JOURNAL OF OBJECT TECHNOLOGY

Online at http://www.jot.fm. Published by ETH Zurich, Chair of Software Engineering ©**JOT, 2004**

Vol. 3, No. 10, November-December 2004

The Theory of Classification Part 15: Mixins and the Superclass Interface

Anthony J H Simons, Department of Computer Science, University of Sheffield, U.K.

1 INTRODUCTION

This is the fifteenth article in a regular series on object-oriented type theory for non-specialists. Earlier articles have built up λ-calculus models of objects [1], classes [2], inheritance [3, 4] and generic template types [5]. These features are common to a number of popular object-oriented languages, such as C++, Eiffel and Java (which now has templates in the latest version). In this article, we look at a less well known, but once popular construct in object-oriented languages called a *mixin*.

Mixins were first proposed in the language Flavors [7]. A mixin is best described as a freestanding component extension, something that is intended to be added onto another class using the inheritance mechanism. A mixin can be combined with many different base classes, to yield different extended classes which contain the combined base and mixin features. Some mixins provide orthogonal functionality that can be added to any class. Other mixins expect the class with which they are combined to provide certain operations, because the mixin's own methods depend on them. In other words, a mixin has a superclass interface, describing the kind of class from which it expects to inherit. By examining mixins formally, we can learn more about the type constraints on inheritance.

2 FLAVORS AND MIXINS

Flavors [6, 7] was an important early object-oriented language, developed at MIT in the late 1970s. It was the first to introduce *multiple inheritance*, the idea that a child class may have more than one parent class and combine all the inherited features in some principled way. As legend has it, this idea was inspired by the presence of several famous ice-cream parlours in the vicinity of MIT. When visiting these emporia, you could choose

Cite this column as follows: Anthony J.H. Simons: "The Theory of Classification, Part 15: Mixins and the Superclass Interface", in *Journal of Object Technology*, vol. 3, no. 10, November-December 2004, pp. 7-18. http://www.jot.fm/issues/issue_2004_11/column1

your basic vanilla ice-cream and then mix in one of several other flavours[1], such as pistachio or strawberry. By analogy, the root class in the new language was called "Vanilla Flavor" and other classes were extensions of this. The idea of a "mixin" was inspired by the extra sauces and toppings you could add to your ice-cream. A mixin is not itself a whole class, but rather a package of optional features that you can choose to add to a class. It is "mixed in" in the sense that, through the mechanism of inheritance, the mixin's features may become interwoven with the features of the class with which it is combined. Mixins were the first attempt to provide flexible solutions to some of the same problems that are currently addressed using *aspects* in aspect-oriented programming, which are woven together in a similar way.

A simple example of a mixin might be an extension that adds a coordinate position to any other object. The classes in a library may exist without reference to any coordinate position, for example, a Truck class might describe the intrinsic properties of trucks, and might be just one of many Vehicle subclasses. Then, for a given simulation application, it is desired that some of these classes be locatable within a coordinate system. In Flavors you could create the extended types quickly by combining the canonical types with a Locatable mixin, to yield locatable versions of each class:

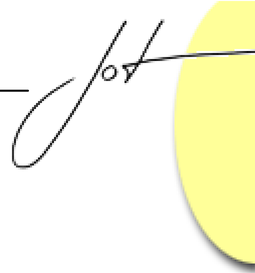
```
(defflavor simulation-truck () (truck locatable-mixin))
```

The syntax of Flavors may not be familiar to many readers. The language was built on top of Lisp. To construct a class, you called the Lisp function *defflavor* and provided it with a list of attributes (methods were defined separately). If this class inherited from other classes, they were supplied in a second list. The syntax looked something like:

```
(defflavor class-name (var-1, var-2, … var-n)
      (super-1, super-2, … super-n))
```

where the various *super* classes are the names of other classes to be "mixed in" with the new class. There was no real syntactic distinction between a class and a mixin, merely a naming convention, whereby mixin names always ended in "-mixin". Folding in a number of mixins required the linearisation of their properties – in fact, this was no different from the general problem of multiple inheritance. In Flavors, the inheritance algorithm combined features from the superclasses in left-to-right order, merging identically-named attributes, provided that all the recursive orderings declared by the superclasses could be preserved [7].

[1] With apologies to US readers, I and my spell-checker prefer British spelling, but proper names like "Flavors" are allowed to remain in their proprietary form!



3 TEMPLATES AND ABSTRACT SUBCLASSES

To illustrate the idea of mixins in another way, we may use the template mechanism in C++ to define “abstract subclasses”. The Locatable mixin described above might be simulated in C++ as the following “abstract subclass”:

```
template <class Any>
class Locatable : public Any {
public:
      Locatable();
      void moveTo(int x, int y);
      Point position() const;
private:
      Point point;  // my position
};

template <class Any>
Locatable::Locatable() : Any(), point(0, 0) {}

template <class Any>
void LocatableMixin::moveTo(int x, int y) {
      point.moveTo(x, y);
}

template <class Any>
Point Locatable::position() const {
      return point;
}
```

Listing 1: C++ definition of an “abstract subclass”

Locatable is defined like a C++ subclass, but inherits from a *type parameter* Any, which has the effect of delaying the combination of local features with the (as yet unknown) inherited features from some eventual base class. Locatable versions of the various Vehicle subclasses may be constructed on the fly, in the style:

```
Locatable<Car> car1;
Locatable<Bus> bus1;
```

at which point the Any parameter is bound to the specific types Car and Bus, respectively. Any C++ class which inherits from a type parameter is an “abstract subclass”. The parameter helps to underline how it expects to be combined with some unknown base class.

4 MIXINS VERSUS ABSTRACT SUBCLASSES

The difference between this “abstract subclass” approach and the earlier “mixin” approach is that Flavors defines each mixin component independently, as a freestanding extension. Combining mixins is more like multiple inheritance in C++ (with virtual base

classes), in which the programmer derives a new subclass which "mixes" all the components. The abstract subclass approach is more like a wrapper function, which expects to be applied to some base object denoting *super*, and then combines the additional fields with the base fields yielding the subtype directly.

Bracha and Cook described a mixin as "an abstract subclass" or "a subclass definition that may be applied to different superclasses" [8]. Here, we would prefer a slightly more careful use of the term "mixin", since we want to draw a formal difference between a subclass and a mixin, which we can illustrate using the model of inheritance from earlier articles [3, 4]. Recall that inheritance is modelled as the combination of records using the $\oplus$ union with override operator:

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

In this, *base* is the parent object and *derived* is the subclass object, constructed by combining *base* with a record of *extra* fields. It is clear that *derived* is the result of the combination, a whole subclass object, rather than just an extension. On the other hand, the *extra* record of additional fields is exactly what we mean by a mixin. The notion of an abstract subclass should therefore be modelled as a function:

$$\begin{aligned} &\text{absub} = \lambda b.(b \oplus \text{extra}) \\ &\text{derived} = \text{absub}(\text{base}) \end{aligned}$$

and this illustrates the difference : the abstract subclass is a function which includes the inheritance operator, whereas the mixin is simply a record of extra fields.

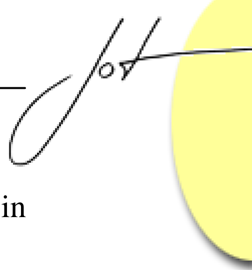
5 MIXINS AS EXTENSION TYPES

Most object-oriented languages don't encourage the specification of mixins *in isolation*, but instead, records of extra fields are typically declared within the scope of a complete subclass definition (like the example in section 3 above). The reasons for this have to do with self-reference and the typing of inheritance. When a conventional object subtype is defined (in the style of Java or C++), references to the self-type in the extension are equivalent to the eventual subtype [2], rather than the type of the extension. We can illustrate this with a Point subtype that extends an Object base type:

$$\begin{aligned} \text{Object} &= \mu\sigma.\{\text{identity} : \rightarrow \sigma\} \\ &\Rightarrow \{\text{identity} : \rightarrow \text{Object}\}, \qquad \text{after unrolling.} \end{aligned}$$

$$\begin{aligned} \text{Point} &= \mu\sigma.(\text{Object} \cup \{x :\rightarrow \text{Integer}, y :\rightarrow \text{Integer}, \text{equal} : \sigma \rightarrow \text{Boolean}\}) \\ &\Rightarrow \{\text{identity} : \rightarrow \text{Object}, x :\rightarrow \text{Integer}, y :\rightarrow \text{Integer}, \\ &\qquad \text{equal} : \text{Point} \rightarrow \text{Boolean}\}, \qquad \text{after unrolling} \end{aligned}$$

Note that the extension record has an *equal* method accepting the self-type σ, which after unrolling the recursion, is equivalent to Point. This is because σ is recursively bound



outside the union of base fields and extra fields, to the result of the union. The self-type in the extension record is therefore not independent, but refers to the subtype.

If Java or C++ allowed the programmer to declare mixins as freestanding records of extra fields to be added to any class, these would have an independent self-type of their own:

$$\begin{aligned} \text{PointMixin} &= \mu\sigma.\{x :\rightarrow \text{Integer}, y :\rightarrow \text{Integer}, \text{equal} : \sigma \rightarrow \text{Boolean}\} \\ &\Rightarrow \{x :\rightarrow \text{Integer}, y :\rightarrow \text{Integer}, \\ &\qquad \text{equal} : \text{PointMixin} \rightarrow \text{Boolean}\}, \qquad \text{after unrolling} \end{aligned}$$

$$\begin{aligned} \text{Point} &= \text{Object} \cup \text{PointMixin} \\ &\Rightarrow \{\text{identity} : \rightarrow \text{Object}, x :\rightarrow \text{Integer}, y :\rightarrow \text{Integer}, \\ &\qquad \text{equal} : \text{PointMixin} \rightarrow \text{Boolean}\}, \qquad \text{after unrolling} \end{aligned}$$

Note how the self-type σ of the PointMixin is bound independently, to refer recursively to the PointMixin type. After combination with the Object type, this yields a Point type in which self-type reference is entirely schizophrenic [3]: the inherited self-type is Object, and the extension self-type is PointMixin, but nowhere is the self-type equivalent to the Point type! So, for this reason, languages based on subtyping would have trouble dealing with self-reference if they wished to admit freestanding types as mixins.

6 MIXINS AS EXTENSION GENERATORS

Flavors is a language, like Smalltalk and Eiffel, in which *self* is rebound during inheritance to refer to the subclass instance. Accordingly, the self-type evolves during inheritance to refer to the subclass's type. In earlier articles, we found that *type generators* could be used to describe this kind of flexibility in the self-type [2, 3]. The type of a mixin can be expressed using a type generator, instead of a fixed type:

$$\text{GenPointMixin} = \lambda\sigma.\{x :\rightarrow \text{Integer}, y :\rightarrow \text{Integer}, \text{equal} : \sigma \rightarrow \text{Boolean}\}$$

In this, the self-type σ is a parameter introduced by λ, and is not yet bound to any specific type. Given a similar generator for the Object type, we can construct a generator for the Point type, by unifying the self-types of the base and mixin generators in the combination:

$$\text{GenObject} = \lambda\sigma.\{\text{identity} : \rightarrow \sigma\}$$

$$\begin{aligned} \text{GenPoint} &= \lambda\tau.(\text{GenObject}[\tau] \cup \text{GenPointMixin}[\tau]) \\ &\Rightarrow \lambda\tau.\{\text{identity} : \rightarrow \tau, x :\rightarrow \text{Integer}, y :\rightarrow \text{Integer}, \\ &\qquad \text{equal} : \tau \rightarrow \text{Boolean}\} \end{aligned}$$

$$\text{Point} = (\mathbf{Y}\ \text{GenPoint})$$
$$\Rightarrow \{\text{identity} : \rightarrow \text{Point},\ x :\rightarrow \text{Integer},\ y :\rightarrow \text{Integer},$$
$$\text{equal} : \text{Point} \rightarrow \text{Boolean}\}, \quad \text{after unrolling.}$$

Here, the subclass generator GenPoint introduces a new self-type τ and propagates this into both the base generator GenObject and the extension generator GenPointMixin (by applying them to the new type), so creating two record types which refer to the same self-type τ. After the union of fields, τ refers homogeneously to the self-type. After fixing the recursion, τ is bound to the desired Point type.

7 BOUND AND FREE MIXINS

We characterise mixins as either *bound* or *free*, to denote whether or not they depend on their superclass. The GenPointMixin type generator above was defined as though it were the type of a free mixin, capable of being combined with any other object, since its methods were assumed not to interact with any superclass behaviour. To examine this in more detail, we can build an *object generator* to represent the implementation of the mixin [4]:

$$\text{freePointMixin} = \lambda\text{self.}\{\ x \mapsto 2,\ y \mapsto 3,$$
$$\text{equal} \mapsto \lambda p.(\ \text{self.}x = p.x \wedge \text{self.}y = p.y)\}$$

The body of the implementation has no dependency on any super-object, illustrating the independence of the mixin. One possible weakness in this design is that the *equal* method can only compare the local *x* and *y* values. If this were "mixed in" with some other base object with an *equal* method, the inherited method would be overridden by the mixin's version.

To illustrate this, we introduce the object generator genSquare, representing a geometric square with its own *side* and *equal* methods:

$$\text{genSquare} = \lambda\text{self.}\{\text{side} \mapsto 5,\ \text{equal} \mapsto \lambda s.(\text{self.side} = s.\text{side})\}$$

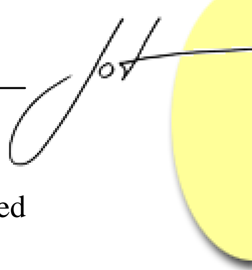
We may seek to combine the genSquare and freePointMixin generators by inheritance; the resulting generator genLocSquare represents a locatable square:

$$\text{genLocSquare} = \lambda\text{self.}(\ \text{genSquare(self)} \oplus \text{freePointMixin(self)}\ )$$

$$= \lambda\text{self.}(\ \{\text{side} \mapsto 5,\ \text{equal} \mapsto \lambda s.(\text{self.side} = s.\text{side})\} \ \oplus$$
$$\{x \mapsto 2,\ y \mapsto 3,\ \text{equal} \mapsto \lambda p.(\text{self.}x = p.x \wedge \text{self.}y = p.y)\}\ )$$

$$= \lambda\text{self.}\{\text{side} \mapsto 5,\ x \mapsto 2,\ y \mapsto 3,$$
$$\text{equal} \mapsto \lambda p.(\text{self.}x = p.x \wedge \text{self.}y = p.y)\}$$

Although two versions of *equal* are present before record combination, the operator $\oplus$ prefers fields from the right-hand side, replacing any identically-named fields on the left.



The inherited version of *equal* is therefore overridden and the version of *equal* obtained in the result is insufficient, because we can no longer compare the *sides* of squares.

Since *equal* is a common method and is likely to exist in most classes, we would prefer our mixin to adapt the *equal* method of its base class, rather than replace it wholesale. In an earlier article [9] we showed how inherited methods could be adapted by *method combination*, in which a redefined version of the method calls the original version through the *super* variable. To make inherited methods available to a mixin, we have to supply it with a variable standing for the *super*-object. The following is an *object generator* for a bound mixin, whose implementation depends on both *self* and *super* variables. The *super* variable will be bound later to a superclass instance:

$$\text{boundPointMixin} = \lambda\text{self. } \lambda\text{super.}\{\text{x} \mapsto 2,\ \text{y} \mapsto 3,\ \text{equal} \mapsto \lambda\text{p.(super.equal(p)} \wedge \text{self.x} = \text{p.x} \wedge \text{self.y} = \text{p.y)}\}$$

The dependency on the super-object is evident in the revised body of the *equal* method, which calls *super.equal(p)* before comparing the respective *x* and *y* values. Note how a bound mixin generator must always have an extra argument, λ*super*, to bind to the eventual base object.

Once again, we combine the genSquare generator with the boundPointMixin generator to obtain a generator for the locatable square, genLocSquare. Note in passing how *self* is reintroduced outside of the record combination, and how both generators accept this new value of *self*, to ensure uniform *self*-reference. In addition, the boundPointMixin receives an actual argument *genSquare(self),* representing the value of *super*; in fact this same expression denotes the base object on the left-hand side of record combination:

$$\text{genLocSquare} = \lambda\text{self.(genSquare(self)} \oplus \text{boundPointMixin(self, genSquare(self)))}$$

$$= \lambda\text{self.(}\{\text{side} \mapsto 5,\ \text{equal} \mapsto \lambda\text{s.(self.side} = \text{s.side)}\} \oplus \{\text{x} \mapsto 2,\ \text{y} \mapsto 3,\ \text{equal} \mapsto \lambda\text{p.((}\lambda\text{s.(self.side} = \text{s.side) p)} \wedge \text{self.x} = \text{p.x} \wedge \text{self.y} = \text{p.y)}\}\text{)}$$

$$= \lambda\text{self.}\{\text{side} \mapsto 5,\ \text{x} \mapsto 2,\ \text{y} \mapsto 3,\ \text{equal} \mapsto \lambda\text{p.(self.side} = \text{p.side} \wedge \text{self.x} = \text{p.x} \wedge \text{self.y} = \text{p.y)}\}$$

After simplification, the result has exactly the desired implementation of a generator for a locatable square object. The *super.equal(p)* expression is expanded during this simplification to yield the body of the inherited method, which is combined using logical *and* with the further parts of the redefined *equal* method.

8 THE SUPERCLASS INTERFACE

We now want to add types to the bound mixin implementation described above. In this, we shall need to provide polymorphic types for *self* and for *super*. The typing of *self* is

relatively straightforward, but the typing of *super* is shown below to be much more difficult. It is particularly desirable to try to establish the type of *super*, since this captures exactly the *superclass interface* of a mixin, describing the type of object with which it expects to be combined. However, the type of *super* is made more complex by the fact that the eventual *self*- and *super*-types must stand in a subtyping relationship. So, although these two types would appear on the surface to be independent, they are in fact related. Other treatments of typed mixins [11] have expressed this by introducing the super-type first, then a dependent self-type. Our novel approach reverses the order of dependency, introducing the self-type first, then a dependent super-type.

Previously, we showed how the type of *self* in an object generator could be given by an F-bound constructed from the corresponding type generator [10]. All combinations with the boundPointMixin form a class with at least the *x, y* and *equal* methods in their interface. The type generator GenPointMixin (from section 6) already describes this interface:

$$\text{GenPointMixin} = \lambda\sigma.\{\text{x} :\rightarrow \text{Integer}, \text{y} :\rightarrow \text{Integer}, \text{equal} : \sigma \rightarrow \text{Boolean}\}$$

such that we may give a polymorphic type σ for *self* ranging over all those types with (at least) these methods:

$$\forall(\sigma <: \text{GenPointMixin}[\sigma]) \text{ . self} : \sigma$$

What is unusual about this is that it does not depend on the type of *super* in any way. We would normally have expected to use a type generator of two arguments, σ and τ, reflecting the two-argument structure of the object generator. The reason why we can ignore the *super*-type τ is because it *never appears* in the public interface of the class – it is irrelevant! This allows us to introduce the self-type σ independently, using the simpler type generator.

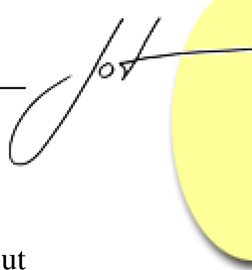
The polymorphic type τ of *super* may now be expressed as a range of types within certain bounds. The lower bound is the type σ of *self* (because *self* : σ must eventually stand in a subtyping relationship with *super* : τ), which gives rise to the lower bound condition:

$$\forall(\sigma <: \text{GenPointMixin}[\sigma]) \text{ . } \forall(\tau \mid \sigma <: \tau) \text{ . super} : \tau$$

The upper bound is expressed in terms of a minimum type, determined by examining the methods that are invoked on the *super* variable, then constructing an interface which supports at least these methods. We can only do this by internal inspection of the object generator implementation. In the example above, the boundPointMixin expects the *super* object to have at least an *equal* method. Accordingly, we may construct a type-generator representing the interface of all objects possessing (just) the *equal* method:

$$\text{GenEqual} = \lambda\tau.\{\text{equal} : \tau \rightarrow \text{Boolean}\}$$

and from this create the upper bound condition:



$$\forall(\sigma <: GenPointMixin[\sigma]) . \forall(\tau <: GenEqual[\sigma]) . super : \tau$$

What is different here is that the constraint is not expressed as: $\tau <: GenEqual[\tau]$, but rather in terms of the self-type σ. This is because, at the time the super-type is bound, all generator-types will be adapted to the current self-type σ. Combining the lower and upper bound constraints on the *super*-type yields the range:

$$\forall(\sigma <: GenPointMixin[\sigma]) . \forall(\tau \mid \sigma <: \tau <: GenEqual[\sigma]) . super : \tau$$

This, finally, is the type of the superclass interface! It is quite complicated, but intuitively expresses the idea that the eventual type of *super* is a supertype of *self* and a subtype of the interface providing the *equal* method.

9 TYPED MIXIN COMBINATION

We may now observe the interplay of types when we combine a typed version of the boundPointMixin with a typed version of the canonical square. First, we attach types to the mixin, as determined above:

$$\begin{aligned}
&boundPointMixin : \forall(\sigma <: GenPointMixin[\sigma]) . \\
&\qquad\qquad \forall(\tau \mid \sigma <: \tau <: GenEqual[\sigma]) . \sigma \rightarrow \tau \rightarrow GenPointMixin[\sigma] \\
&= \lambda(\sigma <: GenPointMixin[\sigma]). \lambda(\tau \mid \sigma <: \tau <: GenEqual[\sigma]). \\
&\qquad \lambda(self : \sigma). \lambda(super : \tau).\{x \mapsto 2, y \mapsto 3, \\
&\qquad\qquad equal \mapsto \lambda(p : \sigma).(super.equal(p) \wedge \\
&\qquad\qquad\qquad self.x = p.x \wedge self.y = p.y)\}
\end{aligned}$$

The typed version of the canonical square is given in the usual way by:

$$GenSquare = \lambda\sigma.\{side : \rightarrow Integer, equal : \sigma \rightarrow Boolean\}$$

$$\begin{aligned}
&genSquare : \forall(\sigma <: GenSquare[\sigma]). \sigma \rightarrow GenSquare[\sigma] \\
&= \lambda(\sigma <: GenSquare[\sigma]). \lambda(self : \sigma). \\
&\qquad \{side \mapsto 5, equal \mapsto \lambda(s : \sigma).(self.side = s.side)\}
\end{aligned}$$

The typed locatable square is to be derived by adding the point mixin to the canonical square. First, we establish the resulting type generator, GenLocSquare:

$$GenLocSquare = \lambda\sigma. (GenSquare[\sigma] \cup GenPointMixin[\sigma])$$

$$= \lambda\sigma.\{ side : \rightarrow Integer, x : \rightarrow Integer, y : \rightarrow Integer, equal : \sigma \rightarrow Boolean \}$$

The typed locatable square is then given by the "mixed in" combination:

$$\begin{aligned}
&genLocSquare : \forall(\sigma <: GenLocSquare[\sigma]). \sigma \rightarrow GenLocSquare[\sigma] \\
&= \lambda(\sigma <: GenLocSquare[\sigma]). \lambda(self : \sigma) . \\
&\qquad (genSquare(self) \oplus boundPointMixin(self, genSquare(self)))
\end{aligned}$$

$$= \lambda(\sigma <: \text{GenLocSquare}[\sigma]).\ \lambda(\text{self} : \sigma)\ .\{\text{side} \mapsto 5,\ x \mapsto 2,\ y \mapsto 3,\ \text{equal} \mapsto \lambda(p : \sigma).(\text{self.side} = \text{p.side} \wedge \text{self.x} = \text{p.x} \wedge \text{self.y} = \text{p.y})\}$$

The result is a typed object generator for a locatable square, exactly as desired.

We now want to check whether the type constraints of the typed mixin generator were properly observed. In the combination, the mixin generator was called with:

- the new value of *self*, having the reintroduced self-type: $\sigma <$ GenLocSquare[σ]
- the value of *super*, *genSquare(self),* having the adapted type: GenSquare[σ]

The self-type σ was expected to satisfy: $\forall(\sigma <:$ GenPointMixin[σ]). This holds by the rule of classification [3]. Since the two generators stand in a pointwise subtyping relationship:

$$\forall\tau\ .\ \text{GenLocSquare}[\tau] <: \text{GenPointMixin}[\tau]$$

then if $\sigma <:$ GenLocSquare[σ] then it follows that $\sigma <:$ GenPointMixin[σ].

The super-type GenSquare[σ] was expected to satisfy: $\forall(\tau \mid \sigma <: \tau <:$ GenEqual[σ]). We can check this by the substitution of {GenSquare[σ] / τ} to see if both the lower and upper bound conditions hold. Firstly, we examine the lower bound:

$$\forall(\sigma < \text{GenLocSquare}[\sigma])\ .\ \sigma <: \text{GenSquare}[\sigma]$$

This holds, because the two generators stand in a pointwise subtyping relationship:

$$\forall\tau\ .\ \text{GenLocSquare}[\tau] <: \text{GenSquare}[\tau]$$

therefore if $\sigma <:$ GenLocSquare[σ] then it follows that $\sigma <:$ GenSquare[σ]. Secondly, we examine the upper bound:

$$\forall(\sigma < \text{GenLocSquare}[\sigma])\ .\ \text{GenSquare}[\sigma] <: \text{GenEqual}[\sigma]$$

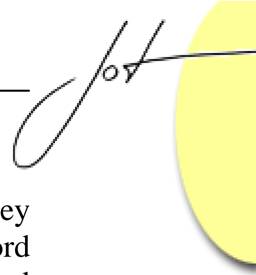
Again, we can show that the two generators stand in a pointwise subtyping relationship for all possible types τ:

$$\forall\tau\ .\ \text{GenSquare}[\tau] <: \text{GenEqual}[\tau]$$

therefore they still stand in this relationship for some subset of types $\sigma \subseteq \tau$. So, we have demonstrated that mixins can be typed and applied to base classes which properly satisfy the superclass interface.

10 CONCLUSION

We started this article by presenting the concept of a mixin. A mixin is a freestanding record of extra fields, intended to be combined with any other object. In some sense, mixins are the primitive building blocks in languages with inheritance-like mechanisms. Bracha and Cook revived interest in mixins when they demonstrated how the models of inheritance in languages as diverse as Smalltalk, Beta and CLOS could all be mapped



onto a simpler model based on the composition of mixins [8]. However, the typing they gave was based on simple first-order types, resulting in fragmented self-types after record combination. This is why mixins don't receive so much attention in Java and C++, and one reason why multiple inheritance is less useful in languages like C++ which are based on simple subtyping.

Mixins are much more interesting in those languages which modify self-reference and the self-type during inheritance. They then have some of the power of *aspects* in aspect-oriented programming, since they can weave in extra attributes and methods and even adapt the course of an inherited method through method combination. However, the formal characterisation of mixins was previously thought difficult. In particular, it was thought that the type of the superclass interface was impossible to express without going to higher order logics. This is because *super* apparently ranges over a set of classes, so quantification should have to range over sets of generators (type *functions*), rather than just over sets of simple types. In earlier work by Harris and others [11], the higher-order type of *super* was introduced first, and then the type of *self*, which depended on the type of *super*. Here, we showed how it is possible to provide a second-order type for *super*, by reversing the order in which the self-type and super-type are introduced. This provided an elegant typing for mixins, which was checked using an example of typed mixin combination.

REFERENCES

[1] A J H Simons: "The theory of classification, part 3: Object encodings and recursion", in *Journal of Object Technology*, vol. 1, no. 4, September-October 2002, pp. 49-57. http://www.jot.fm/issues/issue_2002_09/column4

[2] A J H Simons: "The theory of classification, part 7: A class is a type family", in *Journal of Object Technology*, vol. 2, no. 3, May-June 2003, pp. 13-22. http://www.jot.fm/issues/issue_2003_05/column2

[3] A J H Simons: "The theory of classification, part 8: Classification and inheritance", in *Journal of Object Technology*, vol. 2, no. 4, July-August 2003, pp. 55-64. http://www.jot.fm/issues/issue_2003_07/column4

[4] A J H Simons: :The theory of classification, part 9: Inheritance and self-reference", in *Journal of Object Technology*, vol. 2, no. 6, November-December 2003, pp. 25-34. http://www.jot.fm/issues/issue_2003_11/column2

[5] A J H Simons: "The theory of classification, part 13: Template Classes and Genericity", in *Journal of Object Technology*, vol. 3, no. 7, July-August 2004, pp. 15-25. http://www.jot.fm/issues/issue_2004_07/column2

[6] H Cannon, *Flavors, Technical Report* (Cambridge: MIT AI Laboratory, 1980).

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

[8] G Bracha and W Cook, "Mixin-based inheritance", *Proc. 5th ACM Conf. Object-Oriented Prog. Sys., Lang. and Appl.* and *Proc. 4th European Conf. Object-Oriented Prog., pub. ACM Sigplan Notices, 25(10)* (ACM Sigplan, 1990), 303-311.

[9] A J H Simons: "The theory of classification, part 10: Method combination and super-reference", in *Journal of Object Technology*, vol. 3, no. 1, January-February 2004, pp. 43-53. http://www.jot.fm/issues/issue_2004_01/column4

[10] A J H Simons: "The theory of classification, part 11: Adding class types to object implementations", in *Journal of Object Technology*, vol. 3, no. 3, March-April 2004, pp. 7-19. http://www.jot.fm/issues/issue_2004_03/column1

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

About the author

Anthony Simons is a Senior Lecturer and Director of Teaching in the Department of Computer Science, University of Sheffield, where he leads object-oriented research in verification and testing, type theory and language design, development methods and precise notations. He can be reached at a.simons@dcs.shef.ac.uk.