JOURNAL OF OBJECT TECHNOLOGY

Online at www.jot.fm. Published by ETH Zurich, Chair of Software Engineering ©**JOT, 2002**

Vol. 1, no. 4, September-October 2002

The Theory of Classification Part 3: Object Encodings and Recursion

Anthny J. H. Simons, Department of Computer Science, University of Sheffield, UK

1 INTRODUCTION

This is the third article in a regular series on object-oriented type theory, aimed specifically at non-theoreticians. Eventually, we aim to explain the behaviour of languages such as Smalltalk, C++, Eiffel and Java in a consistent framework, modelling features such as classes, inheritance, polymorphism, message passing, method combination and templates or generic parameters. This will be the "Theory of Classification" of the series title. Along the way, we shall look at some important theoretical approaches, such as subtyping, F-bounds, matching and, in this article, the primitive object calculus and the fixpoint theorem for recursion.

The first article [1] introduced the notion of type from both the practical and mathematical points of view and the second article [2] introduced some examples of type rules for constructing and checking simple expressions. Using a starter-kit containing only set theory and boolean logic, we built models for pairs and functions, eventually encoding objects as records, a kind of finite function mapping from labels to values. However, this is only one of three fundamentally different approaches to encoding objects in the primitive model [3, 4, 5]. The first two are based on set theory and the λ-calculus [6], the calculus of primitive functions, and the last on the ς-calculus [5], the calculus of primitive objects. In this article, we investigate the benefits and disadvantages of different object encodings.

2 EXISTENTIAL OBJECT ENCODING

The first encoding style is based on *data abstraction* [3, 4]. It represents an object as an explicit pair of *state* and *methods* (rules for constructing pairs were given in the previous article [2]). In this approach, a simple Cartesian Point type is defined as follows:

Cite this column as follows: Anthony 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, pages 49-57. http://www.jot.fm/issues/issue_2002_09/column4

$$\text{Point} = \exists\ \text{rep}\ .\ (\text{rep} \times \{x : \text{rep} \to \text{Integer};\ \ y : \text{rep} \to \text{Integer};\ \ \text{equal} : \text{rep} \times \text{rep} \to \text{Boolean}\})$$

This definition has the sense of "let there be some representation type *rep*, such that the Point type is defined as a pair of *rep* × *methods*, where *methods* is a record of functions that manipulate the *rep*-type." This clearly bears some similarity with abstract data types (see article [1]), since the state of the Point, *rep*, is existentially quantified using ∃. This has the effect of declaring the existence of state, but preventing any direct access to it. The *rep* is some hidden concrete type (like a *sort*), about which nothing further is known. The record of methods is visible by virtue of not being ∃-quantified.

An instance of a Point type may be defined with a particular concrete representation (here, we assume that *rep* = Integer × Integer) as follows:

$$\text{aPoint} = \langle\langle 2, 3\rangle, \{\ x \mapsto \lambda(s : \text{rep}).\pi_1(s),\ y \mapsto \lambda(s : \text{rep}).\pi_2(s),$$
$$\text{equal} \mapsto \lambda(p : \text{rep} \times \text{rep}).(\pi_1(\pi_1(p)) = (\pi_1(\pi_2(p)) \wedge \pi_2(\pi_1(p)) = \pi_2(\pi_2(p)))\ \}\rangle$$

As this looks rather dense, break it down as follows: aPoint is defined as a pair <r, m>, where r is the concrete state, a pair of Integers <2, 3>, and m is a record of methods that access different projections of the state. The x and y functions both accept a single *rep* argument, whereas the equal function accepts an argument which is a pair of *reps*, hence the nested use of projections to get at "the first of the first of p" and so on.

Existential encoding models the hiding of state, rather like the use of *private* declarations in C++ and Java. It can be used to model packages, whose contents are only revealed within certain scopes [7]. The other advantage of this approach is that types, such as Point, are non-recursive, since all its methods are defined to accept a *rep*, rather than the Point type itself. A disadvantage of this approach is the inelegance of method invocation. Recall that a Point p is a pair, so to invoke one of its methods requires accessing the first projection $\pi_1(p)$ to get at its state and second projection $\pi_2(p)$ to get at its methods. Simply to invoke the x-method requires the complicated construction: $\pi_2(p).x(\pi_1(p))$ in the calculus. Instead, we would like the model to reflect more directly the natural syntax of object-oriented languages.

One way would be to define a special method invocation operator "•" to hide the ungainly syntax, such that the expression:

$$\text{obj} \bullet \text{msg}(\text{arg}) \ \Leftrightarrow\ \pi_2(\text{obj}).\text{msg}(\pi_1(\text{obj}), \pi_1(\text{arg})).$$

However, this has several drawbacks. Firstly, separate versions of "•" would be needed for methods accepting zero, or more arguments. Secondly, "•" would have to accept objects, messages and arguments of all types, requiring a much more complicated higher-order type system to express well-typed messages.

3 FUNCTIONAL OBJECT ENCODING

For this reason, we prefer the second encoding, in which objects are represented as *functional closures* [3, 4]. A closure is essentially a function with an implicit state. A function can acquire hidden state variables due to the way in which it was defined. For example:

let y = 3 **in**
inc = λx.(x + y)

defines *inc* inside the scope of y. The function[1] accepts x as an argument (x is a *bound variable*), but y is a *free variable* in the body of *inc*, with the value 3. Applications of *inc* produce results that depend on more than the argument x: inc(2) ⇒ 5; inc(4) ⇒ 7; showing how the function has "remembered" some state. In pure functional languages, this state cannot be modified (free variables have *static binding*, as in Common Lisp).

Using this encoding, objects can be modelled directly as functions. This may sound strange, but recall how a record is really a finite set of label-to-value mappings, while a function is a general set of value-to-value mappings [2]. Records are clearly a subset of functions. In this view, any object is a function: λ(a : A).e, where the argument a : A is a label and the function body e is a multibranch if-statement, returning different values for different labels. We can model method invocation directly as function application, for example if we have Point p, then p.x in the program may be interpreted as: p(x) in the calculus. In an untyped universe, untyped functions are sufficient to model objects.

However, in a typed universe, records are subtly different from functions, in that each field may hold a value of a different type. For this reason, we use a special syntax for records and record selection [2], which allows us to determine the types of particular fields. In this approach, a simple Cartesian Point type is defined as follows:

Point = μ pnt . {x : → Integer; y : → Integer; equal : pnt → Boolean}

This definition has the sense of "let *pnt* be a placeholder standing for the eventual definition of the Point type, which is defined as a record type whose methods may recursively manipulate values of this *pnt*-type." In this style, "μ *pnt*" (sometimes notated as "**rec** *pnt*") indicates that the following definition is recursive. We explore the issue of recursion below.

An instance of this Point type may be defined as follows:

let xv = 2, yv = 3 **in**
aPoint = { x ↦ xv, y ↦ yv, equal ↦ λ(p : Point).(xv = p.x ∧ yv = p.y) }

[1] If the λ-calculus syntax still puzzles you, consider that: inc = λx.(x + y) is saying the same thing as the engineer's informal notation: inc(x) = (x + y). The λx simply identifies the formal argument x and the dot "." separates this from the body expression.

in which xv and yv are state variables in whose scope aPoint is defined. The constructor function make-point from the previous article [2] serves exactly the same purpose as the **let**...**in** syntax, by establishing a scope within which aPoint is defined.

In this encoding, method invocation has a direct interpretation. In the program, we may have a Point p and invoke p.x; the model uses exactly the same syntax and furthermore, we can determine the types of selection expressions using the dot "." operator from the record elimination rule [2]. Note how, in this encoding, the functions representing methods have one fewer argument each. This is because we no longer have to supply the *rep* as the first argument to each method. Instead, variables such as xv and yv are directly accessible, as all of aPoint's methods are defined within their scope. This exactly reflects the behaviour of methods in Smalltalk, Java, C++ and Eiffel, which have direct access to attributes declared in the surrounding class-scope. A disadvantage of the functional closure encoding is the need for recursive definitions, which requires a full theoretical explanation.

4 RECURSION EVERYWHERE

Objects are naturally recursive things. The methods of an object frequently invoke other methods in the same object. To model this effectively, we need to keep a handle on *self*, the current object. Using the μ-convention, we may define aPoint's equal method in terms of its other x and y methods (rather than directly in terms of variables xv, yv), as follows:

let xv = 2, yv = 3 **in**
aPoint = μ self . { x ↦ xv, y ↦ yv,
equal ↦ λ(p : Point).(self.x = p.x ∧ self.y = p.y) }

This declares *self* as the placeholder variable, equivalent to the eventual definition of the object aPoint, which contains embedded references to *self* (technically, we say that μ binds *self* to the resulting definition). This is exactly the same concept as the pseudo-variable *self* in Smalltalk, also known as *this* in Java and C++, or *Current* in Eiffel. In the formal model, all nested method invocations on the current object must be selected from *self*.

An *object* is recursive if it calls its own methods, or passes itself as an argument or result of a method. Above, we saw that the Point *type* is also recursive, because equal accepts another Point object. Object *types* are typically recursive, because their methods frequently deal in objects of the same type. Object-recursion and type-recursion are essentially independent, but related (for example, a method returning *self* will have the *self*-type as its result type).

As programmers, we take recursion for granted. However, it is a considerable problem from a theoretical point of view. So far, we have not demonstrated that recursion exists in the model, nor have we constructed it from first principles. Consider that the so-called "definition" of a recursive Point type in the (deliberately faulty) style:

$$Point = \{x : \rightarrow Integer;\ y : \rightarrow Integer;\ equal : Point \rightarrow Boolean\}$$

is not actually a *definition*, but rather an *equation* to which we must find a solution, since Point appears on both left- and right-hand sides. It is exactly like the form of an equation in high school algebra: $x = x^2/3$. This is not a definition of x, but an equation to be solved for x. Note that, for some equations, there may be more than one solution, or no solutions at all! So, does recursion really exist, and is there a unique solution?

5 THE FIXPOINT THEOREM

In high school algebra, the trick is to isolate the variable x: the above becomes: $x^2 - 3x = 0$, which we can factorize to obtain: $x\ (x - 3) = 0$, and from this the two solutions: $x = 0$, $x = 3$. Exactly the same kind of trick is used to deal with recursion. We try to isolate the recursion in the definition and replace this by a variable. Rather than define recursive Point outright, we define a function GenPoint with a single parameter in place of the recursion:

$$GenPoint = \lambda\ pnt\ .\ \{x : \rightarrow Integer;\ y : \rightarrow Integer;\ equal : pnt \rightarrow Boolean\}$$

Note that GenPoint is not recursive. GenPoint is a *type function* - it accepts one type argument, *pnt*, and returns a record type, in which *pnt* is bound to the supplied argument. We can think of GenPoint as a *type generator* (hence the name). We may apply GenPoint to any type we like, and so construct a record type that looks something like a Point. However to obtain exactly the Point record type we desire, we must substitute Point/*pnt*:

$$GenPoint[Point] = \{x : \rightarrow Integer;\ y : \rightarrow Integer;\ equal : Point \rightarrow Boolean\}$$

which is fine, except that it doesn't solve the recursion problem. All we have managed to do is rephrase it as: Point = GenPoint[Point], with Point still on both sides of the equation.

This is nonetheless interesting, in that Point is *unchanged* by the application of GenPoint to itself, hence it is called a *fixpoint* of the generator GenPoint. The *fixpoint theorem* in the λ-calculus states that a recursive function is equivalent to the limit of the self-application of its corresponding generator. To understand this, we shall apply GenPoint to successive types and gradually approximate the Point type we desire. Let the first approximation be defined as: $Point_0 = \perp$. In this, ⊥ stands for the undefined type[2], meaning that we know nothing at all about it. The next approximation is:

$$Point_1 = GenPoint[Point_0] = \{x : \rightarrow Integer;\ y : \rightarrow Integer;\ equal : \perp \rightarrow Boolean\}$$

[2] The symbol ⊥ has the name "bottom" (seriously). It is typically used to denote the "least defined" element.

$Point_1$ can be used as the type of points whose x and y methods are well-typed, but equal is not well-typed, so we cannot use it safely. The next approximation is:

$$Point_2 = GenPoint[Point_1] = \{x : \rightarrow Integer;\ y : \rightarrow Integer;\ equal : Point_1 \rightarrow Boolean\}$$

$Point_2$ can be used as the type of points whose equal method is also well-typed, because although its argument type is the inadequate $Point_1$, we only access the x and y methods in the body of equal, for which $Point_1$ gives sufficient type information. The $Point_2$ approximation is therefore adequate here, because the equal method only "digs down" through one level of recursion. In general, methods may "dig down" an arbitrary number of levels. What we need therefore is the infinitely-long approximation (the limit of the self-application of GenPoint):

$$Point = GenPoint[GenPoint[GenPoint[GenPoint[GenPoint[\ ...\ ]]]]]$$

which, finally, is a non-recursive definition of Point. Point is called the *least fixed point* of the generator GenPoint, and fortunately there is a unique solution. In λ-calculus [6] recursion is not a primitive notion, but infinitely-long expressions are allowed; so recursion can be constructed from first principles. To save writing infinitely-nested generator expressions, a special combinator function **Y**, known as the *fixpoint finder*, can be used to construct these from generators on the fly. One suitable definition of **Y** is:

$$\mathbf{Y} = \lambda f.(\ \lambda s.(f\ (s\ s))\ \lambda s.(f\ (s\ s))\ )$$

and, for readers prepared to attempt the following exercise, you can show that:

$$\mathbf{Y}\ [GenPoint] \Rightarrow GenPoint[GenPoint[GenPoint[GenPoint[GenPoint[\ ...\ ]]]]]$$

6 THE OBJECT CALCULUS

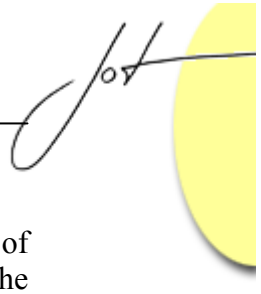
The third and most radical encoding changes the underlying calculus on which the model is based. To appreciate this contrast, we must understand something of the λ-calculus [6], which was invented by Church in the late 1930s as a primitive model of computation. There are only two fundamental rules of the calculus: function definition (known as λ-*abstraction*):

$\lambda x.e$ denotes a function of x, with body e, in which x is bound;

and function application (known as β-*reduction*):

$\lambda x.e\ \ v \Rightarrow e\{v/x\}$ denotes application of $\lambda x.e$ to v, yielding $e\{v/x\}$.

These notions are familiar to anyone who has ever programmed in a language with functions. The β-reduction rule has the sense: "a function of x with body e, when applied to a value v, is simplified to yield a result, which is the body e in which all occurrences of



the variable x have been replaced by v". As programmers, we like to think in terms of passing actual argument v to formal argument x and then evaluating body e. From the point of view of the calculus, this is simply a mechanical substitution, written e{v/x} and meaning "v substituted for x in e"; and "evaluation" simply corresponds to further symbolic simplification.

Abadi and Cardelli's theory of primitive objects [5] introduced the ς-calculus in which the fundamental operations are the construction of objects, the invocation of methods, and the replacement of methods (useful for explaining field updates and overriding):

[m = ς(x) e]	denotes an object with a method labelled m
o.m	invokes (the value of) method m on object o
o.m ⇐ ς(x) f	replaces the value of m in o with ς(x) f

Primitive operators include brackets [], the sigma-binder ς, the dot selector "." and the ⇐ override operator. In particular, the behaviour of ς(x) is different from that of λx in the λ-calculus, in that it automatically binds the argument x to the object from which the method is selected. In the expression: o.m, the value of m is selected *and applied* to the object o, such that we obtain e{o/x} in the method's body. This is an extremely clever trick, as it completely side-steps all the recursive problems to do with self-invocation[3]. To illustrate, a simple point object may be defined as:

aPoint = [x = ς(self) 2, y = ς (self) 3, equal = ς (self) λ(p) self.x = p.x ∧ self.y = p.y]

in which all methods bind the *self*-argument, *by definition of the calculus*. The x and y methods simply return suitable values. The equal method, after binding *self*, returns a normal function, expecting another Point p. Although we use non-primitive λ(p) and boolean operations in the body of equal, these notions can all be defined from scratch in the ς-calculus. For example, a Boolean object may provide suitable logical operations as its methods; and even a λ-abstraction can be defined as an object that looks like a program stack frame, with methods returning its argument value and code-body [5].

We cannot dispense with recursion altogether, for the Point type requires another Point as the argument of the equal method. The Point type is defined as:

Point = μ pnt [x : Integer, y : Integer, equal : pnt → Boolean]

where μ is understood to bind *pnt* recursively, and the existence of recursion is justified by the fixpoint theorem. When giving types to the methods, ς(self) is not considered to contribute anything to the type signature (the binding is internal); methods have the

[3] Somewhat similar to finding out that a crafty accountant has redefined the meaning of death for tax purposes. But seriously, a calculus may adopt any primitive rules it likes, within credible bounds of minimality.

public types of their released bodies. The resulting object type is quite similar in appearance to a record type in the functional encoding scheme. The binding of self-arguments in every method is also reminiscent of the existential encoding scheme. Overall, the ς-calculus uses more primitive operators and has a more sophisticated binding rule than the λ-calculus.

7 CONCLUSION

We have compared three formal encodings for objects and their types. The existential encoding avoided recursion but suffered from an ungainly method invocation syntax. The functional encoding was more direct, but used recursion everywhere. The primitive object encoding avoided recursion for self-invocation but needed it elsewhere. Choosing any of these encoding schemes is largely a matter of personal taste. In later articles, we shall use the functional closure encoding, partly because it has few initial primitives and reflects the syntax of object-oriented languages directly, but also because the notion of generators and fixpoints later proves crucial to understanding the distinct notions of *class* and *type*. In presenting the fixpoint theorem for solving recursive definitions, we also gave a notional meaning to the pseudo-variables standing for the current object in object-oriented languages. In the next article, we shall develop a theory of types and subtyping, seeing how recursion interacts with subtyping.

REFERENCES

[1] A J H Simons, *The Theory of Classification, Part 1: Perspectives on Type Compatibility*, in Journal of Object Technology, vol. 1, no. 1, May-June 2002, pages 55-61. http://www.jot.fm/issues/issue_2002_05/column7.

[2] 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, pages 47-54. http://www.jot.fm/issues/issue_2002_07/column4.

[3] J C Reynolds, *User defined types and procedural data structures as complementary approaches to data abstraction*, in: Programming Methodology: a Collection of Articles by IFIP WG2.3, ed. D Gries, 1975, 309-317; reprinted from *New Advances in Algorithmic Languages*, ed. S A Schuman, INRIA, 1975, 157-168.

[4] W Cook, *Object-oriented programming versus abstract data types*, in: Foundations of Object-Oriented Languages, LNCS 489, eds. J de Bakker et al., Springer Verlag, 1991, 151-178.

[5] M Abadi and L Cardelli. *A Theory of Objects. Monographs in Computer Science,* Springer-Verlag, 1996.

[6] A Church, *A formulation of the simple theory of types*, Journal of Symbolic Logic, 5 (1940), 56-68.

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

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