 0,\ y \mapsto 0,\ z \mapsto 0,\\ &\qquad\qquad \text{equal} \mapsto \lambda(\text{other}: s).(\text{super.equal(other)}\\ &\qquad\qquad\qquad \wedge\ \text{self.z} = \text{other.z})\}\\ &\quad(\Phi\text{point}\ [t, s]\ (\text{self})))\end{aligned}$$

The resulting 3D *point* class will admit of further extensions, but will only allow its coordinates to be filled with objects whose type satisfies the bound $\forall(t \subseteq \Phi\text{INTEGER}[t])$, such as SMALL_INTEGER or INTEGER. The technique for method-combination is used again here. The modified parent record is obtained by Φpoint [t, s] (self). The type-application Φpoint [t] is correct, since any integer type *t* satisfying the new bound will satisfy the old bound:

$$\forall(t \subseteq \Phi\text{INTEGER}[t]).\Phi\text{INTEGER}[t] \subseteq \Phi\text{NUMERIC}[t]$$

The dependent type application Φpoint [t, s] is therefore also correct, since all 3D *integer* points are also members of the wider *integer* point class:

$$\forall(t \subseteq \Phi\text{INTEGER}[t]).\forall(s \subseteq \Phi\text{3DPOINT}[t, s]).\\ \Phi\text{3DPOINT}[t, s] \subseteq \Phi\text{POINT}[t, s]$$

Finally, Φpoint [t, s] (self) produces a value in this type, which is bound internally to *super*. Consider now the operation of $\oplus$, the higher-order record combination operator. It is given two records to combine:

$$\{x \mapsto \perp, y \mapsto \perp, \text{identity} \mapsto \text{self}, \text{equal} \mapsto \lambda(\text{other: s}).\\ (\text{self.x} = \text{other.x} \wedge \text{self.y} = \text{other.y})\}$$

$$\oplus \ \{x \mapsto 0, y \mapsto 0, z \mapsto 0, \text{equal} \mapsto \lambda(\text{other: s}).\\ (\text{super.equal(other)} \wedge \text{self.z} = \text{other.z})\}$$

whose fields now have the types:

$$\forall(t \subseteq \Phi\text{INTEGER}[t]).\forall(s \subseteq \Phi\text{3DPOINT}[t, s]).\\ \{x: t, y: t, \text{identity: s}, \text{equal: s} \rightarrow \text{BOOLEAN}\}\\ \oplus \ \{x: t, y: t, z: t, \text{equal: s} \rightarrow \text{BOOLEAN}\}$$

The rule ΩRECD requires common fields to satisfy Ω in turn. Common fields are *x, y* and *equal()*. These trivially satisfy the rule ΩPARM since all pairwise comparable signatures have the same bounded parameter.

This example also overrides the *undefined* values $\perp$ for *x, y*, which were polymorphically typed $\forall(t \subseteq \Phi\text{NUMERIC}[t])$, with defined values 0 in the more specific type $\forall(t \subseteq \Phi\text{INTEGER}[t])$. This captures exactly the *deferred feature* mechanism in *Eiffel* [Meye88, Meye92] and *pure virtual* mechanism in *C++* [Stro91]. Signatures may be given for methods whose implementation is only supplied later in descendants.

6.3.6 Inheritance with Construction and Abstraction

The extended model will permit a slightly different style of inheritance with higher classes, in which we choose partially to construct and then to inherit, in the style:

$$\Phi\text{3DINT_POINT} = \Lambda\sigma.\{x: \text{INTEGER}, y: \text{INTEGER}, z: \text{INTEGER}, \text{identity}: \sigma, \text{equal}: \sigma \to \text{BOOLEAN}\}$$

$$\Phi\text{3dint_point} : \forall(s \subseteq \Phi\text{3DINT_POINT}\ [s]).s \to \Phi\text{3DINT_POINT}\ [s]$$

$$\Phi\text{3dint_point} = \Lambda(s \subseteq \Phi\text{3DINT_POINT}\ [s]).\lambda(\text{self}: s).$$
$$(\lambda(\text{super}: \Phi\text{POINT}\ [\text{INTEGER}, s]).$$
$$\text{super} \oplus \{x \mapsto 0, y \mapsto 0, z \mapsto 0,$$
$$\text{equal} \mapsto \lambda(\text{other}: s).(\text{super.equal(other)} \wedge \text{self.z} = \text{other.z})\}$$
$$(\Phi\text{point}\ [\text{INTEGER}, s]\ (\text{self})))$$

Here, the parameterised class of 2D *points* is applied to an argument for the coordinate type, INTEGER. From the earlier discussion on binding, it is clear that this constructs an *integer point* generator which can be extended in the normal way. The example creates a class of 3D *integer points* to illustrate full use of the *super* mechanism.

It seems clear that higher-order inheritance can be used if the child class has the same number, or fewer type parameters than its parent. It is more difficult to see how a child class can introduce additional type parameters. It turns out that this is not too difficult. Further internal type abstraction may be introduced when deriving the parameterised class of 2D *points* from our basic *object* class:

$$\Phi\text{POINT} = \Lambda\tau.\Lambda\sigma.\{x: \tau, y: \tau, \text{identity}: \sigma, \text{equal}: \sigma \to \text{BOOLEAN}\}$$

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

$$\Phi\text{point} = \Lambda(t \subseteq \Phi\text{NUMERIC}\ [t]).\Lambda(s \subseteq \Phi\text{POINT}\ [t, s]).\lambda(\text{self}: s).$$
$$(\Phi\text{object}\ [s]\ (\text{self}) \oplus \{x \mapsto \bot, y \mapsto \bot,$$
$$\text{equal} \mapsto \lambda(\text{other}: s).(\text{self.x} = \text{other.x} \wedge \text{self.y} = \text{other.y})\})$$

Here, even though Φobject [s] only accepts one type parameter, this is the *self*-type of *points*, which is bound over the coordinate type *t*. This may be thought of as an implicit type dependency in the *object* class which is made explicit later in the parameterised 2D *point* class. Our formulation of inheritance allows type abstraction to be introduced in an intuitive way.

6.3.7 Higher-Order Multiple Inheritance

Finally, it is important to observe the behaviour of the higher-order *merge* constraint M during multiple inheritance. We repeat our earlier example of inheriting multiply from a 2D *point* and a *z-coordinate* class, over whose coordinate types we abstract here, in order to derive the greatest lower bound parameterised 3D *point* class. The example assumes the existence of a parameterised 2D *point* class, as described in the previous section, whose *x* and *y* fields are constrained by the bound $\forall(t \subseteq \Phi\text{NUMERIC}\ [t])$. Another parameterised *z-coordinate* class is given, which also inherits from *object*, but which introduces a different bound $\forall(t \subseteq \Phi\text{REAL}\ [t])$ on its *z* field:

$$\Phi\text{ZCOORD} = \Lambda\tau.\Lambda\sigma.\{\text{z: } \tau, \text{ identity: } \sigma, \text{ equal: } \sigma \rightarrow \text{BOOLEAN}\}$$

$$\Phi\text{zcoord} : \forall(t \subseteq \Phi\text{REAL}\ [t]).\forall(s \subseteq \Phi\text{ZCOORD}\ [t, s]).\ s \rightarrow \Phi\text{ZCOORD}\ [t, s]$$

$$\Phi\text{zcoord} = \Lambda(t \subseteq \Phi\text{REAL}\ [t]).\Lambda(s \subseteq \Phi\text{ZCOORD}\ [t, s]).\lambda(\text{self: } s).\ (\Phi\text{object}\ [s]\ (\text{self}) \oplus \{z \mapsto 0.0, \text{ equal} \mapsto \lambda(\text{other: } s).\ (\text{self.z} = \text{other.z})\})$$

Inheriting multiply from these classes is complicated by the fact that they have different bounds on their polymorphic parameters. To establish whether the parent classes *meet*, MPARM tells us to compute their intersection type:

$$\forall(t \subseteq \Phi\text{NUMERIC}\ [t]).\forall(s \subseteq \Phi\text{POINT}\ [t, s]).\ \forall(\tau \subseteq \Phi\text{REAL}\ [\tau]).\forall(\sigma \subseteq \Phi\text{ZCOORD}\ [\tau, \sigma]).\ s \cap \sigma$$

The recursive definition of MRECD will dig through any parameterised types on which the two *self*-types eventually depend, computing intersections until simple types are reached. Here, common fields are only in the *self*-type. To compute $s \cap \sigma$ this must be transformed into a greatest lower bounds condition:

$$\forall(t \subseteq (\Phi\text{NUMERIC}\ [t] \cap \Phi\text{REAL}\ [t])).\ \forall(s \subseteq (\Phi\text{POINT}\ [t, s] \cap \Phi\text{ZCOORD}\ [t, s]))$$

Assuming $\forall(t \subseteq \Phi\text{REAL}\ [t]).\Phi\text{REAL}\ [t] \subseteq \Phi\text{NUMERIC}\ [t]$, this simplifies to:

$$\forall(t \subseteq \Phi\text{REAL}\ [t]).\forall(s \subseteq (\Phi\text{POINT}\ [t, s] \cap \Phi\text{ZCOORD}\ [t, s]))$$

Now, with all other type intersections determined, the final one may be calculated by merging the two generators. Calling this Φ3DPOINT:

$$\Phi\text{3DPOINT} = \Lambda\tau.\Lambda\sigma.\{\text{x: } \tau, \text{ y: } \tau, \text{ z: } \tau, \text{ identity: } \sigma, \text{ equal: } \sigma \rightarrow \text{BOOLEAN}\}$$

a well-typed inheritance expression may be defined, yielding a parameterised 3D *point* class with a more restricted bound $\forall(t \subseteq \Phi\text{REAL}\ [t])$ on its coordinate:

$$\Phi\text{3dpoint} : \forall(t \subseteq \Phi\text{REAL}\ [t]).\forall(s \subseteq \Phi\text{3DPOINT}\ [t, s]).\ s \rightarrow \Phi\text{3DPOINT}\ [t, s]$$

$$\begin{aligned}
&\Phi\text{3dpoint} = \Lambda(t \subseteq \Phi\text{REAL}\ [t]).\Lambda(s \subseteq \Phi\text{3DPOINT}\ [t, s]).\lambda(\text{self: } s).\\
&\quad(\lambda(\text{father: } \Phi\text{POINT}\ [t, s]).\lambda(\text{mother: } \Phi\text{ZCOORD}\ [t, s]).\\
&\qquad(\text{father} \otimes \text{mother}) \oplus\\
&\qquad\quad\{x \mapsto 0.0,\ y \mapsto 0.0,\ \text{equal} \mapsto \lambda(\text{other: } s).\\
&\qquad\qquad(\text{father.equal(other)} \wedge \text{mother.equal(other)})\}\\
&\quad(\Phi\text{point}\ [t, s]\ (\text{self}))\ (\Phi\text{zcoord}\ [t, s]\ (\text{self})))
\end{aligned}$$

Again, the method combination technique is used to resolve the conflicting *equal()* methods; and the opportunity is taken to supply effective values for the 2D *poly-point*'s deferred *x* and *y* values. As before, the distribution of multiple new type parameters satisfies old bounds. In particular, this is due to the operation of ⊗ which will not combine two records unless their types intersect. Here, a strategy was adopted of seeking an intersection type for the result according to M that will allow ⊗ to operate without error.

This chapter has developed a general theory of classification, which gives a fairly full understanding of the types and behaviours of objects in object-oriented programming. In passing, intuitive explanations have been found in the λ-calculus for many of the popular features found in object-oriented languages; and occasionally good grounds have been determined for rejecting certain others. A noteworthy achievement of our model is to unite all desirable forms of polymorphism under a single F-bounded scheme. Although much ground has been covered here, there are many loose ends to tidy up. The focus here was on explaining object *behaviour*. The following chapter turns to address the issues in the modelling of object *state* and *identity*.