Chapter 5

A Theory of Classification

A simple theory of classification is presented. The theory is "simple" in the sense that classes are considered second-order generalisations over recursive types. Classes are defined using generators which abstract over the polymorphic type of self, but all other types are assumed to be simple, static types. For completeness' sake, the theory expounds the properties of objects from both the implementation and type perspective. First, classification is defined in terms of ordering relationships between type- and implementation-generators. Then, the process of deriving classes incrementally through inheritance is examined in greater detail .

5.1 Classification and Hierarchy

Classification inevitably carries with it the notion of hierarchy, the subdivision of sets of objects into smaller sets. Classification therefore admits of an ordering relationship between classes. Here, the first part of our theory seeks to establish the exact nature of that ordering relationship.

The notion of *class* cannot be properly developed without considering the properties of objects from both the implementation and type perspectives. In the same way that *types* can be considered from a concrete or abstract point of view, we see no reason why these two aspects of *class* cannot be treated together, in contrast with [Snyd87, Amer90]. It is reasonable to suppose that transformations upon an object's structure will be reflected by transformations in its type. Our model eventually links the type of an object with its implementation, in the style proposed in [CHC90].

5.1.1 Typed Objects

We adopt a Cook-style [Cook89a, CCHO89b, CHC90] typed functional model to represent objects and their associated types. An object is written as a labelled record of functions, representing methods, which has a corresponding type:

$$\{a_1 \mapsto e_1, \dots a_n \mapsto e_n\} : \{a_1 \colon t_1, \dots a_n \colon t_n\}$$

Here, the a_i are labels and the e_i are method expressions having the types t_i, which are released from the record by application of an object to a label. A record is essentially an associative map from unique labels to values, which can be modelled using finite functions in the λ-calculus (see Appendix 1).

Access to encapsulated object data is handled through unary methods, which in turn are modelled using nullary functions of the form: $unit \rightarrow v$, where v is the value to be returned. For the sake of uniformity, all methods are assumed to expect an argument of the trivial type UNIT and method invocation is assumed implicitly to apply the selected method to the sole element of this type. Where a_1 is a unary method and a_2 is a binary method expecting one argument v, we assume the following implicit translations:

$$\text{obj}.a_1 \;\Leftrightarrow\; \text{obj}.a_1\ (\text{unit})$$

$$\text{obj}.a_2\ (v) \;\Leftrightarrow\; \text{obj}.a_2\ (\text{unit}, v)$$

In general, an object's methods are mutually recursive, since they may refer to the object as a whole, which we denote by *self*. Methods may accept arguments or return results in the type of *self*, which we denote by σ. A recursive object is written using μ to bind the recursion variable *self*; similarly μ binds the recursion variable σ in recursive types:

$$\mu \text{self}.\{a_1 \mapsto e_1, \dots a_n \mapsto e_n\} : \mu\sigma.\{a_1 \colon t_1, \dots a_n \colon t_n\}$$

The methods e_i may contain free occurrences of *self* and the type signatures of the methods t_i may contain free occurrences of the *self*-type σ. The recursions at term- and type-level are essentially independent, since it is possible for a method to use *self* but not accept or return a value in the type σ; similarly a method may not refer to *self* but still accept or return another object in the same type σ as the *self*-type.

Recursive structures exhibit an *unrolling* or *unfolding* property. A recursive object is equivalent to its infinite expansion; likewise a recursive type:

$$\mu \text{self}.f(\text{self}) \equiv f(\mu \text{self}.f(\text{self}))$$

$$\mu\sigma.F(\sigma) \equiv F(\mu\sigma.F(\sigma))$$

where f(self) and F(σ) are term- and type-expressions standing respectively for the record body and type of an object. Abstracting over the recursive structure of objects and types yields recursive object generators and recursive type generators, which are written as functions of *self* and the *self*-type σ:

λself.f(self) $\Lambda\sigma$.F(σ)

These are functionals whose fixpoints (*ie* least fixed points) correspond to the original recursive objects and types. In order to motivate fixpoints at both the term- and type-level, our theory appeals to Bruce and Mitchell's partial equivalence relations [BM92] for its semantic domains. In order to preserve the convergence of recursive functions representing objects, it must avoid constructing the object containing only the identity value. Using the technique described in chapter 3 involving the trivial type UNIT, all object fields are written as functions, in which case convergence is guaranteed.

With these assumptions, generators may be related to objects and types in the following way:

μself.f(self) = ($\mathbf{Y}$ λself.f(self))

$\mu\sigma$.F(σ) = ($\mathbf{Y}$ $\Lambda\sigma$.F(σ))

where $\mathbf{Y}$ is the fixpoint finder. The treatment below deals mainly with object generators and type generators; it is assumed that the exact recursive structure and recursive type of an object can always be recovered by application of the fixpoint finder.

5.1.2 Object Classes

In object-oriented programming, a *class* describes a family of objects bearing some similarity in structure and type. The exact nature of this similarity will be motivated below by appealing to intuitive notions about class membership. For the moment, let us assert that a class may contain objects having different types. So, whereas the two truncated types:

INTα = {identity: INTEGER, equal: INTEGER $\rightarrow$ BOOLEAN}

CHARα = {identity: CHARACTER, equal: CHARACTER $\rightarrow$ BOOLEAN}

do not stand in any simple type relationship with each other, they both belong to the same class. A class defines an implementation and an interface for the methods of its objects. These are represented using generators to abstract over any particular *self* and *self*-type:

ϕobject = λself.{identity $\mapsto$ self, equal $\mapsto$ λother.(self = other)}

ΦOBJECT = $\Lambda\sigma$.{identity: σ, equal: $\sigma \rightarrow$ BOOLEAN}

which capture the shared implementation and specification of INTα and CHARα. The simple structure and behaviour of an object of the truncated type

INTα can be recovered by application of the generators to an object, 3, of type INTEGER:

```
obj = φobject (3);            INTα = ΦOBJECT [INTEGER];
obj.identity ⇒ 3;             INTα.identity: INTEGER;
obj.equal(4) ⇒ false;         INTα.equal [INTEGER] : BOOLEAN;
```

similarly, an object of the truncated type CHARα can be constructed by applying the generators to 'a' and CHARACTER. It is clear from the correspondence between terms and types that a straightforward relationship exists between objects created using φ*object* and types created using ΦOBJECT.

Application of generators produces objects and types which share the same implementation strategy and offer the same operations in their interfaces. For example, the body of *equal()* uses a primitive state comparison operator = which is assumed to work uniformly over any concrete representation. The types INTα and CHARα offer exactly the same operations in their interfaces, with homomorphic type signatures (although these function types do not stand in any straightforward typing relationship with each other).

The truncated type INTα is not exactly the same as the type INTEGER, which possesses other operations; this is evident from the fact that the expression:

```
obj : INTα;
obj.identity.plus(4)
```

is legal if INTEGER possesses a *plus()* operation; whereas INTα does not possess such an operation. The application of generators does not, in general, produce results that are closed over their arguments. However, a special case is found by taking the fixpoints of the generators:

```
object = (Y φobject)              OBJECT = (Y ΦOBJECT)
```

and applying the generators to these fixpoints produces results that *are* closed over their arguments, by the unrolling property of recursive types:

```
object ≡ φobject (object)         OBJECT ≡ ΦOBJECT [OBJECT]
```

The type OBJECT is considered the *least defined* type of its class and the instance *object* is a member of exactly this type. The property of being *least defined* comes from Y's making minimal assumptions when fixing the structure and behaviour of a recursive type. The types INTα and CHARα are *more defined*, since they place additional constraints on the types of their fields and exhibit more structure after one level of unfolding. These are called *pre-fixpoints* of the generator. The instances of all three types are pairwise incomparable, since they have different types. The type OBJECT is also considered the *most general* type of its class, since there exist other types, possessing more operations than OBJECT, that one may reasonably expect to include in the class.

5.1.3 Class Membership

Class membership is determined in object-oriented programming according to the services objects provide. An object belonging to a class must provide *at least* the services described by the class. This places an upper bound on the kinds of object that may be considered members of a class. An object satisfies the requirements for class membership if it implements equivalent, or strictly more methods than those expected by the class. An *equivalent* method is one with a compatible type signature which computes the same operation as the method it replaces. This is the substitutablility criterion for two object implementations, guaranteeing that one object may be substituted in place of another and still execute correctly all requests made of it.

To model extended objects with some replaced methods, we assume the existence of $\oplus$, a record combination operator with override, such that:

$$\text{obj} = \text{base} \oplus \text{extra}$$

constructs the record *obj* from all the fields of *base* and *extra*, where fields in *extra* replace (*ie* override) fields of the same name in *base*. Given this operator, we can define $\Re$ as the substitutability criterion between two object implementations:

$$\text{obj}\ \Re\ \text{imp} \iff (\text{imp} \oplus \text{obj}) = \text{obj}$$

which says that *obj* respects implementation *imp* if *obj* has at least all the named fields defined in *imp*, some of which may have been replaced. The equality constraint ensures that every field in *imp* is present, or replaced, in *obj*.

Class membership may now be formalised in two parts: firstly, as a constraint on method implementations; secondly as a constraint on method type signatures.

A method implementation constraint is introduced that allows instances *x* to belong to our example *object* class if:

$$x\ \Re\ \phi\text{object}(x)$$

which says that any object *x* must respect the implementation of the truncated record ϕ*object*(*x*). The application of the generator here ensures that occurrences of *self* in recursive records are co-referential for the purposes of field-by-field comparison.

A method type signature constraint is introduced that allows types τ to belong to our example object class if:

$\tau \subseteq \Phi$OBJECT $[\tau]$

which says that any type τ belonging to a class must be a subtype of the truncated type ΦOBJECT $[\tau]$. This construction is the *F-bound*, or *functional bound* [CCHO89a, CHC90] introduced in section 4.4 above. Subtyping between record types [CW85] was introduced in section 3.3.4 above.

Together, these constraints formalise the notion of a class as a family of objects sharing a *minimum* common recursive structure and recursive type. We may combine the implementation and type constraints by moving from an untyped object calculus (with separate types) to a typed calculus. Given the type generator function ΦOBJECT:

ΦOBJECT = $\Lambda\sigma$.{identity: σ, equal: $\sigma \rightarrow$ BOOLEAN}

we may replace the *untyped* object generator ϕ*object* by a *typed* object generator Φ*object* [1] having the following type signature and definition:

Φobject : $\forall$(t $\subseteq \Phi$OBJECT [t]).t $\rightarrow \Phi$OBJECT [t]

Φobject = Λ(t $\subseteq \Phi$OBJECT [t]).λ(self: t).
{identity $\mapsto$ self, equal $\mapsto$ λ(other: t).(self = other)}

and so provide a well-typed definition in the F-bounded second-order λ-calculus for the class of all objects having identity and equality. The function Φ*object* is a generator of polymorphic objects that maps objects in the type t to records in the constructed type ΦOBJECT [t]. The exact type of t is unknown - it is supplied later when objects are created. For example:

Φobject [INTEGER] (3) $\Rightarrow$
{identity $\mapsto$ 3, equal $\mapsto$ λ(other: INTEGER).(3 = other)} : INTα

constructs a truncated integer record, whereas:

(**Y** Φobject [**Y** ΦOBJECT]) $\Rightarrow$ object : OBJECT

= μself.{identity $\mapsto$ self, equal $\mapsto$ λ(other: OBJECT).(self = other)}
: OBJECT

constructs a recursive instance of exactly the type OBJECT. The latter syntax formalises the recovery of the simple recursive structure and type of objects when they are created in programs. Such objects are understood to be instances of the most general, least defined type of their class. The former syntax is not used explicitly in object-oriented languages, but is useful in our

[1] *ie* we use lower-case ϕ for untyped and upper-case Φ for typed object generators.

model, since it allows partial records to be modified in structure and type when they are combined with additional fields in subclass objects.

5.1.4 Class Hierarchy

In object-oriented programming, classes are hierarchically ordered according to the services they provide. A class is ordered below another if it provides more services and so describes a subset of the objects that may possibly be included in the other. Classes lower in the order are sometimes said to *inherit* the structure and type of the classes above them. Formally, *inheritance* is rightly to be considered as a separate notion from the *ordering* on classes; for the moment, let us focus on the properties of this order.

A simple Cartesian *point* class is introduced, bearing a deliberate similarity to our earlier *object* class. It provides the *identity* and *equal()* methods as before, with extra methods *x* and *y* to access the location of points (by default, the origin [0, 0]). For the moment, we shall ignore issues such as initialisation parameters and state update for the *x* and *y* fields of *point* objects. The typed definition for a *point* class is given by:

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

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

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

It is clear that this *point* class provides more services than our earlier *object* class; and that it replaces the earlier definition of *equal()* by a new version. The separate untyped object generator and type generator for this class are given by:

$$\phi\text{point} = \lambda\text{self}.\{x \mapsto 0,\ y \mapsto 0,\ \text{id} \mapsto \text{self},\ \text{eq} \mapsto \lambda\text{other}.(\text{self.x} = \text{other.x} \wedge \text{self.y} = \text{other.y})\}$$

$$\Phi\text{POINT} = \Lambda\sigma.\{\text{x: int, y: int, id: } \sigma\text{, eq: } \sigma \rightarrow \text{bool}\}$$

and these will be used to motivate an ordering relationship between our *point* class and *object* class by considering implementation and type issues independently.

The ordering relationship for object implementations is now formalised as a constraint linking the generators ϕ*point* and ϕ*object*:

$$\forall(x\ \Re\ \phi\text{point}(x)).\phi\text{point}(x)\ \Re\ \phi\text{object}(x)$$

which says that for all objects *x* with at least the methods of ϕ*point(x)*, it must be the case that ϕ*point(x)* has at least the methods of ϕ*object(x)*. From this it is intuitively apparent that any conceivable object *x* with at least the methods of ϕ*point(x)* will also have at least the methods of ϕ*object(x)*. We deliberately restrict the scope of quantification to $\forall(x \Re \phi point(x))$ in order to permit the immediate inference: $x \Re \phi object(x)$. If we expressed the constraint linking generators in an unrestricted pointwise fashion, we might still infer that *x* is a member of the ϕ*object* class using a rule:

$$\frac{\Gamma \vdash x \Re \phi f(x),\ \ \Gamma \vdash \forall y.\phi f(y) \Re \phi g(y)}{\Gamma \vdash x \Re \phi g(x)}$$

Put another way, we are not interested in whether the constraint linking generators ϕ*point(x)* $\Re$ ϕ*object(x)* holds for those objects which do not respect the implementation of the ϕ*point* class.

Similarly, the ordering relationship for object types is formalised as a constraint linking the type generators ΦPOINT and ΦOBJECT:

$$\forall(t \subseteq \Phi\text{POINT}\ [t]).\Phi\text{POINT}\ [t] \subseteq \Phi\text{OBJECT}\ [t]$$

which is a pointwise subtyping constraint requiring the generator ΦPOINT to be type-for-type in a subtype relationship with the generator ΦOBJECT for all types that are legal members of the point class [CHC90]. Given this, it is clear from the transitivity of $\subseteq$ that any conceivable type t satisfying $t \subseteq \Phi$POINT [t] will also satisfy $t \subseteq \Phi$OBJECT [t]. Cardelli has a more general pointwise subtyping rule linking generators in [AC95] without the restriction imposed above: $\forall(t \subseteq \Phi$ POINT [t]); this requires a further rule of the form:

$$\frac{\Gamma \vdash t \subseteq \Phi F[t],\quad \Gamma \vdash \forall s.\Phi F[s] \subseteq \Phi G[s]}{\Gamma \vdash t \subseteq \Phi G[t]}$$

in order to infer that $t \subseteq \Phi$OBJECT [t]. Bruce gives a proof of soundness for such a rule in [Bruc94, p158], Lemma 4.3. Again, we are not especially interested in whether the pointwise constraint holds between two F-bounds for types not satisfying the more restrictive F-bound.

Our theory deliberately relates the subclass generators to the superclass generators, so that important properties such as transitivity, reflexivity and antisymmetry may be preserved. Attempts to describe this ordering by relating the subclass objects directly to the superclass generators [Bruc94] results in a loss of transitivity [AC95] when seeking to compose ordering relations. We cannot infer with any certainty that the following holds:

$(F \subseteq \Phi G[F]) \wedge (G \subseteq \Phi H[G]) \Rightarrow (F \subseteq \Phi H[F])$

since there exist counter examples of the form:

$$\begin{aligned}
&\Phi F = \lambda\sigma.\{p : \sigma \to \text{INTEGER}, q : \text{INTEGER}\}\\
&\Phi G = \lambda\sigma.\{p : \sigma \to \text{INTEGER}\}\\
&\Phi H = \lambda\sigma.\{p : G \to \text{INTEGER}\}
\end{aligned}$$

where the antecedents hold, but not the consequent:

$$\begin{aligned}
&F \subseteq \Phi G[F] \;\Leftrightarrow\; \{p : F \to \text{INTEGER}, q : \text{INTEGER}\} \subseteq \{p : F \to \text{INTEGER}\}\\
&G \subseteq \Phi H[G] \;\Leftrightarrow\; \{p : G \to \text{INTEGER}\} \subseteq \{p : G \to \text{INTEGER}\}\\
&F \not\subset \Phi H[F] \;\Leftrightarrow\; \{p : F \to \text{INTEGER}, q : \text{INTEGER}\} \not\subset \{p : G \to \text{INTEGER}\}
\end{aligned}$$

because $G \not\subset F$ violates contravariance. Rather than make the membership of subclass objects in superclasses a primary assertion, our theory chooses instead to derive this property *from* an ordering rule linking the pointwise instantiation of the two class generators.

In consequence, the class hierarchy is a partial order which preserves the properties of reflexivity, transitivity and antisymmetry. First, the properties of implementations are listed:

Implementation reflexivity:

$\Gamma \vdash \forall(x \,\Re\, \phi f(x)).\phi f(x) \,\Re\, \phi f(x)$

because $\forall x.(x \,\Re\, x)$ from definition of $\Re$.

Implementation transitivity:

$$\frac{\Gamma \vdash \forall(x \,\Re\, \phi f(x)).\phi f(x) \,\Re\, \phi g(x), \quad \Gamma \vdash \forall(x \,\Re\, \phi g(x)).\phi g(x) \,\Re\, \phi h(x)}{\Gamma \vdash \forall(x \,\Re\, \phi f(x)).\phi f(x) \,\Re\, \phi h(x)}$$

because $\forall x.\forall y.\forall z.(x \,\Re\, y) \wedge (y \,\Re\, z) \Rightarrow (x \,\Re\, z)$ from the monotonicity of overriding in the $\oplus$ record combination operator.

Implementation antisymmetry:

$$\frac{\Gamma \vdash \forall(x \,\Re\, \phi f(x)).(\phi f(x) \,\Re\, \phi g(x)) \wedge (\phi g(x) \,\Re\, \phi f(x))}{\Gamma \vdash \forall(x \,\Re\, \phi f(x)).\phi f(x) = \phi g(x)}$$

because $\forall x.\forall y.(x \,\Re\, y) \wedge (y \,\Re\, x) \Rightarrow (x = y)$ from the monotonicity of overriding in the $\oplus$ record combination operator; we assume that the order of record fields is not significant when judging equality.

These are the properties that a partial order on implementations must observe. Similar properties apply to types:

Type reflexivity:

$$\Gamma \vdash \forall(t \subseteq \Phi F[t]).\Phi F[t] \subseteq \Phi F[t]$$

because $\forall t.t \subseteq t$ is true.

Type transitivity:

$$\frac{\Gamma \vdash \forall(t \subseteq \Phi F[t]).\Phi F[t] \subseteq \Phi G[t], \qquad \Gamma \vdash \forall(t \subseteq \Phi G[t]).\Phi G[t] \subseteq \Phi H[t]}{\Gamma \vdash \forall(t \subseteq \Phi F[t]).\Phi F[t] \subseteq \Phi H[t]}$$

since $\forall t.(t \subseteq \Phi F[t]) \wedge (\Phi F[t] \subseteq \Phi G[t]) \Rightarrow (t \subseteq \Phi G[t])$ from the first antecedent; and therefore the two antecedents are composable.

Type antisymmetry:

$$\frac{\Gamma \vdash \forall(t \subseteq \Phi F[t]).(\Phi F[t] \subseteq \Phi G[t]) \wedge (\Phi G[t] \subseteq \Phi F[t])}{\Gamma \vdash \forall(t \subseteq \Phi F[t]).\Phi F[t] = \Phi G[t]}$$

because $\forall s.\forall t.(s \subseteq t) \wedge (t \subseteq s) \Rightarrow (s = t)$ by definition; we assume that the ordering of record fields is not significant when judging equality.

These are the properties that a partial order on types must observe.

Since the orders on implementations and types have been modelled separately, the effect of their combination must be considered. We assert that an ordering on typed object generators can only exist if both the orders on types and implementations are observed. Fortuitously, it is never possible to create antisymmetric ordering conditions on implementation and type, since the type of a record is dependent on the number of fields it possesses: adding a field will produce a subtype. However, it is theoretically possible to define a subclass which overrides one method, but whose type is unchanged. Similarly, it is theoretically possible to define a subclass whose *self*-type is specialised, but whose implementation is unchanged.

5.2 Classification and Derivation

The chief advantage of inheritance in object-oriented languages is that it supports *incremental classification*, the derivation of a subclass from an existing class. Starting with the typed model of inheritance given by Cook *et al* in [CHC90], our theory illustrates how classes derived through inheritance may be shown nonetheless to observe the ordering properties of the class hierarchy.

Later, our theory extends Cook's approach, devising a more general typing for his record combination operator $\oplus$. Our operator differs from earlier formulations by being fully polymorphic rather than simply-typed. In order to support this, the notion of *dependent second-order types* is first introduced. Later, this will allow incremental extensions to be decoupled from the classes they extend.

5.2.1 Typed Inheritance

Inheritance is a shorthand mechanism for defining a subclass with respect to some chosen class. The new class extends, or simply reimplements, the services of the other and so describes a subset of the objects that may possibly be included in the other. Inheritance requires an ability to derive subclass object and type generators by modifying existing generators in order to make them refer to the new subclass and at the same time incorporate an extension record of additional typed fields.

Consider a child *point*-instance that is somehow derived from the parent ϕ*object* generator. When inheriting the parent's methods, occurrences of *self* must be redirected to refer to the child. This is important, because inherited methods may refer to *self* and we should want this to mean the child *point*-instance and not an instance of the parent. To achieve this, occurrences of *self* may be bound in the parent's generator, ϕ*object*, to the eventual child *point*-instance:

$$\phi\text{object}(\text{point}) \Rightarrow \{\text{identity} \mapsto \text{point}, \text{equal} \mapsto \lambda\text{other}.(\text{point} = \text{other})\}$$

This produces a partial record for a *point* object. So far, *identity* has been specialised to return the *point* instance, but *equal()* has the wrong implementation. Using $\oplus$ this partial record may now be combined with extra fields. New fields for *x* and *y* are added and the field *equal()* is replaced with the new desired implementation:

$$\begin{aligned}&\phi\text{object}(\text{point}) \oplus \\ &\quad \{x \mapsto 0, y \mapsto 0, \text{equal} \mapsto \lambda\text{other}.(\text{point}.x = \text{other}.x \\ &\qquad\qquad\qquad\qquad\qquad\qquad \wedge \text{point}.y = \text{other}.y)\}\end{aligned}$$

$$\begin{aligned}\Rightarrow\ &\{x \mapsto 0, y \mapsto 0, \text{identity} \mapsto \text{point}, \\ &\qquad \text{equal} \mapsto \lambda\text{other}.(\text{point}.x = \text{other}.x \wedge \text{point}.y = \text{other}.y)\}\end{aligned}$$

This yields a record having the intended structure of a recursive point instance after one level of unfolding. Generalising this technique, a child object generator may be derived from a parent generator by abstracting over *self* in the result:

$$\begin{aligned}&\phi\text{point} = \lambda\text{self}.(\phi\text{object}(\text{self}) \oplus \\ &\quad \{x \mapsto 0, y \mapsto 0, \text{equal} \mapsto \lambda\text{other}.(\text{self}.x = \text{other}.x \wedge \text{self}.y = \text{other}.y)\})\end{aligned}$$

$$\begin{aligned}&= \lambda\text{self}.\{x \mapsto 0, y \mapsto 0, \text{identity} \mapsto \text{self}, \\ &\qquad \text{equal} \mapsto \lambda\text{other}.(\text{self}.x = \text{other}.x \wedge \text{self}.y = \text{other}.y)\}\end{aligned}$$

This construction binds the parent generator's *self* to the resulting child's *self*, before combining this record with the extension record, which may also contain free references to *self*, bound only in the resulting generator.

A similar process binds the parent's *self*-type to the resulting child's *self*-type, before combining the partial record type with the remaining field types:

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

This ensures that the new *self*-type σ has the bound $\forall(t \subseteq \Phi\text{POINT }[t])$ in the point class, rather than the bound $\forall(t \subseteq \Phi\text{OBJECT }[t])$; and this illustrates in turn how an inherited function like *identity* may change in type even when its implementation is not changed. For the moment, it is assumed that $\oplus$ replaces the types of record fields along with their values.

Combining the two approaches, typed inheritance may be modelled by specialising the inherited type and structure of the parent in one operation, before combining this with new methods. This is achieved by distributing both the *self*-type and *self* of the child class to the typed form of parent generator:

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

$$\begin{aligned}\Phi\text{point} &= \Lambda(t \subseteq \Phi\text{POINT }[t]).\lambda(\text{self: t}).\\ &\quad(\Phi\text{object }[t]\text{ (self)} \oplus \{x \mapsto 0, y \mapsto 0,\\ &\qquad\quad \text{equal} \mapsto \lambda(\text{other: t}).(\text{self.x} = \text{other.x} \wedge \text{self.y} = \text{other.y})\})\\ &= \Lambda(t \subseteq \Phi\text{POINT }[t]).\lambda(\text{self: t}).\\ &\quad\{x \mapsto 0, y \mapsto 0, \text{identity} \mapsto \text{self},\\ &\qquad\quad \text{equal} \mapsto \lambda(\text{other: t}).(\text{self.x} = \text{other.x} \wedge \text{self.y} = \text{other.y})\}\end{aligned}$$

which produces exactly the desired form of typed object generator for the *point* class. It is also clear by inspection that combining the record implementations yields a result with the desired type $\forall(t \subseteq \Phi\text{POINT }[t]).t \to \Phi\text{POINT }[t]$, where:

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

Since we are now dealing with a typed system, it is important to ensure that this style of derivation for inheritance is type correct. The internal type application Φ*object* [t] is correct, because $\Phi\text{POINT }[t] \subseteq \Phi\text{OBJECT }[t]$ and therefore any type satisfying *point*'s type generator will also satisfy the bound on o*bject*'s type generator. The internal *self*-application Φ*object* [t] (*self*) is also correct since *object*'s generator has now been specialised to *point*'s *self*-type and will accept an argument in this type.

5.2.2 Typed Record Combination

The record combination operator $\oplus$ must also be demonstrably type correct. Cook considered that $\oplus$ joins values whose types are constant [CHC90]. In this case, each occurrence of $\oplus$ has a particular simply-typed form:

$$\oplus : \beta \to \varepsilon \to t$$

for each eventual record type t, in which the types of the *base* record β and *extra* extension record ε are related to the type of the result, due to the presence of the *self*-type t in the fields of β and ε. Whereas *extra* is always a truncated version of the result, *base* may legally contain fields that are supertypes (allowed by overriding). Accordingly, this relationship may be qualified as:

$$t = \beta \cap \varepsilon \;\Rightarrow\; (t \subseteq \beta) \wedge (t \subseteq \varepsilon)$$

making t the greatest lower bound on the types β and ε. This suggests the notion of an *intersection type* [Pier92, CP93] derived from the usual notion of subtyping. To have a Cook-style simply-typed record combination operator, we must assume that there are many different versions of $\oplus$, each typed over a different t and then over different supertypes β and ε of t, such that $t = \beta \cap \varepsilon$.

This is not especially satisfying. Instead, we wish to generalise $\oplus$ to a second-order typed operator, *combine*. However, this requires resolving the mutual type dependency between β, ε and t:

letrec $t = \beta \cap \varepsilon$ *in*

combine : $\forall(\beta \supseteq t).\forall(\varepsilon \supseteq t).\beta \to \varepsilon \to t$

```
combine = $\Lambda(\beta \supseteq t).\Lambda(\varepsilon \supseteq t).\lambda(base: \beta).\lambda(extra: \varepsilon).$
    { label $\mapsto$ value | (label $\in$ dom(base) $\cup$ dom(extra))
                          $\wedge$ (if label $\in$ dom(extra)
                                then value = extra.label
                                else value = base.label) }
```

Our aim is to prohibit the combination of two types β and ε which cannot be related to a common subtype t. Unfortunately, a type derivation may not be specified in this way. It is not clear that we could make a type assumption about the result and discharge it later, since we would have to invoke the rule we are defining to discharge the assumption on which it depends.

The mutually recursive type dependency is curious but necessary. Without the type constraints on its arguments, the result of *combine* is not guaranteed to have an intersection type. To see this, consider overriding a *base* record with an *extra* record having incomparable types in some common fields. The result is not a subtype of *base*. Critically, we want to preserve the pointwise subtyping relationship between child and parent classes and in particular the result of *combine* must be a subtype of the *base* argument for any pair of record types.

To avoid the mutually recursive type dependency, this condition may be re-expressed as a more complex type constraint linking the types β and ε. Since $\oplus$ is not commutative, every field of *extra* is always present in *base* $\oplus$ *extra*. Therefore, the result is always a subtype of *extra*. To ensure that the result is also always a subtype of *base*, we require that *base* fields can only ever be replaced by subtype fields taken from *extra*. This may be expressed as a type introduction rule for $\oplus$:

$$\frac{\begin{array}{ll}\Gamma \vdash b\text{: }\{a_1\text{: } s_1, \ldots a_j\text{: } s_j, \ldots a_k\text{: } s_k\}, & \\ \Gamma \vdash e\text{: }\{a_j\text{: } t_j, \ldots a_k\text{: } t_k, \ldots a_n\text{: } t_n\}, & \text{provided that } \sigma \text{ is uniform} \\ \qquad \Gamma \vdash t_j \subseteq s_j, \ldots t_k \subseteq s_k & \text{in b, e and b} \oplus \text{e}\end{array}}{\Gamma \vdash b \oplus e\text{: }\{a_1\text{: } s_1, \ldots a_j\text{: } t_j, \ldots a_k\text{: } t_k, \ldots a_n\text{: } t_n\}}$$

Since $\oplus$ is always used in a context where occurrences of *self* are co-referential, type complications due to non-uniform *self*-types σ are avoided. Otherwise, records could be constructed using $\oplus$ whose *self* was not uniform: occurrences of *self* in *extra* always refer to the result, but ocurrences of *self* in the result might refer either to the result or to the *base* record. It would still be possible to preserve record subtyping, but only at the cost of restricting overriding further: we could not replace any method having the *self*-type as an argument, since this would violate contravariance. Fortunately, an F-bounded type system promotes uniform *self*-types through the application of generators.

A type override constraint Ω is now defined, linking the record types β and ε. Ω is used below to provide a regular typing for second-order record combination:

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

$$\text{combine} : \forall\beta.\forall(\varepsilon \mid \varepsilon\ \Omega\ \beta).\beta \rightarrow \varepsilon \rightarrow \beta\cap\varepsilon$$

```
combine = $\Lambda\beta.\Lambda(\varepsilon \mid \varepsilon\ \Omega\ \beta).\lambda(\text{base: } \beta).\lambda(\text{extra: } \varepsilon)$.
    { label $\mapsto$ value | (label $\in$ dom(base) $\cup$ dom(extra))
                              $\wedge$ (if label $\in$ dom(extra)
                                      then value = extra.label
                                      else value = base.label) }
```

This condition is sufficient to type record combination for both uniform and non-uniform *self*-reference. We shall continue to use $\oplus$ as a convenient abbreviation with the meaning:

$$\oplus = \{ \oplus_{\beta,\varepsilon} \mid \forall\beta.\forall(\varepsilon \mid \varepsilon\ \Omega\ \beta).\ \oplus_{\beta,\varepsilon} : \beta \rightarrow \varepsilon \rightarrow \beta\cap\varepsilon \ \wedge\ \oplus_{\beta,\varepsilon} = \text{combine}\ [\beta\ \varepsilon] \}$$

5.2.3 Typed Method Combination

A further feature of object-oriented languages is the ability to extend the behaviour of inherited methods [Moon86, Keen89]. Instead of replacing the parent's method, a child class may combine this with additional code. The child class defines a *code wrapper* [Cook89a, CP89, CCHO89b] for the extended method, in which a call is made to the inherited version. This is best illustrated through an example.

Suppose that a three-dimensional point class were required to have the following specification:

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

A natural way to provide this would be through inheritance, basing the new class on the existing two-dimensional Cartesian point class, which has the specification:

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

This class already provides much of the desired behaviour:

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

$$\begin{aligned}&\Phi\text{point} = \Lambda(t \subseteq \Phi\text{POINT}\ [t]).\lambda(\text{self: t}).\\ &\quad \{x \mapsto 0,\ y \mapsto 0,\ \text{identity} \mapsto \text{self},\\ &\qquad \text{equal} \mapsto \lambda(\text{other: t}).(\text{self.x} = \text{other.x} \wedge \text{self.y} = \text{other.y})\}\end{aligned}$$

except that the new class requires an additional *z* method and a modified version of *equal()* that compares all three scalar values of a three-dimensional coordinate. This new version is only minimally different from the old version.

Rather than replace *equal()* wholesale, it would be preferable to make use of the existing version using a technique for method combination. Accordingly, a wrapper method is written for the modified *equal()*. Inside the wrapper, two recursion variables are used - *self* denotes the child and *super* denotes the parent, through which the inherited method *super.equal()* may be accessed:

$$\text{equal} \mapsto \lambda(\text{other: t}).(\text{super.equal(other)} \wedge \text{self.z} = \text{other.z})$$

In order to be meaningful, *super.equal()* must be equivalent to its inline expansion in the wholesale replacement of the *equal()* method:

$$\text{equal} \mapsto \lambda(\text{other: t}).(\text{self.x} = \text{other.x} \wedge \text{self.y} = \text{other.y} \wedge \text{self.z} = \text{other.z})$$

in which *self*-reference is uniform, such that the combined parts of the method refer to the same object.

This in turn requires a more detailed consideration of the binding of *self* and *super*, with their associated types. It is clear that all occurrences of *self* in a combined method should be co-referential, irrespective of whether they appear in the inherited method or wrapper. The desired goal is for *super* to refer to a version of the parent record in a context where *self* has been rebound to refer to the child. This may be achieved by distributing the new *self* and *self*-type of the child to the parent generator. Inside the 3D point class, *super* must have the form:

Φpoint [t] (self)

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

for $\forall(t \subseteq \Phi$3DPOINT [*t*]) and *self* : t. This is essentially a retyped form of the parent record. The method *equal()* has the desired implementation of the parent, in which *self* refers to the child, having the type $\forall$(t $\subseteq$ Φ3DPOINT [t]).

During inheritance, an object of exactly this type is routinely obtained on the left-hand side of the record combination operator $\oplus$, before these fields are combined with the extension record. The method combination technique must simply maintain a handle on this object so that the original method implementations are available after record combination. This is achieved by abstracting over *super* internally during inheritance:

Φ3dpoint : $\forall$(t $\subseteq$ Φ3DPOINT [t]).t $\rightarrow$ Φ3DPOINT [t]

Φ3dpoint = Λ(t $\subseteq$ Φ3DPOINT [t]).λ(self: t).
(λ(super: ΦPOINT [t]).
super $\oplus$ {z $\mapsto$ 0, equal $\mapsto$ λ(other: t).
(super.equal(other) $\wedge$ self.z = other.z)}
(Φpoint [t] (self)))

This construction binds *super* to the value Φpoint [*t*] (*self*). This value has the derived type: ΦPOINT [*t*], which can be inferred from the type of *self*. The internal *super* variable is therefore given exactly this type.

This construction is provably equivalent to the simpler form of inheritance in which references to *super* are expanded inline. By reducing the internal abstraction over *super*, we obtain the usual expression generating a modified form of the parent record, in which *self* denotes the child. This record also becomes available in the body of the wrapper method *equal()*. The sub-expression *super.equal(other)* is therefore reducible and will select a method from the parent record, applying it to the variable *other*, yielding an expression:

(self.x = other.x $\wedge$ self.y = other.y)

in which *self* denotes the child. All references to *super* may therefore be eliminated, yielding directly the simpler form of inheritance:

Φ3dpoint = Λ(t $\subseteq$ Φ3DPOINT [t]).λ(self: t).
Φpoint [t] (self) $\oplus$
{z $\mapsto$ 0, equal $\mapsto$ λ(other: t).
(self.x = other.x $\wedge$ self.y = other.y $\wedge$ self.z = other.z)}

in which the *equal()* method is replaced wholesale by a redefined version and all references to *self* are coreferential, denoting the child.

Any *super* method may be invoked inside a wrapper method, not just the inherited version of the combined method concerned. The effect of invoking a *super* method is always to leapfrog any locally-redefined version. In cases where no locally-redefined version exists, the inherited method is selected, since there is then no difference between the *self* and *super* versions. It is a static type error to invoke a non-existent *super* method.

5.3 Classification and Components

The breaking down of classification into incremental steps now makes it possible to describe independently a component extension. The earliest object-oriented language with such a notion was Flavors [Moon86], which referred to component extensions as *mixins*. Following the ice-cream metaphors used in the language, a *mixin* represented a particular set of behaviours that could be added to a basic, or *vanilla* flavoured[2] class. In practice, a mixin looked like any other class, with the distinction that it was not intended to be instantiated independently.

Bracha and Cook [BC90] revived interest in mixins when it became clear that models of inheritance for languages as diverse as Smalltalk [GR83], Beta [Mads93] and CLOS [Keen89] could all be mapped onto a simpler model based on the composition of mixins. A mixin is described as an *abstract subclass*, or

> "a subclass definition that may be applied to different superclasses to create a related family of modified classes" [BC90, p303].

More correctly, a mixin is a free-standing object extension function that abstracts over its own parent. It is not itself *abstract*, since it provides concrete services. Cook was unable to type mixins in [CHC90], because his record combination operator was not expressive enough to combine free-standing record extensions. Our theory replaces Cook's simply-typed operator with a polymorphic operator which has the required properties. In the unfinished Abel report [Harr91a] an attempt is made to type mixins using a higher-order *kinded* calculus. Here, a simpler way to type mixins is presented that requires only the notion of *type dependency* in a second-order calculus.

5.3.1 Typed Extension Records

In the model of inheritance given above, extension records are only defined within the scope of the enclosing inheritance expression - they have no independent existence. The *self* occurring in an extension record is bound only in the result of record combination $\oplus$, and refers to the result, rather than to the extension itself. Now, we choose to decompose inheritance further, by

[2] "Flavors" is a trademarked name. Elsewhere British spelling is used. ☺

abstracting over the *self* of extension records. This aims to give extension records a certain limited independent existence.

Abstracting over *self* yields a free-standing generator for a family of extension records, to which we may give a polymorphic type. Such a generator looks much like a class, except that its methods are intended to supplement the methods of other classes. An extension generator destined to provide a two dimensional coordinate system for any class has the form:

$$\Delta\text{XYCOORD} = \Lambda\sigma.\{\text{x: INTEGER, y: INTEGER, equal: } \sigma \rightarrow \text{BOOLEAN}\}$$

$$\Delta\text{xycoord} : \forall(\varepsilon \subseteq \Delta\text{XYCOORD } [\varepsilon]).\varepsilon \rightarrow \Delta\text{XYCOORD } [\varepsilon]$$

$$\begin{aligned}\Delta\text{xycoord} = \Lambda&(\varepsilon \subseteq \Delta\text{XYCOORD } [\varepsilon]).\lambda(\text{self: } \varepsilon).\\ &\{\text{x} \mapsto 0, \text{y} \mapsto 0, \text{equal} \mapsto \lambda(\text{other: } \varepsilon).\\ &\qquad(\text{self.x} = \text{other.x} \wedge \text{self.y} = \text{other.y})\}\end{aligned}$$

By convention, typed extension record generators are indicated using Δ-prefixes, to distinguish them from the Φ-prefixes of class generators. The constraint on the type of *self* arises from the fact that any class with which Δ*xycoord* is combined will have at least the methods *x, y* and *equal*.

To illustrate the use of this record extension generator during inheritance, it is combined with the *object* class to derive the Cartesian *point* class shown above. Since Φ*object* and Δ*xycoord* are now both generators, it is necessary to apply them to arguments standing for *self* before combining the resulting records using ⊕. In particular, combination must ensure that references to *self* in the base and extension record are co-referential. To achieve this, inheritance must distribute the *self*-type and *self* of the result to both Φ*object* and Δ*xycoord* generators:

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

$$\begin{aligned}\Phi\text{point} &= \Lambda(\text{t} \subseteq \Phi\text{POINT } [\text{t}]).\lambda(\text{self: t}).\\ &\quad \Phi\text{object } [\text{t}] \text{ (self)} \;\oplus\; (\Delta\text{xycoord } [\text{t}] \text{ (self)})\\ &= \Lambda(\text{t} \subseteq \Phi\text{POINT } [\text{t}]).\lambda(\text{self: t}).\\ &\quad \{\text{identity} \mapsto \text{self, equal} \mapsto \lambda(\text{other: t}).(\text{self} = \text{other})\} \;\oplus\\ &\quad \{\text{x} \mapsto 0, \text{y} \mapsto 0, \text{equal} \mapsto \lambda(\text{other: t}).\\ &\qquad\qquad (\text{self.x} = \text{other.x} \wedge \text{self.y} = \text{other.y})\}\\ &= \Lambda(\text{t} \subseteq \Phi\text{POINT } [\text{t}]).\lambda(\text{self: t}).\\ &\quad \{\text{x} \mapsto 0, \text{y} \mapsto 0, \text{identity} \mapsto \text{self},\\ &\qquad \text{equal} \mapsto \lambda(\text{other: t}).(\text{self.x} = \text{other.x} \wedge \text{self.y} = \text{other.y})\}\end{aligned}$$

This has a pleasing symmetry. The application Δxycoord [t] (self) is correct, because we can assert ∀(t ⊆ ΦPOINT [t]).t ⊆ ΔXYCOORD [t]. This follows from the observation ∀(t ⊆ ΦPOINT [t]).ΦPOINT [t] ⊆ ΔXYCOORD [t], which is a straightforward generalisation of the simple subtyping t ⊆ ε that obtains between any particular result and extension in record combination, where

$t = \beta \cap \varepsilon$. The above construction is very similar to the idea of multiple inheritance, which is explored later in chapter 6. Although this strategy gives independence to extension records, the type of the result, $\forall(t \subseteq \Phi\text{POINT}\ [t])$, is still used to demonstrate that record combination is well-typed according to the polymorphic definition of $\oplus$.

5.3.1 Typed Free Mixins

A genuine mixin cannot use the type of the result in this way, rather it must derive the result type from its own type and the type of the class it is mixed with. A mixin is an object extension function which abstracts over the *super* object with which it is mixed. Since objects are in general recursive, we define a mixin as a function of both *self* and *super*:

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

$$\begin{aligned}\Sigma\text{xycoord} = {} & \Lambda(\varepsilon \subseteq \Delta\text{XYCOORD}\ [\varepsilon]).\Lambda(\beta \supset \varepsilon).\lambda(\text{self: } \varepsilon).\lambda(\text{super: } \beta).\\ & \quad \text{super} \ \oplus\ \{\text{x} \mapsto 0,\ \text{y} \mapsto 0,\ \text{equal} \mapsto \lambda(\text{other: } \varepsilon).\\ & \qquad\qquad (\text{self.x} = \text{other.x} \wedge \text{self.y} = \text{other.y})\}\end{aligned}$$

By convention, Σ-prefixes are used to distinguish mixins from the extension record generators, which have Δ-prefixes. In the type signatures for mixins, the order of quantification for the base and extension record types is deliberately reversed. This is in order to force the type of *super* to depend directly on the type of *self*, reflecting the earlier strategy for typing *super* during method combination. The minimum type of *self* may be quantified independently of *super*, since *self* does not depend on any of *super*'s methods. Although this mixin may be combined with any parent class, the eventual type of *super* must be a supertype of *self*, so that record combination is well-typed internally. To preserve this condition, it is important to prohibit subsumption in the values supplied for *super* and *self*: once the types β and ε are given, *super* and *self* must have exactly these types. The result is an intersection type respecting the interfaces of both the base class and the extension record.

The Σ*xycoord* mixin function may be applied to any suitable parent class, such as ϕ*object*, to derive extended object generators. Since the final value and type of *self* are unknown, these are distributed as parameters in the resulting mixed class, which is temporarily called Φ*mixed1*. We distribute to Σ*xycoord* two types, standing for the types of *self* and *super*, followed by two values in these types. If σ is the type of *self*, then ΦOBJECT $[\sigma]$ is the appropriate *super* type adapted for σ and Φ*object* $[\sigma]$ (*self*) is the *super* record.

$$\begin{aligned}\Phi\text{mixed1} = {} & \Lambda\sigma.\lambda(\text{self: } \sigma).\\ & \quad \Sigma\text{xycoord}\ [\sigma, \Phi\text{OBJECT}\ [\sigma]]\ (\text{self}, \Phi\text{object}\ [\sigma]\ (\text{self}))\end{aligned}$$

$$\begin{aligned}= {} & \Lambda\sigma.\lambda(\text{self: } \sigma).\\ & \quad \{\text{identity} \mapsto \text{self},\ \text{equal} \mapsto \lambda(\text{other: } \sigma).(\text{self} = \text{other})\}\\ & \qquad \oplus\ \{\text{x} \mapsto 0,\ \text{y} \mapsto 0,\ \text{equal} \mapsto \lambda(\text{other: } \sigma).\\ & \qquad\qquad (\text{self.x} = \text{other.x} \wedge \text{self.y} = \text{other.y})\}\end{aligned}$$

$$= \Lambda\sigma.\lambda(\text{self}: \sigma).\ \{x \mapsto 0,\ y \mapsto 0,\ \text{identity} \mapsto \text{self},\ \text{equal} \mapsto \lambda(\text{other}: \sigma).(\text{self.x} = \text{other.x} \wedge \text{self.y} = \text{other.y})\}$$

This has exactly the form of the typed object generator Φ*point*, but lacks the F-bound on the *self*-type σ. Looking at the type constraints in Σ*xycoord*, we know that $\sigma \subseteq \Delta\text{XYCOORD}\ [\sigma]$ must hold for the *self*-type and that $\sigma \subset \Phi\text{OBJECT}\ [\sigma]$ must then hold for the *super*-type. The minimum type satisfying this is given by the intersection $\Phi\text{OBJECT}\ [\sigma] \cap \Delta\text{XYCOORD}\ [\sigma] = \Phi\text{POINT}\ [\sigma]$, to give this type a name. The result is therefore well-typed for $\forall(\sigma \subseteq \Phi\text{POINT}\ [\sigma])$ and this condition can be constructed from the result type of Σ*xycoord*.

The kind of mixin shown here is called a *free mixin*, since it makes few assumptions about the class with which it is to be combined. None of the methods in the extension record interact with base class methods. It is also possible to provide *bound mixins*, which depend on their base class having certain methods, often because they wish to specialise these methods.

5.3.2 Typed Bound Mixins

A bound mixin is also a free-standing object extension function that abstracts over both *self* and *super*. However, the *super* recursion variable refers to an object which possesses one or more methods on which *self*'s methods depend. The type of *super* expresses a minimum requirement on the interface of the base generator, such that combining this with the extension generator yields meaningful methods.

This can be illustrated by converting the earlier example of method combination into a free-standing bound mixin. Before, a class Φ*3dpoint* was derived by extending the Φ*point* class with a record providing a new *z* method and specialising the *equal()* method inherited from its parent. The freestanding version of this extension record is a generator:

$$\lambda(\text{self}: \varepsilon).\lambda(\text{super}: \beta).\{z \mapsto 0,\ \text{equal} \mapsto \lambda(\text{other}: \varepsilon).\ (\text{super.equal(other)} \wedge \text{self.z} = \text{other.z})\}$$

for which the types β and ε must be established. Even though *self*'s method *equal()* depends on *super*'s inherited method, the *super* type β is unusual in that it never appears in the interface (references to *self* appearing in the interface of inherited *super* methods will have the rebound type ε rather than β). This suggests that ε may be bound independently of β. Furthermore, it is clear that β depends directly on ε, since the *super* record is always constructed by applying a parent generator to *self*. Based on these insights, ε is bound before β.

In order to constrain the type ε of *self* independently, we appeal to the existence of an extension record type generator ΔZCOORD:

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

This is the type generator for an extension record generator in which references to *super* have been expanded inline. We may always suppose that such a generator exists, since its type signature does not depend on the type of *super*. Accordingly, *self* may now be given the polymorphic type:

$$\text{self} : \varepsilon \subseteq \Delta\text{ZCOORD } [\varepsilon]$$

It was established above that the type β of *super* is dependent on ε. It is also clear that β must possess a minimum interface containing those *super* methods that are invoked within the extension generator. Since *super.equal()* is the only *super* method invoked in the extension generator, any valid parent class must have some type $t \subseteq \Phi\text{EQUAL } [t]$, where:

$$\Phi\text{EQUAL} = \Lambda\tau.\{\text{equal: } \tau \rightarrow \text{BOOLEAN}\}$$

If ε is the type of *self*, then *super* must have at least the type $\Phi\text{EQUAL } [\varepsilon]$, since it must specialise the inherited *self*-type to ε; in fact *super* may take any dependent type β in the range:

$$\text{super} : (\beta \mid \varepsilon \subset \beta \subseteq \Phi\text{EQUAL } [\varepsilon])$$

This allows the bound mixin Σ*zcoord* finally to be given a type:

$$\begin{aligned}\Sigma\text{zcoord} : {} & \forall(\varepsilon \subseteq \Delta\text{ZCOORD } [\varepsilon]).\forall(\beta \mid \varepsilon \subset \beta \subseteq \Phi\text{EQUAL } [\varepsilon]). \\ & \varepsilon \rightarrow \beta \rightarrow \beta \cap \varepsilon\end{aligned}$$

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

The $\beta \subseteq \Phi\text{EQUAL } [\varepsilon]$ constraint ensures that *super* has at least an *equal()* method retyped in the self-type ε. The $\varepsilon \subset \beta$ constraint ensures that *super* still has a more general type than *self*. This is often overlooked - if $\varepsilon = \beta$, this does not produce sensible method combination, but wraps methods which have already been wrapped. If $\varepsilon \supset \beta$, the record combination operator $\oplus$ is supplied with arguments having the wrong types and fails to type-check. Generally, the lower bound on the type of *super* prevents a mixin from being combined incorrectly with a proper subclass.

Σ*zcoord* may be applied to any suitable parent, such as Φ*point*, to turn a two-dimensional coordinate object into a three-dimensional one:

$$\begin{aligned}
&\Phi\text{mixed2} = \Lambda\sigma.\lambda(\text{self: } \sigma).\\
&\quad \Sigma\text{zcoord } [\sigma, \Phi\text{POINT } [\sigma]] \text{ (self, } \Phi\text{point } [\sigma] \text{ (self))}\\[1ex]
&= \Lambda\sigma.\lambda(\text{self: } \sigma).\\
&\quad \{x \mapsto 0,\ y \mapsto 0,\ \text{identity} \mapsto \text{self},\\
&\qquad\quad \text{equal} \mapsto \lambda(\text{other: } \sigma).(\text{self.x} = \text{other.x} \wedge \text{self.y} = \text{other.y})\}\\
&\qquad\quad \oplus\ \{z \mapsto 0,\ \text{equal} \mapsto \lambda(\text{other: } \sigma).\\
&\qquad\qquad\quad (\text{self.x} = \text{other.x} \wedge \text{self.y} = \text{other.y}) \wedge (\text{self.z} = \text{other.z})\}\\[1ex]
&= \Lambda\sigma.\lambda(\text{self: } \sigma).\\
&\quad \{x \mapsto 0,\ y \mapsto 0,\ z \mapsto 0,\ \text{identity} \mapsto \text{self},\ \text{equal} \mapsto \lambda(\text{other: } \sigma).\\
&\qquad\qquad\quad (\text{self.x} = \text{other.x} \wedge \text{self.y} = \text{other.y} \wedge \text{self.z} = \text{other.z})\}
\end{aligned}$$

The result has exactly the form of Φ*3dpoint*. As before, the type of the result may be inferred. The *self*-type must obey $\sigma \subseteq \Delta$ZCOORD $[\sigma]$ and then the *super*-type must obey $\sigma \subset \Phi$POINT $[\sigma] \subseteq \Phi$EQUAL $[\sigma]$. The minimum type satisfying both is given by ΔZCOORD $[\sigma] \cap \Phi$POINT $[\sigma] = \Phi$3DPOINT $[\sigma]$, to give it a name. The result is well-typed for $\forall(\sigma \subseteq \Phi$3DPOINT $[\sigma])$.

It is sometimes illuminating to verify this using actual types. The least type in the result class is the fixpoint 3DPOINT. It clearly satisfies the type constraints on *self* and *super* in Σ*zcoord*:

$$\text{3DPOINT} \subseteq \Delta\text{ZCOORD [3DPOINT]},$$

$$\text{3DPOINT} \subset \Phi\text{POINT [3DPOINT]} \subseteq \Phi\text{EQUAL [3DPOINT]}.$$

Clearly, the first relationship ensures that 3DPOINT has at least the methods provided in the mixin. The second relationship has two parts: the first part ensures that 3DPOINT is a proper subtype of the *super* type with which it was combined; the second part ensures that the *super* type chosen has at least an *equal()* method, without which the *super.equal()* call would be incorrect.

5.3.3 On Dependent Second-Order Types

The *Abel* typing of mixins quantified over *classes*, rather than over *types*. [Harr91a] allowed variables to range over the *self*- and *super*-generators. This is because the *super*-interface apparently ranges over a set of classes and therefore the type of *self*, which is assumed to depend on the particular *super* chosen, must also range over a family of classes. Our approach avoids higher-order complications in two stages. Firstly, a second-order typing is provided for the freestanding *self* of the extension record, irrespective of whatever type is eventually given to *super*. Secondly, a second-order type expression is constructed for *super* that depends directly on the *self*-type; we are therefore not forced to quantify over class generators, as in [Harr91]. The typing given here is technically more accurate than a proposed typing of the *super*-interface in [BC90], which makes no distinction between *class* and *type*, resulting in fragmented notions of *self* after record combination. A related compositional model for extending classes is given in [Hauc93]. Like [CHC90], the scheme adopted is not expressive enough to type mixins, although *self*-reference is properly handled.

For simple theories of classification, in which the type of *self* is polymorphic but other types are static, dependent second order types provide a useful mechanism for typing programs. Dependent types are of the form:

$\forall\sigma.\forall(\tau \mid \tau\ \rho\ \sigma)$ where "ρ" denotes a relational constraint.

This kind of typing is a simple extension of the idea of functional bounds; and it is no more difficult to implement. Both F-bounds and the kinds of dependent second-order types shown here require a typechecking algorithm that compares interfaces for structural subsumption. It is relatively easy to type-check expressions with dependent type. In the scheme for polymorphic record-combination, the base type is made available before the dependent type has to be checked. In the scheme for mixins, a minimum type for the base type must be calculated from the constraint provided by the chosen super class.

More flexible theories of classification may not be amenable to this kind of checking. Chapter 6 reintroduces polymorphic attributes and functions bound over other polymorphic types than the *self*-type. While quantifying over single-argument type constructors may only require a third-order theory, the arbitrary stacking of type parameter arguments seems to indicate that a higher-order treatment is more suitable.