Mixins: Typing the Superclass Interface

A J H Simons, Department of Computer Science, University of Sheffield,
Regent Court, 211 Portobello Street, Sheffield S1 4DP, United Kingdom.
Email: A.Simons@dcs.shef.ac.uk

Abstract

A mixin is a free-standing class extension function, describing a set of behaviours that may be combined with potentially many other classes. Although informal, operational descriptions of mixins are given in languages like Flavors and CLOS, a truly satisfying formal description of mixins has so far eluded researchers in the field. Previous attempts have either restricted the interpretation of classes to simple types, or required a complex form of higher-order quantification. We describe a new typing for mixins, based on Cook's F-bounded model of inheritance, that uses dependent second-order types. In particular, we are able to type the superclass interface, expressing how a mixin expects in general to be combined with some class possessing *at least* a particular set of methods.

1. Introduction

Mixins are of interest in object-oriented programming because they describe the notion of a component extension, in much the same way that languages with encapsulation support the notion of component sub-parts. The difference is that mixins are combined using the inheritance rules of the language, rather than using ordinary composition. Whereas composition respects the client interface of the component, preserving the abstraction barrier between the part and the whole, mixin combination involves a subtle linking of inherited methods and mixin methods. Figure 1 illustrates a mixin class designed to add a two-dimensional co-ordinate system onto any class with which it is combined. In any resulting combination, the methods *x*, *y* and *move* do not interact with inherited methods, since they provide orthogonal functionality. However, the method *equal* extends the functionality of some basic *equal* method in order to compare object co-ordinates as part of the equality-test. The basic method, which is assumed to compare the states of two objects, is accessed through *super.equal(...)*, where *super* is a self-referential variable denoting the inherited part of the combination. At the time the mixin is defined, the exact binding of *super* and its type are not known. However, *super* must eventually refer to some object possessing at least an *equal* method for the combination to be well-defined.

```
mixin class XYCOORD
private attributes
        xcoord, ycoord : INTEGER;
public methods
        x : INTEGER
                { return xcoord }
        y : INTEGER
                { return ycoord }
        equal (other : SELF) : BOOLEAN
                { return (super.equal (other) and xcoord = other.x
                        and ycoord = other.y) }
        move (newx, newy : INTEGER) : SELF
                { xcoord := newx;  ycoord := newy;  return self }
end
```

Figure 1: Mixin Supplying a 2D Co-ordinate System

A mixin therefore has a *client interface*, describing the methods that it exports, and a *superclass interface*, expressing a minimum requirement on the methods owned by any class with which it is to be combined. This is because the services offered by a mixin depend in general on services inherited from the class with which it is eventually combined. The superclass interface is analogous to a socket, expecting to receive a plug-in parent class *possessing* at least a certain set of methods. By this, we do not necessarily mean *exporting*, since languages with both *public* and *protected* modes of inheritance allow classes to bequeath methods to their descendants which are not in their client interface.

This notion of the superclass interface has proven extremely difficult to capture, both practically in languages like Flavors and CLOS [Moon86, Keen89], which cannot express it, and in theoretical treatments which either over-simplify the typing issues [BC90, Hauc93] or have problems typing mixin inheritance in more complex type models [CHC90, Harr91]. We describe some of these difficulties in sections 2, 3 and 4 respectively.

The fact that a mixin extends a class, yielding a modified class leads naturally to the idea of a *class extension function*, having the form: $\Delta : \mathrm{CLASS} \rightarrow \mathrm{CLASS}$, where the domain and codomain range over classes. This notion has only been captured imperfectly in earlier treatments of mixins [BC90], which implicitly assume extension functions having the less satisfying form: $\Delta : \mathrm{TYPE} \rightarrow \mathrm{TYPE}$, ranging over simple types in the place of classes. Cook *et al*. [Cook89, CCHO89a, CCHO89b, CHC90] have elsewhere promoted the view that a class is not really a type τ, but a family of related types constrained by an F-bound: $\forall(\tau \subseteq F[\tau])$. However, in [CHC90], mixins could not be typed in the F-bounded model, because the combination operator would only work for simply-typed records. In section 5, we develop a dependent second-order typing for Cook's combination operator, which allows us to combine polymorphic typed records, in the manner of multiple *class* inheritance with linearisation. Finally, by appealing to the close relationship between the type of *self* and the type of *super*, we construct in section 6 a dependent second-order typing for mixins having the desired form: $\Delta : \mathrm{CLASS} \rightarrow \mathrm{CLASS}$, which captures exactly the required constraint on the superclass interface.

2. Untyped Treatments of Mixins

The widespread use of the term *mixin* to describe a component extension to a class dates from its appearance in Flavors [Moon86], the earliest object-oriented language with such a concept. 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 class.

In practice, a mixin looked like any other class, with the distinction that it was "not intended to be instantiated independently". This operational criterion was also adopted by the Common

Lisp Object System [Keen89]. The "no separate instantiation" requirement is mostly one of semantic intention, since there are otherwise no major formal differences between mixins and classes in these languages. For example, the multiple inheritance rules of Flavors and CLOS, though they linearise the class hierarchy in different ways[1], bind occurrences of *super* (Flavors) and *call-next-method* (CLOS) for mixins and classes in much the same way.

Neither language supports the notion of a superclass interface for mixins (nor for ordinary classes). No attempt is made to check that calls to inherited methods accessed through *super* (Flavors) or *call-next-method* (CLOS) are well-typed. It is perfectly possible to combine a mixin with a class which does not provide a required basic method. Super-method calls are resolved at run-time, leading to not-found errors for undefined super-methods.

3. Naïve Treatments of Mixins

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 combination of mixins. This work revealed important symmetries across these languages: a Beta prefix pattern binds the keyword *inner* to an extensional subpattern, while a Smalltalk subclass is defined as an extension binding the keyword *super* to a parent class and likewise a CLOS subclass method binds *call-next-method* to a parent method. Inheritance in these languages can be decomposed into the combination of an extension record of supplementary methods with a base record denoting the prefix pattern (Beta) or parent class (Smalltalk, CLOS). A mixin is initially described as an "abstract[2] subclass", or:

---

[1] Flavors uses a left-to-right, depth-first combination, whereas CLOS uses a more complex topological sorting algorithm based on ordered pairs.

[2] An unfortunate term, since a mixin is not *abstract* in the normal sense: it provides a concrete set of services.

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

Stated more correctly, a mixin is a free-standing *class extension function* that abstracts over its own superclass, having the form: $\Delta : \text{CLASS} \rightarrow \text{CLASS}$. Bracha and Cook were clearly reaching for such a definition, but the examples of mixins presented in [BC90, p304] fall short of this, since they are neither extension *functions* (they are simply free-standing extensions), nor do they abstract over *classes* (they abstract over types, a subtle but important distinction).

Initially, Bracha and Cook develop an untyped model of mixin combination. Their idea of a mixin is a parameterised free-standing extension record of methods, having the form:

$$\Delta = \lambda\text{super}.\{a_1 \mapsto e_1, \ldots\ a_n \mapsto e_n\}$$

where the a_i are labels and the e_i are method expressions which may contain further occurrences of *super*, through which other methods may be invoked. During mixin combination, *super* is bound to the parent object with which the mixin is combined. Ordinary objects are modelled as records of methods:

$$P = \{b_1 \mapsto e_1, \ldots\ b_n \mapsto e_n\}$$

A combined object C is derived from a parent object P using:

$$C = \Delta\ (P) \oplus P$$

where $\oplus$ is an asymmetric record combination operator preferring the fields from its *left*-hand argument. Δ (P) applies the extension to the parent P, such that *super*-reference in the extension is redirected onto the parent. Messages sent to *super* in the extension will invoke the parent's methods. It is clear from this definition that Δ is not a *class extension function*, since the extension of the parent class is achieved using $\oplus$, after the application of the mixin. Bracha and Cook's mixins are perhaps better described as *adaptable extensions*. More seriously, problems arise when we start to add types to this model.

The later development [BC90, p308-9] reveals that classes are given simple types. This interpretation is only adequate for languages such as Modula-3, in which subclassing is subtyping and methods are external to recursive objects [CDJG89]. Generally, this is not consistent with other work by Cook *et al.* [Cook89, CCHO89a, CCHO89b, CHC90] which has found that in practice most object-oriented languages assume a different type model. The weakness of the simply-typed model is revealed when P, Δ and C have recursive types. This is a common occurrence, for example when methods of the parent P make calls through *self* to other parent methods, or when the extension Δ provides a coherent set of methods which call each other. In this case, the mixin and the parent have the form:

$$\Delta = \lambda \text{super}.\mu \text{self}.\{a_1 \mapsto e_1, \ldots\ a_n \mapsto e_n\}$$

$$P = \mu \text{self}.\{b_1 \mapsto e_1, \ldots\ b_n \mapsto e_n\}$$

in which the method expressions e_i contain further references to *self*, through which other methods are invoked, recursively. The notation: $\mu \mathit{self}.\phi(\mathit{self})$ represents the recursive object which is the fixed point of a generator: $\lambda \mathit{self}.\phi(\mathit{self})$. In the generator, *self* is a parameter, but in the fixed point it is bound to the structure of the recursive object, $\mathit{self} = \phi(\mathit{self})$. When we add types to this model, we discover that the type of *self* is different in the parent object and in the mixin. Let us give the parent the recursive type: $P : \sigma$. Assuming that *super* does not appear in the client interface of the mixin, we may give the adapted mixin a different recursive type: $\Delta\ (P) : \tau$, to indicate the fact that *self* is bound over a different recursive record of methods. The mixin itself is a function having the type: $\Delta : \forall(t \subseteq \Theta).t \rightarrow \tau$, where $t \subseteq \Theta$ is a type constraint on the *super* argument, expressing the fact that it must provide at least a particular set of methods (those methods which are invoked through *super* in the mixin). Provided that $P \subseteq \Theta$, then the application $\Delta\ (P)$ is type correct, and has the result type τ. However, the combination $C = \Delta\ (P) \oplus P$ does not yield an object with a useful recursive type, since self-reference is non-uniform in the result. In methods a_i occurrences of *self* have the type τ, whereas in methods b_i occurrences of *self* have the type σ. Nowhere in the combined object C does *self* refer to C, but rather to disjoint sub-parts of C. This leads to type-unsafe method overriding, for example, where mixin methods a_i are wrappers which adapt inherited methods b_i, and both return *self*. Then, $b_i : \sigma$ is replaced by $a_i : \tau$ in a context where σ and τ have unrelated, disjoint types. We shall call this approach the *naïve* typing of mixins. This naïveté is present to a certain degree in other recent efforts to type the superclass [Hauc93] and subclass [Lamp93] interfaces.

Lamping concentrates on distinguishing the client interface from the *specialisation* interface (i.e. subclass interface), the Beta-style dual of our superclass interface [Lamp93]. The focus of Lamping's work is on identifying what (hidden) protocols of a parent class are relied on by its descendants and therefore which methods must be protected from change. Unfortunately, no treatment of recursion is given, so the primary requirement that *self*-reference should be kept consistent under inheritance is not addressed. Without such a guarantee, it is impossible to determine whether protected methods are available to descendants or not.

Hauck recommends that inheritance should be understood as a kind of composition [Hauc93], in which subclass objects *contain* an object of the superclass type, but repeat their interface, delegating repeated methods to the attribute *super*. Recognising the binding problem associated with recursion, Hauck uses Cook's technique [Cook89, CP89] to redirect the *self* of the parent object P onto the child C. Cook and Hauck give an object definition the form of a generator ϕP parameterised over *self*:

$$\phi P = \lambda \mathrm{self}.\{b_1 \mapsto e_1, \ldots\ b_n \mapsto e_n\}$$

The recursive object P is implicitly constructed as the fixed point of the generator ϕP (in [Hauc93], the fixed form of *self* is referred to as *here*):

$$P = (\mathbf{Y}\ \phi P) \;\Leftrightarrow\; P = \phi P(P)$$

in which *self* is bound recursively to P. However, the component attribute *super* in any child C is constructed as: *super* = ϕP(C). This insight is important, since it ensures that *self*-reference in P's methods is redirected onto the child C, such that *self*-reference in all methods (whether new in C, or delegated to P) consistently refers to the child C.

Unfortunately, Hauck's extension to mixin-based composition [Hauc93, p238] is faulty, since it does not deal properly with the binding of *self*. A mixin definition has the form of an extension generator parameterised over *self* and *super*:

$$\phi\Delta = \lambda\text{self}.\lambda\text{super}.\{a_1 \mapsto e_1, \ldots\ a_n \mapsto e_n\}$$

From the diagram in [Hauc93, p238] the adapted mixin object is expected to have the form:

$$\mu\text{self}.\phi\Delta\ (\text{self}, \phi P(\text{self}))$$

in which the mixin's *super* is bound to some adapted form of the parent P and *self*-reference in the parent's methods is redirected onto the mixin. This means that *self*-reference in both the mixin and the parent refer to the *extension record* alone. Unlike Hauck's other derived child classes, a mixin cannot repeat the full interface of the parent. Mutually recursive calls among P's methods will now no longer work, since they access a *self* which only knows about the additional methods provided by the mixin.

4. Incomplete Treatments of Mixins

Cook's F-bounded model of inheritance [Cook89, CP89, CCHO89b, CHC90] deals properly with recursion, but stops short of handling mixins. In Cook's approach, free-standing extension records are not allowed, because of the complications this raises in the typing of the record combination operator ⊕. For comparison, we first describe Cook's untyped model.

A child object C is defined by first modifying a parent generator ϕP to a child generator ϕC:

$$\phi C = \lambda\text{self}.(\phi P\ (\text{self}) \oplus \{a_1 \mapsto e_1, \ldots\ a_n \mapsto e_n\})$$

and then taking the fixed point: $C = (\mathbf{Y}\ \phi C)$. Here, ⊕ is an asymmetric record combination operator that prefers fields from its *right*-hand argument. Note especially how *self* is only bound in the result of record combination. Free occurrences of *self* in the extension record's method expressions e_i refer to the *self* of the child generator ϕC, rather than to the extension record itself. For this reason, the extension record cannot exist outside the scope of the child object definition. Within these constraints, it is possible to model single inheritance with method combination, replacing an inherited method b_i with a new version a_i which invokes the original through a call to *super*. So long as the extension record is defined within the scope of *self* and *super* in ϕC, it is possible to parameterise inheritance internally over *super*:

$$\phi C = \lambda self.(\lambda super.(super \oplus \{a_1 \mapsto e_1, \ldots a_n \mapsto e_n\})\ (\phi P\ (self)))$$

and bind *super* to a value representing the adapted parent object ϕP (*self*), in which *self* has been redirected to refer to the child. Now, method expressions e_i in the extension record may contain free occurrences of *super* and *self*. Methods a_i in the extension record may override or wrap methods b_i in the parent class, but in contrast with [BC90], *self*-reference in the parent and the extension both refer consistently to the child.

Intuitively, to develop the notion of a mixin which deals properly with the binding of *self*, we want a free-standing extension record to have the form of a generator:

$$\phi\Delta = \lambda self.\lambda super.\{a_1 \mapsto e_1, \ldots a_n \mapsto e_n\}$$

which abstracts over both *self* and *super*. We intend to develop a mixin-style of inheritance having the form:

$$\phi C = \lambda self.(\phi P\ (self) \oplus (\phi\Delta\ (self, \phi P\ (self))))$$

in which the *self* of the mixin is adapted to the child and the *super* is adapted to a modified form of the parent in which *self*-reference denotes the child. Unfortunately, it is not possible to type this in Cook's model without first redefining his record combination operator.

To see why this is necessary, we shall add types to the model and introduce some more concrete examples, in preparation for a formal treatment of the mixin from section 1. Figure 2 illustrates a basic typed *square* class. ΦSQUARE is a type generator, describing the recursive type of a simple geometric square shape, having the methods *side, area* and *equal*. It is parameterised in σ, the *self*-type of *square*s. In generator-form, it can be adapted by application to new types: ΦSQUARE [*t*], which distributes *t* to σ.

ΦSQUARE = $\Lambda\sigma$.{side: INTEGER, area: INTEGER, equal: $\sigma \rightarrow$ BOOLEAN}

Φsquare : $\forall$(t $\subseteq$ ΦSQUARE [t]).t $\rightarrow$ ΦSQUARE [t]

Φsquare = Λ(t $\subseteq$ ΦSQUARE [t]).λ(self: t).
{side $\mapsto$ 10, area $\mapsto$ (self.side * self.side),
equal $\mapsto$ λ(other: t).(self.side = other.side)}

Figure 2: Basic Square Class

The fixed point of the generator: SQUARE = (**Y** ΦSQUARE) yields the exact recursive type of instances of the basic *square* class. Unrolling the recursion, the type SQUARE is equal to: {side: INTEGER, area: INTEGER, equal: SQUARE $\rightarrow$ BOOLEAN}. The associated function Φ*square* is a typed object generator, describing the recursive form of such instances, parameterised in *self* and the *self*-type. To create an object *sqr*, a type must be supplied for *self*, then the fixed point taken: *sqr* = (**Y** (Φ*square* [SQUARE])), in order to bind *self* recursively to *sqr*. The type of *self* is expressed as an *F-bound*: $\forall$($t \subseteq$ ΦSQUARE [t]).t, indicating that *self* may be safely redirected onto other recursive structures so long as their type satisfies $t \subseteq$ ΦSQUARE [t]. This expresses the constraint that any *self : t* must have at least the interface of the adapted type ΦSQUARE [t], where $\subseteq$ means "is a subtype of" and here can be interpreted as "has more methods than", since apart from distributing types to the parameter t, we never change the types of existing methods.

Figure 3 illustrates a more elaborate typed *xy-square* class, the class of all squares with a two-dimensional co-ordinate system, which is derived from the *square* class using inheritance with method combination. Note how the inherited *super* object is a form of the parent generator adapted to the *self*-type and *self* of the child using: Φ*square* [t] (*self*). This *super*, which has the adapted type: ΦSQUARE [t], is then combined using $\oplus$ with an extension record that adds the methods *x*, *y* and *move* and redefines *equal* to compare the co-ordinate positions as well as the sides of *xy-square*s. The redefined *equal* wraps the inherited method, by calling *super.equal(...)* in its body. In the inheritance expression, *super.equal*(*other*) is a reducible expression, which selects the body of the *equal* method from the typed *super* object. The body: (*self.side* = *other.side*) is substituted inline in place of the *super*-method invocation in the wrapper, which then overrides the inherited method.

$$
\begin{aligned}
&\Phi\text{XY-SQUARE} = \Lambda\sigma.\{\text{x: INTEGER, y: INTEGER, equal: } \sigma \rightarrow \text{BOOLEAN},\\
&\qquad\qquad \text{move : INTEGER} \times \text{INTEGER} \rightarrow \sigma,\\
&\qquad\qquad \text{side: INTEGER, area: INTEGER}\}
\end{aligned}
$$

$$
\Phi\text{xy-square} : \forall(\text{t} \subseteq \Phi\text{XY-SQUARE [t]}).\text{t} \rightarrow \Phi\text{XY-SQUARE [t]}
$$

$$
\begin{aligned}
&\Phi\text{xy-square} = \Lambda(\text{t} \subseteq \Phi\text{XY-SQUARE [t]}).\lambda(\text{self: t}).\\
&\quad (\lambda(\text{super: } \Phi\text{SQUARE [t]}).\\
&\qquad (\text{super} \;\oplus\; \{\text{x} \mapsto 0,\ \text{y} \mapsto 0,\ \text{equal} \mapsto \lambda(\text{other: t}).\\
&\qquad\quad (\text{super.equal(other)} \wedge \text{self.x} = \text{other.x} \wedge \text{self.y} = \text{other.y}),\\
&\qquad \text{move} \mapsto \lambda(\text{a: INTEGER}).\lambda(\text{b: INTEGER}).(\text{x := a; y := b; self})\})\\
&\quad (\Phi\text{square [t] (self)}))
\end{aligned}
$$

$$
\begin{aligned}
&= \Lambda(\text{t} \subseteq \Phi\text{XY-SQUARE [t]}).\lambda(\text{self: t}).\{\text{x} \mapsto 0,\ \text{y} \mapsto 0,\ \text{equal} \mapsto \lambda(\text{other: t}).\\
&\qquad\quad (\text{self.side} = \text{other.side} \wedge \text{self.x} = \text{other.x} \wedge \text{self.y} = \text{other.y}),\\
&\qquad \text{move} \mapsto \lambda(\text{a: INTEGER}).\lambda(\text{b: INTEGER}).(\text{x := a; y := b; self}),\\
&\qquad \text{side} \mapsto 10,\ \text{area} \mapsto (\text{self.side * self.side})\ \}
\end{aligned}
$$

Figure 3: Derived XY-Square Class

Since we are now dealing with a typed system, it is important to ensure that this style of derivation is type correct. The internal type application Φ*square* [*t*] is correct provided that any $t \subseteq \Phi\text{SQUARE}\ [t]$. Given the new F-bound: $\forall(t \subseteq \Phi\text{XY-SQUARE}\ [t])$ and the observation: $\forall t.\Phi\text{XY-SQUARE}\ [t] \subseteq \Phi\text{SQUARE}\ [t]$, a pointwise subtyping relationship between the interface generators [AC95], any type satisfying *xy-square*'s type generator will also satisfy the bound on *square*'s type generator, because it can be shown [Bruc94, p158] that the following transitivity rule holds:

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

The internal *self*-application Φ*square* [t] (*self*) is now correct since *square*'s generator has been specialised to *xy-square*'s *self*-type and will accept a new *self*-argument in this type.

The record combination operator $\oplus$ must also be demonstrably type correct. Cook *et al.* considered that $\oplus$ joins "values whose types are constant" [CHC90, p128]. To achieve this interpretation, we must consider that, during object-creation, inheritance expressions are β-reduced in normal order and record combination is performed *last*, after the recursive type of the new object has been fixed using **Y**. In this case, each occurrence of $\oplus$ has a particular simply-typed form:

$$\oplus : \beta \rightarrow \varepsilon \rightarrow \sigma$$

for each eventual record type σ, in which the types of the *base* record β and *extra* extension record ε are related to the type σ of the result, due to the presence of the *self*-type σ in the fields of β and ε. In our model, both *extra* and *base* are truncated versions of the resulting record (we do not allow, nor need, subsumption in the types of individual fields, as do other record subtyping models [CW85]). Accordingly, we may qualify this relationship as:

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

making σ the greatest lower bound on the types β and ε. This suggests the notion of an *intersection type* [Pier92, CP93, Comp94] 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 σ and then over different supertypes β and ε of σ, such that $\sigma = \beta \cap \varepsilon$. This is not especially satisfying, since it fails to generalise the notion of record combination for typed records. More seriously, it means that we cannot type mixin-based inheritance.

In particular, we cannot use a simply-typed $\oplus$ to combine an extension with a parameterised base class whose *self*-type is unfixed. In contrast to the *bound* extension records used in ordinary inheritance, we need *free-standing* extension records for mixin-based inheritance, which abstract over the types of *self* and *super*. Because of this, Cook *et al.* were unable to provide useful types for "abstract subclasses", or free-standing extensions, in [CHC90, p129].

5. Second-Order Typed Record Combination

To overcome this problem, we wish to generalise $\oplus$ to a second-order typed operator, *combine*. However, this requires resolving the mutual type dependency between β, ε and σ:

$$\textit{letrec}\ \sigma = \beta \cap \varepsilon\ \textit{in}$$

$$\text{combine} : \forall(\beta \supseteq \sigma).\forall(\varepsilon \supseteq \sigma).\beta \rightarrow \varepsilon \rightarrow \sigma$$

Our aim is to prohibit the combination of two types β and ε which cannot be related to a common subtype σ. Unfortunately, we may not specify a type derivation in this way. It is not clear that we could make the initial type assumption about the result, 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, we can re-express this condition 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 be replaced by fields taken from *extra* if the replacement fields have the same types. We express this as a type introduction rule for $\oplus$:

$$\frac{\begin{array}{c}\Gamma \vdash \text{base: } \{a_1\text{: } s_1, \ldots\ a_j\text{: } s_j, \ldots\ a_k\text{: } s_k\}, \\ \Gamma \vdash \text{extra: } \{a_j\text{: } t_j, \ldots\ a_k\text{: } t_k, \ldots\ a_n\text{: } t_n\}, \\ \Gamma \vdash t_j = s_j, \ldots\ t_k = s_k\end{array}}{\Gamma \vdash \text{base} \oplus \text{extra: } \{a_1\text{: } s_1, \ldots\ a_j\text{: } t_j, \ldots\ a_k\text{: } t_k, \ldots\ a_n\text{: } t_n\}} \quad \begin{array}{l}\text{provided that } \textit{self} \text{ is uniform} \\ \text{in } \textit{base}, \textit{extra} \text{ and } \textit{base} \oplus \textit{extra}\end{array}$$

Since we shall always use $\oplus$ in a context where occurrences of *self* are co-referential, we avoid type complications due to non-uniform *self*-types σ. Fortunately, an F-bounded type system promotes uniform *self*-types through the application of generators.

We can now define a type override constraint Ω linking the record types β and ε and then provide a regular typing for second-order record combination:

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

$$\mathrm{combine} : \forall\beta.\forall(\varepsilon \mid \varepsilon\ \Omega\ \beta).\beta \to \varepsilon \to \beta{\cap}\varepsilon$$

$$\begin{aligned}\mathrm{combine} = {} & \Lambda\beta.\Lambda(\varepsilon \mid \varepsilon\ \Omega\ \beta).\lambda(\mathrm{base}{:}\ \beta).\lambda(\mathrm{extra}{:}\ \varepsilon).\\ & \quad \{\ \mathrm{label} \mapsto \mathrm{value} \mid (\mathrm{label} \in \mathrm{dom(base)} \cup \mathrm{dom(extra)})\\ & \qquad\qquad \wedge (\text{if label} \in \mathrm{dom(extra)}\\ & \qquad\qquad\qquad \text{then value} = \mathrm{extra.label}\\ & \qquad\qquad\qquad \text{else value} = \mathrm{base.label})\ \}\end{aligned}$$

This condition is sufficient to type record combination. Where common fields exist, the equality constraint enforces uniform instantiation of the *self*-type. We shall continue to use $\oplus$ below as an abbreviation, with the expanded meaning:

$$\oplus = \forall\beta.\forall(\varepsilon \mid \varepsilon\ \Omega\ \beta).\mathrm{combine}\ [\beta\ \varepsilon]$$

We note that this form of typing has avoided explicit higher-order quantification because it exploits a dependency between two type arguments. We call this style *dependent second-order quantification*.

Now, we aim to give recursive extension records a certain limited independent existence. By abstracting over the *self* and *self*-type of extension records, we obtain a free-standing generator, 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. Figure 4 illustrates a simple typed extension generator destined to provide a two dimensional co-ordinate system for any class without an *equal* method. By convention, we adopt Δ-prefixes for typed extension record generators, 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*, *equal* and *move*. So far, this class is not a true mixin because it does not also abstract over *super*.

$$\begin{aligned}\Delta\text{XYCOORD} = \Lambda\sigma.\{&\text{x: INTEGER, y: INTEGER, equal: } \sigma \rightarrow \text{BOOLEAN},\\ &\text{move : INTEGER} \times \text{INTEGER} \rightarrow \sigma\}\end{aligned}$$

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

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

Figure 4: Extension Class for a 2D Co-ordinate System

We could imagine combining this extension class with a basic *object* class (not illustrated here) to derive a *point* class, using the style:

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

$$\begin{aligned}&\Phi\text{point} = \Lambda(\text{t} \subseteq \Phi\text{POINT [t]}).\lambda(\text{self: t}).\\ &\qquad \Phi\text{object [t] (self)} \;\oplus\; (\Delta\text{xycoord [t] (self)})\end{aligned}$$

Since Φ*object* and Δ*xycoord* are both generators, it is necessary to apply them to uniform arguments standing for the *self*-type and *self* before combining the resulting records. The above construction is very similar to the idea of *multiple inheritance with linearisation*, since repeated combination with $\oplus$ prefers fields from the right-most argument; however it is still not *mixin inheritance*, since it anticipates the type of the result, $\forall(\text{t} \subseteq \Phi\text{POINT [t]})$.

6. Typing the Superclass Interface

A genuine mixin should derive the result type from its own type and the type of the class it is mixed with. Earlier, we defined a mixin as a *class extension function which abstracts over its own superclass*. Since classes are in general recursive, a mixin is a function of both *self* and *super*, which constructs a new class by internal application of the record combination operator.

Figure 5 illustrates the mixin corresponding to the extension class in Figure 4:

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

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

Figure 5: Open Mixin for a 2D Co-ordinate System

We adopt Σ-prefixes to distinguish mixins from extension classes, which have Δ-prefixes. In the type signatures for mixins, we deliberately order the quantification to force the type of *super* to depend directly on the type of *self*, reflecting our earlier strategy for typing the inherited *super* object. Recall that in Figure 3, *super* had the type ΦSQUARE [*t*], where *t* is the new *self*-type. It is always the case that *super*'s type is a proper supertype of *self*'s type; consider that, for the application ΦSQUARE [*t*] to be correct, $t \subseteq$ ΦSQUARE [*t*] must hold. Accordingly, we insist on a type relationship $\beta \supset \varepsilon$ between the types of *super* and *self* in a mixin. We do not allow 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 of ⊕ is an intersection type respecting the interfaces of both the base class and the extension class.

We call the mixin shown in Figure 5 an *open mixin*, since it makes few assumptions about the class with which it is to be combined. None of the methods in the extension record are assumed to interact with any base class methods. It is also possible to provide *bounded mixins*, which depend on their base class having certain methods, often because they wish to specialise these methods. The type of *super* then must express a minimum requirement on the interface of the base generator, such that method combination yields meaningful methods. We now seek to type such a bounded mixin, corresponding to the concrete example given in Figure 1, by further constraining the type of *super* in Figure 5.

Bounded mixins apparently introduce a mutual dependency: methods of *self* may now depend on methods of *super*, since one of *self*'s method results may be passed back directly from the *super*-method invocation. It is usual to quantify in order of dependency, leading not unnaturally to the assumption that *super* should be bound before *self*. However, even though *self*'s method *equal* depends on *super*'s inherited method, the *super* type β is unusual in that it never appears in *self*'s interface (references to *self* appearing in the interface of inherited *super* methods will have the rebound type ε rather than β). This suggests that we can bind ε independently of β. Furthermore, from the previous discussion 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, we bind ε before β.

In order to constrain the type ε of *self* independently, we appeal to the existence of the type generator ΔXYCOORD for an extension class, given in Figure 4. We may always suppose that such a generator exists independently, since its type signature does not depend on the type of *super*. Accordingly, we may legitimately still give *self* the polymorphic type:

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

We have established that the type β of *super* is a supertype of ε. It is also clear that β must possess a minimum interface containing those *super* methods that are invoked within the extension class. Since *super.equal(...)* is the only *super* method we wish to invoke, 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 ΦEQUAL [ε], since it must specialise the inherited *self*-type to ε; ΦEQUAL [ε] is the upper bound on the type of *super*. As we have already determined that ε is the lower bound, *super* may take any dependent type β in the range:

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

This allows us to type the bounded version of the mixin Σ*xycoord*, illustrated in Figure 6. 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 $\varepsilon = \beta$, we do not obtain sensible method combination, but wrap methods which have already been wrapped once. If $\varepsilon \supset \beta$ were allowed, the mixin might be combined incorrectly with a proper subclass.

$$\begin{array}{l} \Sigma\text{xycoord} : \forall(\varepsilon \subseteq \Delta\text{XYCOORD }[\varepsilon]).\forall(\beta \mid \varepsilon \subset \beta \subseteq \Phi\text{EQUAL }[\varepsilon]). \\ \qquad\qquad\qquad \varepsilon \to \beta \to \beta \cap \varepsilon \end{array}$$

$$\begin{array}{l} \Sigma\text{xycoord} = \Lambda(\varepsilon \subseteq \Delta\text{XYCOORD }[\varepsilon]).\Lambda(\beta \mid \varepsilon \subset \beta \subseteq \Phi\text{EQUAL }[\varepsilon]). \\ \quad \lambda(\text{self: } \varepsilon).\lambda(\text{super: } \beta). \\ \quad \text{super } \oplus \{\text{x} \mapsto 0, \text{y} \mapsto 0, \text{equal} \mapsto \lambda(\text{other: } \varepsilon). \\ \qquad\qquad (\text{super.equal(other)} \wedge \text{self.x} = \text{other.x} \wedge \text{self.y} = \text{other.y}), \\ \qquad \text{move} \mapsto \lambda(\text{a: INTEGER}).\lambda(\text{b: INTEGER}).(\text{x} := \text{a; y} := \text{b; self})\} \end{array}$$

Figure 6: Bounded Mixin for a 2D Co-ordinate System

We may apply the Σ*xycoord* mixin directly to any suitable parent class owning an *equal* method. Figure 7 illustrates an example of mixin inheritance in which the mixin Σ*xycoord* is combined with the ϕ*square* generator shown in Figure 2, in order to derive an extended *xy-square* class with a 2D co-ordinate system. Looking at the type constraints in Σ*xycoord*, we know that for any particular *self*-type $\sigma \subseteq \Delta$XYCOORD [σ], any *super*-type τ must be a proper supertype satisfying $\sigma \subset \tau \subseteq \Phi$EQUAL [$\sigma$]. We know from Figure 5 that a proper supertype may be created by application of some superclass generator, here ΦSQUARE, such that $\tau = \Phi$SQUARE [σ]. In order to verify that ΦSQUARE is indeed the generator for a legitimate superclass, we must be able to show that $\sigma \subset \Phi$SQUARE [σ], for some σ. In order to verify that ΦSQUARE generates an interface with at least the methods expected by the mixin, we must be able to show that ΦSQUARE [σ] $\subseteq \Phi$EQUAL [σ], for some σ. The latter immediately follows from the observation: $\forall t.\Phi$SQUARE [t] $\subseteq \Phi$EQUAL [t], the pointwise subtyping relationship that obtains between the two generators. We now have two constraints on the type of *self*: $\sigma \subseteq \Delta$XYCOORD [σ] and $\sigma \subset \Phi$SQUARE [σ]. The minimum type satisfying this is the intersection ΦSQUARE [σ] $\cap$ ΔXYCOORD [σ]; let us give this type the name: ΦXY-SQUARE [σ]. The result is therefore only well-typed for

$\forall(\sigma \subseteq \Phi$XY-SQUARE $[\sigma])$. Note that we have constructed this constraint from the argument types of the mixin and the chosen superclass, without requiring foreknowledge of the result type. In a type-checking algorithm, the form of the generator ΦXY-SQUARE can be constructed mechanically from the fields of the generators ΔXYCOORD and ΦSQUARE.

$$
\begin{array}{l}
\Phi\text{xy-square} = \Lambda(\sigma \subseteq (\Delta\text{XYCOORD}\ [\sigma] \cap \Phi\text{SQUARE}\ [\sigma])).\lambda(\text{self}: \sigma). \\
\qquad \Sigma\text{xycoord}\ [\sigma, \Phi\text{SQUARE}\ [\sigma]]\ (\text{self}, \Phi\text{square}\ [\sigma]\ (\text{self})) \\
\\
= \Lambda(\sigma \subseteq (\Delta\text{XYCOORD}\ [\sigma] \cap \Phi\text{SQUARE}\ [\sigma])).\lambda(\text{self}: \sigma). \\
\qquad \{\text{side} \mapsto 10,\ \text{area} \mapsto (\text{self.side} * \text{self.side}), \\
\qquad\qquad \text{equal} \mapsto \lambda(\text{other: t}).(\text{self.side} = \text{other.side})\} \\
\qquad \oplus\ \{\text{x} \mapsto 0,\ \text{y} \mapsto 0,\ \text{equal} \mapsto \lambda(\text{other}: \sigma). \\
\qquad\qquad\qquad (\text{self.side} = \text{other.side} \wedge \text{self.x} = \text{other.x} \wedge \text{self.y} = \text{other.y}), \\
\qquad\qquad \text{move} \mapsto \lambda(\text{a: INTEGER}).\lambda(\text{b: INTEGER}).(\text{x} := \text{a};\ \text{y} := \text{b};\ \text{self})\} \\
\\
= \Lambda(\sigma \subseteq (\Delta\text{XYCOORD}\ [\sigma] \cap \Phi\text{SQUARE}\ [\sigma])).\lambda(\text{self}: \sigma). \\
\qquad \{\text{x} \mapsto 0,\ \text{y} \mapsto 0,\ \text{equal} \mapsto \lambda(\text{other}: \sigma). \\
\qquad\qquad\qquad (\text{self.side} = \text{other.side} \wedge \text{self.x} = \text{other.x} \wedge \text{self.y} = \text{other.y}), \\
\qquad\qquad \text{move} \mapsto \lambda(\text{a: INTEGER}).\lambda(\text{b: INTEGER}).(\text{x} := \text{a};\ \text{y} := \text{b};\ \text{self}), \\
\qquad\qquad \text{side} \mapsto 10,\ \text{area} \mapsto (\text{self.side} * \text{self.side})\ \}
\end{array}
$$

Figure 7: XY-Square Class Derived by Mixin Inheritance

In the construction of the mixed class, which we shall call Φ*xy-square*, both *self* and σ are parameterised. 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 ΦSQUARE $[\sigma]$ is the appropriate *super* type and Φ*square* $[\sigma]$ (*self*) is the *super* record. Internally, the mixin function combines two records: (Φ*square* $[\sigma]$ (*self*)) $\oplus$ (Δ*xycoord* $[\sigma]$ (*self*)). According to the polymorphic definition of $\oplus$, ΔXYCOORD $[\sigma]$ Ω ΦSQUARE $[\sigma]$ must hold in order for the result to have a well-defined type. Since Φ*square* and Δ*xycoord* have an *equal* field in common, Ω requires these fields to have the same type; in particular, instantiations of the *self*-type must be identical. This is observed by distributing σ to both generators.

7. Evaluating Dependent Second-Order Types

Our approach to typing mixins is essentially a trick that exploits second-order type dependency to avoid having to go to explicit higher-order quantification. The usual expectation is for a mixin to depend on a *range of superclasses*; intuitively, this leads one to give *super* the type of a *type function*, quantifying over generators:

$$\forall(\beta :: \text{TYPE} \rightarrow \text{TYPE}).\forall(\varepsilon \subseteq \Phi[\varepsilon]).\beta \rightarrow \varepsilon \rightarrow \beta[\varepsilon]\cap\varepsilon$$

The *Abel* group's final report provided a higher-order typing for mixins [Harr91a], in which *self*- and *super*-type variables ranged over type functions, rather than bounded types. We avoid higher-order complications by reversing the order of quantification for *super*- and *self*-types. Firstly, we are able to provide a second-order typing for the free-standing *self* of the extension record, irrespective of whatever type we eventually give to *super*. Secondly, we are able to construct a second-order type expression for *super* that depends directly on the *self*-type; we are therefore not forced to quantify over generators.

Our typing is technically more accurate than a proposed typing of the *super*-interface for mixins in [Hauc93], and more generally useful than the restricted scheme proposed in [BC90], which really only covers non-recursive simple *types*; the scheme presented here covers recursive polymorphic *classes*. Treatments of mixins which do not establish the pattern of mutual recursion [CP89] between *self* and *super* lose call-backs from *super*-methods to *self*. This can produce unwanted retrograde behaviour where a base class also provides versions of methods added by the mixin. Mitchell [Mitc90, CM92] has developed a type scheme which reasons negatively about methods which a record *must not possess*, to cover this contingency (and also because his type rules allow subsumption in the number of fields matched to a rule). Our scheme does not require this added safeguard, since we bind recursion variables consistently to the extended object's structure *after* mixin combination has taken place.

For simple theories of classification, in which the type of *self* is polymorphic but other types are monomorphic, 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.

We think that this kind of constraint 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, parameterised bounds for the *super*-type may be precalculated. In both schemes, assumptions about the result-type may be discharged by mechanically constructing new generators with intersection types, using the record combination algorithm.

The author would like to thank William Cook for inspiration and Kim Bruce for corrections and useful discussions which influenced the development of these ideas.

References

[AC95] M Abadi and L Cardelli (1995), 'On subtyping and matching', *Proc. 9th European Conf. Object-Oriented Prog.*, Aarhus, Denmark.

[Bruc94] K Bruce (1994), 'A paradigmatic object-oriented programming language: design, static typing and semantics', *J. of Func. Prog., 4(2)*, 127-206.

[BC90] G Bracha and W Cook (1990), 'Mixin-based inheritance', Proc. 5th *ACM Conf. Object-Oriented Prog. Sys., Lang. and Appl.* and *Proc. 9th European Conf. Object-Oriented Prog.*, 303-311.

[CCHO89a] P Canning, W Cook, W Hill, W Olthoff and J Mitchell (1989), 'F-bounded polymorphism for object-oriented programming', *Proc. 4th Int. Conf. Func. Prog. Lang. and Arch.*, Imperial College London, September, 273-280.

[CCHO89b] P Canning, W Cook, W Hill and W Olthoff (1989), 'Interfaces for strongly-typed object-oriented programming', *Proc. 4th ACM Conf. Object-Oriented Prog. Sys., Lang and Appl.*, 457-467.

[CDJG89] L Cardelli, J Donahue, L Glassman, M Jordan, B Kalsow and G Nelson (1989), 'Modula-3 report (revised)', *Tech. Rep. 52*, Digital Equipment Corporation Systems Research Centre.

[CHC90] W Cook, W Hill and P Canning (1990), 'Inheritance is not subtyping', *Proc. 17th ACM Symp. Principles of Prog. Lang.*, 125-135.

[CM92] L Cardelli and J Mitchell (1992), 'Operations on records (summary)', *Proc. 5th Int. Conf. Math. Found. Prog. Lang. Semantics*, pub. *LNCS, 442*, Springer Verlag, 22-52.

[Comp94] A Compagnoni (1994), 'Subtyping in $F^{\omega}_{\wedge}$ is decidable', *Technical Report ECS-LFCS-94-281*, University of Edinburgh, LFCS.

[Cook89] W Cook (1989), *A denotational semantics of inheritance*, PhD Thesis, Brown University.

[CP89] W Cook and J Palsberg (1989), 'A denotational semantics of inheritance and its correctness', *Proc. 4th ACM Conf. Object-Oriented Prog. Sys., Lang. and Appl.*, 433-443.

[CP93] A Compagnoni and B Pierce (1993), 'Multiple inheritance via intersection types', *Technical Report ECS-LFCS-93-275*, University of Edinburgh, LFCS.

[CW85] L Cardelli and P Wegner (1985), 'On understanding types, data abstraction and polymorphism', *ACM Computing Surveys, 17(4)*, 471-521.

[Harr91] W Harris (1991), *Typed Object-Oriented Programming: ABEL Project Posthumous Report*, Hewlett-Packard Laboratories.

[Hauc93] F J Hauck (1993), 'Inheritance modelled with explicit bindings: an approach to typed inheritance', *Proc. 8th ACM Conf. Object-Oriented Prog. Sys., Lang. and Appl.*, pub. *Sigplan Notices, 28(10)*, 231-239.

[Keen89] S E Keene (1989), *Object-Oriented Programming in Common Lisp: a Programmer's Guide to CLOS*, Addison-Wesley and Symbolics Press, Reading MA.

[Lamp93] J Lamping (1993), 'Typing the specialisation interface', *Proc. 8th ACM Conf. Object-Oriented Prog. Sys., Lang. and Appl.*, pub. *Sigplan Notices, 28(10)*, 201-214.

[Mads93] O L Madsen (1993), *Object-Oriented Programming in the Beta Programming Language*, Addison-Wesley.

[Mitc90] J C Mitchell (1990), 'Towards a typed foundation for method specialisation and inheritance', *Proc. 17th ACM Symp. on Principles of Prog. Langs.*, 109-124.

[Moon86] D A Moon (1986), 'Object-oriented programming with Flavors', *Proc. 1st ACM Conf. Object-Oriented Prog. Sys., Lang. and Appl.*, pub *ACM Sigplan Notices, 21(11)*, 1-6.

[Pier92] B Pierce (1992), 'Intersection types and bounded polymorphism', *Technical Report ECS-LFCS-92-200*, University of Edinburgh, LFCS.