White Rose Research Online

eprints@whiterose.ac.uk
https://eprints.whiterose.ac.uk
Universities of Leeds, Sheffield and York

Deposited via The University of Sheffield.

White Rose Research Online URL for this paper:
https://eprints.whiterose.ac.uk/id/eprint/79265/

Version: Published Version

Article:
Simons, A.J.H. (2004) The theory of classification part 11: adding class types to object implementations. Journal of Object Technology, 3 (3). 7 - 19. ISSN: 1660-1769

Reuse
Items deposited in White Rose Research Online are protected by copyright, with all rights reserved unless indicated otherwise. They may be downloaded and/or printed for private study, or other acts as permitted by national copyright laws. The publisher or other rights holders may allow further reproduction and re-use of the full text version. This is indicated by the licence information on the White Rose Research Online record for the item.

Takedown
If you consider content in White Rose Research Online to be in breach of UK law, please notify us by emailing eprints@whiterose.ac.uk including the URL of the record and the reason for the withdrawal request.

UNIVERSITY OF LEEDS
University of Sheffield
UNIVERSITY of York

JOURNAL OF OBJECT TECHNOLOGY

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

Vol. 3, No. 3, March-April 2004

The Theory of Classification Part 11: Adding Class Types to Object Implementations

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

1 INTRODUCTION

This is the eleventh article in a regular series on object-oriented type theory, aimed specifically at non-theoreticians. In the *Theory of Classification*, we have so far considered the typeful aspect of classes [1, 2] and their implementation aspect [3, 4] separately. We have been concerned to point out how the notion of classification has a fully formal interpretation [2], in which, at the typeful level, a *class* is distinct from a *type* [1]. Likewise, we have explored the implementation level, in order to understand the operation of inheritance on objects [3] and give a precise meaning to the pseudo-variables *self* [3] and *super* [4] in different object-oriented languages.

Eventually, we must link the type and implementation aspects together, since this is how type rules are properly presented [5]. In a type rule, the aim is to be able to derive the resulting type of some expression, given the types of the values that make up this expression. We would like to show, for example, that the result of extending an object with extra fields is itself a well-typed expression. To do this, we must somehow attach the class-type information to the object-values. Furthermore, we introduced a special operator ⊕ to model inheritance [3]. We must also show that inheritance itself is a well-typed operation. This will involve examining the types of the objects that we pass as operands to ⊕, which was defined in a polymorphic way.

We started with a calculus of class-types [1, 2] and developed a separate, but related, calculus of object-values [3, 4]. In this article, we seek to develop a calculus of *typed objects*, in which the type information is attached to the object information. With this, we shall be able to infer the types of object definitions created via inheritance.

Cite this column as follows: Anthony 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

2 LINKING VALUES AND SIMPLE TYPES

To introduce the new typed calculus, we shall review the different λ-calculus styles presented so far. Imagine a functional language (like Lisp, ML, or the functional subset of C) in which we want define our own *negate* function to flip the sign of a number. In the *untyped* λ-calculus, we can define *negate* as follows:

$$\text{negate} = \lambda x.(-x)$$

since it takes an argument *x* and returns a body, in which *x* is negated using primitive minus "-" (which we assume exists already). What is the type of this function? So far, this is not specified – we could be negating an Integer, a Real or even a Natural (unsigned!) number, or worse still, something which is not even a number. Let us further assume that we want *negate* to apply to Integers, rather than any of the other types. To assert this, we give *negate* a type signature and attach type information to the implementation of the function:

$$\text{negate} : \text{Integer} \rightarrow \text{Integer}$$
$$= \lambda(x : \text{Integer}).(-x)$$

The type declaration says that *negate* takes an Integer and returns an Integer result. On the next line, the implementation is given in the *simply typed* λ-calculus, in which λ(*x* : Integer) declares again that the argument *x* is of the Integer type. In this style of writing, we don't bother to annotate the type of the function body explicitly, since the result type was declared beforehand. It could also be inferred using other type rules for "-", which are not shown here.

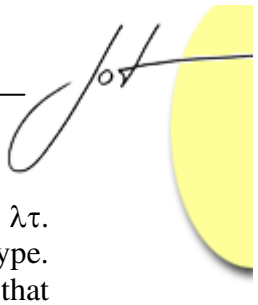
The main thing to note is the style of declaration. A typed function is always declared by giving its type signature, then defining its implementation, in which type information is attached to the argument variables. We first discussed this in the earlier article [1].

3 LINKING VALUES AND POLYMORPHIC TYPES

We now want to generalise the typing of *negate*, to indicate that it can flip the sign of all kinds of numeric types. The *polymorphic typed* λ-calculus allows us to define functions that accept both type- and value-arguments. We could define a very general version of *negate* as follows:

$$\text{negate} : \forall\tau.\ \tau \rightarrow \tau$$
$$= \lambda\tau.\lambda(x : \tau).(-x)$$

This is rather more general than we actually want. In the type signature, it says that *negate* is defined for *all types* τ, then accepts an argument in this type τ and returns a result of the same type. We shall fix this later, so that *negate* only applies to signed, numeric types. For



the moment, note how the implementation is prefixed with an extra type parameter: λτ. This says that *negate* accepts an actual type argument, followed by a value in this type. This means that we have to apply *negate* to two arguments, first a type, then a value of that type:

negate [Integer] (3 : Integer) ⇒ -3
negate [Real] (2.1 : Real) ⇒ -2.1

To distinguish type-application from value-application, we conventionally use [] to supply type arguments and () to supply value arguments. Type-application is equivalent to *instantiating* the type of the function. This follows naturally from the rules of λ-calculus: by applying *negate* to a type Integer, you substitute the actual type for the parameter: {Integer/τ} in the function body. The body is everything to the right of λτ. So, the value returned after type-application is identical to a simply-typed version of the function (like that shown above):

negate [Integer] ⇒ λ(x : Integer).(-x)
negate [Real] ⇒ λ(x : Real).(-x)

in which the type of *x* is now fixed. This typed function may now be applied to a value of the appropriate type.

In the theoretical model, we always have to supply the desired type of the function, before we can apply it to a value of this type. We cannot perform type-inference in the style: *negate (3 : Integer),* because this breaks the convention on the ordering of arguments in the declaration. The main thing to note is that the type parameter is always introduced before the value argument, so these arguments are always supplied in this order. We first discussed this idea in [1].

4 TYPE PARAMETERS AND KINDS

Since we are now dealing with a typed calculus, what is the "type" of the parameter τ? Technically, type variables like τ also have a meta-type, known as a *kind*. This is the "next level up" in the type system. We could show the meta-type of variables like τ by introducing them in a style in which the kind is explicit: λ(τ :: TYPE), to indicate that τ is a type parameter which can range over all types in the set TYPE. However, since the *second order, polymorphic typed* λ-calculus only has one main kind (the set of all simple types, TYPE), we shall later omit mentioning TYPE explicitly. For a discussion of orders of calculus, see the earlier article [1].

Above, we noted that we wanted to restrict the type of the *negate* function, such that it applied only to the signed, numeric types. This can be done by filtering the set of possible types in TYPE to those of interest. Let us assume that there is a type-filtering function Filter-Signed that returns true only if the type is a numerical, signed type. We can define a signed, numerical subset of all types:

$$\text{SIGNED} = \{\ \tau :: \text{TYPE} \mid \text{Filter-Signed}[\tau]\ \}$$

This defines SIGNED as "all those types τ, for which Filter-Signed[τ] is true". It should be clear that SIGNED $\subseteq$ TYPE. We can then express the type of *negate* as:

$$\text{negate} : \forall(\tau :: \text{SIGNED}).\ \tau \rightarrow \tau$$
$$= \lambda(\tau :: \text{SIGNED}).\lambda(x : \tau).(-x)$$

In the definition of *negate*, the type variable τ ranges only over those types in the SIGNED subset. Restrictions like this are extremely useful in object-oriented programming, where we wish polymorphic methods to apply only to certain sets of types. A set of types sharing some common structure is a *class* [1, 2] in our Theory of Classification.

5 GENERATORS USED AS CLASS TYPE-FILTERS

In earlier articles [1, 2] we introduced the notion of a *function bound*, often abbreviated to *F-bound* [6, 7], to describe a similar restriction. Literally, a *bound* means a restriction, and a *function bound* is a restriction expressed using a function. Let us define a type function, a record type generator, for a simple class of two-dimensional Cartesian Points:

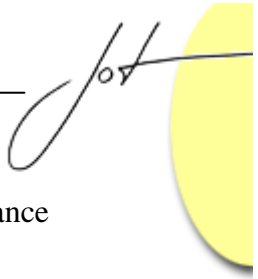
$$\text{GenPoint} = \lambda\sigma.\{x : \rightarrow \text{Integer},\ y : \rightarrow \text{Integer},\ \text{equal} : \sigma \rightarrow \text{Boolean}\}$$

This expresses the interface of a family of Point-like types that have at least the three methods *x, y* and *equal*. The generator parameterises the self-type σ, which eventually could stand for different types of Point, such as a Point3D [4] or a HotPoint (a selectable Point, see below). We can use this type generator as a filter to restrict the polymorphic application of these methods to only those types which could be considered at least "some kind of Point".

Recall that the typeful notion of a *class* is a group of (possibly recursive) types sharing a minimum common structure. We may express the *class of Points* as: $\forall(\tau <: \text{GenPoint}[\tau])$, because it restricts the types over which τ can range to those types which are a subtype of the instantiated generator GenPoint[τ]. Earlier, we found that an extended interface is a subtype [2, 8], so this captures precisely the object-oriented notion of families of object-types that share a minimum common set of methods. We may now give the methods *x, y* and *equal* a type signature which restricts their applicability to the class of Points:

$$\forall(\tau <: \text{GenPoint}[\tau]).\ \tau.x : \rightarrow \text{Integer}$$
$$\forall(\tau <: \text{GenPoint}[\tau]).\ \tau.y : \rightarrow \text{Integer}$$
$$\forall(\tau <: \text{GenPoint}[\tau]).\ \tau.\text{equal} : \tau \rightarrow \text{Boolean}$$

These type signatures say that the methods are selected *only* from those types τ which satisfy the membership criteria of the Point class. Note in passing how the *equal* method is a *binary method*, accepting another argument of the same type as the owning object itself.



We shall be interested to see how the type of a binary method evolves, when inheritance comes into play.

Formally, an F-bound is always expressed using a subtyping constraint: $\forall(\tau <: G[\tau])$, for some type generator function G. For comparison with the previous section, this can be thought of as a type filtering constraint: $\forall(\tau \mid F[\tau])$, where F is defined as: $F = \lambda\tau.(\tau <: G[\tau])$.

6 LINKING OBJECTS AND CLASS-TYPES

We are about to define a *typed object generator* for a class of Points. We introduced *type generators* in [1] and *object generators* in [3]. This time, we are going to attach type information to the object generator for a Point instance at the co-ordinate <2, 3>. We proceed exactly as in the sections above, by first declaring the type signature, then giving the full typed definition:

$$\begin{aligned}&\text{genAPoint} : \forall(\tau <: \text{GenPoint}[\tau]) \,.\, \tau \to \text{GenPoint}[\tau]\\ &= \lambda(\tau <: \text{GenPoint}[\tau]).\ \lambda(\text{self} : \tau).\\ &\qquad \{\ x \mapsto 2,\ y \mapsto 3,\ \text{equal} \mapsto \lambda(p : \tau).(\text{self.x} = p.x \wedge \text{self.y} = p.y)\ \}\end{aligned}$$

At first sight, this may look rather daunting! In fact, it is no more complex than the style of typed definitions given above. To motivate this structure, we shall build up to it more slowly.

Recall that an untyped object generator [3] is a function of *self*, whose body is a record describing the method implementations of an object instance. Our first version of the generator (omitting all details of the actual methods) is:

$$\text{gen}_1 = \lambda\text{self}.\{\ldots\}$$

If we wish to add types to this, we must prefix the value-argument λ*self* with a type-argument, $\lambda\sigma$, where σ stands for the type of *self*. We shall also attach the σ type explicitly to the *self*-variable. Our second version is a universally-typed generator:

$$\text{gen}_2 = \lambda\sigma.\lambda(\text{self} : \sigma).\{\ldots\}$$

in which σ still ranges over all types in the universe of types. We want to restrict σ so that it ranges over only those types in the class of *self*. To do this, we must have a separate type generator *Gen*, which has a type-shape matching the value-shape of the object generator, *gen*. We can then use it as a filter on the type parameter σ, giving a third, F-bounded version of the generator:

$$\text{gen}_3 = \lambda(\sigma <: \text{Gen}[\sigma]).\lambda(\text{self} : \sigma).\{\ldots\}$$

This is now in the same form as the typed object generator *genAPoint*, above. To see the correspondence, note how the second line in the definition of *genAPoint* introduces first a self-type parameter: $\lambda(\tau <: \text{GenPoint}[\tau])$, then the *self*-argument: $\lambda(\textit{self} : \tau)$, followed on the

third line by the record-body, representing the implementation of the Point instance. This follows the general form of second-order typed definitions: first, we introduce the type parameter, then the value parameter, then the body of the function.

The type signature for *genAPoint* also deserves some discussion. It says that *genAPoint* is well-defined for all types τ in the class of Points: $\forall(\tau <: \text{GenPoint}[\tau])$, and then that it accepts a value (ie an actual value for the *self*-argument) in the type τ and returns a record-body having the type GenPoint[τ]. This does in fact accurately describe the type of the record body. If we supply some standard *p*:Point as the *self*-argument, we get a result with the type: GenPoint[Point][1]. If we supply some more more specific *hp*:HotPoint as the *self*-argument, we get a result with the "truncated" type: GenPoint[HotPoint]. By "truncated", we mean a type that looks like a HotPoint, but with only those methods that were listed in the Point-interface.

[1] Readers following this series will recall that GenPoint[Point] is a fixpoint of the generator, ie that Point = GenPoint[Point], so we could equally say that the result is of the exact type Point.

7 STRONGLY TYPED INHERITANCE

To study the workings of inheritance when types are added, we shall attempt to extend the *typed object generator* for a Point to yield a *typed object generator* for a HotPoint, a selectable kind of point. As before, we shall first provide a *type generator* for the HotPoint type (we shall need this later to express F-bounds). GenHotPoint can be defined by extension, based on the GenPoint type generator. The additional fields include the types of the new method *selected* and redefined method *equal* (which, in a HotPoint, must also compare selected states):

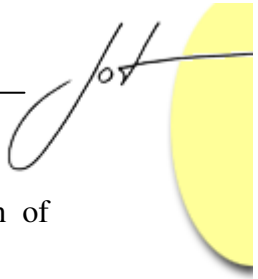
$$\text{GenHotPoint} = \lambda\tau.(\ \text{GenPoint}[\tau] \cup \{\ \text{equal} : \tau \rightarrow \text{Boolean}, \text{selected} : \rightarrow \text{Boolean}\ \}\ )$$

The simplified form of this type generator is well-formed, after computing the union of fields:

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

The only interesting consideration is what happens with GenPoint[τ]. This causes a substitution of type parameters in the body of GenPoint: $\{\tau/\sigma\}$, and has the consequence that all references to the self-type are uniformly changed to τ, before the union of fields is computed. This means that the identically-typed *equal* method type appears twice, once on either side of the $\cup$ operator, but only one copy is retained after the union.

We are now about to define a typed object generator for an instance of HotPoint, at the coordinate <2, 3> and whose selected state = true. We shall attempt to derive this by inheritance, in accordance with the model given in the earlier articles [3, 4]. This time, however, we shall be careful to attach type information, in the style presented above, to all



parts of the definition. First, we give the type declaration, then the full definition of *genAHotPoint*:

$$\begin{aligned}&\text{genAHotPoint} : \forall(\sigma <: \text{GenHotPoint}[\sigma]).\ \sigma \rightarrow \text{GenHotPoint}[\sigma]\\&= \lambda(\sigma <: \text{GenHotPoint}[\sigma]).\ \lambda(\text{self} : \sigma).\ (\ \lambda(\text{super} : \text{GenPoint}[\sigma]).\\&\qquad (\text{super} \oplus \{\ \text{equal} \rightarrow \lambda(q : \sigma).(\text{super.equal}(q) \wedge \text{self.selected} = q.\text{selected}),\\&\qquad\qquad \text{selected} \mapsto \text{true}\ \}\ )\\&\qquad \text{genAPoint}\ [\sigma]\ (\text{self})\ )\end{aligned}$$

This looks fabulously complicated! However, if you mentally put on one side the whole body expression inside the bold parentheses, the prequel leading up to it is in exactly the same form as all our other definitions. First, the type signature of *genAHotPoint* is given. Then, on the second line, its full definition is given, starting with the type parameter σ and value parameter *self*, followed by the body (everything contained within the bold parentheses).

Looking now at the body expression, this is exactly the construction we used to explain super-method combination in the previous article [4], except that types have now been attached to all value parameters. The body is a nested function application that first binds *super*, then performs a record combination using the ⊕ operator [3]. We shall want to simplify this (*viz* evaluate the combination expression), to assure ourselves that we have in fact defined a suitable generator for a HotPoint instance. However, we must first establish whether the body expression is properly typed.

8 TYPE SOUNDNESS OF *SUPER*

We first want to satisfy ourselves that the binding of *super* is type-sound. From the previous article [4] we learned that *super* is an adapted form of the parent object, in which self-reference is redirected to refer to the child[2]. Does our typed model reflect this faithfully?

The type and binding of *super*

At the start of the body, the *super* variable is declared with the type: λ(*super* : GenPoint[σ]). So, the type of *super* is structurally "like" the type of the parent Point, except that, within this structure, the self-type is replaced by σ, which is the new self-type of the child. GenPoint[σ] is a "truncated type" in which the self-type refers to a HotPoint, but which offers only those methods available to a Point. You can think of a generator as a mask, and GenPoint[HotPoint] "masking out" all the methods of HotPoint, that were not in the interface of a Point (*viz* in the body of the GenPoint-generator). This is the appropriate

[2] We restrict ourselves in this article to explaining the more sophisticated model of inheritance, in which self and the self-type evolve. This is, after all, the more interesting, relatively novel concept that needs explanation.

type to give to *super*, since it captures exactly the type of a "mofidied parent instance" in which *self* is redirected to refer to the child [4].

To understand what is happening inside the body expression above, notice how it consists of a super-function, $\lambda(super : \text{GenPoint}[\sigma]).(\ldots)$ which is applied to an object, denoting the value to bind to *super*, given right at the end of the body expression (mentally skip over the body of the *super*-function, which consists of the record-combination expression). This super-object is given at the end by the expression: *genAPoint* [σ] (*self*).

The next question we must ask is: does the *super*-variable receive an object-value of the right type? We need to work out the type of the expression: *genAPoint* [σ] (*self*) and see if this corresponds to something with the declared type: GenPoint[σ]. The super-object is clearly constructed from the typed object generator *genAPoint*, after supplying a type argument σ and a value argument *self* : σ. The result of this is a record, the body of the generator *genAPoint* (see section 6 above). The type of the result was declared in the type signature: $\forall(\tau <: \text{GenPoint}[\tau]).\ \tau \rightarrow \text{GenPoint}[\tau]$, which says that, by supplying σ and *self* : σ, we obtain a result having the type: GenPoint[σ]. This is exactly the type of object expected for *super*, above.

However, before we assume this happy outcome, we should check first whether it is actually type-safe to apply the generator to the type parameter σ, and value parameter *self* : σ Are these suitable types and values for this generator?

Rebinding type parameters

We will look at the type substitution first. The generator *genAPoint* was declared to be safe with all types satisfying $\tau <: \text{GenPoint}[\tau]$. Technically, the application *genAPoint*[σ] is simply a matter of substituting one type parameter for another: {σ/τ}. All type parameters have the same kind, TYPE, so this should not be a problem. However, a more subtle thing is happening. By substituting {σ/τ}, we are changing the restriction on the types which may instantiate the parameter. The parameters implicitly carry attached type constraints (from the F-bounds), so we have to worry about whether changing these makes a formal difference.

Although we cannot compare two type parameters directly, we can make a judgement about all the types which could possibly instantiate the respective parameters. Fortunately, it turns out that any type we could supply for σ will also satisfy the type constraint on τ. This is because of the point-wise subtyping condition between the two type generators:

$$\forall\sigma\ .\ \sigma <: \text{GenHotPoint}[\sigma] \ \Rightarrow\ \sigma <: \text{GenPoint}[\sigma]$$

which lies at the basis of the *Classify* rule in [2]. Because the type generators GenHotPoint and GenPoint stand in the right structural relationship, we can safely replace the self-type τ of the parent class by the self-type σ of the child class.

Rebinding value parameters

We will now look at the value substitution. The second argument in *genAPoint* [σ] (*self*) is a self-reference that refers to a HotPoint instance, with the type: *self* : σ. The generator *genAPoint* was originally declared to accept a value parameter having the type: *self* : τ <: GenPoint[τ]. However, we have just replaced τ by a new parameter: σ <: GenHotPoint[σ], by applying the generator to this type: *genAPoint*[σ]. From section 3 above, we know that this has the effect of re-typing the body of the function. All former references to τ are now replaced by σ, so the value parameter *self* : τ has been modified to *self* : σ. We may therefore supply an actual argument of this type, directly.

9 TYPE SOUNDNESS OF INHERITANCE

We now want to satisfy ourselves that the record combination expression with ⊕, which models the extension of an object by inheritance, is itself type sound. This expression is the whole body of the *super*-function, which we skipped over, above:

$$\text{super} \quad \oplus \quad \{\ \text{equal} \rightarrow \lambda(q : \sigma).(\text{super.equal}(q) \wedge \text{self.selected} = q.\text{selected}),\ \text{selected} \mapsto \text{true}\ \}$$

in which *super* is now bound, and refers to the *super*-object described in the previous section. Also, *self* refers to the *self* : σ introduced as the value parameter in the generator *genAHotPoint*. In order to understand whether the operator ⊕ is being applied to values of suitable types, we need to simplify the left-hand and right-hand operands until they have the form of object records.

The base record

The left-hand operand to ⊕ is *super*, and this is bound to the object *genAPoint* [σ] (*self*), which simplifies to a record, after σ and *self* : σ have been supplied as arguments:

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

To see where this came from, refer back to the body of the typed object generator *genAPoint*, given in section 6 above. The only difference is that we have substituted {σ/τ} and (self:σ/self:τ} as a result of instantiating the generator. One interesting thing to notice is that, in the "inherited" version of *equal*, both *self* and the compared argument *p* are now of the child-type, σ.

The extension record

The right-hand operand to ⊕ is a record of extra methods for a HotPoint, the fields contained in the braces {…}. The first is a redefinition of the *equal* method; the second is the new *selected* method (returning *true* for the instance we are defining). The body of

equal contains a super-method invocation. We would like to satisfy ourselves that this *equal* is in fact equivalent to a regular method, by simplifying the super-method invocation.

The super-invocation has the form: *super.equal(q).* We know that $q : \sigma$ from the immediately preceding declaration: $\lambda(q : \sigma)$. Fortunately, the *equal* method selected from the body of the super-object (given above) expects an argument $p : \sigma$ of exactly the same type. After substituting $\{q:\sigma/p:\sigma\}$ in the body, we obtain the simplified result:

$$\text{super.equal}(q : \sigma) \;\Rightarrow\; \text{self.x} = \text{q.x} \;\wedge\; \text{self.y} = \text{q.y}$$

Checking this out, we know already that *self* : σ, so we are comparing a *q* and a *self* which have the same type. Now, we substitute this into the body of the redefined *equal*, in place of the original super-invocation (which we have now simplified away altogether [4]), to yield the form of a regular record of methods:

$$\{\ \text{equal} \rightarrow \lambda(q : \sigma).(\ \text{self.x} = \text{q.x} \;\wedge\; \text{self.y} = \text{q.y} \wedge \text{self.selected} = \text{q.selected}),\ \text{selected} \mapsto \text{true}\ \}$$

in which *self* and *q* are uniform throughout the body of *equal*, referring to different child-instances, and are both of the same type, the child-type σ.

The record combination

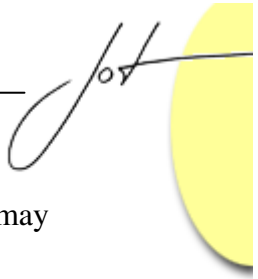
The record combination expression, modelling the extension of an object by inheritance, has now been reduced to the form:

$$\{\ x \mapsto 2,\ y \mapsto 3,\ \text{equal} \mapsto \lambda(p : \sigma).(\text{self.x} = \text{p.x} \wedge \text{self.y} = \text{p.y})\ \}\ \oplus$$
$$\{\ \text{equal} \rightarrow \lambda(q : \sigma).(\ \text{self.x} = \text{q.x} \;\wedge\; \text{self.y} = \text{q.y} \wedge \text{self.selected} = \text{q.selected}),\ \text{selected} \mapsto \text{true}\ \}$$

and it only remains to simplify ⊕. This was declared as a rather liberally-typed polymorphic operator [3], in the style of *function override*, accepting any two maps with the same domain-type, and yielding a map with this domain-type and a derived codomain-type that was the union of the arguments' codomains:

$$\forall \alpha, \beta, \gamma\ .\ \oplus : (\alpha \rightarrow \beta) \times (\alpha \rightarrow \gamma) \rightarrow (\alpha \rightarrow \beta \cup \gamma)$$
$$= \lambda(f:\alpha \rightarrow \beta).\lambda(g:\alpha \rightarrow \gamma).$$
$$\{\ k \mapsto v \mid (k \in \text{dom}(f) \cup \text{dom}(g)) \wedge (k \in \text{dom}(g) \Rightarrow v = g(k)) \wedge (k \notin \text{dom}(g) \Rightarrow v = f(k))\ \}$$

In a later article, we will reconsider this type signature, to better constrain the "legitimate" types of record arguments supplied in inheritance expressions. For the moment, let us note that the domain-type α will be bound to the type Label, from which we draw all the names of methods. The range-type β will be bound to a union of all the method-signature types of the base record; likewise γ is a union of all the method-signature types of the extension record. This seems to "lump together" all the different types in each union. However, the



definition of the operator $\oplus$ explains how individual fields are overridden, so we may obtain the detail from this.

So, on the left-hand side, we have a base record with the (more detailed, record) type:

$$\{ x : \rightarrow \text{Integer}, y : \rightarrow \text{Integer}, \text{equal} : \sigma \rightarrow \text{Boolean}\}$$

and on the right-hand side, we have an extension record with the type:

$$\{ \text{equal} : \sigma \rightarrow \text{Boolean}, \text{selected} : \rightarrow \text{Boolean} \}$$

and after combining the base and extra records according to the definition of $\oplus$, we seem to obtain, experimentally speaking, a record with the type:

$$\{ x : \rightarrow \text{Integer}, y : \rightarrow \text{Integer}, \text{equal} : \sigma \rightarrow \text{Boolean}, \text{selected} : \rightarrow \text{Boolean}\}$$

Looking at the pooled method types: {Integer, $\sigma \rightarrow$ Boolean, Boolean} in the result, this does seem to be a union of {Integer, $\sigma \rightarrow$ Boolean} $\cup$ {$\sigma \rightarrow$ Boolean, Boolean}, as the type-signature of $\oplus$ declared. So, this is at least consistent, even if it is not yet very informative.

The result of record combination is therefore the body of an extended generator, suitable for a HotPoint instance, in which *equal* is re-typed in terms of the child's self-type σ, and the extra method *selected* is included. It is as if we had defined *genAHotPoint* from first principles, without inheritance, with the (simplified) form:

$$\begin{aligned}&\text{genAHotPoint} : \forall(\sigma <: \text{GenHotPoint}[\sigma]).\ \sigma \rightarrow \text{GenHotPoint}[\sigma]\\ &= \lambda(\sigma <: \text{GenHotPoint}[\sigma]).\ \lambda(\text{self} : \sigma).\ \{ x \mapsto 2, y \mapsto 3,\\ &\qquad \text{equal} \rightarrow \lambda(q : \sigma).(\ \text{self.x} = \text{q.x}\ \wedge\ \text{self.y} = \text{q.y} \wedge \text{self.selected} = \text{q.selected}),\\ &\qquad\qquad \text{selected} \mapsto \text{true}\ \}\end{aligned}$$

so demonstrating that strongly-typed inheritance is just a short-hand for defining larger objects by extension, but with all the relevant type information attached.

10 CONCLUSION

We have presented a model of strongly-typed object generators, in which the class-type information is attached to the object-information. We may now formally claim to have defined the notion of *class* from *both* the implementation *and* type perspectives, combined. A class is, from a concrete perspective, a family of objects that share a similar (but overridable) implementation strategy and, from an abstract perspective, a family of types that share a similar (minimum common) method interface. This provides a good foundation for developing further model interpretations of other object-oriented concepts, such as class hierarchies, abstract classes and interfaces.

We also used the new typed calculus to present a model of strongly-typed inheritance. This combined two aspects of the sophisticated model of inheritance put forward previously in the *Theory of Classification*. Firstly, when a class inherits methods from its

parent, object self-reference is redirected to refer to a child instance [3, 4]. Secondly, the type signatures of inherited methods adapt, such that methods that referred to a parent-type now refer to a child-type [1, 2]. This is important in languages like Smalltalk and Eiffel, where binary methods like *equal* or *plus* may evolve under inheritance, but always apply to objects of the appropriate specific type. Attaching the self-type parameter σ to any other variable in the model exactly explains the novel typing construction in Eiffel, where a variable is declared to have the type "like current". It anchors the type of the variable to the type of *self*. This is an extremely satisfying way of providing types for binary methods, which expect to receive another object with the same type as *self*.

We also fulfilled a formal obligation to demonstrate that aspects of inheritance were type-sound. Super-method invocation was shown to be well-typed, yielding results as expected. We shall have to return to the typing of ⊕, in order to restrict the legitimate types of extra methods added to an object during inheritance, but our liberally-typed version of ⊕ works as intended with the object implementations shown. Technically, the definition given for ⊕ is an abbreviation of a more complete definition in the polymorphic typed λ-calculus, in which both type parameters and value parameters are supplied explicitly. We can think of our operator as a short-hand for expressing a type-instantiated version of the full-length *combine* function:

$$\text{combine} = \lambda\sigma.\lambda\tau.\lambda(\text{base} : \sigma).\lambda(\text{extra} : \tau).(\ldots)$$
$$\oplus_{A,B} = \text{combine}\ [A, B]$$

which is instantiated for each pair of record types A, B that we wish to combine.

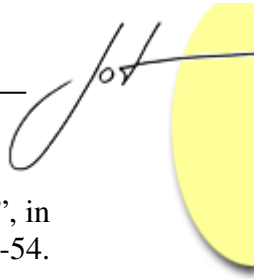
REFERENCES

[1] 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.

[2] 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. pp. 55-64. http://www.jot.fm/issues/issue_2003_07/column4

[3] 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

[4] 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



[5] A J H Simons, "The theory of classification, part 2: The scratch-built typechecker", in *Journal of Object Technology*, vol. 1, no. 2, July-August 2002, pp. 47-54. http://www.jot.fm/issues/issue_2002_07/column4

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

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

[8] A J H Simons, "The theory of classification, part 4: Object types and subtyping", in *Journal of Object Technology*, vol. 1, no. 5, November-December 2002, pp. 27-35. http://www.jot.fm/issues/issue_2002_11/column2

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.