View metadata, citation and similar papers at core.ac.uk

brought to you by CORE

provided by White Rose Research Online

This is a repository copy of *The theory of classification: part 14: modification and objects like myself*.

White Rose Research Online URL for this paper:
http://eprints.whiterose.ac.uk/79269/

Version: Published Version

Article:

Simons, A.J.H. (2004) The theory of classification: part 14: modification and objects like myself. Journal of Object Technology, 3 (8). 15 - 26. ISSN 1660-1769

Reuse

Unless indicated otherwise, fulltext items are protected by copyright with all rights reserved. The copyright exception in section 29 of the Copyright, Designs and Patents Act 1988 allows the making of a single copy solely for the purpose of non-commercial research or private study within the limits of fair dealing. The publisher or other rights-holder may allow further reproduction and re-use of this version - refer to the White Rose Research Online record for this item. Where records identify the publisher as the copyright holder, users can verify any specific terms of use on the publisher's website.

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.

White Rose university consortium
Universities of Leeds, Sheffield & York

eprints@whiterose.ac.uk
https://eprints.whiterose.ac.uk/

JOURNAL OF OBJECT TECHNOLOGY

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

Vol. 3, No. 8, September-October 2004

The Theory of Classification Part 14: Modification and Objects like Myself

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

1 INTRODUCTION

This is the fourteenth article in a regular series on object-oriented theory for non-specialists. Previous articles have built up models of objects [1], types [2] and classes [3] in the λ-calculus. Inheritance has been shown to extend both types [4] and implementations [5, 6], in the contrasting styles found in the two main families of object-oriented languages. One group is based on (first-order) *types* and *subtyping*, and includes Java and C++, whereas the other is based on (second-order) *classes* and *subclassing*, and includes Smalltalk and Eiffel. The most recent article demonstrated how *generic types* (templates) and *generic classes* can be added to the Theory of Classification [7], using various *Stack* and *List* container-types as examples.

The last article concentrated just on the typeful aspects of generic classes, avoiding the tricky issue of their implementations, even though previous articles have demonstrated how to model both types and implementations separately [4, 5] and in combination [6]. This is because many of the operations of a *Stack* or a *List* return modified objects "like themselves"; and it turns out that this is one of the more difficult things to model in the Theory of Classification. The attentive reader will have noticed that we have so far avoided the whole issue of object modification. This is, after all, quite an important area to consider, since one of the main benefits of object-oriented programming is to encapsulate state and handle state updates in a clean fashion. In the current article, we consider the whole area of environment modelling and the creation of modified objects. Eventually, this leads to an extension to the theory to handle constructor-methods, through the use of which an object can create another object "like itself".

Cite this column as follows: Anthony J.H. Simons: "The Theory of Classification, Part 14: Modification and Objects like Myself", in *Journal of Object Technology*, vol. 3, no. 8, September-October 2004, pp. 15-26. http://www.jot.fm/issues/issue_2004_09/column2

2 THE GLOBAL ENVIRONMENT

The whole idea of object modification, modelled as updates to the values stored in attribute variables, is problematic in a functional calculus like the λ-calculus. This is because pure functional languages do not support the notion of reassignment to variables. This would be a side-effect, and pure functional languages are intentionally free of side-effects, a property known as *referential transparency*. However, the effect of assignment may be approximated using a global set of variable bindings, called the *environment*, which is passed from function to function as the program is executed. The environment is an *associative map* from variable names to their bound values. For example, the following map is an environment which contains two variables *p1, p2* which map to records representing simple coordinates:

$$\text{globalEnv} = \{\text{p1} \mapsto \{\text{x} \mapsto 2, \text{y} \mapsto 3\}, \text{p2} \mapsto \{\text{x} \mapsto 4, \text{y} \mapsto 7\}, \ldots\}$$

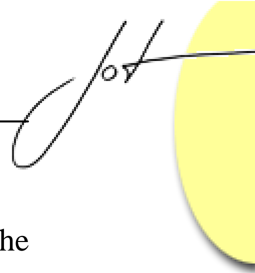
During program execution, we may want certain statements to update the global environment. Modifications cannot literally change the state of the environment, since we are working in a pure functional language without side-effects; instead, they construct new environments in which appropriate changes have been made. The environment must therefore be passed in and out of each function, since assignments may occur at any point and their effect must be recorded. Every function accepts the environment as an extra first argument. Likewise, every function returns a packaged result, which is a pair of the environment and the function's usual return value. The caller of a function must unpack the returned result to determine the state of the environment, as well as accessing the ordinary return value.

Initially, the environment is empty. As variables are declared and initialised, these are added to the environment using a function like *env-add*:

$$\begin{aligned}&\text{env-add} : \text{Map} \rightarrow \text{Label} \rightarrow \text{Value} \rightarrow \text{Map}\\&= \lambda(\text{env} : \text{Map}).\lambda(\text{var} : \text{Label}).\lambda(\text{val} : \text{Value}).\text{env} \oplus \{\text{var} \mapsto \text{val}\}\end{aligned}$$

which takes an environment, a variable name and a value to bind to this variable in the environment. The function returns the new environment, in which the old environment *env* is combined with a maplet from the variable *var* to the value *val*. Since the function override operator ⊕ is used [5], this will ensure that $\{var \mapsto val\}$ is added to the bindings in the environment, replacing any existing binding for *var*. This is useful to model both variable declaration with initialisation, when a variable is first added to the environment, and variable reassignment, when the value associated with a variable is replaced.

At any moment, the environment contains the most recently-bound version of each of the variables. In the body of a function, access to any global variable is modelled by looking up the value stored in the environment. Since the environment is basically a map (which is the same thing as a finite function [1]) this can be done by *applying* the



environment, like a function, to the labels used as variable names. For example, the following expression looks up the value of *p1* in *globalEnv*:

$$\text{globalEnv (p1)} \Rightarrow \{x \mapsto 2, y \mapsto 3\}$$

To change the coordinate associated with variable *p1*, we may execute the expression:

$$\begin{aligned} &\text{env-add (globalEnv) (p1) } (\{x \mapsto 5, y \mapsto 1\}) && \text{-- } \textit{supply 3 arguments} \\ &\Rightarrow \text{globalEnv} \oplus \{p1 \mapsto \{x \mapsto 5, y \mapsto 1\}\} && \text{-- } \textit{the body of env-add} \\ &\Rightarrow \{p1 \mapsto \{x \mapsto 5, y \mapsto 1\}, p2 \mapsto \{x \mapsto 4, y \mapsto 7\}, \ldots\} && \text{-- } \textit{new globalEnv} \end{aligned}$$

and this rebinds the value of the variable *p1*, returning a new environment in which *p1* maps to a different coordinate instance. This models the notion of reassignment.

3 LOCAL AND GLOBAL UPDATES

Things are made slightly more complicated if we want both global and local variable bindings. If all functions merely had local variables, upon function exit the environment could revert to the environment that was originally passed to the function. This would ensure that the bindings set up on entry to the function were forgotten and any older bindings for the same variables were restored. However, we want the global bindings to persist between each function call. The environment passed to a function may possibly be modified and should therefore be handed back as part of the result.

One possible approach is to try to distinguish between the global environment, and local variables, which are bound on entry to functions in the usual way. The problem with this approach is that a local variable may shadow the name of a global variable in the environment. When such a variable is updated, we would expect the local copy to be modified, since the global variable would be hidden. Upon exit from this scope, the global binding would be restored. However, it is hard to imagine how we could integrate primitive λ-calculus binding with looking up variables in a constructed environment. In any case, most state variables are introduced as local variables within the scope of some object or function, so it is hard to distinguish between the two kinds of variables in practice.

Another approach is to use a *multimap* for the environment, that is, a kind of map with duplicated keys, rather like the *association list* provided in Common Lisp. Whenever a scope is entered and a new variable binding is added, it is inserted *ahead* of any existing bindings for the same variable. Whenever a value is looked up, the lookup function returns the *first* bound value it finds, which hides any other older bindings found later in the list. Whenever a scope is exited, all local variables are descoped by explicitly removing the *first* binding found for each variable, so restoring any older bindings. Whenever a variable is reassigned, the *first* occurrence of the variable is rebound to the new value; and this works whether the variable is global or local. To do this, we need a family of functions to manipulate the environment:

env-insert : Multimap → Label → Value → Multimap
env-remove : Multimap → Label → Multimap
env-replace : Multimap → Label → Value → Multimap
env-lookup : Multimap → Label → Value

which behave as described above. The implementations of these functions would use primitive list operations, such as *cons, head* and *tail* to search through the lists representing the multimaps. We defer a fuller treatment of this until a later article.

4 MODELLING UPDATES AS NEW OBJECTS

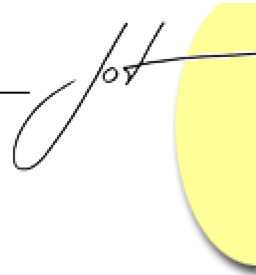
For the moment, the simplest approach is to assume that objects in the theoretical model are "pure functional objects", that is, all modifications to object state do not literally modify the state of the object, rather they create and return a new object in which the changes are manifest. There is no fundamental reason why an imperative language cannot be approximated by a functional calculus in this way. The only difference is that all state-modifying methods, which are typically void-methods in a concrete language, now have to return a new instance of their owning type. Sequences of modifications have to be modelled as nested method invocations. As an example, consider the following Point type:

$$\text{Point} = \mu\sigma.\{\text{x : Integer, y : Integer, equal : } \sigma \rightarrow \text{Boolean,}$$
$$\text{move : Integer} \times \text{Integer} \rightarrow \sigma\}$$

which, in addition to the usual *x, y* and *equal* methods, has a *move* method to update its position. This method is typed to return another *Point* object, reflecting the fact that the modified position is in fact a newly-created instance of *Point*. Issuing a sequence of *move* instructions to a *Point* object *p* could be represented by the following nested method invocations:

p.move(2, 3).move(4, 5).move(6, 7)

since each *move* returns a new *Point*, which becomes the receiver of the subsequent *move* message. Although, in the model, the final *Point* instance at (6, 7) is a distinct object from the original *p : Point*, we can still reason about such sequences of update operations. But there is a catch: while the idea sounds straightforward in principle, it turns out that implementing the *move* method in practice is quite difficult to accomplish in the theoretical model. So far, none of our object types has had the ability to create new instances "like itself". As we shall see below, this requires yet another level of recursion, in which objects contain their own object constructors.



5 CONSTRUCTORS FOR OBJECTS

In an earlier article [6], we established the basic strategy for constructing new objects. It involved first constructing the object type, then the object implementation, from generators. The basic *Point* record type (without the *move* method) must be constructed from a generator, because it refers to itself recursively:

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

and the recursion variable σ is later bound to the record type by taking the fixpoint using **Y**.

$$\text{Point} = \mathbf{Y}\ [\text{GenPoint}]$$
$$\Rightarrow \{\text{x : Integer, y : Integer, equal : Point} \to \text{Boolean}\} \qquad \text{-- } \textit{after unrolling}$$

The basic *point* record instance must also be constructed from a generator, because it refers to itself recursively. A *typed* object generator is used, so that we can attach types to the bound variables. This is the generator for a specific *point* instance, at the location (2, 3):

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

The recursive *point* instance is then constructed by supplying a type *Point* as the first type-argument, and then taking the fixpoint using **Y**, to bind the recursion variable *self* over the rest of the record:

$$\text{point} = \mathbf{Y}\ (\text{genPoint [Point]})$$
$$\Rightarrow \{\text{x} \mapsto 2,\ \text{y} \mapsto 3,\ \text{equal} \mapsto \lambda(\text{p} : \tau).(\text{point.x} = \text{p.x} \wedge \text{point.y} = \text{p.y})\}$$

While this works well for examples of specific points, we wanted to allow the creation of points that were initialised to different coordinates. To do this, we extended the generator to accept an extra initialisation argument, a pair of Integers [6]:

$$\text{initPoint} : \forall(\tau <: \text{GenPoint}[\tau]).\ (\text{Integer} \times \text{Integer}) \to \tau \to \text{GenPoint}[\tau]$$
$$= \lambda(\tau <: \text{GenPoint}[\tau]).\lambda(\text{a, b : Integer} \times \text{Integer}).\lambda(\text{self} : \tau).$$
$$\{\text{x} \mapsto \text{a},\ \text{y} \mapsto \text{b},\ \text{equal} \mapsto \lambda(\text{p} : \tau).(\text{self.x} = \text{p.x} \wedge \text{self.y} = \text{p.y})\}$$

And from this, we could define a simple object constructor, *makePoint*, which uses the type generator *GenPoint* and the extended object generator *initPoint* internally to establish the recursive *Point* type, and the recursive *point* instance, respectively:

$$\text{makePoint : Integer} \times \text{Integer} \to \text{Point}$$
$$= \lambda(\text{a, b : Integer} \times \text{Integer}).\ \mathbf{Y}\ (\text{initPoint}\ [\mathbf{Y}\ \text{GenPoint}]\ (\text{a, b}))$$

An example of creating a point instance at a different location is given by:

$$
\begin{aligned}
&\text{makePoint}(4, 5)\\
&\Rightarrow \mathbf{Y}\ (\text{initPoint}\ [\mathbf{Y}\ \text{GenPoint}]\ (4, 5))\\
&\Rightarrow \mathbf{Y}\ (\ \lambda(a, b : \text{Integer} \times \text{Integer}).\lambda(\text{self} : \text{Point}).\\
&\qquad \{x \mapsto a, y \mapsto b, \text{equal} \mapsto \lambda(p : \text{Point}).(\text{self}.x = p.x \wedge \text{self}.y = p.y)\}\ (4, 5))\\
&\Rightarrow \mathbf{Y}\ (\ \lambda(\text{self} : \text{Point}).\\
&\qquad \{x \mapsto 4, y \mapsto 5, \text{equal} \mapsto \lambda(p : \text{Point}).(\text{self}.x = p.x \wedge \text{self}.y = p.y)\}\ )\\
&\Rightarrow \mu\text{self}.\{x \mapsto 4, y \mapsto 5, \text{equal} \mapsto \lambda(p : \text{Point}).(\text{self}.x = p.x \wedge \text{self}.y = p.y)\}
\end{aligned}
$$

The main thing to notice about all this is that object generators like *initPoint* contain within them the structure of the instance that they create. The *initPoint* function accepts some arguments (a type argument, then a pair of *Integers*, then the value for *self*) and returns, as the body, the structure of the *point* instance. The generator must logically exist *prior* to the creation of any object instance. It is therefore hard to imagine how we might create an object instance that contains its own generator! This is rather like trying to pull yourself up by your own shoelaces.

6 CREATING OBJECTS LIKE MYSELF

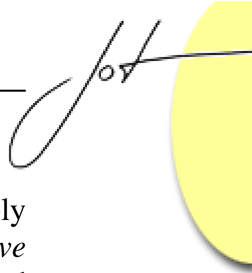
However, as we anticipated in section 4 above, every time the *move* method is invoked, we require a *Point* object to create a new instance like itself, except that the *x* and *y* coordinates will take on different values. This is exactly like requiring an object to have its own constructor as one of its methods. To see this, we shall change the definition of the *Point*-type, so that it now also has a *move* method. Defining the type is straightforward:

$$
\begin{aligned}
\text{GenPoint} = \lambda\sigma.\{&x : \text{Integer}, y : \text{Integer}, \text{equal} : \sigma \rightarrow \text{Boolean},\\
&\text{move} : \text{Integer} \times \text{Integer} \rightarrow \sigma\}
\end{aligned}
$$

$$
\begin{aligned}
&\text{Point} = \mathbf{Y}\ [\text{GenPoint}]\\
&\Rightarrow \{x : \text{Integer}, y : \text{Integer}, \text{equal} : \text{Point} \rightarrow \text{Boolean},\\
&\qquad \text{move} : \text{Integer} \times \text{Integer} \rightarrow \text{Point}\} \qquad \text{-- } \textit{after unrolling}
\end{aligned}
$$

However, the implementation is less straightforward. If we could assume that an object constructor *makePoint* already existed, we could provide the extended generator for moveable points with extra initialisation arguments as the following:

$$
\begin{aligned}
&\text{initPoint} : \forall(\tau <: \text{GenPoint}[\tau]).\ (\text{Integer} \times \text{Integer}) \rightarrow \tau \rightarrow \text{GenPoint}[\tau]\\
&= \lambda(\tau <: \text{GenPoint}[\tau]).\lambda(a, b : \text{Integer} \times \text{Integer}).\lambda(\text{self} : \tau).\\
&\qquad \{x \mapsto a, y \mapsto b, \text{equal} \mapsto \lambda(p : \tau).(\text{self}.x = p.x \wedge \text{self}.y = p.y),\\
&\qquad\qquad \text{move} \mapsto \lambda(u, v : \text{Integer} \times \text{Integer}).\mathbf{makePoint}(u, v)\ \}
\end{aligned}
$$



Here, the implementation of the *move* method has been added, in which the body simply calls *makePoint* with the same pair of *Integer* arguments that were given to the *move* method. This should in principle create and return a new *Point* instance at the desired location.

However, the above is not yet a legal definition. The reason for this is that *makePoint* must use *initPoint* internally to construct the new object instance. As a result, the above implies a recursive definition of *initPoint*. To see the recursion more clearly, we can replace the occurrence of *makePoint* by its expansion into generators (this comes from the body of the definition of *makePoint* in section 5 above):

$$\begin{aligned}&\textbf{initPoint} : \forall(\tau <: \text{GenPoint}[\tau]).\ (\text{Integer} \times \text{Integer}) \rightarrow \tau \rightarrow \text{GenPoint}[\tau]\\ &= \lambda(\tau <: \text{GenPoint}[\tau]).\lambda(a, b : \text{Integer} \times \text{Integer}).\lambda(\text{self} : \tau).\\ &\qquad \{x \mapsto a,\ y \mapsto b,\ \text{equal} \mapsto \lambda(p : \tau).(\text{self}.x = p.x \wedge \text{self}.y = p.y),\\ &\qquad\quad \text{move} \mapsto \lambda(u, v : \text{Integer} \times \text{Integer}).\ \mathbf{Y}\ (\textbf{initPoint}\ [\mathbf{Y}\ \text{GenPoint}]\ (u, v))\ \}\end{aligned}$$

From the bold highlight, it is clear that *initPoint* occurs both on the left and right-hand sides of this "definition". As readers of this series will appreciate, a recursive definition is not a proper definition in the λ-calculus [1], but merely an equation that must be solved for some value of *initPoint*.

7 CONSTRUCTOR-LEVEL RECURSION

It is interesting to note that the model now requires recursion on three different levels:

- object-level recursion, because objects frequently need to refer internally to self, so that methods can call other methods of the same object;
- type-level recursion, because object types frequently define methods that accept and return objects of the same type as themselves;
- constructor-level recursion, because objects frequently have methods that construct and return other objects like themselves.

Constructor-level recursion was first identified by Cook and others [8]. In the cited paper, they referred to this initially as "class-level" recursion. Later, this was changed to "constructor-level" recursion, to better reflect the facts [9].

The technique for solving constructor-level recursion is the same one we have used for solving recursive definitions before [1], in which we abstract at the point of recursion and introduce a new variable standing for the recursively-defined thing, here the object constructor for *Points*. At first, it is tempting to think that we need to introduce a recursion variable standing for the whole of *initPoint*. However, the type-argument of this function doesn't enter into the constructor-recursion: we know in advance that we are always going to create things of type τ, where τ is eventually bound to *Point*. The object constructor function is something that takes a pair of *Integers* and returns *genPoint* : $\tau \rightarrow \tau$, a simple object generator for building recursive *Point* instances after their fields have

been initialised. We shall therefore introduce a constructor recursion variable *ctor*, with the type:

$$\text{ctor} : (\text{Integer} \times \text{Integer}) \rightarrow (\tau \rightarrow \tau)$$

standing for the *Point* constructor. We will introduce this recursion variable *after* the type argument τ (so that τ will be bound) but *before* the other arguments from *initPoint*, because we need to fix the constructor-level recursion before we accept *Integer* arguments and fix the object-level recursion. The result is a *typed generator* for an *object constructor*, which we shall call *genInitPoint* and which has the type signature:

$$\begin{aligned}\text{genInitPoint} : \forall(\tau <: \text{GenPoint}[\tau]).((\text{Integer} \times \text{Integer}) \rightarrow (\tau \rightarrow \tau)) \rightarrow \\ (\text{Integer} \times \text{Integer}) \rightarrow \tau \rightarrow \text{GenPoint}[\tau]\end{aligned}$$

This looks a little daunting, but essentially it is similar to the type signature for *initPoint*, with an extra type argument in the signature, giving the type of the recursion variable *ctor*, standing for the constructor. The full definition of *genInitPoint* is given by:

$$\begin{aligned}\text{genInitPoint} = \lambda(\tau <: \text{GenPoint}[\tau]).\lambda(\text{ctor} : (\text{Integer} \times \text{Integer}) \rightarrow (\tau \rightarrow \tau)). \\ \lambda(\text{a, b} : \text{Integer} \times \text{Integer}).\lambda(\text{self} : \tau). \\ \{\text{x} \mapsto \text{a, y} \mapsto \text{b, equal} \mapsto \lambda(\text{p} : \tau).(\text{self.x} = \text{p.x} \wedge \text{self.y} = \text{p.y}), \\ \text{move} \mapsto \lambda(\text{u, v} : \text{Integer} \times \text{Integer}).\ \mathbf{Y}\ (\text{ ctor (u, v)) }\}\end{aligned}$$

In this, *ctor* is introduced as the extra recursion variable standing for the *Point* constructor. This allows the use of *ctor* in the body of the *move* method. The application of the fixpoint finder **Y** is essentially to bind the object-level recursion inside instances created by *ctor*. We shall return to this below. Note that *genInitPoint* is now properly defined, without having to refer recursively to itself.

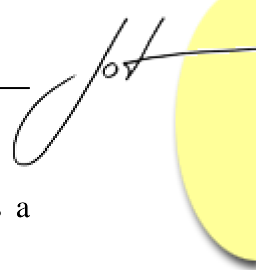
8 OBJECTS WITH CONSTRUCTOR METHODS

Point instances, which now have their own constructor method *move*, are created from this generator in several stages. First, we supply the desired type argument *Point*:

$$\text{genInitPoint[Point]} \qquad \text{-- } \textit{step 1, supply the type argument}$$

$$\begin{aligned}= \lambda(\text{ctor} : (\text{Integer} \times \text{Integer}) \rightarrow (\text{Point} \rightarrow \text{Point})). \\ \lambda(\text{a, b} : \text{Integer} \times \text{Integer}).\lambda(\text{self} : \text{Point}). \\ \{\text{x} \mapsto \text{a, y} \mapsto \text{b, equal} \mapsto \lambda(\text{p} : \text{Point}).(\text{self.x} = \text{p.x} \wedge \text{self.y} = \text{p.y}), \\ \text{move} \mapsto \lambda(\text{u, v} : \text{Integer} \times \text{Integer}).\ \mathbf{Y}\ (\text{ ctor (u, v)) }\}\end{aligned}$$

This yields a typed function in which $\{\text{Point}/\tau\}$ has been substituted throughout. The first argument to this function is the recursion variable *ctor*. We want to bind *ctor* recursively over the rest of the body, using the fixpoint finder **Y**. But first, let us consider the type



signature of the above function (after step 1), to see whether taking the fixpoint is a legitimate operation. It has the signature:

$$((\text{Integer} \times \text{Integer}) \rightarrow (\text{Point} \rightarrow \text{Point})) \rightarrow (\text{Integer} \times \text{Integer}) \rightarrow \text{Point} \rightarrow \text{Point}$$

in other words, it takes a first argument of the *ctor* constructor-type and then returns something with exactly the same type signature (if we ignore the bracketing of the remaining types). This is useful, because it satisfies the conditions for a generator. Generators must always have the form: $gen : \tau \rightarrow \tau$, since, when they are applied to some argument, they must return that argument unchanged [1]. So, the step-1 result is indeed a generator-for-a-constructor, which we can now fix in step 2:

Y (genInitPoint[Point]) -- *step 2, fix the constructor recursion*

$$= \mu\text{ctor}.\ \lambda(\text{a, b : Integer} \times \text{Integer}).\lambda(\text{self : Point}).$$
$$\{\text{x} \mapsto \text{a, y} \mapsto \text{b, equal} \mapsto \lambda(\text{p : Point}).(\text{self.x = p.x} \wedge \text{self.y = p.y}),$$
$$\text{move} \mapsto \lambda(\text{u, v : Integer} \times \text{Integer}).\ \mathbf{Y}\ (\text{ ctor (u, v)) }\}$$

This yields a *Point*-constructor function beginning λ(*a, b : Integer* × *Integer*) ... and in whose body *ctor* is recursively fixed to refer to this, the same *Point*-constructor function. We denote this fact by prefixing the function with μ*ctor*, according to convention [1]. Below, we must remember that *ctor* now refers to this function, the step-2 result. In the next step, we supply the desired coordinate position for a particular point instance:

Y (genInitPoint[Point]) (2, 3) -- *step 3, supply instance coordinates*

$$= \lambda(\text{self : Point}).$$
$$\{\text{x} \mapsto \text{2, y} \mapsto \text{3, equal} \mapsto \lambda(\text{p : Point}).(\text{self.x = p.x} \wedge \text{self.y = p.y}),$$
$$\text{move} \mapsto \lambda(\text{u, v : Integer} \times \text{Integer}).\ \mathbf{Y}\ (\text{ ctor (u, v)) }\}$$

This yields a function in which {2/a, 3/b} have been substituted, thereby supplying the coordinate position (2, 3) for the first point instance. The step-3 result is a function beginning λ(*self : Point*)…, in other words, a simple object generator, whose object-level recursion we can fix in the usual way using **Y**:

Y (**Y** (genInitPoint[Point]) (2, 3)) -- *step 4, fix the object recursion*

$$= \mu\text{self}.\ \{\text{x} \mapsto \text{2, y} \mapsto \text{3, equal} \mapsto \lambda(\text{p : Point}).(\text{self.x = p.x} \wedge \text{self.y = p.y}),$$
$$\text{move} \mapsto \lambda(\text{u, v : Integer} \times \text{Integer}).\ \mathbf{Y}\ (\text{ ctor (u, v)) }\}$$

This is now a point instance, in which *self* refers recursively to this instance, and *ctor* refers back to the constructor which built this instance! Note how the formula for creating an object that contains its own constructor requires an additional fixpoint operation. The above formula could be rewritten to show all three fixpoints, including the type-level fixpoint:

Y (**Y** (genInitPoint[**Y** GenPoint]) (2, 3))

In this, the innermost fixpoint: [**Y** *GenPoint*] fixes the type-level recursion, yielding the recursive *Point* type. The next outermost fixpoint: **Y** (*genInitPoint*[**Y** *GenPoint*]) fixes the constructor-level recursion, yielding the recursive *ctor* constructor. The outermost fixpoint: **Y** (**Y** (*genInitPoint*[**Y** *GenPoint*]) (2, 3)) fixes the object-level recursion, after the constructor *ctor* has been applied to some initialisation arguments (2, 3), yielding the recursive *point* instance.

9 UPDATING A POINT

Let us call this initial *Point* instance *p1*, and construct it using the formula:

$$p1 : \text{Point} = \mathbf{Y}\ (\mathbf{Y}\ (\text{genInitPoint[Point]})\ (2, 3))$$

$$\Rightarrow \{x \mapsto 2,\ y \mapsto 3,\ \text{equal} \mapsto \lambda(p : \text{Point}).(p1.x = p.x \wedge p1.y = p.y),$$
$$\text{move} \mapsto \lambda(u, v : \text{Integer} \times \text{Integer}).\ \mathbf{Y}\ (\ \text{ctor}\ (u, v))\ \}$$

This object has a *move* method, which contains a recursive reference to the object constructor *ctor* that was used in the building of *p1*. We would like to see the effect of invoking the *move* method on *p1*, to see what kind of object this returns. We should like it to return a new *Point* instance at a different location, so we shall call this instance *p2*:

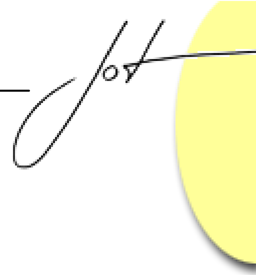
$$p2 : \text{Point} = p1.\text{move}(4, 5)$$
$$\Rightarrow \lambda(u, v : \text{Integer} \times \text{Integer}).\ \mathbf{Y}\ (\ \text{ctor}\ (u, v))\ (4, 5) \qquad \text{-- } \textit{select move}$$
$$\Rightarrow \mathbf{Y}\ (\ \text{ctor}\ (4, 5)) \qquad \text{-- } \textit{bind \{4/u, 5/v\}}$$

At this stage, we have to refer back to the definition of *ctor* to understand how to simplify this any further. We obtain this definition formally by *unrolling* the value of the variable *ctor*, which was recursively bound at the end of step 2, above. The following sidebar shows this:

$$\text{ctor} \Rightarrow \qquad \text{-- } \textit{unroll ctor}$$
$$\lambda(a, b : \text{Integer} \times \text{Integer}).\lambda(\text{self} : \text{Point}).$$
$$\{x \mapsto a,\ y \mapsto b,\ \text{equal} \mapsto \lambda(p : \text{Point}).(\text{self}.x = p.x \wedge \text{self}.y = p.y),$$
$$\text{move} \mapsto \lambda(u, v : \text{Integer} \times \text{Integer}).\ \mathbf{Y}\ (\ \text{ctor}\ (u, v))\ \}$$

So we can now continue the main simplification of *p1.move(4,5)* with the substitution:

$$\mathbf{Y}\ (\ \text{ctor}\ (4, 5))$$
$$\Rightarrow \mathbf{Y}\ (\lambda(a, b : \text{Integer} \times \text{Integer}).\lambda(\text{self} : \text{Point}). \qquad \text{-- } \textit{unroll ctor}$$
$$\{x \mapsto a,\ y \mapsto b,\ \text{equal} \mapsto \lambda(p : \text{Point}).(\text{self}.x = p.x \wedge \text{self}.y = p.y),$$
$$\text{move} \mapsto \lambda(u, v : \text{Integer} \times \text{Integer}).\ \mathbf{Y}\ (\ \text{ctor}\ (u, v))\ \}\ (4, 5))$$



$\Rightarrow$ **Y** λ(self : Point). *-- bind {4/a, 5/b}*

{x $\mapsto$ 4, y $\mapsto$ 5, equal $\mapsto$ λ(p : Point).(self.x = p.x $\wedge$ self.y = p.y),
move $\mapsto$ λ(u, v : Integer $\times$ Integer). **Y** (ctor (u, v)) }

$\Rightarrow$ μself. *-- take the fixpoint*

{x $\mapsto$ 4, y $\mapsto$ 5, equal $\mapsto$ λ(p : Point).(self.x = p.x $\wedge$ self.y = p.y),
move $\mapsto$ λ(u, v : Integer $\times$ Integer). **Y** (ctor (u, v)) }

This is the final result, showing that *p2* is another *Point* instance at the coordinates (4, 5) and with its own copy of the constructor *ctor* embedded inside its *move* method. So, we have shown that it is possible to create objects with their own constructor-methods embedded inside them. However, it was theoretically dense and required three different levels of fixpoints.

10 CONCLUSION

We started this article with a discussion of how to model object state updates in the Theory of Classification. In λ-calculus, variable reassignment is prohibited, but the same effect may be approximated by extending all functions to accept and return an *environment* argument, which is some kind of map storing the current variable bindings. The machinery for binding and unbinding variables is quite complex: eventually, we must use a multimap and handle all binding, unbinding and lookup explicitly. Furthermore, we need implementations of *List*-methods like *cons, head* and *tail* to manipulate the multimaps, which are essentially *association lists* with duplicated keys.

An alternative approach is to model state updates as the creation of new objects. A sequence of updates is modelled as a nested series of method invocations. However this requires a new level of sophistication in the model, in which objects contain their own constructors. The major part of this article was devoted to explaining *constructor-level* recursion. This is the third kind of recursion, after *object-level* and *type-level* recursion. With this facility, we were able to provide *Point* objects with a method: *move : Integer* × *Integer* → *Point*, which returns a new *Point* instance. This same facility would be required to define the *List*-methods *cons* and *tail*, which both return new *List* instances. So, constructor-level recursion is a generally useful feature, essential for the definition of more complex kinds of datatype. With this facility, we may now provide implementations for the container-classes like *List* and *Stack*, which had been deferred in the previous article [7].

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 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

[3] 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

[4] 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

[5] 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

[6] A J H Simons: "The Theory of Classification, Part 12: Building the Class Hierarchy", in *Journal of Object Technology, vol. 3, no. 5, May-June 2004*, pp. 13-24. http://www.jot.fm/issues/issue_2004_05/column2

[7] 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

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

[9] 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.