JOURNAL OF OBJECT TECHNOLOGY

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

Vol. 1, No. 1, May-June 2002

The Theory of Classification

Part 1: Perspectives on Type Compatibility

Anthony J H Simons, Department of Computer Science, University of Sheffield

1 INTRODUCTION

This is the first article in a regular series on object-oriented type theory, aimed specifically at non-theoreticians. The object-oriented notion of classification has for long been a fascinating issue for type theory, chiefly because no other programming paradigm has so sought to establish systematic sets of relationships between all of its types. Over the series, we shall seek to find answers to questions such as: What is the difference between a type and a class? What do we mean by the the notion of plug-in compatibility? What is the difference between subtyping and subclassing? Can we explain inheritance, method combination and template instantiation? Along the way, we shall survey a number of different historical approaches, such as subtyping, F-bounds, matching and state machines and seek to show how these models explain the differences in the behaviour of popular object-oriented languages such as Java, C++, Smalltalk and Eiffel. The series is titled "The Theory of Classification", because we believe that all of these concepts can be united in a single theoretical model, demonstrating that the object-oriented notion of class is a first-class mathematical concept!

In this introductory article, we first look at some motivational issues, such as the need for plug-in compatible components and the different ways in which compatibility can be judged. Reasons for studying object-oriented type theory include the desire to explain the different features of object-oriented languages in a consistent way. This leads into a discussion of what we really mean by a type, ranging from the concrete to the abstract views.

2 COMPONENTS AND COMPATIBILITY

The eventual economic success of the object-oriented and component-based software industry will depend on the ability to mix and match parts selected from different suppliers [1]. In this, the notion of component compatibility is a paramount concern:

- the *client* (component user) has to make certain assumptions about the way a component behaves, in order to use it;
- the *supplier* (component provider) will want to build something which at least satisfies these expectations;

But how can we ensure that the two viewpoints are compatible? Traditionally the notion of *type* has been used to judge compatibility in software. We can characterise type in two fundamental ways:

- *syntactic* compatibility - the component provides all the expected operations (type names, function signatures, interfaces);
- *semantic* compatibility - the component's operations all behave in the expected way (state semantics, logical axioms, proofs);

and these are both important, although most work published as "type theory" has concentrated on the first aspect, whereas the latter aspect comes under the heading of "semantics" or "model checking". There are many spectacular examples of failure due to type-related software design faults, such as the Mars Climate Orbiter crash and the Ariane-5 launch disaster. These recent high-profile cases illustrate two different kinds of *in*compatibility.

In the case of the Mars Climate Orbiter, the failure was due to inadequate characterisation of *syntactic type*, resulting in a confusion of metric and imperial units. Output from the spacecraft's guidance system was re-interpreted by the propulsion system in a different set of measurement units, resulting in an incorrect orbital insertion manoeuvre, leading to the crash [2]. In the case of the Ariane 5 disaster, the failure was due to inadequate characterisation of *semantic type*, in which the guidance system needlessly continued to perform its pre-launch self-calibration cycle. During launch, the emission of larger than expected diagnostic codes caused arithmetic overflow in the data conversion intended for the propulsion system, which raised an exception terminating the guidance system, leading to the violent course-correction and break-up of the launcher [3]. This last example should be of particular interest to object-oriented programmers, since it involved the wholesale reuse of the previously successful guidance software from the earlier Ariane 4 launcher in a new context.

3 DEGREES OF STRICTNESS AND SOPHISTICATION

How strictly must a component match the interface into which it is plugged? In Pascal, a strongly-typed language, a variable can only receive a value of exactly the same type, a property known as *monomorphism* (literally, the same form). Furthermore, types are checked on a *name equivalence*, rather than *structural equivalence* basis. This means that, even if a programmer declared *Meter* and *Foot* to be synonyms for *Integer*, the Pascal type system would still treat the two as non-equivalent, because of their different names (so avoiding the Martian disaster). In C++, *typedef* synonyms are all considered to be the same type and you would have to devise wrapper classes for *Meter* and *Foot* to get the same strict separation.

Furthermore, all object-oriented languages are *polymorphic* (literally, having many forms), allowing variables to receive values of more than one type.[1] From a practical point of view, polymorphism is regarded as an important means of increasing the generality of an interface, allowing for a wider choice of components to be substituted, which are said to *satisfy* the interface. Informally, this is understood to mean supplying *at least* those functions declared in the interface. However, the theoretical concept of polymorphism is widely misunderstood and the term mistakenly applied, by object-oriented programmers, variously to describe dynamic binding or subtyping. The usage we shall adopt is consistent with established work in the functional programming community, in that it requires at least a second-order typed lambda calculus (with type parameters) to model formally [4]. However, we must lay more foundations before introducing such a calculus.

A simple approach to interface satisfaction is *subtyping*. This is where an object of one type may safely be substituted where another type was expected [5]. This involves no more than coercing the supplied subtype object to a supertype and executing the supertype's functions. The coerced object then behaves in *exactly* the same way as expected. An example of this is where two *SmallInt* objects are passed to an *Integer plus* function and the result is returned as an *Integer*. The function originally expected *Integer*s, but could handle subtypes of *Integer* and convert them. Note that no dynamic binding is implied or required. Also, a simply-typed first-order calculus (with subtyping) is adequate to explain this behaviour.

We shall call the more complex, polymorphic approach *subclassing*. This is where one type is replaced by another, which also systematically replaces the original functions with new ones appropriate to the new type. An example of this is where a *Numeric* type, with abstract *plus, minus, times* and *divide*, is replaced by a *Complex* type, having appropriately-retyped versions of these (as in Eiffel [6]). Rather than coerce a *Complex* object to a *Numeric*, the call to *plus* through *Numeric* should execute the *Complex plus* function. Also, there is an obligation to propagate type information about the arguments

[1] Beware object-oriented textbooks! Polymorphism does *not* refer to the dynamic behaviour of objects aliased by a common superclass variable, but to the fact that variables may hold values of more than one type in the first place. This fact is independent of static or dynamic binding.

and result-type of *Complex's plus* back to the call-site, which needs to supply suitable arguments and then know how to deal with the result. In a later article, we shall see why this formally requires a parametric explanation.

To summarise so far, there are three different degrees of sophistication when judging the type compatibility of a component with respect to the expectations of an interface:

- correspondence: the component is identical in type and its behaviour exactly matches the expectations made of it when calls are made through the interface;
- subtyping: the component is a more specific type, but behaves exactly like the more general expectations when calls are made through the interface;
- subclassing: the component is a more specific type and behaves in ways that exceed the more general expectations when calls are made through the interface.

Certain object-oriented languages like Java and C++ practise a halfway-house approach, which is *subtyping* with *dynamic binding*. This is similar to subtyping, except that the subtype may provide a replacement function that is executed instead. Adapting the earlier example, this is like the *SmallInt* type providing its own version of the *plus* function which wraps the result back into the *SmallInt* range. Syntactically, the result is acceptable as an *Integer*, but semantically it may yield different results from the original *Integer plus* function (when wrap-around occurs). The selection mechanism of dynamic binding is formally equivalent to higher-order functional programming [7], in which functions are passed as arguments and then are dynamically invoked under program control. So, languages with apparently simple type systems are more complex than they may at first seem.

4 CONCRETE AND ABSTRACT TYPES

How can we explain the behaviour of languages such as Smalltalk, C++, Eiffel and Java in a consistent framework? Our goal is to find a *mathematical model* that can describe the features of these languages; and a *proof technique* that will let us reason about the model. To do this, we need an adequate definition of type that will allow reasoning about syntactic and semantic type compatibility. This brings into question what we mean exactly by a *type*.

Bit-Interpretation Schemas

There are various definitions of type, with increasing formal usefulness. Some approaches are quite concrete, for example a programmer sometimes thinks of a type as a *schema for interpreting bit-strings* in computer memory, eg the bit-string 01000001 is 'A' if interpreted as a *Character*; but 65 if interpreted as an *Integer*. This approach is concerned more with machine-level memory storage requirements than with formal properties necessary to reason about types.

Model-Based and Constructive Types

An afficionado of formal methods (such as Z [8], or VDM) likes to think of types as *equivalent to sets*: $x : T \Leftrightarrow x \in T$.

This is called the *model-based approach*, in which the notion of type is grounded in a set-theoretic model, that is, having type (x : T, "x is of type T") is equivalent to set membership ($x \in T$, "x is a member of set T") All program operations can be modelled as set manipulations. The constructive approach [9] also translates a program into a simpler *concrete* model, like set-theory, whose formal mathematical properties are well understood.

Concrete approaches have their limits [10], for example, how would you specify an *Ordinal* type? You merely want to describe something that is countable, whose elements are ordered, but not assert that any particular set "is" the set of *Ordinals*. The set of *Natural* numbers: *Natural* = {0, 1, 2...} is too specific a model for *Ordinal*, since this excludes other ordered things, like *Characters*, and the *Natural* numbers are subject to further operations (such as arithmetic) which the *Ordinals* don't allow (although strictly the set-theoretic model only enumerates the membership of a type and does not describe how elements behave).

Syntactic and Existential Abstract Types

A type theorist typically thinks of a type as a *set of function signatures*, which describe the operations that a type allows. This characterises the type in a more *abstract* way, by enumerating the operations that it allows. The *Ordinal* type is defined as:

$$\text{Ordinal} = \exists\, \text{ord} . \{\text{first:} \rightarrow \text{ord};\ \ \text{succ: ord} \rightarrow \text{ord}\}$$

in which $\exists$ *ord* can be read as "let there be an uninterpreted set *ord*", such that the following operations accept and return elements from this (as yet undefined) set. Ordinal is then defined as the type providing *first* and *succ*; and we don't care about the representation of *ord*. This approach is variously called *syntactic*, since it is based on type signatures, or *existential*, since it uses $\exists$ to reveal the existence of a representation, but refuses to qualify *ord* any further.

Although syntactic types reach the desired degree of abstraction away from concrete models, they are not yet precise. Consider that the following faulty expressions are still possible:

succ('b') = first() = 'a'	- an undesired possibility;
succ(1) = 1	- another undesired possibility;

This is because the signatures alone fail to capture the intended meaning of functions.

Axioms and Algebraic Types

A mathematician considers a type as a *set of signatures and constraining axioms*. The type *Ordinal* is fully characterised by:

$$
\begin{aligned}
&\text{Ordinal} = \exists\, \text{ord} \,.\, \{\text{first}: \rightarrow \text{ord};\ \ \text{succ}: \text{ord} \rightarrow \text{ord}\} \\
&\forall x : \text{Ordinal} \,.\, (\text{succ}(x) \neq \text{first}()) && (1) \\
&\qquad\qquad \wedge\ (\text{succ}(x) \neq x) && (2)
\end{aligned}
$$

This form of definition is known as an *algebra*. Formally, an algebra consists of: a *sort* (that is, an uninterpreted set, *ord*, acting as a placeholder for the type); and a set of functions defined on the sort (*first, succ*), whose meaning is given by axioms. The two axioms (1) and (2), plus the logical rule of induction, are sufficient to make *Ordinal* behave in exactly the desired way. But how do the axioms work? Let us arbitrarily label: x = first().

- From (1), succ(x) ≠ first(), so we know succ(x) is distinct from x; let us choose another arbitrary label: y = succ(x).
- From (2) succ(y) ≠ y; from (1) succ(y) ≠ x, so we know succ(y) is distinct from x and y; let us therefore label: z = succ(y) = succ(succ(x)).
- From (2) succ(z) ≠ z; from (1) succ(z) ≠ x; but could succ(z) = y? Although there is no ground axiom that instantly forbids this, induction rules it out, because:

  by substitution of y and z, we get: succ(succ(succ(x))) = succ(x)
  by unwinding succ, we get: succ(succ(x)) = x, which is false by (1),
  so succ(z) is also distinct; and so on...

Once the algebra is defined, we can disregard the sort, which is no longer needed, since every element of the type can now be expressed in a purely syntactic way:

first(); succ(first()); succ(succ(first())); ...

The algebraic definition of *Ordinal* says exactly enough and no more [11]; it is both more *abstract* than a concrete type - it is not tied to any particular set representation - and is more *precise* - it is inhabited exactly by a monotonically-ordered sequence of abstract objects.

5 CONCLUSION

We are motivated to study object-oriented type theory out of a concern to understand better the notion of syntactic and semantic type compatibility. Compatibility may be judged according to varying degrees of strictness, which each have different consequences. Likewise, different object-oriented languages seem to treat substitutability in different ways. As a preamble to developing a formal model in which languages like

Smalltalk, C++, Eiffel and Java can be analysed and compared, increasingly abstract definitions of type were presented. The next article in this series builds on the foundation laid here and deals with models of objects, methods and message-passing.

REFERENCES

[1] B J Cox, *Object-Oriented Programming: an Evolutionary Approach*, 1st edn., Addison-Wesley, 1986.

[2] Mars Climate Orbiter Official Website, *http://mars.jpl.nasa.gov/msp98/orbiter/*, September 1999.

[3] J L Lions, Ariane 5 *Flight 501 Failure, Report of the Inquiry Board*, http://sunnyday.mit.edu/accidents/Ariane5accidentreport.html, July 1996.

[4] J C Reynolds, *Towards a theory of type structure, Proc. Coll. sur la Programmation*, New York; pub. LNCS 19, Springer Verlag, 1974, 408-425.

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

[6] B Meyer, *Object-Oriented Software Construction, 2nd edn*., Prentice Hall, 1995.

[7] W Harris, *Contravariance for the rest of us*, J. of Obj.-Oriented Prog., Nov-Dec, 1991, 10-18.

[8] J M Spivey, *Understanding Z: a Specification Language and its Formal Semantics*, CUP, 1988.

[9] P Martin-Löf, *Intuitionistic type theory, lecture notes*, Univ. Padova, 1980.

[10] J H Morris, *Types are not sets*, Proc. ACM Symp. on Principles of Prog. Langs., Boston, 1973, 120-124.

[11] K Futatsugi, J Goguen, J-P Jouannaud and J Messeguer, *Principles of OBJ-2*, Proc. 12th ACM Symp. Principles of Prog. Langs., 1985, 52-66.

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

JOURNAL OF OBJECT TECHNOLOGY

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

Vol. 1, No. 2, July-August 2002

The Theory of Classification Part 2: The Scratch-Built Typechecker

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

1 INTRODUCTION

This is the second 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. Along the way, we shall look at some important theoretical approaches, such as subtyping, F-bounds, matching and the object calculus. Our goal is to find a *mathematical model* that can describe the features of these languages; and a *proof technique* that will let us reason about the model. This will be the "Theory of Classification" of the series title.

	Schemas	**Interfaces**	**Algebras**
Exact	1	2	3
Subtyping	4	5	6
Subclassing	7	8	9

Figure 1: Dimensions of Type Checking

The first article [1] introduced the notion of type, ranging from the programmer's concrete perspective to the mathematician's abstract perspective, pointing out the benefits of abstraction and precision. From these, let us choose three levels of type-checking to consider: representation-checking (bit-schemas), interface-checking (signatures) and behaviour-checking (algebras). Component compatibility was judged according to whether exact type correspondence, simple subtyping or the more ambitious subclassing was expected. Combining these two dimensions, there are up to nine combinations to consider, as illustrated in figure 1. However, we shall be mostly interested in the darker shaded areas. In this second article, we shall build a typechecker that can determine the exact syntactic type (box 2, in figure 1) of expressions involving objects encoded as simple records.

2 THE UNIVERSE OF VALUES AND TYPES

Rather like scratch-building a model sailing ship out of matchsticks, all mathematical model-building approaches start from first principles. To help get off the ground, most make some basic assumptions about the universe of values. Primitive sets, such as *Boolean, Natural* and *Integer* are assumed to exist (although we could go back further and construct them from first principles, in the same way as we did the *Ordinal* type [1]; this is quite a fascinating exercise in the λ-calculus [2]). All other kinds of concept have to be defined using rules to say how the concept is formed, and how it is used. We shall assume that there are:

- *sets* A, B, ..., corresponding to the given primitive types in the universe; and
- *elements* a, b, ..., of these sets, corresponding to the values in the universe; and
- *set operations* such as membership $\in$, inclusion $\subseteq$, and union $\cup$; and
- *logical operations* such as implication $\Rightarrow$, equivalence $\Leftrightarrow$ and entailment $\vdash$.

With this starter-kit, we can determine whether a value belongs to a type, since: $x : X$ ("x is of type X") can be interpreted as $x \in X$ in the model; or whether two types are related, for example: $Y <: X$ ("Y is a subtype of X") can be interpreted as the subset relationship $Y \subseteq X$ in the model [3].

3 RULES FOR PAIRS

An immediately useful construction which we do not yet have is the notion of a pair of values, $\langle n, m \rangle$, possibly taken from different types N and M. The type of pairs is known as a *product type*, or *Cartesian product*, since there are $N \times M$ possible parings of elements $n \in N$, and $m \in M$. Formally, we require a rule to introduce the syntax for a product type. This is called a *type introduction* rule. In its simplest form (ignoring the notion of *context*, which is roughly the same idea as variable scope), the rule for forming a product is:

$$\frac{n : N,\;\; m : M}{\langle n, m \rangle : N \times M} \qquad \text{[Product Introduction]}$$

This rule is expressed in the usual style of *natural deduction*, with the premises above the line and the conclusions below. In longhand, it says "if n is of type N and m is of type M, then we may conclude that a pair $\langle n, m\rangle$ has the product type $N \times M$". The rule introduces the syntax for writing pair-values and pair-types, but also defines the relationship between the values and the types (the order of the values n and m determines the structure of the type $N \times M$).

For pair constructions to be useful, we need to know how to access the elements of a pair, and determine their types. We define the *first* and *second projections* of a pair, usually written in the style: $\pi_1(e)$, $\pi_2(e)$ applied to some pair-value e. The projections are defined formally in an *elimination rule*, so called because it deconstructs the pair to access its elements:

$$\frac{e : N \times M}{\pi_1(e) : N,\ \pi_2(e) : M} \quad \text{[Product Elimination]}$$

"If e is a pair of the product type $N \times M$, then the first projection $\pi_1(e)$ has the type N, and the second projection $\pi_2(e)$ has the type M." Note that, in this rule, e is presented as a single expression-variable, but we know it stands for a pair from its type $N \times M$. In both rules, the horizontal line has the force of an implication, which we could also write using $\Rightarrow$.

4 RULES FOR FUNCTIONS

Consider an infinite set of pairs: $\{\langle 1, \text{false}\rangle, \langle 2, \text{true}\rangle, \langle 3, \text{false}\rangle, \langle 4, \text{true}\rangle, \langle 5, \text{false}\rangle \ldots\}$. This set is an enumeration of a relationship between Natural numbers and Boolean values - it is in fact one possible representation of the function *even()*. Since functions have this clear, natural interpretation in our model, we are justified in introducing a special syntax for them:

$$\frac{x : D \ \vdash\ e : C}{\lambda x.e : D \rightarrow C} \quad \text{[Function Introduction]}$$

"If variable x has the type D and, as a consequence, expression e has the type C, we may conclude that a function of x with body e has the function type $D \rightarrow C$." This rule introduces the λ-syntax for functions and the arrow-syntax for function types. If you happen to be a hellenophobic[1] engineer, simply consider that: $\lambda x.e \Leftrightarrow f(x) = e$. The type

[1] Hellenophobe: a hater of Greek symbols.

signature for a function is always written as an *arrow type* $D \rightarrow C$, with the function's *domain* (input set) D on the left and the *codomain* (output set) C on the right. The premise of this rule relates the required types of $x : D$, $e : C$ using entailment $\vdash$, since the type of the body expression $e : C$ is not independent, but "follows from" the type of the variable $x : D$. This is because the body expression will contain occurrences of x, the variable. Consider that a function may return its argument - in that case, the result type *is* the argument type; there is clearly a dependency.

The function elimination rule explains the type of an expression involving a function application. In so doing, it also defines the parenthesis syntax for function application:

$$\frac{f : D \rightarrow C,\ \ v : D}{f(v) : C} \qquad \text{[Function Elimination]}$$

"If f is a function from $D \rightarrow C$, and v is a value of type D, then the result of the function application f(v) is of type C". This rule also expresses the notion of type-sound function application: it allows f to be applied *only* to values of the domain type D (technically, the rule allows you to deduce that the result of function application is well-typed in this circumstance, but is otherwise undefined).

Do we need rules for multi-argument functions? Not really, because we already have the separate product rules. The domain D in the function rules could correspond, if we so wished, to a type that was a product: $D \Leftrightarrow N \times M$. In this case, the argument value would in fact be a pair $v : N \times M$. We assume that any single type variable in one rule can be matched against a constructed type in another rule, if we so desire.

5 RULES FOR RECORDS

Most model encodings for objects [4, 5, 6] treat them as some kind of record with a number of labelled fields, each storing a differently-typed value. So far, we do not have a construction for records in our model. However, consider that a record is rather like a finite set of pairs, relating *labels* to *values:* {⟨name, "John"⟩, ⟨surname, "Doe"⟩, ⟨age, 25⟩, ⟨married, false⟩}. Since a record has this clear, natural interpretation in the model, we are justified in introducing a special syntax. If A is the primitive type of labels:

$$\frac{\alpha_i : A, \qquad e_i : T_i}{\{\alpha_1 \mapsto e_1, ..., \alpha_n \mapsto e_n\} : \{\alpha_1 : T_1, ..., \alpha_n : T_n\}} \ \textit{for}\ i = 1..n \qquad \text{[Record Introduction]}$$

"If there are n distinct labels α_i, and n values e_i of different corresponding types T_i then a record, constructed by pairing the ith label with the ith value, has a record type, which is constructed by pairing the ith label with the corresponding ith type". This rule uses the index i to link corresponding labels, values and types. In the conclusion, the set-braces are used for records and record types deliberately, since these are *sets of pairs*. The label-to-value pairings are notated as mappings $\alpha_i \mapsto e_i$, for visual clarity and convenience.

The corresponding *record elimination* rule introduces the *dot* or *record selection* operator, defining how to deconstruct a record to select a field and then determine its type:

$$\frac{e : \{\alpha_1 : T_1, ..., \alpha_n : T_n\}}{e.\alpha_i : T_i,} \quad \textit{for } i = 1..n \qquad \text{[Record Elimination]}$$

"If e has the type of a record, with n labels α_i, mapping to types T_i, then the result of selecting the ith field $e.\alpha_i$ has the ith type T_i."

6 APPLYING THE RULES

We have a set of rules for constructing pairs, functions and records. With this, we can model simple objects as records. Ignoring the issue of encapsulation for the moment, a simple Cartesian point object may be modelled as a record whose field labels map to simple values (attributes) and to functions (methods). We require a function for constructing points:

$$\text{make-point} : \text{Integer} \times \text{Integer} \rightarrow \text{Point}$$

This is a type declaration, stating that *make-point* is a function that accepts a pair of Integers and returns a Point type (which is so far undefined). The full definition of *make-point:*

$$\begin{aligned}\text{make-point} = \lambda(e : \text{Integer} \times \text{Integer})\ .\\ \{\ x \mapsto \pi_1(e),\ y \mapsto \pi_2(e),\ \text{equal} \mapsto \lambda(p : \text{Point}).(\pi_1(e) = p.x \wedge \pi_2(e) = p.y)\ \}\end{aligned}$$

names the argument expression *e* supplied upon creation and returns a record having the fields *x, y* and *equal*. The *x* and *y* fields map to simple values, projections of e; the *equal* field maps to a function, representing a method. Note that *make-point* is built up in stages according to the type rules. The *product introduction* rule can construct a pair type: Integer × Integer from primitive Integers. The function type of *equal*: Point → Boolean can be inferred from the type Point supplied as the argument, and the type of the body expression, using the *function introduction* rule: the body is a logical "and" ∧ expression, a primitive Boolean operation provided with the starter-kit. The record type Point is the type of the value returned by *make-point*. Using the *record introduction* rule, we determine that this is equivalent to a record type: { x : Integer, y : Integer, equal : Point → Boolean }, by examining the individual types of the label-value pairs supplied as its fields. Finally, even *make-point* is properly constructed using the *function introduction* rule, with Integer × Integer as the domain type and Point as the codomain type.

The rules permit us to deduce that objects can be constructed using *make-point*, and also that they are well-typed. Let us now construct some points and expressions involving points, to see if these are well-typed. The *let ... in* syntax is a way of introducing a scope for the values p1 and p2, in which the following expressions are evaluated:

let p1 = make-point(3, 5) **in**
    p1.x;

Is p1.x meaningful, and does it have a type? The record elimination rule says this is so, provided p1 is an instance of a suitable record type. Working backwards, p1 must be a record with at least the type: {... x : X ... } for some type X. Working forwards, p1 was constructed using *make-point*, so we know it has the Point type, which, when expanded, is equivalent to the record type: { x : Integer, y : Integer, equal : Point → Boolean }, which also has a field x : Integer. Matching up the two, we can deduce that p1.x : Integer.

let p1 = make-point(3, 5), p2 = make-point(3, 7) **in**
    p1.equal(p2);

Is p1.equal(p2) meaningful, and does it have a type? Again, by working backwards through the record elimination rule, we infer that p1 must have at least the type {... equal : Y ...} for some type Y. Working forwards, we see that p1 has a field equal : Point → Boolean, so by matching up these, we know Y ⇔ Point → Boolean. So, the result of selecting p1.equal is a function expecting another Point. Let us refer to this function as f. In the rest of the expression, f is applied to p2, but is this type correct? Working forwards, p2 was defined using *make-point*, so has the type Point. Working backwards through the function elimination rule, the function application f(p2) is only type-sound if f has the type Point → Z, for some type Z. From above, we know that f : Point → Boolean, so by matching Z ⇔ Boolean, we confirm that the expression is well-typed and also can infer the expression's result type: p1.equal(p2) : Boolean.

7 CONCLUSION

We constructed a mathematical model for simple objects from first principles, in order to show how it is possible to motivate the existence of something as relatively sophisticated as an object with a (constant) state and methods, using only the most primitive elements of set theory and Boolean logic as a starting point. The type rules presented were of two kinds: introduction rules describe how more complex constructions, such as functions and records, are formed and under what conditions they are well-typed; elimination rules describe how these constructions may be decomposed into their simpler elements, and what types these parts have. Both kinds of rule were used in a typechecker, which was able to determine the syntactic correctness of method invocations. The formal style of reasoning, chaining both forwards and backwards through the ruleset, was illustrated. The simple model still has a number of drawbacks: there is no updating or encapsulation of state; there are problems looming to do with recursive definitions; and we ignored the context (scope) in which the definitions take effect. In the next article, we shall examine some different object encodings that address some of these issues.

REFERENCES

[1] A J H Simons, Perspectives on type compatibility, *Journal of Object Technology 1(1),* May, 2002.

[2] A J H Simons, Appendix 1 : λ-Calculus, in: *A Language with Class: the Theory of Classification Exemplified in an Object-Oriented Language, PhD Thesis,* University of Sheffield, 1995, 220-238. See http://www.dcs.shef.ac.uk/~ajhs/classify/.

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

[4] 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.

[5] 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.

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

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

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:

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

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**
    $inc = \lambda x.(x + y)$

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: $inc(2) \Rightarrow 5$; $inc(4) \Rightarrow 7$; 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: $\lambda(a : A).e$, 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:

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

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**
    $aPoint = \{\ x \mapsto xv,\ y \mapsto yv,\ equal \mapsto \lambda(p : Point).(xv = p.x \wedge yv = p.y)\ \}$

[1] If the λ-calculus syntax still puzzles you, consider that: $inc = \lambda x.(x + y)$ 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:

$$\begin{array}{l} \textbf{let}\ xv = 2,\ yv = 3\ \textbf{in} \\ \quad aPoint = \mu\, self\,.\ \{\ x \mapsto xv,\ y \mapsto yv, \\ \qquad equal \mapsto \lambda(p : Point).(self.x = p.x \wedge self.y = p.y)\ \} \end{array}$$

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:

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

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:

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

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*:

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

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: $\text{Point}_0 = \bot$. In this, $\bot$ stands for the undefined type[2], meaning that we know nothing at all about it. The next approximation is:

$$\text{Point}_1 = \text{GenPoint}[\text{Point}_0] = \{x : \to \text{Integer};\ y : \to \text{Integer};\ \text{equal} : \bot \to \text{Boolean}\}$$

[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:

$$\text{Point}_2 = \text{GenPoint}[\text{Point}_1] = \{x : \rightarrow \text{Integer};\ \ y : \rightarrow \text{Integer};\\ \text{equal} : \text{Point}_1 \rightarrow \text{Boolean}\}$$

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):

$$\text{Point} = \text{GenPoint}[\text{GenPoint}[\text{GenPoint}[\text{GenPoint}[\text{GenPoint}[\ \dots\ ]]]]]$$

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:

$$\mathbf{Y}\ [\text{GenPoint}]\ \Rightarrow\ \text{GenPoint}[\text{GenPoint}[\text{GenPoint}[\text{GenPoint}[\text{GenPoint}[\ \dots\ ]]]]]$$

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):

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

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:

$$aPoint = [x = \varsigma(self)\ 2,\ y = \varsigma\ (self)\ 3,\ equal = \varsigma\ (self)\ \lambda(p)\ self.x = p.x \wedge self.y = p.y]$$

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:

$$Point = \mu\ pnt\ [x : Integer,\ y : Integer,\ equal : pnt \rightarrow Boolean]$$

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

JOURNAL OF OBJECT TECHNOLOGY

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

Vol. 1, no. 5, November-December 2002

The Theory of Classification Part 4: Object Types and Subtyping

Anthony J H Simons, Department of Computer Science, University of Sheffield, UK

1 INTRODUCTION

This is the fourth article in a regular series on object-oriented type theory, aimed specifically at non-theoreticians. The "Theory of Classification" will 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. So far, we have covered the foundations of type theory and symbolic calculi. Along the way, we shall look at some further theoretical approaches, such as F-bounds and matching. In this article, we focus on the fundamental notion of subtyping.

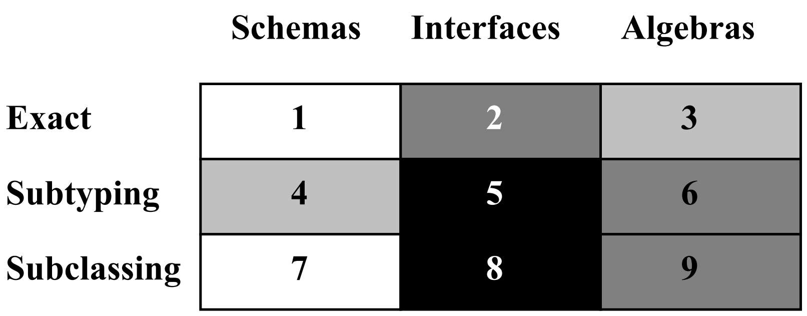


Figure 1: Dimensions of Type Checking

Previous articles introduced the notion of type [1], motivated rules for constructing types [2] and compared different formal encodings for objects [3]. We consolidate here on the *functional closure* encoding of objects and object types that we intend to use in the rest of the series. Previously, we noted how component compatibility can be judged from different perspectives, according to *representation*, *interface* or full *behavioural* correspondence; and type checking may be *exact*, or according to more flexible schemes which we called *subtyping* and *subclassing* [1, 2]. This yields the nine possible combinations shown again in figure 1, of which the darker shaded areas interest us the most. In this fourth article, we shall construct syntactic subtyping rules (box 5 in figure 1) to determine when it is safe to substitute one object in place of another, where a different type was possibly expected.

2 OBJECTS AND OBJECT TYPES

In the following, we refer to *objects* and *object types*. Objects are modelled in the calculus as simple records, whose fields consist of labels which map to values [2, 3]. The values can be of any kind, for example, simple integer values, or functions (representing methods), or indeed other objects. In the model, each object contains all of its own methods - the calculus does not bother to represent indirection to a shared table of methods, which is merely an implementation strategy in object-oriented languages. The calculus only has to capture the notion of field selection, using the record selection "dot" operator, which then works for both attribute access and method invocation [2].

In the calculus, we define things using "name = expression" to associate names with equivalent values (or types, below). The names are simply convenient abbreviations for the associated definitions. Simple objects may be written directly in the calculus, for example an arbitrary instance may be defined as a literal record:

$$\text{aPerson} = \{\text{name} \mapsto \text{"John"};\ \text{surname} \mapsto \text{"Doe"};\ \text{age} \mapsto 25;\ \text{married} \mapsto \text{false}\}$$

The type of an object follows automatically from the types of the values used for its fields. A suitable corresponding type for the instance above may be given as:

$$\text{Person} = \{\text{name} : \text{String};\ \text{surname} : \text{String};\ \text{age} : \text{Integer};\ \text{married} : \text{Boolean}\}$$

Object types are modelled in the calculus as record types, that is, records whose fields consist of labels which map to types (note the use of ":" to indicate "has the type", rather than "$\mapsto$" meaning "maps to the value"). We always use the term *object type* to denote the type of an object, to distinguish straightforward types from the more subtle notion of *class*, which potentially has other interpretations.

3 RECURSIVE OBJECTS AND OBJECT TYPES

Objects are frequently recursive, where they refer to their own methods, via *self*. Likewise, object types are recursive where the type signatures of their methods accept or return objects of the same type [3]. Since recursion is not built-in as a primitive notion, we must appeal to the fixpoint theorem to construct recursive objects and types [3]. A conventional notation is used to denote an object that has been recursively constructed:

$$\text{aPoint} = \mu\text{self} . \{x \mapsto 2;\ y \mapsto 3;\ \text{equal} \mapsto \lambda p.(p.x = \text{self}.x \wedge p.y = \text{self}.y)\}$$

in which μ*self* indicates that *self* is recursively bound in the record after the dot. This is really a short-hand for defining an *object generator*, a function of *self* (note the difference: λ*self*):

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

and then constructing aPoint from this generator, using the fixpoint combinator **Y** (see [3]):

$$\text{aPoint} = (\mathbf{Y}\ \text{genAPoint}) \;\Rightarrow\; \text{genAPoint}(\text{genAPoint}(\text{genAPoint}(...)))$$

Since it is inconvenient to have to keep on appealing to this construction in every recursive definition, we use the μ-convention to denote recursive definitions directly. Recursive object types may also be denoted using this convention. A suitable corresponding type for the instance above is given as:

$$\text{Point} = \mu\sigma . \{x : \text{Integer};\ y : \text{Integer};\ \text{equal} : \sigma \to \text{Boolean}\}$$

in which μσ indicates that σ, the self-type of points, is recursively bound in the record type after the dot. Note in passing how the type of equal is a function type (arrow type) and that equal accepts an argument having the same self-type, which accounts for Point being a recursive type. In type definitions, the μ-convention is a short-hand for the whole rigmarole of defining a *type generator*, a type function of σ (σ is the function's type parameter):

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

and then constructing the recursive Point type using **Y** (in a similar manner [3]):

$$\text{Point} = [\mathbf{Y}\ \text{GenPoint}] \;\Rightarrow\; \text{GenPoint}[\text{GenPoint}[\text{GenPoint}[...]]]$$

In the rest of this article, we will use the μ-convention throughout for recursive objects and recursive types. In later articles, object generators and type generators will acquire a new importance in the definition of the notion of *class*. It is theoretically sound to use **Y** to "fix" both recursive objects and recursive types in the same universe, provided that records contain fields of functions and do not refer directly to themselves [4].

4 SIMPLE SUBTYPING

Object-oriented languages take the ambitious view that, in the universe of types, all types are *systematically related* in a type hierarchy. Types lower in the hierarchy are somehow compatible with more general types higher in the hierarchy. By contrast, older languages like Pascal treated every type as distinct and unrelated[1] The more flexible object-oriented notion of type compatibility is based on a safe substitution property. Liskov is frequently cited as the source of this idea [5], popularly known as the *Liskov Substitutability Principle* (LSP), although Cardelli and Wegner [6] deserve shared credit:

> "What is wanted here is something like the following substitution property: if for each object o_1 of type S there is an object o_2 of type T such that, for all programs P defined in terms of T, the behavior of P is unchanged when o_1 is substituted for o_2, then S is a subtype of T." [5]

These authors all identify substitutability with subtyping. In fact, subtyping is just one formal approach which satisfies the principle of safe substitution - there are other more subtle and equally formal approaches, which we shall consider in later articles.

Henceforth, we shall use "<:" to mean "is a subtype of". The notion of subtyping derives ultimately from subsets in set theory. In exactly the same way that: x : X ("x is of type X") can be interpreted as $x \in X$ ("x is a member of the set X") in the model, so the subtyping relationship, for example: Y <: X ("Y is a subtype of X") can be interpreted consistently as the subset relationship $Y \subseteq X$ ("Y is a subset of X") in the model. Objects may belong to a hierarchy of increasingly more general types: consider that if y : Y and Y <: X holds, then y : X is also true. The set-theoretic model supports this directly: If $y \in Y$ and $Y \subseteq X$, then it follows that $y \in X$, by the definition of subsets:

$$Y \subseteq X ::= \forall y \, . \, y \in Y \Rightarrow y \in X \qquad \text{[Definition of a subset]}$$

While this translation works immediately for primitive types, the object substitutability principle is couched in terms of *equivalent behaviour* after a substitution. The behaviour of an object is characterised by its methods; so we are obliged to consider the type-relationships between all corresponding methods in the original and substitute objects.

5 FUNCTION SUBTYPING

For a substitute object y : Y to behave exactly like the original object x : X, then every method of X must have a corresponding method in Y that behaves like the original method of X. If we are only interested in *syntactic* compatibility [1] (that is, between interfaces), this reduces to checking the type signatures of related pairs of methods. In many object-oriented languages, the correspondence between the methods of X and Y is at least partly assured by defining Y as an extension of X - in this case, Y may inherit the majority of X's methods unchanged. However, we must cater for the general case, in which Y substitutes different methods in place of those in X (known as method *redefinition*, or *overriding*).

Consider a method f of X, which we shall call $f_X : D_X \rightarrow C_X$, to indicate that it is a function accepting an argument of some type D and yielding a result of some other type C. This is to be replaced by a substitute method f of Y, which we shall call $f_Y : D_Y \rightarrow C_Y$, with the intention that Y should still behave like X. Under what conditions can f_Y be safely substituted in place of f_X? From the syntactic point of view, there are two obligations:

- f_Y must be able to handle at least as many argument values as f_X could accept; we express this as a constraint on the domains (argument types): $D_Y \supseteq D_X$; and

- f_Y must deliver a result that contains no more values than the result of f_X expected; we express this as a constraint on the codomains (result types): $C_Y \subseteq C_X$.

A helpful way to think about these obligations is to consider how a program might fail if they were broken. What if f_Y accepted fewer argument values than f_X? In this case, there might be some valid arguments supplied to f_X in the original working program that were not recognised by f_Y after the substitution, causing a run-time exception. What if f_Y delivered more result values than f_X? In this case, the call-site expecting the result of f_X might receive a value that was outside the specified range when f_Y was invoked instead.

From this, we can motivate the *function subtyping* rule, which expresses under what conditions a function type $D_Y \rightarrow C_Y$ is a subtype of another function type $D_X \rightarrow C_X$. This uses the same style of natural deduction rule as before [2]:

$$\frac{D_X <: D_Y,\ \ C_Y <: C_X}{D_Y \rightarrow C_Y\ <:\ D_X \rightarrow C_X} \qquad \text{[Function Subtype]}$$

"If the domain (argument type) D_Y is larger than the domain D_X, and the codomain (result type) C_Y is smaller than the codomain C_X, then the function type $D_Y \rightarrow C_Y$ is a subtype of the function type $D_X \rightarrow C_X$." Note how there is an assymmetry in this rule: in the subtype function, the codomain is also a subtype, but the domain is a supertype. For this reason, we sometimes say that the domains are *contravariant* (they are ordered in the opposite direction) and the codomains are *covariant* (they are ordered in the same direction) with respect to the subtyping relationship between the functions.

The function subtyping rule is expressed formally using a single argument and result type. To extend this to methods accepting more than one argument, we recall the fact that a single type variable in one rule may be deemed equivalent to a product type in another rule [2]. In this case, the contravariant constraint applies to the two products as a whole, and so we need a *product subtyping* rule to break this down further:

$$\frac{S_1 <: T_1,\ \ S_2 <: T_2}{S_1 \times S_2\ <:\ T_1 \times T_2} \qquad \text{[Product Subtype]}$$

"The product type $S_1 \times S_2$ is a subtype of the product type $T_1 \times T_2$, if the corresponding types in the products are also in subtyping relationships: $S_i <: T_i$." The consequence for function subtyping is that, in a multi-argument function, all of the overriding (subtype) function's arguments must be supertypes of those in the function they replace.

6 RECORD SUBTYPING

The last piece of the subtyping jigsaw is to determine under what conditions a whole object type Y is a subtype of another object type X. Recall that object types are basically record types, whose fields are function types, representing the type signatures of the object's methods. According to Cardelli and Wegner [6], one record type is a subtype of another if it can be coerced to the other. For example, a Person type with the fields:

Person = {name : String; surname : String; age : Integer; married : Boolean}

might be considered a subtype of a DatedThing type having fewer fields:

DatedThing = {name : String; age : Integer}

because a Person can always be coerced to a DatedThing by "forgetting" the extra surname and married fields. Intuitively, this also satisfies the LSP, since a Person instance may always be used where a DatedThing was expected; any method invoked through a DatedThing variable will also exist in a Person object. This motivates the first part of the record subtyping rule (*record extension*):

$$\frac{a_1, \ldots a_k, \ldots a_n : A}{\{a_1 : T_1, \ldots a_n : T_n\} <: \{a_1 : T_1, \ldots a_k : T_k\}} \; \textit{for } 1 \le k \le n \qquad \text{[Record Extension]}$$

"If there are distinct labels a_i, then a longer record constructed with n fields, having the types $a_i : T_i$, is a subtype of a shorter record constructed with only the first k fields, provided that the common fields all have the same types." This rule basically asserts that adding to the fields of a record creates a subtype record.

Defining new types of object by extension is clearly a common practice in object-oriented programming languages. However, there is also the possibility of overriding some methods in the extended object. What requirement should we place on a record type if *all* of its fields were to change their types? According to the reasoning above which led to the function subtyping rule, any replacement methods should be subtypes of the originals. We can confirm this using a simpler example of field-type redefinition. If PositiveInteger <: Integer, then

PositiveCoordinate = {x : PositiveInteger; y : PositiveInteger}

is intuitively a subtype of the more general:

Coordinate = {x : Integer; y : Integer} because the set of all PositiveCoordinates is contained within the set of all Coordinates (the positive subset occupies the upper right quadrant of the Cartesian plane). This motivates the second part of the record subtyping rule (*record overriding*):

$$\frac{a_i : A,\ \ S_i <: T_i}{\{a_i : S_i\} <: \{a_i : T_i\}}\ \ \textit{for}\ i = 1..n \qquad\qquad \text{[Record Overriding]}$$

"A record that has n labelled fields of the types $a_i : S_i$ is a subtype of a record having the same labelled fields of the different types $a_i : T_i$, provided that each type S_i is a subtype of the corresponding type T_i." This rule uses the index i to link the corresponding labels, types and subtypes appropriately. Combining the record extension rule with the overriding rule gives the complete, but more complicated looking, *record subtyping* rule:

$$\frac{a_i : A,\ \ S_1 <: T_1, \ldots\ S_k <: T_k}{\{a_1 : S_1, \ldots\ a_n : S_n\} <: \{a_1 : T_1, \ldots\ a_k : T_k\}}\ \ \textit{for}\ 1 \leq k \leq n \qquad \text{[Record Subtyping]}$$

"A longer record type $\{a_i : S_i\}$ with n fields is a subtype of a shorter record type $\{a_i : T_i\}$ with only k fields, provided that, in the first k fields that they have in common, every type S_i is a subtype of the corresponding type T_i." The record subtyping rule generalises the previous two rules. If $S_i = T_i$, for i = 1..k, then it reduces to the record extension rule. If k = n, then it reduces to the record overriding rule.

To complete the picture, we must say a little about recursion and subtyping. The subtyping rules given above depend on having complete type information about all the elements that make up the types. One of these elements could be the self-type, in a recursive type. We cannot make definite assertions about subtyping between recursive types, unless we first make some assumptions about the corresponding self-types. Cardelli [7] expresses this (here slightly simplified, ignoring the context) as the rule:

$$\frac{\sigma <: \tau\ \vdash\ S <: T}{\mu\sigma.S\ <:\ \mu\tau.T}\ \ \sigma \text{ free only in S, } \tau \text{ free only in T} \qquad \text{[Recursive Subtype]}$$

"If assuming that σ is a subtype of τ allows you to derive that S is a subtype of T, then the recursive type $\mu\sigma.S$ is a subtype of the recursive type $\mu\tau.T$, where S may contain occurrences of the self-type σ and T may contain occurrences of the self-type τ." This rule sets up the relationship between the self-type variables and the types which depend on them. In a later article, we shall revisit the interactions between recursion and subtyping, which proves to be quite a thorny problem for object-oriented type systems.

7 CONCLUSION

We have reconstructed the classic set of subtyping rules for object types, including rules for recursive types. These rules have the following impact on object-oriented languages that wish to preserve subtyping relationships. A class or interface name may be understood as an abbreviation for the type of an object, where the type is expressed in full as the set of method type signatures owned by the class (or interface). A subclass or derived interface may be understood as a subtype, if it obeys the rule of record subtyping. In particular, the subtype may add (but not remove) methods, and it may replace methods with subtype methods.

A method is a valid replacement for another if it obeys the function subtyping rule. In particular, the subtype method's arguments must be of the same, or more general (but not more restricted) types; and its result must be of the same, or a more restricted (but not more general) type. Very few languages obey both the covariant and contravariant parts of the function subtyping rule (Trellis [8] is one example). Languages such as Java and C++ are less flexible than these rules allow, in that they require replacement methods to have exactly the same types. Partly, this is due to interactions with other rules for resolving name overloading; but also it reflects a certain weakness in the type systems of languages based on subtyping, which we will start to explore in the next article.

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/column5.

[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] 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, pages 49-57. http://www.jot.fm/issues/issue_2002_09/column4

[4] K Bruce and J Mitchell, PER models of subtyping, recursive types and higher-order polymorphism, *Proc. 19th ACM Symp. Principles of Prog. Langs.,* (1992), 316-327.

[5] B Liskov, Data abstraction and hierarchy, *ACM Sigplan Notices, 23(5),* (1988), 17-34.

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

[7] L Cardelli, Amber, *Combinators and Functional Programming Languages, LNCS, 242* (1986), 21-47.

[8] C Schaffert, T Cooper, B Bullis, M Kilian and C Wilpolt, An introduction to Trellis/Owl, *Proc. 1st ACM Conf. Object-Oriented Prog. Sys., Lang. and Appl., pub. ACM Sigplan Notices, 21(11),* (1986), 9-16.

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

[1] Apart from subrange types in Pascal, which were considered compatible with their base types.

JOURNAL OF OBJECT TECHNOLOGY

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

Vol. 2, No. 1, January-February 2003

The Theory of Classification Part 5: Axioms, Assertions and Subtyping

Anthony J H Simons, Department of Computer Science, University of Sheffield, UK

1 INTRODUCTION

This is the fifth article in a regular series on object-oriented type theory, aimed specifically at non-theoreticians. The series has been investigating the notion of simple object types and subtyping from the *syntactic* point of view, that is, judging type compatibility by the type signatures of an object's methods. In terms of the dimensions of type checking in figure 1, we have considered *exact* type correspondence (box 2, see earlier article [1]) and *subtyping*, a more flexible kind of type correspondence (box 5, see previous article [2]). This allows us to determine whether a given type provides enough operations to satisfy a given interface and whether the supplied operations have suitable type signatures.

	Schemas	**Interfaces**	**Algebras**
Exact	1	2	3
Subtyping	4	5	6
Subclassing	7	8	9

Figure 1: Dimensions of Type Checking

However, component compatibility is not just a matter of observing the conventions on type signatures. An object could offer all the expected operations, but still execute in a completely perverse way (see earlier article [3]). It is equally important to know whether a component *behaves* in the way expected by the program in which it is used. For this, an approach is required which can model the *semantics* of object types and capture precisely how they execute. Semiformal methods for capturing behaviour include statecharts [4] and various assertion languages, such as OCL [5]. A means of incorporating assertions

[6] into a practical programming language was first introduced by Eiffel [7]. This expresses the meaning of operations in terms of the *preconditions* and *postconditions* that they satisfy, together with *invariants* characterising the unchanging properties of object types.

All of these approaches are incomplete realisations of the more fundamental *algebraic* approach to defining the meaning of abstract datatypes. In this article, we consider the exact specification of a type's *behaviour* (box 3 in figure 1) and also the relationship between algebraic specification and subtyping, which will allow us to prove when one object *behaves in a subtype-conformant way* to another (box 6 in figure 1).

2 INITIAL AND FINAL ALGEBRAS

Mathematicians have been experimenting with notions of abstract types and classification since the early part of the 20th century. Much of this work falls within the remit of *formal algebra* and *category theory*. An algebra is an abstract type definition, consisting of a set of elements (a *sort*) and a collection of operations acting on the set, characterised by their type signatures and logical axioms (see earlier article [3]). Some algebras are related to each other, in that they have the same operations, but the properties of the operations may vary slightly from algebra to algebra. As an example, consider that the List type, with *cons, head* and *tail* can be mapped onto an abstract Stack type with *push, top* and *pop*. Such a mapping relationship is called a *homomorphism* (literally, "same form"), and where an inverse mapping also exists, this is called an *isomorphism* ("identical form").

In the universe of algebras, families of algebras exist in which elements and operations that are distinct in one algebra become merged and indistinguishable in other algebras. Consider that if "+" is an abstract operation with just the property of *associativity* (for example, like "+" used to append Strings in Java), then "a + b" and "b + a" will mean different things. However, if in another algebra, "+" also has the property of *commutativity* (for example, like "+" used to add Integers), then "a + b" and "b + a" will mean the same thing. In the universe of algebras, homomorphisms are arrows running from the algebras with more distinguishable elements to the algebras with fewer distinguishable elements. At one end of this universe, an algebra called the *initial algebra* exists[1], whose elements are more distinguishable than in any other. At the opposite end, a *final algebra* exists, whose elements are the least distinguishable.

[1] Technically, an *initial algebra* is one from which a unique homomorphism maps to every other algebra. Similarly, a *final algebra* is one into which a unique homomorphism maps from every other algebra. By this definition, initial and final algebras may not exist in certain semantic domains.

3 AXIOMATIC SEMANTICS

When defining an algebraic type, the first concern is to establish under what semantics the axioms of the algebra will be interpreted. The Ordinal type[2] from an earlier article [3] was defined using *final algebra semantics*, in which we assume that all elements of the type are equivalent, unless we can prove them to be distinct:

$$\begin{array}{lll}
\multicolumn{3}{l}{\text{Ordinal} = \exists\ \text{ord}\ .\ \{\text{first:} \rightarrow \text{ord};\ \ \text{succ: ord} \rightarrow \text{ord}\}} \\
\multicolumn{3}{l}{\forall x, y : \text{Ordinal}\ .} \\
 & \text{succ}(x) \neq \text{first}() & (1) \\
\wedge & \text{succ}(x) \neq x & (2) \\
\wedge & \text{succ}(x) = \text{succ}(y) \Leftrightarrow x = y & (3)
\end{array}$$

The onus is on showing that the first() element is distinct from any successor succ(x), that any element x is distinct from its immediate successor succ(x), and that by induction all the elements of the type are eventually distinct, such that Ordinal is inhabited by a series of monotonically increasing elements: first(), succ(first()), succ(succ(first())) ...

We change our approach to define the behaviour of object types. The Stack algebra below is defined using *initial algebra semantics*, in which we assume that all the elements of the algebra are distinct, unless we can prove them to be equal. Note in particular how axiom (6) asserts when two Stacks can be judged to be equivalent; and how axiom (5) asserts under what conditions the element you retrieve is equivalent to the one you previously inserted:

$$\begin{array}{lll}
\multicolumn{3}{l}{\text{Stack} = \forall T.\ \mu\text{stk}.\{\text{push} : T \rightarrow \text{stk};\ \ \text{pop} : \rightarrow \text{stk};\ \ \text{top} : \rightarrow T;\ \ \text{empty} : \rightarrow \text{Boolean};} \\
 & \multicolumn{2}{l}{\text{size} : \rightarrow \text{Natural}\};\ \ \text{newStack} : \ \rightarrow \text{Stack};} \\
\multicolumn{3}{l}{\forall e : T\ .\ \forall s : \text{Stack}\ .} \\
 & \text{newStack().empty()} & (1) \\
\wedge & \neg\text{s.push(e).empty()} & (2) \\
\wedge & \text{newStack().size()} = 0 & (3) \\
\wedge & \text{s.push(e).size()} = 1 + \text{s.size()} & (4) \\
\wedge & \text{s.push(e).top()} = e & (5) \\
\wedge & \text{s.push(e).pop()} = s & (6)
\end{array}$$

In the signature of Stack's operations, the use of ∀T "for all types T" indicates a *generic* Stack definition, since T is later used as the element-type. μ *stk* indicates that Stack is a recursive record type in which *stk* refers to the eventual Stack type. All Stack operations

[2] I am indebted to Kim Bruce for pointing out that the third Ordinal axiom is required to allow the rule of induction to operate as intended. Originally, I missed this.

are defined as methods that accept and return elements of these and other types (which we assume are defined in associated algebras). The initial constructor for a Stack is not a member of the record, since a Stack instance does not create itself. The axioms which define the meaning of Stack's operations are equations which refer to arbitrary stacks s : Stack and elements e : T. Each axiom asserts a boolean expression which holds true for the type. Axioms are combined using "$\wedge$" logical *and*. All other properties of Stacks can be inferred from these axioms.

4 INDUCTIVE DEFINITIONS

The strategy for defining the behaviour of an object type uses an inductive approach, similar to recursive function definition in a functional programming language. In the axioms defining the meaning of *size*, note how there is a base case (3) for new Stacks, and a step case (4) for arbitrary Stacks *s*. The step case always defines a property in terms of something simpler that is closer to the base case (a *recurrence relationship*). Thereafter, the size of any Stack can be derived using repeated application of these rules.

The equations always relate pairs of methods on the left-hand side and assert that a nested invocation of these methods is equivalent to something else on the right-hand side (think of Stack axioms (1) and (2) as being "equivalent to true" on the right-hand side). How do you decide which pairs of methods to relate; and how do you know when sufficient axioms have been defined? If too many axioms are supplied, a theorem prover might waste time exploring redundant solutions that could be derived from other axioms. To help with this problem, the functions of the type are sometimes divided into three groups:

- constructors - the smallest set of functions returning the type, which, taken together, can generate *every single instance* of the type;
- transformers - the remaining functions which return the type, but which are non-primitive in the sense that they can be defined in terms of primitive constructors;
- observers - functions returning something other than the type, typically because they inspect part of the type or compute some value from it.

Note that *constructors* mean more than the usual object-oriented sense of the word. In a pure functional calculus, pushing an element onto a Stack means creating a new Stack object onto which the extra element has been added. Accordingly, *new* and *push* are both algebraic constructors for the Stack. With these two operations, we can create every single possible Stack instance. Consequently, *pop* is a non-primitive transformer and its result can be defined in terms of other constructors. Likewise, *top, empty* and *size* are observers.

Thereafter, the maximum number of axioms to define is decided. You need no more than an axiom for *each constructor paired with every other non-constructor*. From this, you would expect to define at most $2 \times 4 = 8$ axioms for Stack. However, only 6 were supplied above; this means that in certain contexts, some of Stack's methods are undefined, an issue to which we shall return below.

5 DEDUCTIVE REASONING

Let us now assume that we wish to derive some property of Stacks. For example, what is the *size* of a Stack after a sequence of *push* and *pop* operations? The problem corresponds to simplifying a nested method expression, such as:

newStack().push(e1).push(e2).pop().size()

To simplify this, we look for axioms which relate suitable pairs of operations on their left-hand side, and substitute the corresponding equivalent expressions on their right-hand side. Working backwards from the end of the expression, and given ∀e : T, ∀s : Stack:

- There is no axiom for s.pop().size(); but this is not an omission, since pop is not a primtive constructor and we can derive its meaning elsewhere. Instead, we look further.

- There is an axiom (6) with s.push(e).pop() on the left-hand side, so if we make the substitutions: s ← newStack().push(e1) and e ← e2, then the corresponding right-hand side resubstitution is: s → newStack().push(e1), giving the simplification:

  newStack().push(e1).push(e2).pop().size() ⇒ newStack().push(e1).size() [by axiom 6].

- There is an axiom (4) with s.push(e).size() on the left-hand side, so if we make the substitutions: s ← newStack() and e ← e1, then the corresponding right-hand side resubstitution is: 1+s.size() → 1+newStack().size(), giving the simplification:

  newStack().push(e1).size() ⇒ 1+newStack().size() [by axiom 4].

- Finally, there is an axiom (3) giving newStack().size() directly:

  1+newStack().size() ⇒ 1+0 [by axiom 3].

and the final answer 1+0 = 1 is obtained using the algebra for Natural numbers in a similar fashion. This is the right answer and, hopefully, the one the reader expected!

6 ERRORS AND DEFERRED DEFINITIONS

The two axioms omitted from Stack's equations were those relating: newStack().top() and newStack().pop(). This is because the meanings of *top* and *pop* are deliberately left undefined for *new* Stacks. "Undefined" can be interpreted variously to signify that a method's meaning has *not yet* been fully specified, or that the method's result *is an error* in this context. In the case of *top* and *pop*, it is clear that these should always raise exceptions with a new Stack. A function that is not defined in all contexts is known as a *partial function*.

There may be other valid reasons for "saying less about" a type than just to specify error cases. Consider first that a Queue type looks quite similar to a Stack, except that some of its functions behave in a slightly different way:

$$
\begin{array}{lll}
\multicolumn{3}{l}{\text{Queue} = \forall T.\ \mu \text{que}.\{\text{push} : T \rightarrow \text{que};\ \ \text{pop} : \rightarrow \text{que};\ \ \text{top} : \rightarrow T} \\
 & \multicolumn{2}{l}{\text{empty} : \rightarrow \text{Boolean};\ \ \text{size} : \rightarrow \text{Natural}\};\ \ \text{newQueue} : \rightarrow \text{Queue}} \\
\multicolumn{3}{l}{\forall e : T\ .\ \forall q : \text{Queue}\ .} \\
 & \text{newQueue().empty()} & (1) \\
\wedge & \neg \text{q.push(e).empty()} & (2) \\
\wedge & \text{newQueue().size()} = 0 & (3) \\
\wedge & \text{q.push(e).size()} = 1 + \text{q.size()} & (4) \\
\wedge & \text{q.push(e).top()} = \text{e}\ \textbf{if}\ \text{q.empty()} & (5a) \\
 & \qquad\qquad\qquad = \text{q.top()}\ \textbf{otherwise} & (5b) \\
\wedge & \text{q.push(e).pop()} = \text{q}\ \textbf{if}\ \text{q.empty()} & (6a) \\
 & \qquad\qquad\qquad = \text{q.pop().push(e)}\ \textbf{otherwise} & (6b)
\end{array}
$$

The differences are in axioms (5) and (6), which assert FIFO properties for a Queue, contrasting with the LIFO properties asserted above for a Stack. Queue's equations also demonstrate how a right-hand side can be split into several cases, using **if**-clauses. The reader should experiment with some examples, in the deductive style shown above, to see that the recurrence relation in axiom 6 causes elements to be added and removed in the right order. (The recurrence works by driving *pop* backwards, until it encounters a base case).

Given that Queue and Stack are *syntactically* identical (their methods have identical type signatures), it should be possible to create the type of an interface, or supertype, to which both Stack and Queue conform. Since both of these collections dispense their elements in a particular order, we shall call their abstract supertype *Dispenser*:

$$
\begin{array}{lll}
\multicolumn{3}{l}{\text{Dispenser} = \forall T.\ \mu \text{dsp}.\{\text{push} : T \rightarrow \text{dsp};\ \ \text{pop} : \rightarrow \text{dsp};\ \ \text{top} : \rightarrow T;} \\
 & \multicolumn{2}{l}{\text{empty} : \rightarrow \text{Boolean};\ \ \text{size} : \rightarrow \text{Natural}\};\ \ \text{newDispenser} : \rightarrow \text{Dispenser}} \\
\multicolumn{3}{l}{\forall e : T\ .\ \forall d : \text{Dispenser}\ .} \\
 & \text{newDispenser().empty()} & (1) \\
\wedge & \neg \text{d.push(e).empty()} & (2) \\
\wedge & \text{newDispenser().size()} = 0 & (3) \\
\wedge & \text{d.push(e).size()} = 1 + \text{d.size()} & (4)
\end{array}
$$

It is clear that both Stack and Queue satisfy the above definition, since they have the identical signatures, and obey the identical axioms (1) - (4). The fact that axioms (5) and (6) are missing means that, at the level of generality described by *Dispenser*, we cannot yet say anything about the order in which elements are inserted, accessed and removed: the specifications of *top* and *pop* are deferred. We say that Dispenser is *underspecified*.

7 UNDERSPECIFICATION AND SUBTYPING

The rules governing axioms and semantic subtyping follow from this. If a type is underspecified, then a subtype may be created by adding suitable axioms giving the missing meanings of the underspecified operations. Note that Stack and Queue define mutually exclusive axioms (5) and (6), such that one could never be a subtype of the other; yet both are semantic subtypes of Dispenser, since they only add to Dispenser's existing axioms and do not violate any of them.

Consider now that, in just one case, Stacks and Queues actually do behave identically in regard to *push*, *pop* and *top* - this is when they contain a single element. We could add this common information to the Dispenser type by writing *partial* axioms:

$\wedge$	d.push(e).top() = e **if** d.empty()	(5a)
$\wedge$	d.push(e).pop() = d **if** d.empty()	(6a)

These two axioms express, for both Stacks and Queues, that if you *push* an element into an empty container, this is the *top* element, and the one that is removed by *pop*. From this, it is clear that some notion of *axiom refinement* must happen in subtypes. In the case of Queue, we simply *complete* the partial axioms by adding parts (5b) and (6b). In the case of Stack, we drop the **if**-condition instead, such that Stack's axioms (5) and (6) cover more than the 1-element case. Dispenser's partial axioms are therefore still satisfied by both Stack and Queue. If we refine an axiom in a subtype, the subtyping condition is this: the refined axiom *must logically entail* the original axiom.

Finally, we can show a relationship between semantic and syntactic subtyping. Why does adding axioms, or strengthening axioms to cover more cases, create a subtype? Consider defining a set S by comprehension in relation to a set T, such that S contains all those elements in T which satisfy the extra axiom p(x):

$$S = \{\forall x \in T \mid p(x)\}$$

It is clear that, if all elements of T pass the test p(x), then S = T. However, if some elements of T fail the test, then $S \subset T$. Therefore, we can assert that: $S \subseteq T$, and this also means that S is a subtype of T [2].

8 CONCLUSION

We have developed an algebraic calculus for reasoning about the *complete* behaviour of object types, and demonstrated the effects of axioms upon subtyping. When seeking to apply the results of this analysis to assertion languages like OCL [5] and object-oriented languages like Eiffel [7] we have to translate from pure algebra into a piecemeal treatment in terms of invariants, pre- and postconditions. It is useful to think in terms of *strengthening* assertions:

- strengthening an *invariant* is identical to strengthening the axioms of an algebra, since the invariant applies constantly to the object type as a whole;
- strengthening a *method postcondition* corresponds either to strengthening the axioms on the result-type of the method, or strengthening the axioms on the object type itself; or possibly to both of these;
- strengthening a *method precondition* corresponds either to strengthening the axioms on the argument-types of the method, or strengthening the axioms on the object type itself; or possibly to both of these.

Since there is a direct relationship between axiom strengthening and subtyping, we can immediately apply our existing object subtyping rules [2] to derive subtyping rules governing the strengthening, or weakening of assertions. Recall that an object type which adds to the methods of another object type is a subtype. The subtype will define the semantics of the extra methods, providing more axioms, which is consistent with being a subtype. Some of these extra axioms may appear as strengthened invariants, which is also consistent. Apart from this, an object type may sometimes replace methods, so we must consider under what conditions this results in a subtype.

A method is a valid replacement for another if it obeys the function subtyping rule [2]. In particular, the subtype method's arguments must be of the same, or more general (but not more restricted) types; and its result must be of the same, or a more restricted (but not more general) type. Translating this into assertions, the method's preconditions may *possibly be weakened* (but never strengthened) and the method's postconditions may *possibly be strengthened* (but never weakened). In this regard, assertions also follow the contravariant argument-type and covariant result-type rules. Eiffel [7] obeys similar rules regarding the weakening of its preconditions and strengthening of its postconditions. The only area of conflict is where a precondition also affects the object type itself. To satisfy the contravariant rule, the precondition can only be weaker, but to satisfy object subtyping, the invariant can only be stronger. In practice, weaker preconditions can co-exist with stronger invariants, since the same object must satisfy the stronger of the two and doesn't care that the method would accept something less strict than itself.

REFERENCES

[1] A. J. H. Simons: *The Theory of Classification, Part 2: The Scratch-Built Typechecker*, Journal of Object Technology, vol. 1, no. 2, July-August 2002, pages 47-54. http://www.jot.fm/issues/issue_2002_07/column4

[2] A. J. H. Simons, *The Theory of Classification, Part 4*: *Object Types and Subtyping*, Journal of Object Technology, vol. 1, no. 5, November-December 2002, pages 27-35. http://www.jot.fm/issues/issue_2002_11/column2

[3] A. J. H. Simons, *The Theory of Classification, Part 1: Perspectives on Type Compatibility*, Journal of Object Technology, vol. 1, no. 1, May-June 2002, pages 55-61. http://www.jot.fm/issues/issue_2002_05/column5

[4] D. Harel and A. Naamad, The STATEMATE semantics of statecharts, *ACM Trans. Soft. Eng. Method., 5(4),* 1996, pages 293-333.

[5] J. Warmer and A. Kleppe, *The Object Constraint Language: Precise Modeling with UML*, (Reading MA : Addison Wesley, 1999).

[6] C A R Hoare, Proof of correctness of data representations, *Acta Informatica, 1,* 1972, pages 271-281.

[7] B. Meyer, *Object-Oriented Software Construction*, 2nd edn., Prentice Hall, 1995.

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

JOURNAL OF OBJECT TECHNOLOGY
Online at www.jot.fm. Published by ETH Zurich, Chair of Software Engineering **©JOT, 2003**

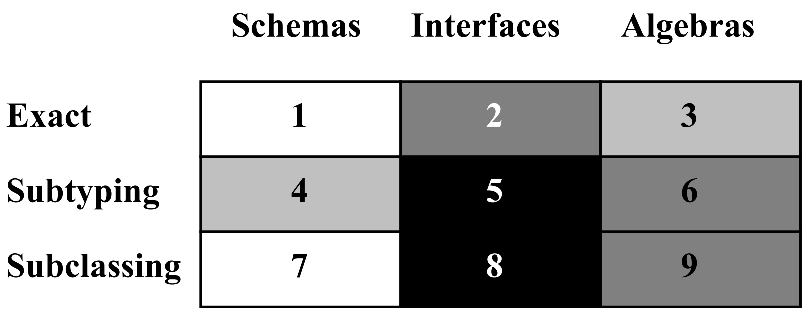
Vol. 2, No. 2, March-April 2003

The Theory of Classification Part 6: The Subtyping Inquisition

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

1 INTRODUCTION

This is the sixth article in a regular series on object-oriented type theory, aimed specifically at non-theoreticians. The series has been investigating the notion of simple object types and has so far developed a *theory of subtyping* which judges object type compatibility from both the *syntactic* point of view, that is, by the type signatures of an object's methods [1] and from the *semantic* point of view, that is, by the logical axioms asserted on an object's methods [2]. In terms of the dimensions of type checking in figure 1, we have considered the *exact* type correspondence of signatures (box 2) and behaviour (box 3), and the *subtype* correspondence of modified signatures (box 5) and refined behaviour (box 6).



	Schemas	Interfaces	Algebras
Exact	1	2	3
Subtyping	4	5	6
Subclassing	7	8	9

Figure 1: Dimensions of Type Checking

This kind of theory provides a much-needed tool for analysing the type safety and behavioural correctness of programming languages. In the late 1980s and early 1990s, the possibility that object-oriented languages might be insecure in their type systems, when judged according to subtyping [3], caused quite a stir, particularly in the software engineering community. This led to the greater prominence of subtype-conformant languages [4], but also sparked a new interest in the way object-oriented languages really seemed to behave [5]. The debate swung between the desire to force languages into obeying subtyping and the desire to develop more sophisticated formal models of object-oriented classes and inheritance. In this article, we take the former side (as devil's advocate, since we shall take the latter side in a subsequent article) and examine a number of popular object-oriented languages for their type safety, asking of each candidate subclass: "Are you, or have you ever been, a proper subtype?".

2 A PRACTICAL TEST FOR SUBTYPING

How well do popular object-oriented languages follow the rules of subtyping? How type secure are they generally? A quick survey may reveal hidden weaknesses or unappreciated strengths in your favourite language. Recall that our theory deals in terms of *objects* and *object types* (so far, we have not used the term *class* in any formal sense). Practical object-oriented languages have concrete *classes* which define the structure and behaviour of objects created at runtime. A class in this sense has both an implementation aspect and a typeful aspect, in that class names are usually also treated as type identifiers. We shall assume the same correspondence when making typing judgements here.

The main syntactic type rules of interest derive from the *record subtyping* and *function subtyping* rules [1]. These can be expressed informally for classes-viewed-as-types in the following way:

- Where a class S is intended to be a subtype of a class T, it must obey the extension rule: S may add to the methods of T, but never remove any methods from T; and the overriding rule: S may replace some methods of T, so long as the replacement methods R_i in S are subtypes of the corresponding methods M_i in T that they replace.
- Where a method R is intended to be a subtype replacement for a method M, it must obey the argument contravariance rule: any arguments of R may be more general than corresponding arguments in M, but never more specific; and the result covariance rule: the result of R may be more specific than the result of M, but never more general.

The main semantic behavioural rules of interest derive from the *addition of new constraints* and the *generalisation of constraints* [2], both of which are said to *strengthen* an axiom. These are converted into the more familiar assertion format below:

- Where a class S is intended to be a behavioural subtype of a class T, it must obey the invariant strengthening rule: S may have a stronger invariant than T, but never weaker; and the behavioural conformance rule: any replacement methods R_i in S must be behavioural subtypes of the corresponding methods M_i in T that they replace.
- Where a method R is intended to be a behavioural subtype replacement for a method M, it must obey the precondition weakening rule: R may weaken M's precondition, but never strengthen it; and the postcondition strengthening rule: R may strengthen M's postcondition, but never weaken it.

In Eiffel [6], strengthening is typically obtained by adding extra assertions in conjunction (combined with logical AND) with the existing set, making the constraint harder to satisfy. Weakening is obtained by providing alternative assertions in disjunction (combined with logical OR) with the existing set, making the constraint easier to satisfy.

3 SMALLTALK AND OBJECTIVE C

Smalltalk [7] is an interesting language to evaluate against these rules, since it is considered by some to be an *untyped* language. From a schema-based perspective (the first column in figure 1), everything is implemented as a uniform object type. Further than this, the static types of variables are not given, so they cannot be checked at all. However, it is reasonable to think of objects at runtime as having a type, corresponding to their class identifier. This type is used implicitly during method lookup to select from method dispatch tables. Type checking is dynamic, in the sense that type errors only appear as "message not understood" exceptions at runtime, after a search for a given method in the class hierarchy has failed. Smalltalk is therefore usually considered to be weakly type checked, since it cannot detect incorrect invocations at compile time.

Fundamentally, Smalltalk expects a subclass to add to the methods of its superclass, which follows the extension rule, and method overriding is supported, on a name-equivalence basis. A weakness is where a replacement method sometimes *derails* the operation of the original version: for example, though all *Collection*s may *add:* elements, a *FixedSizeCollection* may not, so it redefines the *add:* method to raise an exception. This is tantamount to removing a method in a subclass, which violates the extension rule. Smalltalk's method overriding rule is slightly stronger than it first appears, due to the syntactic checking that is carried out upon the distinctive infix message syntax. For example, in the Smalltalk expression:

```
myArray at: 3 put: 42.
```

the *at:put:* message requires exactly two arguments, one inserted after each colon, though the precise types of these arguments cannot be specified. If this method were overridden, the replacement method would have exactly the same name and so it would expect exactly the same number of arguments. This allows us to assert that Smalltalk has a rudimentary interface check in its method overriding. However, we cannot say that the arity of methods is statically checked against invocations, since a message with fewer arguments would simply be considered a different method instead.

Objective C [8] allows the programmer to mix plain C code, which is statically checked according to the rules of C, with dynamically checked object message expressions, written in the same style as Smalltalk. By default, all objects have the static type *id*, the base type of all object references. A variation on this is where the programmer may assert that an object is of a more specific object type than *id*. This does not affect the dynamic binding of methods, but does allow a compiler to check whether a method of a given name exists for that type. Given the contrasting declarations:

```
id squareOne;
Shape * squareTwo;
```

messages sent to *squareOne* will not be statically checked, but messages sent to *squareTwo* can be checked to see if they are defined for objects of at least the type *Shape** (pointer to *Shape*). A further feature of Objective C is the ability to attach type *protocols* (an early foreshadowing of Java interfaces) to any class, independent of its position in the class hierarchy. Variables bearing a protocol-type may be checked in a similar fashion. Smalltalk and Objective C ultimately have weak syntactic type checking. It is possible in either language to write expressions which compile, but fail at runtime due to type-related errors.

4 C++ AND JAVA

Two languages which come closer to following the syntactic rules of subtyping are C++ [9] and Java [10]. Variables are strongly typed in both languages, such that all expressions may be checked against the declared types of methods. By default, C++ binds methods statically unless you request dynamic binding (with the *virtual* keyword). Java's methods are dynamically bound, unless marked *final*, in which case compilers may choose to bind them statically, since they will never be overridden.

Like Smalltalk, these languages expect a subclass to add to the methods of a superclass. Although it is still possible to derail methods in C++ and Java, the temptation to do this is much reduced. Smalltalk's dependence on derailment arises from having only a single classification hierarchy in which to factor out all behaviours. As a result, some generic methods are declared which do not strictly apply to every subclass. By contrast, it is possible to apply multiple and overlapping classification schemes in C++ using multiple inheritance; and in Java using interfaces. In any case, the C++ programming culture tends to avoid large, monolithic class hierarchies.

A different threat to C++ comes through *public, protected* and *private* modes of inheritance. Only in the *public* form of inheritance does a subclass inherit the method interface of its superclass unchanged; in the other two forms, inherited methods become secrets of the subclass. This is equivalent to withdrawing a method in a subclass; however, the C++ compiler recognises that this violates subtyping and correctly disallows aliasing through superclass variables. *Friend*-declarations in C++ are a different matter, which we consider alongside *selective exports* in Eiffel, below.

The method overriding rule in C++ and Java expects a replacement method to have *exactly the same* type signature as the original. This is actually more restrictive than the function subtyping rule requires - more general argument types and more specific result types are allowed in a subtype method, even if accepting more general arguments has limited practical application [3]. Another reason lies behind the choice of this simpler, but stricter rule.

In both languages, method names are not unique within a class, but *overloaded* versions may exist, so long as they can be distinguished by the types of their arguments:

```
void setDate(int, int, int);      // set using day, month, year
void setDate(String);             // set using "dd/mm/yy" string
```

A compiler must be able to resolve the most specific type of an expression to select a unique overload. In C++, this requires a complex series of type conversions. It is difficult to combine this with a mechanism for inferring which of several overloaded versions should be replaced by a more specialised method type (the same replacement might apparently override more than one original version). So, for this pragmatic reason, overriding is restricted to methods having exactly the same type. Some recent C++ compilers allow the returned self-type (the type of *this*, the current object) to be specialised during method overriding, since overloading is resolved using argument types alone.

In terms of the syntactic rules of subtyping, Java and C++ are fairly secure. The weakness in C++ comes from the ease with which a programmer can override the type system and convert one type into another, even when this is not suitable. C++ allows both explicit and implicit type coercions between its numeric, character and boolean types, often losing information in the process (eg a *long int* converted up to a *float*). The most potentially damaging use of typecasting is where the type information attached to a pointer is thrown away ("casting to *void**") or the pointer is converted arbitrarily into another pointer type. For example:

```
Square* mySquare = (Square*) myShape;
```

is only safe if *myShape* holds an object pointer of at least the type *Square*,* but in practice it could hold any type of pointer. The typecast (*Square**) is not checked in C++, such that a program could continue to run with an unsuitable value in *mySquare*, leading to a system crash. In Java, this kind of type conversion is checked at runtime, rasing an exception if *myShape* does not refer to an object of at least the *Square* type. Modern C++ has tried to address this problem by advising programmers to use the similarly-checked type-conversion operators *static_cast, dynamic_cast* and *const_cast*, instead of simply retyping variables.

C++ promotes the use of both *value* and *reference* types (here, we mean pointers), which has extra implications for the implementation schema. Exact typing (box 1 in figure 1) allows the compiler to reserve the exact amount of storage required for an object, or a pointer. Subtyping (box 2 in figure 1) is subject to different physical constraints for values and pointers. A pointer can be coerced to a supertype without difficulty and the primary data is preserved via one level of indirection, though access is limited to those fields declared in the supertype. A value can be coerced to a supertype, but any additional subclass fields are truncated, since storage is allocated in the variable itself, rather than via one level of indirection. For this reason, no dynamic binding can be applied to value-types.

C++ also has a parametric type mechanism, known as *templates*. Our theory does not yet include a treatment of parametric types. However, C++ does not check template class definitions; instead, the compiler only checks fully *instantiated* templates, which are no different from regular classes. Our existing rules may handle these.

5 EIFFEL AND TRELLIS

Eiffel [6] is one language that can be evaluated against both the syntactic and semantic subtyping rules, since it supports object and method specification using executable assertions. Checking Eiffel's syntactic type system is still a challenge, as it offers three different type mechanisms, known as *conformance*, *constrained genericity* and *type anchoring*:

- *conformance* is the regular class-subclass type compatibility relationship;
- *constrained genericity* is a parametric typing mechanism with class constraints;
- *type anchoring* is an entirely novel mechanism linking the types of variables.

We only have space to consider the first of these mechanisms in the current article; the other two mechanisms are based on insights that will form the basis for a complete re-appraisal of the formal notion of *class* and *classification* in a later article.

Like other languages, Eiffel expects a subclass to add to the methods of its superclasses and possibly redefine some. Early versions of the language ran into problems over selective inheritance, whereby a subclass could withdraw a method that was exported (ie declared public) in a superclass. From version 3.0 this was fixed and the default behaviour is to inherit all export declarations unchanged. However, selective inheritance crept in via the back door with the *undefinition* mechanism, whereby a method's effective implementation could be suspended in a subclass, turning it back into a *deferred* (ie abstract) method. While this appears to obey the letter of the law (the method remains in the subclass's public interface), it breaks the spirit of the law (it is illegal to invoke non-implemented methods) and we cite method derailment as a precedent!

Eiffel's overriding rules are ambitious. Not only can you replace methods with retyped versions, but you can also redefine attribute types in a subclass. Unfortunately, Eiffel's overriding rules are faulty from the viewpoint of subtyping. This was the infamous "Eiffel type failure" headline in 1989 [3]. Eiffel assumes that *everything* may be uniformly specialised in a subclass: attributes, method arguments and method results.

This rubs up against strict subtyping in two places. Firstly, it is incorrect to specialise method arguments in an overriding method. This violates the contravariant requirement for argument-types, which asserts that these can only be more general in a subtype method [1]. Secondly, the specialisation of attributes only works for attribute access. At some point, an attribute must also be initialised by assignment. The assignment operation may formally be considered to have the type signature: assign : $\tau \rightarrow$ void, where τ is the type of the value being assigned to the attribute. It is clear that contravariance is violated if a more specialised value of type σ is assigned: assign : σ → void. Figure 2 shows how breaking contravariance may potentially lead to a runtime crash (this example uses Eiffel 5.0 syntax):

```
class POINT
create  make                        -- declare initialiser-method for a point
feature {ANY}                       -- public read-only access to attributes and methods
        x, y : INTEGER;             -- coordinates of a point, initially 0 by default
        make(nx, ny : INTEGER) is do x := nx; y := ny end;
        equal(other : POINT) : BOOLEAN is
                do Result := (x = other.x and y = other.y) end
end -- POINT

class HOTPOINT inherit POINT
        redefine equal              -- with unsafe covariant argument specialisation
feature {ANY}
        on : BOOLEAN;               -- currently selected, initially false by default
        toggle is do on := not on end;
        equal(other : HOTPOINT) : BOOLEAN is
                do Result  := (x = other.x and y = other.y and on = other.on) end
end -- HOTPOINT

genpt, point : POINT;               -- declarations
hotpt : HOTPOINT;
same : BOOLEAN;

create point.make(3, 5);            -- create standard point at (3, 5)
create hotpt.make(3, 5);            -- create hotpt at (3, 5) with on = false by default
genpt := hotpt;                     -- alias hotpt through genpt variable
same := genpt.equal(point);         -- invoke hotpt's equal, with only a point arg!!
```

Figure 2: Eiffel Covariant Type Failure Example

The salient issue is that *equal : POINT → BOOLEAN* is replaced by *equal : HOTPOINT → BOOLEAN*, incorrectly specialising the argument type. When the replacement method is invoked through the general variable *genpt : POINT*, statically it appears to be safe to supply *point : POINT* as its argument. However, when *HOTPOINT*'s *equal* method executes by dynamic binding, it tries to access the non-existent *on* attribute of this plain POINT argument. If unchecked, this will cause a memory segmentation fault.

Eiffel's designer eventually decided to fix both of these problems in a non-standard way [11]. Rather than change the type rules to obey strict subtyping, covariant argument-type redefinitions are flagged, such that unsafe combinations of aliasing and polymorphic invocation are detected immediately. The same technique is used to trap the polymorphic invocation of methods which have been suspended in descendent classes (both are known as polymorphic CAT-calls, standing for *change* in *availability* or *type*). Technically, this solution works, although mathematically it does not address the basic soundess issue.

By contrast, Eiffel's semantic redefintion rules follow *exactly* the semantic subtyping requirements as stated in section 2 above. This means there is an inconsistency between Eiffel's syntactic and semantic redefinition rules: the conflict is over redefined method arguments, between the precondition weakening rule and the (incorrect) argument specialisation rule. One accepts more, the other less. Paradoxically, the correct semantic rules were derived by thinking about contracts between client and supplier objects. The language Trellis [4] applies the same thinking to its syntactic type rules, in which redefined method arguments may only become more general. Trellis is the only language in our survey whose syntactic type rules follow exactly the subtyping requirements in section 2.

A different type-related issue is raised by Eiffel's *selective export* mechanism. A class may export separate lists of features to ANY (public visibility), to NONE (protected visibility) or to an arbitrary set of client classes. This has the curious consequence that the public interface of a class may appear to change, depending on who its client is! Types are no longer fixed, but are chameleon-like interfaces that change colour according to context. C++ raises similar issues with its *friend* declarations. Arbitrary functions, or whole classes, may be declared a *friend* of another class, in which case the friends have total freedom of access. So, the public interface of a class may appear different to its friends than to other clients. Friendship declarations are not inherited. So, aliasing an object through a superclass variable offering extra friendship privileges may break its intended encapsulation. Eiffel's selective exports are inherited unchanged, which is better.

6 CONCLUSION

The subtyping inquisition has bared the type systems of several popular object-oriented languages and found most of them guilty of violating subtyping in one way or another. Syntactically sound subtyping is exhibited by Trellis and Java, although Java is less flexible in its redefinition rule. C++ would be as good as Java, were it not for unchecked typecasting. Eiffel is retrospectively type-safe, due to the polymorphic CAT-call rule, even though it strictly violates soundness. None of the surveyed languages apart from Eiffel seriously promote verifying the behaviour of a class. Where full use is made of Eiffel's assertion mechanism, then a subclass may be shown to conform to the behaviour of its parent class. However, in all these languages, it is still possible to redefine methods to execute in arbitrary ways, resulting in unpredictable behaviour in substitutable components.

The fact that these faults do not give rise to system crashes more often than they do is explained mostly by the fact that programmers strive to write code in a consistent way, adopting style guidelines over and above what the type systems are capable of checking. It typically takes more than one unusual circumstance to trigger a type-related fault - for example, the Eiffel type-failure examples [3] were manufactured retrospectively by theoreticians, working backwards from the formal rules of subtyping. Up until that point, no system failures had been reported as being due to this particular fault. In practice, you are less likely to want to compare a HOTPOINT with a more general POINT than you are with another object of the same type. So, how is it that subtyping cannot express this? Such will be the focus of the next article in the series.

REFERENCES

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

[2] A J H Simons. "The theory of classification, part 5: Axioms, assertions and subtyping", *Journal of Object Technology,vol. 2, no. 1, January-February 2003*, pp. 13-21. http://www.jot.fm/issues/issue_2003_01/column2

[3] W Cook. "A proposal for making Eiffel type safe", *Proc. 3rd European Conf. Object-Oriented Prog.*, 1989, 57-70; reprinted in *Computer Journal 32(4),* 1989, 305-311.

[4] C Schaffert, T Cooper, B Bullis, M Kilian and C Wilpolt. "An introduction to Trellis/Owl", *Proc. 1st ACM Conf. Object-Oriented Prog. Sys., Lang. and Appl.*, pub. *ACM Sigplan Notices, 21(11),* 1986, 9-16.

[5] W Cook and J Palsberg. "A denotational semantics of inheritance and its correctness", *Proc. 4th ACM Conf. Object-Oriented Prog. Sys., Lang. and Appl.,* pub. *ACM Sigplan Notices, 24(10),* 1989, 433-443.

[6] B Meyer. *Object-Oriented Software Construction*, 2nd edn., Prentice Hall, 1995.

[7] A Goldberg and D Robson. *Smalltalk-80: The Language and its Implementation,* Addison Wesley, 1983.

[8] B J Cox and A J Novobilski. *Object-Oriented Programming: an Evolutionary Approach,* 2nd edn., Addison Wesley, 1991.

[9] B Stroustrup. *The C++ Programming Language*, 3rd edn., Addison Wesley, 1997.

[10] C S Horstmann and G Cornell. *Core Java 2, Volume 1 - Fundamentals*, Sun Microsystems Press, 2003.

[11] B Meyer. "Beware polymorphic cat-calls", *Eiffel forum at 18th Conf. Tech. Object-Oriented Lang. and Sys. (TOOLS Pacific),* Melbourne, 1995; also available through: http://archive.eiffel.com/doc/manuals/technology/typing/cat.html.

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.

JOURNAL OF OBJECT TECHNOLOGY

Online at www.jot.fm. Published by ETH Zurich, Chair of Software Engineering ©**JOT, 2003**

Vol. 2, No. 3, May-June 2003

The Theory of Classification Part 7: A Class is a Type Family

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

1 INTRODUCTION

This is the seventh article in a regular series on object-oriented type theory, aimed specifically at non-theoreticians. So far, we have built up a model of objects as simple records, which are instances of corresponding record types [1]. Initially, we took the seemingly attractive view that a programmer's *class* in C++ or Java corresponds in some way to a *type* in the formal model, and that a compatible *subclass* therefore corresponds to a *subtype* [2], linking this with an algebraic description of subtype-compatible semantic behaviour [3]. Naturally, the *class*-construct in an object-oriented language also defines an implementation for the instances of the class, but there is no formal contradiction here in considering just the typeful aspects of class declarations. Subtyping does indeed provide a simple, flexible model for type compatibility, but we shall find that it is not always as useful as we might expect.

We shall discover circumstances in which a type system based on subtyping breaks down, providing less than useful information. Object-oriented languages like Smalltalk and Eiffel exhibit sophisticated, systematic kinds of behaviour which cannot adequately be described in terms of types and subtyping. By appealing to natural notions of classification in biology, we shall demonstrate the extent to which the subtyping model fails to capture the intuitive notion of a *class*. In this article, we shall define the notion of *class* formally, and prove that it is more than just a type. To understand this, we will need to extend our formal model to include type polymorphism. This requires the *second-order* λ-calculus and notions of *universal* [4, 5] and *function-bounded quantification* [6, 7].

2 THE PROBLEM OF RECURSIVE CLOSURE

There used to be a popular rhyming couplet that joked about the terminology used in the biological classification of the animal kingdom:

Cats have kittens, dogs have puppies,
But guppies just have baby guppies.

While every species has young of its own kind, the biologists seemed to have left some holes in the taxonomy, surely an oversight! What is more disturbing is that most object-oriented languages will assert that Dog, Cat and Guppy, by virtue of being kinds of Animal, all mate with an Animal and produce an offspring, which is (you guessed) an Animal, something forced upon recursive types by the rules of subtyping. To see why, let us express the reproducing behaviour of all animals (ignoring litters of offspring) formally as:

$$\text{Animal} = \mu\sigma.\{..., \text{mate} : \sigma \rightarrow \sigma, ...\}$$

This defines Animal as a recursive record type whose methods (most of which are not shown) include the *mate* method. In such a recursive definition, σ is the self-type, a placeholder for the eventual type name, bound recursively using μ to refer to the whole record (the earlier articles [1, 2] explain this notation). Once the recursion is established, we can access the type of the *mate* method as:

$$\text{Animal.mate} : \text{Animal} \rightarrow \text{Animal}$$

showing that an Animal mates with an Animal and produces an Animal offspring, suitably capturing the general notion. Intuitively, we should like to introduce the Animal subclasses Dog, Cat (and even Guppy), such that these creatures all mate with, and produce young of their own kind, in the uniformly specialised style:

$$\text{Dog.mate} : \text{Dog} \rightarrow \text{Dog}$$
$$\text{Cat.mate} : \text{Cat} \rightarrow \text{Cat}$$

We might expect Cat and Dog to be subtypes of Animal; however this is not the case, since they both redefine the signature of the *mate* method a way that violates subtyping. The function subtyping rule allows a subtype function to have more general arguments and a more specific result [2]. Here, the Dog type replaces the signature Animal $\rightarrow$ Animal with the retyped signature Dog $\rightarrow$ Dog, which unhelpfully specialises both argument and result. The best we could do while still preserving subtyping is to break with uniform specialisation and invent strangely retyped versions of *mate* which still accept Animal arguments:

$$\text{Dog} = \mu\sigma.\{..., \text{mate} : \text{Animal} \rightarrow \sigma, ...\}$$
$$\text{Dog.mate} : \text{Animal} \rightarrow \text{Dog}$$

This ensures that Dogs produce puppies, but still allows a Dog to mate with any kind of Animal, which seems intuitively wrong, but is formally correct by the rules of subtyping. The Animal type declared that its *mate* method always accepts an argument of at least the Animal type and we cannot go back on this, particularly if we expect to invoke *Dog.mate* dynamically through an Animal variable and supply any legal Animal argument.

In general, recursion interacts poorly with subtyping. If any type T has a method that is *closed* over its own type: $T.m : T \rightarrow T$, then it is impossible to specialise the type of this method uniformly. Let us assume that such a type S existed, with a method $S.m : S \rightarrow S$. If we want to establish a subtype relationship $S <: T$, then for the replaced method m to be a subtype, we have to show, on the result-side, that $S <: T$, which is consistent, and also show, on the argument-side, that $T <: S$, which is precisely the opposite of the relationship we seek. The only condition under which both $S <: T$ and $T <: S$ is if $S = T$!

This is one reason why redefined methods cannot change their types in languages with both type recursion and subtyping. Consider that in Java, the *equals* method is recursively typed at the root: Object.equals : Object $\rightarrow$ Boolean, making every Java type recursive. Recursion fixes the type of this method, which can never be specialised. This leads to a damaging lack of expressiveness: any redefinition of *equals* must still accept a generic Object argument, yet it is usually a semantic error to seek to equate a Cat with anything other than another Cat. In practice, programmers compare like with like, but they can only do this by *type downcasting* the argument of the *equals* method, forcibly overriding the natural type system.

3 THE PROBLEM OF TYPE LOSS

A different but related problem arises when recursively-typed methods are inherited and invoked in a subtype object. So far, we have not modelled the notion of inheritance in any detail, but let us invent a simple rule to create subtypes by record extension [2]. Since records are merely maps from labels to values, and maps are really just sets, we can combine records using set union. Let us assume that we wish to define a hierarchy of numeric types, and that the basic Number type provides a primitive notion of addition:

$$\begin{aligned}\text{Number} &= \mu\sigma.\{\text{plus} : \sigma \rightarrow \sigma\},\\ &= \{\text{plus} : \text{Number} \rightarrow \text{Number}\}\end{aligned}$$

after unrolling the recursion. We can seek to derive other numeric types by extending this, yielding for example the Natural, Integer, Real and Complex numbers. In particular, the Integer type offers a full range of arithmetical methods:

$$\text{Integer} = \mu\sigma.(\text{Number} \cup \{\text{minus} : \sigma \rightarrow \sigma, \text{times} : \sigma \rightarrow \sigma, \text{divide} : \sigma \rightarrow \sigma\})$$

This defines the Integer type by extending the Number type with a record of additional fields, and then fixing the recursion. After unrolling Number to yield the corresponding record type, we can compute the union of fields, yielding the recursive record type:

$$\begin{aligned}\text{Integer} = \mu\sigma.\{&\text{plus} : \text{Number} \rightarrow \text{Number}, \text{minus} : \sigma \rightarrow \sigma,\\ &\text{times} : \sigma \rightarrow \sigma, \text{divide} : \sigma \rightarrow \sigma\},\\ = \{&\text{plus} : \text{Number} \rightarrow \text{Number}, \text{minus} : \text{Integer} \rightarrow \text{Integer},\\ &\text{times} : \text{Integer} \rightarrow \text{Integer}, \text{divide} : \text{Integer} \rightarrow \text{Integer}\}\end{aligned}$$

after unrolling the Integer recursion. Curiously, while the locally declared methods are all typed in terms of Integer, the inherited *plus* method is already fixed in terms of Number. As a consequence, the intuitively reasonable arithmetic expression:

```
i, j, k : Integer
i.plus(j).minus(k)
```

fails even to typecheck! This is because the sub-expression *i.plus(j)* returns a result of the general type Number, for which the *minus* method is not defined. This is the problem of type-loss under inheritance, something with which Java and C++ programmers will be familiar. It arises because the type recursion in Number fixes the signature of the *plus* method, and this more general type is still retained in Integer, after the union of fields. In practice, Java and C++ programmers have to use *type downcasting* to override the natural type system, if they wish expressions like this to compile, something like:

```
i, j, k : Integer
((Integer) i.plus(j)).minus(k)
```

Type downcasting is typically considered a last resort, a dirty trick to be used on occasions when the natural type system doesn't help. Here, we have shown how type downcasting has to be used systematically all the time to overcome deficiencies in the type system. This is a strong indicator that subtyping is not the most appropriate formal model for object-oriented languages. Instead, we need a more expressive type system.

4 QUANTIFICATION OVER TYPES

Working backwards from the desired goal, it seems that our intuitive notion of classification requires a type system in which recursive types can have methods that are closed over their own type, but which are nonetheless related to each other in some systematic way. We want to be able to support *families* of related types that behave in similar ways, such as the numeric types which all provide addition:

$$\begin{array}{l}\text{Integer.plus : Integer} \rightarrow \text{Integer}\\ \text{Complex.plus : Complex} \rightarrow \text{Complex}\\ \text{Natural.plus : Natural} \rightarrow \text{Natural}\end{array}$$

and somehow be able to assert that these all belong to the *class of numbers*. There is clearly a systematic pattern here, in which all related numeric types τ have a *plus* method with the type signature $\tau.\text{plus} : \tau \rightarrow \tau$. We can get close to this idea with universal quantification:

$$\forall\tau \,.\, \tau.\text{plus} : \tau \rightarrow \tau$$

which says that "all types τ have a method *plus* which accepts and returns a value of the same type τ." This is still not quite right, since we want *plus* to be defined only for the numeric types, not for absolutely every type.

Universal quantification was adopted independently by Girard [4] and Reynolds [5] as a way of introducing *type parameters*, variables which range over types (ie which receive types as their bound value). They are used in the *second-order* λ-calculus, in which functions can accept both value- and type-arguments. This was found useful to express the notion of *polymorphism*, describing functions that work uniformly on families of types. *Parametric polymorphism* was built into the early functional programming languages ML and Hope, and exists in object-oriented languages as the *templates* in C++, or *generic types* in Ada and Eiffel. Before we can understand the usefulness of type parameters in modelling polymorphism, we need to explain the idea of different *orders* of calculus, and understand something of how to construct and simplify expressions.

5 SIMPLY-TYPED AND POLYMORPHIC CALCULUS

A zero-order system has no variables, but only sets of values. A first-order system has functions, whose bound variables range over simple values. A second-order system has functions, whose bound variables range over both values and simple types. The λ-calculus is a basic functional language. In its primitive form, it is *untyped*, so we cannot yet say anything about its order. About the simplest function you can write in the untyped λ-calculus is the *identity* function, which accepts an argument and returns it unchanged:

$\lambda x.x$	the untyped identity function
$\lambda x.x\ \ 3 \Rightarrow 3$	apply identity to an Integer
$\lambda x.x\ \ \text{'a'} \Rightarrow \text{'a'}$	apply identity to a Character

Recall that "λx" means "a function of x" and everything after the dot is the function body, here just "x". Placing the function next to a value *applies* the function to this value, rather like calling a function in a programming language. Applying λx.x to the integer 3 simply binds the argument x←3 then returns the body x, which has the substituted value 3 (see also [1]). In the untyped calculus, we can apply λx.x to anything, such as integers, characters or even other functions (in which case we would have a *higher-order* system).

We may attach simple types to the function's arguments in the *simply-typed* λ-calculus. This is a first-order system, since simply-typed variables can only range over basic values. If we so wish, we can restrict the identity function to accept only Integer values:

$\lambda(x{:}\text{Integer}).x$	a typed identity function
$\lambda(x{:}\text{Integer}).x\ \ 3{:}\text{Integer} \Rightarrow 3$	type-safe application
$\lambda(x{:}\text{Integer}).x\ \ \text{'a'}{:}\text{Character} \Rightarrow \text{error}$	type-incorrect application

The difference here is that the types of the formal argument and the actual value must match, otherwise the application is deemed illegal, a type error. We can say that this identity is a *monomorphic* function, since it is defined only for a single type, Integer. We say that identity has the type: Integer → Integer.

In the *second-order* λ-calculus, functions have extra arguments standing for types, which are introduced ahead of the arguments standing for values of these types. A *polymorphic* version of identity is given by the following, in which τ is a *type parameter:*

$\lambda\tau.\lambda(x:\tau).x$	a polymorphic identity function
$\lambda\tau.\lambda(x:\tau).x \text{ Integer} \Rightarrow \lambda(x:\text{Integer}).x$	instantiation with Integer
$\lambda\tau.\lambda(x:\tau).x \text{ Character} \Rightarrow \lambda(x:\text{Character}).x$	instantiation with Character
$\lambda\tau.\lambda(x:\tau).x \text{ Integer } 3:\text{Integer} \Rightarrow 3$	instantiation and application
$\lambda\tau.\lambda(x:\tau).x \text{ Character 'a':Character} \Rightarrow \text{'a'}$	instantiation and application

Identity now expects a type argument: λτ and then a value of this type: λ(x:τ). If we apply identity just to a type, such as Integer, then we bind τ←Integer and return the body, which after the type substitution is the simply-typed version of the function. This models the notion of *type parameter instantiation* in C++ or Eiffel, in which type parameters are replaced by actual types. We may apply the resulting simply-typed function to a value of the expected type, as before. Second-order functions expect to be applied to a type, then to a value in that order. We say that polymorphic identity has the type: $\forall\tau.\tau \to \tau$, since it applies to any type and returns a result of the same type.

6 GENERIC OBJECT TYPES

This kind of construction can be used to extend our formal model of object types and allows us to define polymorphic types (generic, or templated types). Let us start with a *monomorphic* recursive record type for an IntegerStack:

$$\begin{aligned}\text{IntegerStack} = \mu\sigma.\{&\text{push} : \text{Integer} \to \sigma,\ \text{pop} : \to \sigma,\ \text{top} : \to \text{Integer},\\ &\text{empty} : \to \text{Boolean},\ \text{size} : \to \text{Integer}\}\end{aligned}$$

As before, σ is a recursive placeholder for the eventual IntegerStack. We may modify this definition to create a *polymorphic* type if we replace occurrences of Integer by a type parameter. We must introduce the parameter at the head of the type definition:

$$\begin{aligned}\text{Stack} = \lambda\tau.\mu\sigma.\{&\text{push} : \tau \to \sigma,\ \text{pop} : \to \sigma,\ \text{top} : \to \tau,\\ &\text{empty} : \to \text{Boolean},\ \text{size} : \to \text{Integer}\}\end{aligned}$$

Here, λτ introduces the parameter τ standing for the element-type, ahead of μσ, which binds the recursion in the rest of the record. This Stack definition now has the form of a *type function*, that is, a function which expects a type argument: τ and then returns a result, a record type in which τ will be bound to some actual type. To see how this works, we can apply Stack to the Integer type (*ie* call Stack with Integer as its actual type argument):

$$\begin{aligned}\text{Stack[Integer]} = \mu\sigma.\{&\text{push} : \text{Integer} \to \sigma,\ \text{pop} : \to \sigma,\ \text{top} : \to \text{Integer},\\ &\text{empty} : \to \text{Boolean},\ \text{size} : \to \text{Integer}\}\end{aligned}$$

and the result we obtain is identical to the IntegerStack type from above, after substituting {Integer/τ} in the record body. This is interesting, because we may apply Stack to any type we fancy, such as Stack[Character] or Stack[Boolean], creating specific instantiations of the polymorphic type. There is no restriction on the actual element type we could supply, so this kind of polymorphism is sometimes known as *universal polymorphism*. We can express the polymorphic types of individual methods using universal quantification:

$$\forall\tau \,.\, \text{Stack}[\tau].\text{push} : \tau \rightarrow \text{Stack}[\tau]$$
$$\forall\tau \,.\, \text{Stack}[\tau].\text{pop} : \rightarrow \text{Stack}[\tau]$$
$$\forall\tau \,.\, \text{Stack}[\tau].\text{top} : \rightarrow \tau$$

which we read as: "for all types τ, a Stack of τ has a *push* method that accepts an argument of the type τ, and returns a Stack of τ", and similarly for the other methods. Clearly, the generic Stack type expresses something about a family of related Stack-types, but this is still not the notion of *class* that we are seeking to capture.

7 FUNCTION-BOUNDED QUANTIFICATION

It was Cook [6, 7] who first realised that in order to model a class as a polymorphic family of related types, the key lay in making the self-type flexible, so that it could refer to a different actual type in every member of the type family. In an earlier article [1] we introduced *type generators* for recursive types, in which the self-type σ is a parameter and is not yet bound. Type generators are similar to our *type functions* for generic types, except that the self-type parameter σ eventually stands for the whole type, not for a part of it.

Type generators can be used, exactly like type functions above, to create different instantiated versions of a parameterised record type. To see how this works, we revisit the Number type, but this time express it as a type generator, in which the self-type is not recursively fixed, but is a parameter introduced by $\lambda\sigma$:

$$\text{GenNumber} = \lambda\sigma.\{\text{plus} : \sigma \rightarrow \sigma\}$$

GenNumber is a generator for a family of related record types which have the general structure of numbers with a plus method. To show this, we can apply GenNumber to other numeric types, and this has the effect of adapting the self-type σ, which is substituted by whatever type-argument we supply:

$$\text{GenNumber[Integer]} = \{\text{plus} : \text{Integer} \rightarrow \text{Integer}\}$$
$$\text{GenNumber[Real]} = \{\text{plus} : \text{Real} \rightarrow \text{Real}\}$$
$$\text{GenNumber[Complex]} = \{\text{plus} : \text{Complex} \rightarrow \text{Complex}\}$$

This looks promising, in that we are able to construct record types with a *plus* method that is uniformly specialised to specific numeric types. We shall use this adaptive ability below.

The ideal type for an Integer is a recursive type whose methods are all closed over its own type, which we can unroll to a record type expressed in terms of Integers:

$$\begin{aligned}\text{Integer} &= \mu\sigma.\{\text{plus} : \sigma \to \sigma,\ \text{minus} : \sigma \to \sigma,\ \text{times} : \sigma \to \sigma,\ \text{divide} : \sigma \to \sigma\},\\ &= \{\text{plus} : \text{Integer} \to \text{Integer},\ \text{minus} : \text{Integer} \to \text{Integer},\\ &\qquad \text{times} : \text{Integer} \to \text{Integer},\ \text{divide} : \text{Integer} \to \text{Integer}\}\end{aligned}$$

Although this type can never be a subtype of the recursive Number *type* from section 3 above (because it uniformly specialises plus : Number → Number to plus : Integer → Integer), it is nonetheless a subtype of a *specially adapted* GenNumber *generator*, that is:

$$\text{Integer} <: \text{GenNumber}[\text{Integer}],$$

which we can demonstrate by unrolling Integer to a record (on the left-hand side) and then evaluating GenNumber[Integer] (on the right-hand side) and comparing the two records:

$$\begin{aligned}&\{\text{plus} : \text{Integer} \to \text{Integer},\ \text{minus} : \text{Integer} \to \text{Integer},\\ &\qquad \text{times} : \text{Integer} \to \text{Integer},\ \text{divide} : \text{Integer} \to \text{Integer}\}\\ &<: \quad \{\text{plus} : \text{Integer} \to \text{Integer}\}\end{aligned}$$

This satisfies the record subtyping rule [2]. The left-hand side contains more fields than the right-hand side, a simple case of record extension. It turns out that all the other numeric types (with more methods than Number) can be shown to enter into a similar relationship with a suitably-adapted version of the generator, for example:

$$\begin{aligned}&\text{Real} <: \text{GenNumber}[\text{Real}]\\ &\text{Complex} <: \text{GenNumber}[\text{Complex}]\end{aligned}$$

and it follows intuitively that any type τ satisfying: $\tau <: \text{GenNumber}[\tau]$ belongs to the family of numeric types which share at least the plus-method. From this, Cook realised that a class is a polymorphic family of types that satisfy a constraint, or *bound* [6, 7], expressed using a generator function. Whereas universal quantification introduces type parameters that range over any type, Cook's *function-bounded quantification* introduces type parameters that only range over a restricted group of types which satisfy the constraint. The whole *class of numbers* can be expressed formally as the type family:

$$\forall(\tau <: \text{GenNumber}[\tau])$$

meaning "all those types which are subtyes of the adapted GenNumber generator". What is unusual about this special kind of quantification is that the parameter τ appears on both sides of the <: subtyping constraint; but it turns out that this is exactly what is necessary to express the notion of a family of recursively closed types that have a shared minimum structure.

A suitable polymorphic type for the plus method may now be given, by restricting the family of types to those in the class of numbers:

$$\forall(\tau <: \mathrm{GenNumber}[\tau])\ .\ \tau.\mathrm{plus} : \tau \rightarrow \tau$$

The constraint $\tau <: \mathrm{GenNumber}[\tau]$ is known as a *function bound*, or *F-bound* for short. F-bounded quantification was a revolutionary discovery, because it captured exactly the kind of polymorphism present in object-oriented languages, in which methods apply to families of types sharing a minimum common structure.

8 CONCLUSION

We have formally defined the notion of class, using Cook's F-bounded quantification to express the idea that a class is *a family of types that share a minimum common structure.* This is a radical departure from the earlier view that a programmer's class corresponds to a simple type. Although the languages Java and C++ adopt this simpler view, we found that there were sufficient reasons to challenge this view, particularly the evidence from the frequent use of type downcasting needed to overcome inadequacies of first-order type systems based on types and subtyping. We are moving towards a second-order type system, in which a programmer's class really corresponds to a polymorphic type. We showed how polymorphism is modelled systematically using type parameters, and explained the relationship between universal quantification, which supports the definition of generic types, and F-bounded quantification, which supports the definition of classes:

$$\mathrm{GenAnimal} = \lambda\sigma.\{\mathrm{mate} : \sigma \rightarrow \sigma\}$$
$$\forall(\tau <: \mathrm{GenAnimal}[\tau])\ .\ \tau.\mathrm{mate} : \tau \rightarrow \tau$$

and even satisfies natural intuitions about biological classification in which animals reproduce their own kind. Mathematics, as someone once said, is pure poetry.

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-Decembe 2002, pp. 27-35. http://www.jot.fm/issues/issue_2002_11/column2

[3] A J H Simons, "The theory of classification, part 5: Axioms, assertions and subtyping", in *Journal of Object Technology*, vol. 2, no. 1, January-February, pp. 13-21. http://www.jot.fm/issues/issue_2003_01/column2

[4] J-Y Girard, “Interpretation fonctionelle et elimination des coupures de l'arithmetique d'ordre superieur”, *PhD Thesis,* Universite Paris VII, 1972.

[5] J Reynolds, “Towards a theory of type structure”, *Proc. Coll. Prog., New York, LNCS 19* (Springer Verlag, 1974), 408-425.

[6] W Cook, “A denotational semantics of inheritance”, *PhD Thesis,* Brown University, 1989.

[7] 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

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.

JOURNAL OF OBJECT TECHNOLOGY

Online at www.jot.fm. Published by ETH Zurich, Chair of Software Engineering ©**JOT, 2003**

Vol. 2, No. 4, July-August 2003

The Theory of Classification Part 8: Classification and Inheritance

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

1 INTRODUCTION

This is the eighth article in a regular series on object-oriented type theory, aimed specifically at non-theoreticians. In earlier articles, we explored the view that a programmer's *class* in C++ or Java corresponds in some way to a *type* in the formal model, and that a compatible *subclass* therefore corresponds to a *subtype* [1, 2]. In the last article, simple subtyping was found to be inadequate to express systematic relationships between recursive types [3]. Instead, we found that a second-order model with type parameters was needed. In this model, a programmer's class corresponds to a *bounded polymorphic type*, representing a family of similar types which share a minimum common structure and behaviour. The family likeness was expressed using a constraint, known as a *function bound* or *F-bound* [4] on the type parameter, which ensured that the parameter could only be replaced by types having at least the structure and behaviour specified in the *F-bound*.

This opens up a completely different formal notion of *class*, and consequently of *inheritance*. Whereas before, we thought of a class as a type, clearly it is now the pattern for a family of related types. Likewise, whereas inheritance was formerly the simple extension of a type, in the new model it is the extension of a general pattern. In this article, we explore further the differences between classes and types, developing the alternative formal model of classification and inheritance, which is quite different from subtyping [5].

2 SPECIFICATION VERSUS IMPLEMENTATION

So far in the *Theory of Classification*, I have been seeking to show how the intuitive notion of *class* in object-oriented languages has a *strictly formal* interpretation that is more general than the simple notion of *type*. At this point, I usually come up against a long-held prejudice among practically-minded programmers that the "real" difference between a class and a type is that a class is merely a programming language construct, whereas a type is the formal description of this. In other words, there is out there the entrenched view that "type = specification" and "class = implementation". In the following, I hope to show that this view is a red herring in our thinking about classification in object-oriented languages.

In the early days, we struggled to understand the formal nature of novel object-oriented language features, especially inheritance, which could sometimes be used in a strict way, to derive a family of related types, and sometimes in an opportunistic way, to extend implementations [6]. This led some to believe that objects have *class* and *type* independently [7, 8], asserting that an inheritance hierarchy was merely a convenience for describing shared implementation, whereas a separate type hierarchy was necessary to describe the subtyping relationships between the same objects. In some cases, the "classes" and "types" for the very same objects could be linked in different orders (see figure 1).

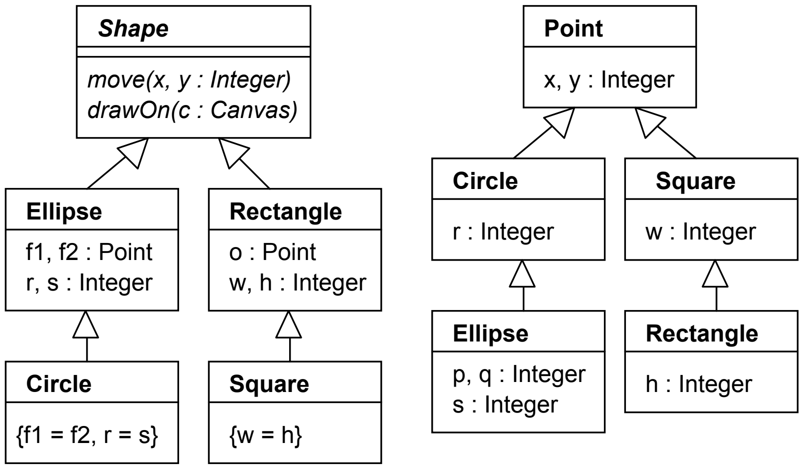


Figure 1: Sharing type (left) and implementation (right)

Now, while this is an interesting issue, it relates to *conceptual design* more than it relates to *type theory* and the notions we have been discussing here. The left-hand hierarchy expresses the conceptual family of Shapes, while the right-hand hierarchy expresses how you might conveniently derive extended records by adding variables, although it leads to conceptual nonsense: a Circle is not a kind of Point, for example. Nonetheless, it is quite possible to define a strange Circle type that is genuinely a subtype of Point - in the theory of subtyping, this would be perfectly legitimate for many definitions of Circle and Point, so long as you obeyed the rules [2]. The real issue here is one of discipline versus opportunism in conceptual design, and is not really related to types and subtyping.

Another blow to the "class = implementation" viewpoint is that programming languages like Pascal and C had types with concrete implementations long before the object-oriented notion of class was popular - nothing *necessarily* forces "class" to mean implementation. In Computer Science, we have always talked in terms of *abstract* and *concrete* types. Abstract types are formal, described in terms of operation signatures and axioms; concrete types have a representation in a programming language. So, why not extend this notion to classes, which can also be both abstract (formal and typeful) and concrete (practical and implemented)?

Modern object-oriented languages, like C++ and Eiffel, link the type hierarchy directly to the implementation hierarchy. Some allow further expression of type compatibility apart from the main implementation hierarchy, like Objective C and Java, which have separate interfaces to express common type relationships that cut across the main divisions in the class hierarchy. Earlier languages like POOL-T [8] developed completely independent implementation- and type-hierarchies, in the hope of preserving "pure subtyping" in a language with multiple, variant implementations. However, even the separate subtype hierarchy of POOL-T is defeated by recursive types and inescapably suffers from the same restrictions and lack of expressiveness that we identified previously for subtyping [3].

3 CLOSED VERSUS OPEN

A more satisfying way of distinguishing object-oriented classes from traditional simple types is to realise that a type is closed, in the sense of being complete, whereas a class is open-ended, in the sense of being subject to arbitrary subdivision and further specialisation. In older object-based languages like Modula-2 and Ada (pre-95, before the addition of inheritance), the type of an object is expressed *exactly* by its interface. In later object-oriented languages with polymorphic inheritance, the class of an object is understood to express only the *minimum* interface which members of the class must satisfy. This subtle difference between *exact* and *minimum* interfaces is what characterises the essential difference between a traditional programmer's type and a modern class. Taxonomic classification in biology also follows this pattern: a mammal is defined as something with (at least) warm blood and hair that bears and suckles live young. This is not a complete or finished definition - biologists don't exclaim: "Look, there's an instance of a mammal!", but rather identify dogs, cats or gerbils and show that these belong to the class of mammals, by virtue of having the four essential mammalian properties.

Mathematically, we can capture the same distinction between simple, closed types and open-ended polymorphic classes. In the previous article [3], we showed that a basic *class of Numbers* may be expressed using the F-bound:

$$\forall(\tau <: \text{GenNumber}[\tau]), \qquad \text{where: GenNumber} = \lambda\sigma.\{\text{plus} : \sigma \rightarrow \sigma\}.$$

According to this definition, the parameter τ may range over all kinds of numeric types, so long as they have *at least* a *plus* method with the specified signature. By contrast, the *exact Number type* is a recursive type, created from first principles as the least fixed point of the generator GenNumber (see earlier article [1] for a full explanation):

$$\text{Number} = \mathbf{Y}\,[\text{GenNumber}] = \text{GenNumber}[\text{GenNumber}[\text{GenNumber}[...]]]$$
$$\Rightarrow \{\text{plus} : \text{Number} \to \text{Number}\}, \qquad \text{at the limit of recursion.}$$

This is a very general numeric type *with only a plus* method - we are unlikely to use direct instances of this type (like mammal, above), but instead we will want to use instances of Integer, Real, or Complex.

Note that exact types, like Number, are fixed, whereas classes are open-ended and flexible. If we say n : Number, we assert that n is a variable that can only receive objects of exactly the Number type. On the other hand, if we say x : $\forall(\tau <: \text{GenNumber}[\tau])$, we assert that x is a variable that can receive objects of any numeric type in the class of Numbers, even types having more than the minimum required methods. This is the difference between monomorphism (exact typing) and bounded polymorphism (constrained flexible typing). In the following, *types* are always exact and *classes* are always polymorphic.

4 RELATING CLASSES AND TYPES

There is an interesting relationship between classes and types. It turns out that the exact Number type is the *least type* which is still a member of the polymorphic Number class. In other words, it has just enough fields to satisfy the F-bound constraint:

$$\text{Number} <: \text{GenNumber}[\text{Number}],$$

which unrolls (on the left) and evaluates (on the right) to give:

$$\{\text{plus} : \text{Number} \to \text{Number}\} \;<:\; \{\text{plus} : \text{Number} \to \text{Number}\}$$

from which it is clear that both sides are equal. In fact, for this one case alone, we could rewrite the subtyping condition as a type equivalence:

$$\text{Number} = \text{GenNumber}[\text{Number}],$$

and the reader may recall that this is exactly the same formula that identifies Number as the fixpoint of the generator GenNumber, that is, a type which is unchanged by the application of the generator [1]. From this, we know that there is exactly one type which "only just" satisfies the conditions for class membership - and this is always the recursive type created from the generator that is used to express the F-bound constraint.

We can visualise this in figure 2 by drawing the Number class as a cone-shaped bounded volume, representing a space of possible numeric types that satisfy the F-bound, and the corresponding "least type" Number, which is only just a member of this class, as the point at the apex of this cone. There are many other numeric types which belong to the Number class, which have more than the minimum required methods: figure 2 also shows the Integer type as a point in the space enclosed by the Number cone.

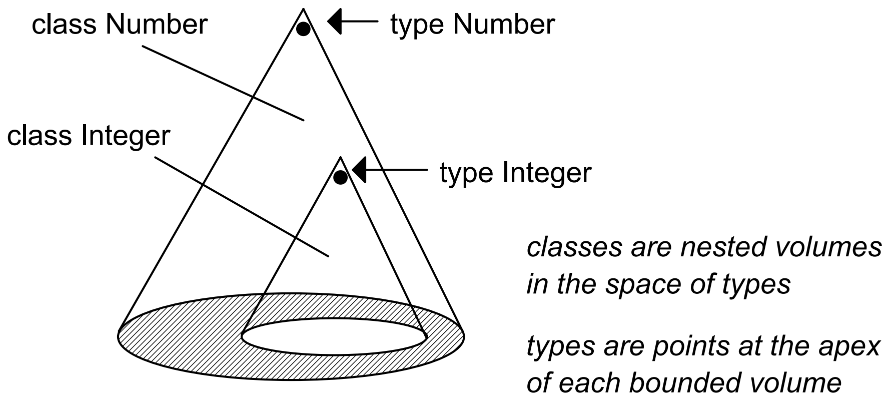

class Number
type Number
class Integer
type Integer
classes are nested volumes in the space of types
types are points at the apex of each bounded volume

Figure 2: Classes as volumes containing types as points

Can we demonstrate this mathematically? Recall that the exact Integer type is given by:

$$\text{Integer} = \mu\sigma.\{\text{plus} : \sigma \to \sigma,\ \text{minus} : \sigma \to \sigma,\ \text{times} : \sigma \to \sigma,\ \text{divide} : \sigma \to \sigma\},$$

$$\Rightarrow \quad \{\text{plus} : \text{Integer} \to \text{Integer},\ \text{minus} : \text{Integer} \to \text{Integer},\ \text{times} : \text{Integer} \to \text{Integer},\ \text{divide} : \text{Integer} \to \text{Integer}\},$$

after unrolling the recursion. We then apply the test for class membership, manipulating the expression on both sides, by unrolling the recursive type (on the left) and applying the type generator (on the right) to yield a comparison between record types:

$$\text{Integer} <: \text{GenNumber[Integer]} \quad \Rightarrow$$

$$\{\text{plus} : \text{Integer} \to \text{Integer},\ \text{minus} : \text{Integer} \to \text{Integer},\ \text{times} : \text{Integer} \to \text{Integer},\ \text{divide} : \text{Integer} \to \text{Integer}\}$$
$$<: \ \{\text{plus} : \text{Integer} \to \text{Integer}\}$$

which is true by the record subtyping rule [2]. We have therefore shown that the Integer type is indeed in the class of Numbers. As well as the least type Number, we may expect many more numeric types like Integer, which have *more than* the minimum required methods, to be members of this class. Such numeric types could include Real, Complex, Fraction and any numeric type with at least a *plus* method. This expresses exactly the object-oriented notion of class membership.

5 A MODEL OF CLASSIFICATION

Figure 2 also visualises how we would expect a class of Integers to nest inside the class of Numbers. This is drawn using "stacking cones" to show that the volume occupied by the Integer class is contained within the Number class. Intuitively, we ought to be able to partition (divide up) the space of Numbers into sub-spaces, corresponding to the spaces occupied by the more specific numeric classes; however, we have yet to demonstrate this idea mathematically.

The Integer class is constructed in the same way as the Number class, first by defining a type generator, a type function which accepts the self-type as an argument:

$$\text{GenInteger} = \lambda\sigma.\{\text{plus} : \sigma \to \sigma,\ \text{minus} : \sigma \to \sigma,\ \text{times} : \sigma \to \sigma,\ \text{divide} : \sigma \to \sigma\}$$

then by expressing the polymorphic Integer class using an F-bound constructed using the same generator:

$$\forall(\tau <: \text{GenInteger}[\tau])$$

This constraint ensures that the parameter τ may range *only* over those numeric types which have *at least* the methods: *plus, minus, times* and *divide* with the specified signatures. It should be clear now that the exact Integer *type* is the least member of this class:

$$\text{Integer} <: \text{GenInteger}[\text{Integer}], \qquad \text{because Integer} = \text{GenInteger}[\text{Integer}].$$

How can we show that the Integer *class* nests inside the Number class? Intuitively, the purpose of classification is to show that objects which belong to specific classes also belong to the more general classes. We therefore want to be able to show that, if any actual type satisfies the constraint for the Integer class, it will also satisfy the constraint for the Number class. In other words, we seek the condition under which:

$$\tau <: \text{GenInteger}[\tau] \Rightarrow \tau <: \text{GenNumber}[\tau]$$

Intuitively, this holds true for these two particular generators, because GenInteger defines strictly more methods than GenNumber and otherwise it has an identical *plus* method. So, any type satisfying $\tau <: \text{GenInteger}[\tau]$ is bound to satisfy $\tau <: \text{GenNumber}[\tau]$. If GenInteger had not defined a *plus* method with a matching signature, then no types could satisfy both constraints. The condition we seek therefore depends on the structure of one generator including all the structure of the other generator.

This sounds somewhat similar to a record subtyping rule [2], except that the generators are not record types, they are type functions with *distinct* self-type arguments. Though we cannot make a rule to compare the generators directly, we can do this indirectly using a rule that compares *instantiations* of the generators, since this yields proper record types. In fact, we want to compare *all possible* instantiations of the two generators with the *same type*. This is known as a *pointwise subtyping* condition, which we write as:

$$\forall\tau\ .\ \text{GenInteger}[\tau] <: \text{GenNumber}[\tau]$$

We read this as: "For all types τ, the record type you get by instantiating GenInteger with τ is always a subtype of the record type you get by instantiating GenNumber with τ". Literally, the constraint expresses two things:

- the subclass generator must have a larger interface than the superclass generator
- the self-types must be made to refer to the same type when making any comparison

and this is the condition under which the GenInteger generator stands in a subclass-to-superclass relationship with the GenNumber generator.

Abstracting away from numeric types, we obtain the subclass rule relating *any* two generators, which we shall name (arbitrarily) GenSub and GenSuper:

$$\frac{\forall\tau\ .\ \text{GenSub}[\tau] <: \text{GenSuper}[\tau]}{\forall t\ .\ t <: \text{GenSub}[t] \ \Rightarrow\ t <: \text{GenSuper}[t]} \qquad [\text{Subclass}]$$

"If the generators GenSub and GenSuper stand in a pointwise subtyping relationship, then any type which satisfies the Sub-class constraint will also satisfy the Super-class constraint". This is the rule which describes how classes are nested inside each other. From this, we may derive another rule, which allows us to infer how types which belong to one class also belong to more general classes:

$$\frac{t <: \text{GenSub}[t],\quad \forall\tau\ .\ \text{GenSub}[\tau] <: \text{GenSuper}[\tau]}{t <: \text{GenSuper}[t]} \qquad [\text{Classify}]$$

"If t is a member type of the Sub-class, and the Sub-class is nested inside the Super-class, then t is also a member type of the Super-class." These are two of the most important rules in the theory of classification describing the subclass relationship and transitive class membership.

6 A MODEL OF INHERITANCE

In our presentation, we want to distinguish carefully the notions of *classification* and *inheritance*. Classification describes the way in which exact types belong to classes and the way in which classes are nested inside one another. Inheritance describes an extension mechanism for defining more specialised classes incrementally. That is, inheritance is merely a short-hand. This means that any class we could develop incrementally using inheritance must look the same as if we had defined it as a whole, from first principles. As a challenge, we shall attempt to derive the Integer class from the Number class.

Previously, we adopted a simplistic model of inheritance, using record type extension [3]. In this approach, a record type is considered to be a set of fields, which can be extended using the set union operator $\cup$ to create a larger record type, which is therefore a subtype of the original record [2]. The basic model for record type extension is:

$$\text{Subtype} = \text{Basetype} \cup \text{Extension}$$

However, we found an anomaly when extending recursive record types, which is that fields obtained from the base type were fixed with the wrong types when combined in the extended record type [3]. If we derive an Integer type naïvely from a Number type, we find that the *plus* method for Integer ends up with intuitively the wrong type signature:

$$\begin{aligned}&\text{Number} = \mu\sigma.\{\text{plus} : \sigma \to \sigma\},\\ &\Rightarrow\ \{\text{plus} : \text{Number} \to \text{Number}\}, \qquad \text{after unrolling the recursion;}\end{aligned}$$

$$\text{Integer} = \mu\sigma.(\text{Number} \cup \{\text{minus} : \sigma \to \sigma, \text{times} : \sigma \to \sigma, \text{divide} : \sigma \to \sigma\})$$

$$\begin{aligned}\Rightarrow\ &\{\text{plus} : \text{Number} \to \text{Number}, \text{minus} : \text{Integer} \to \text{Integer},\\ &\quad \text{times} : \text{Integer} \to \text{Integer}, \text{divide} : \text{Integer} \to \text{Integer}\}\end{aligned}$$

Essentially, the reason why this doesn't work is because we are extending simple types using a subtyping model and are hitting the limits of subtyping in the context of recursion. The self-types of Number and Integer are fixed independently, such that after record union, there is no single, uniform self-type, resulting in a kind of "self-type schizophrenia".

The insight offered by Cook and others [5] is that inheritance should not be modelled as the *extension of simple types*, but rather it is the *extension of the patterns described by their generators*. With generators, we are better able to control the binding of the self-types, so that these refer uniformly to the same type in the combined result. Starting with the GenNumber generator, we shall derive a GenInteger generator incrementally from it, but also rebind the self-type of the base generator at the same time:

$$\text{GenNumber} = \lambda\tau.\{\text{plus} : \tau \to \tau\}$$

$$\begin{aligned}\text{GenInteger} &= \lambda\sigma.(\text{GenNumber}[\sigma] \cup\\ &\qquad\qquad \{\text{minus} : \sigma \to \sigma, \text{times} : \sigma \to \sigma, \text{divide} : \sigma \to \sigma\})\\ &= \lambda\sigma.(\{\text{plus} : \sigma \to \sigma\} \cup \{\text{minus} : \sigma \to \sigma, \text{times} : \sigma \to \sigma, \text{divide} : \sigma \to \sigma\})\\ &= \lambda\sigma.\{\text{plus} : \sigma \to \sigma, \text{minus} : \sigma \to \sigma, \text{times} : \sigma \to \sigma, \text{divide} : \sigma \to \sigma\}\end{aligned}$$

This yields exactly the GenInteger that we hoped for, equivalent to the generator that we defined from first principles in section 5. Note how the inherited *plus* method is typed in terms of σ, the self-type of GenInteger, rather than in terms of τ, the old self-type of GenNumber. All the methods of GenInteger are uniformly typed in terms of σ, solving the "self-type schizophrenia" problem.

The trick at the heart of this derivation is the type-instantiation of the GenNumber generator, obtained through the application: GenNumber[σ]. Recall that GenNumber is a type-function, with a formal argument τ. When GenNumber is applied to σ, we substitute

$\{\sigma/\tau\}$ throughout in the body. The result of the application is a base record of method types: $\{\text{plus} : \sigma \rightarrow \sigma\}$ which is subsequently unioned with the extension record to yield the result.

To obtain the exact Integer type, we may take the fixpoint of the GenInteger generator that we have just derived, to bind the self-type argument recursively:

$$\text{Integer} = \mathbf{Y}\,[\text{GenInteger}] = \text{GenInteger}[\text{GenInteger}[\text{GenInteger}[...]]]$$

$$\Rightarrow \{\text{plus} : \text{Integer} \rightarrow \text{Integer},\ \text{minus} : \text{Integer} \rightarrow \text{Integer},$$
$$\text{times} : \text{Integer} \rightarrow \text{Integer},\ \text{divide} : \text{Integer} \rightarrow \text{Integer}\}, \qquad \text{at the limit.}$$

This has the type signature that we desire, namely one that is uniformly typed in terms of the Integer self-type, rather than one which is schizophrenic. This seems therefore to be a more satisfactory model for explaining incremental derivations in a class hierarchy.

7 CONCLUSION

We have exposed the heart of the theory of classification, using Cook's F-bounded quantification to express the idea that a class is *a family of types that share a minimum common structure.* By contrasting the notion of *type* in older object-based languages with the notion of *class* in newer object-oriented languages with inheritance, we have argued that extensibility and polymorphism are what properly characterise classes apart from traditional programmer's types. Thereafter, we have used *class* to refer to a polymorphic family and *type* to refer to one member of this family.

We developed a second-order model of class membership, in which recursive types with similar structure could be shown to belong in the same class. We extended this to a model of subclassing, based on a pointwise relationship between generators, to show how classes are nested hierarchically. Finally, we showed how generator adaptation provided a more satisfactory model of incremental inheritance, avoiding problems of schizophrenic self-reference. The languages Smalltalk and Eiffel adopt this more sophisticated model of inheritance, in which inherited references to self and the self-type are redirected to refer uniformly to the subclass.

While we dispelled one myth about classes and implementation, it is clear that we have only covered the *typeful* aspects of classes here. We defer a treatment of the *concrete* aspects of classes, especially the inheritance of implementation, to a later article. This will help us further in understanding the differences between the classification model of Smalltalk and Eiffel, and the subtyping model of Java and C++.

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] 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), pp. 273-280.

[5] 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.

[6] M Sakkinen, "Disciplined inheritance", *Proc. 3rd European Conf. Object-Oriented Prog.,* (Nottingham: British Computer Society, 1989), pp. 3-24.

[7] A Snyder, "Encapsulation and inheritance in object-oriented programming languages", *Proc. 1st ACM Conf. Object-Oriented Prog., Sys., Lang. and Appl.,* pub. *ACM Sigplan Notices, 21(11),* (ACM Sigplan, 986), pp. 38-45.

[8] P America , "Designing an object-oriented language with behavioural subtyping", *Proc. Conf. Foundations of Object-Oriented Lang*., (1990), pp. 60-90.

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.

JOURNAL OF OBJECT TECHNOLOGY

Online at http://www.jot.fm. Published by ETH Zurich, Chair of Software Engineering ©**JOT, 2003**

Vol. 2, No. 6, November-December 2003

The Theory of Classification
Part 9: Inheritance and Self-Reference

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

1 INTRODUCTION

This is the ninth article in a regular series on object-oriented type theory, aimed specifically at non-theoreticians. The previous article demonstrated how the intuitive notion of *class* in object-oriented languages has a *strictly formal* interpretation that is more general than the simple notion of *type* [1]. A class can be modelled as an *F-bounded polymorphic type*, representing a family of similar types which share a minimum common structure and behaviour [2, 3]. In this second-order model, which provides an alternative to first-order subtyping [2, 4], even recursively-defined classes can be shown to nest properly inside each other [1], a more sophisticated kind of type compatibility that we call *subclassing* in figure 1 (box 8). Type rules were defined for class membership, sub-classification and class extension by inheritance.

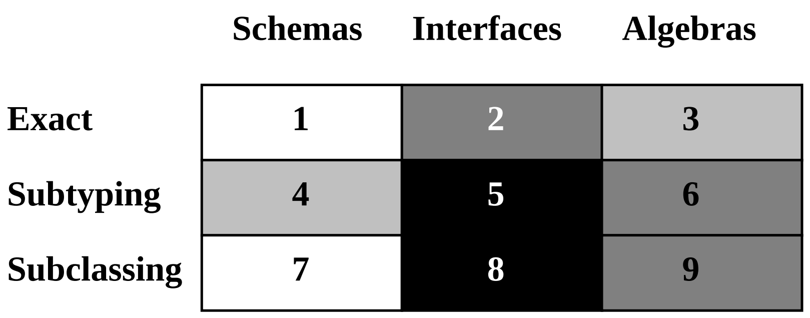


Figure 1: Dimensions of Type Checking

The previous article concentrated only on the *typeful* aspects of classes. In this article, we now turn to the *concrete* aspect of classes and the detailed modelling of method implementations. We want to be able to explain in the formal model how an object's structure is extended or altered during inheritance. In particular, we want to understand the process of method overriding and the meaning, after inheritance, of the special self-referential variable *self*, also known as *this*, or *current* in different object-oriented languages.

2 OBJECT IMPLEMENTATIONS

Several articles ago, we considered three different formal encodings of simple objects [5]. We preferred the λ-calculus encoding, which represents recursive objects as functional closures, denoting simple records of methods. One reason for choosing this model is because it fits so well with the F-bounded explanation of polymorphic classes, since both models rely on the use of *generators*, special functions which accept *self* (or, the *self*-type) as their argument. Later, we shall link the object model with the type model, in a combined second-order F-bounded λ–calculus. For the time being, we shall deal with objects in an untyped way.

A simple two-dimensional point object at the co-ordinate (3, 5) may be represented as a record, whose fields store values and functions, representing the attributes and methods of the object. Each field consists of a label which maps to a value (a simple value, or a function):

$$\begin{aligned} \text{aPoint2D} = \mu\ \text{self}\ .\ \{\ &x \mapsto 3,\ y \mapsto 5,\ \text{identity} \mapsto \text{self}, \\ &\text{equal} \mapsto \lambda p.(\text{self}.x = p.x \wedge \text{self}.y = p.y)\ \} \end{aligned}$$

In the above, *self* is the recursion variable, a placeholder equivalent to the eventual definition of the whole object *aPoint2D*, which contains embedded references to *self* (technically, we say that μ binds *self* to the resulting definition – see [5] for an explanation of this convention). Following the dot is a record of fields, enclosed in braces {…}. The first two fields, labelled *x* and *y*, map to simple values in the model. This is because we want *aPoint2D.x* and *aPoint2D.y* to return the simple values directly, rather like accessor methods. The third field *identity* is a method for returning the object itself; the fourth field *equal* maps to another function λp.(…), representing a method that compares *aPoint2D* with the argument *p*, which is assumed to be another Point2D instance.

This is where it becomes clear why *aPoint2D* is a recursive object: inside the body of the *equal* method, *aPoint2D* must invoke further methods *x* and *y* upon itself, to perform the field-by-field comparison with the argument *p*. To enable this, the body of the *equal* method needs a handle on the top-level object *aPoint2D*, granted through the recursion variable *self*. Any object that needs to invoke nested methods on itself must be recursive, so this is quite a common occurrence in practice. The *identity* method is also recursive, returning the object itself.

This theoretical use of *self* corresponds exactly to the usual meaning of *self* (in Smalltalk), *this* (in Java and C++) and *current* (in Eiffel). In the model, you always invoke nested methods explicitly through the recursion variable *self*. In some of these programming languages, you can omit the receiver of a nested message to the same object, which is implicitly understood to be *self* (*this*, or *current*). This is just a syntactic sugaring in the programming language.

3 OBJECT GENERATORS

Readers who have been following this series will know by now that a recursive definition is created from first principles using a generator, a function which abstracts over the point of recursion. Previously, we used *type generators* to build recursive types [1, 2]. Here, we introduce *object generators* to build recursive objects:

$$\begin{aligned}\text{genAPoint2D} = \lambda\ \text{self} . \{\ & x \mapsto 3,\ y \mapsto 5,\ \text{identity} \mapsto \text{self}, \\ & \text{equal} \mapsto \lambda p.(\text{self}.x = p.x \wedge \text{self}.y = p.y)\ \}\end{aligned}$$

In this function, *self* is not yet bound to any value – it is the argument of the function. The generator can be used to build the recursive object by infinite self-application [5]:

$$\text{aPoint2D} = \text{genAPoint2D}(\text{genAPoint2D}(\text{genAPoint2D}(\ldots)))$$

and for convenience's sake, we may use the fixpoint finder **Y** to build this infinite sequence:

$$\mathbf{Y}\ (\text{genAPoint2D}) = \text{genAPoint2D}(\text{genAPoint2D}(\text{genAPoint2D}(\ldots))) = \text{aPoint2D}$$

Later, we will see more expressions of this kind to describe the final fixing of the recursive structure of an object. Roughly speaking, an object generator *genAPoint2D* is a template for the structure of points like *aPoint2D*, in which *self* does not yet refer to anything. It turns out that this is a useful construction, since it will allow us to explain how *self* can refer to different objects.

4 EXTENDED OBJECT IMPLEMENTATIONS

Our goal is to model the inheritance of implementation. This can be thought of as a kind of extension or adaptation of an object, to produce an extended object with more fields, some of which may be modified, in the sense that the labels map to different values than in the original object. The challenge we set ourselves is to derive a three-dimensional point *aPoint3D* at the co-ordinate (3, 5, 2) by extending the simple two-dimensional *aPoint2D* in some fashion.

To begin with, we jump ahead to describe what we want *aPoint3D* to look like, at the end of the derivation process. Ultimately, we want it to have an extra *z* dimension and a modified version of the *equal* method, as if it had been defined from scratch, as a whole, in the following way:

$$\begin{aligned}\text{aPoint3D} = \mu\ \text{self} . \{\ & x \mapsto 3,\ y \mapsto 5,\ z \mapsto 2,\ \text{identity} \mapsto \text{self}, \\ & \text{equal} \mapsto \lambda p.(\text{self}.x = p.x \wedge \text{self}.y = p.y \wedge \text{self}.z = p.z)\ \}\end{aligned}$$

In the above, *self* is the placeholder variable, equivalent to the eventual definition of the object *aPoint3D*, which also contains embedded references to *self* in its *identity* and *equal* methods. Note the difference in the meaning of *self*: whereas before μ bound *self* $\leftarrow$ *aPoint2D*, here μ binds *self* $\leftarrow$ *aPoint3D*. The object denoted by *self* changes, depending on the binding context. From the practical point of view, this is also desirable, since the body of the modified *equal* method is only valid if *self* stands for *aPoint3D* (*viz*: you could not access the *z*-method of *aPoint2D*).

In the previous article [1], we constructed a simple model for inheritance, in which we treated a record as a set of fields and used set union $\cup$ to build a larger derived record by taking the union of the fields of the base record and a record of extra fields:

derived = base $\cup$ extra

This only works for very simple records, which have unique fields and no recursion:

aCoord2D = $\{ x \mapsto 3, y \mapsto 5 \}$	- base record
zField = $\{ z \mapsto 2 \}$	- extra record
aCoord3D = aCoord2D $\cup$ zField = $\{ x \mapsto 3, y \mapsto 5, z \mapsto 2 \}$	

We cannot construct *aPoint3D* in this way, because the simple union of fields would create a result in which there were two different versions of *equal*, since the original and redefined versions of this field are not actually identical, therefore the union would preserve two copies.

5 UNION WITH OVERRIDE

Instead, a different operator must used, called *union with override*: $\oplus$. This is a standard mathematical operator, defined for maps (rather than sets), that combines two maps in a certain way, which we define below. A *map* is a collection of pairs of values, called *maplets*, with an arrow from the left- to the right-hand value in each pair. A map is written like:

$\{a \mapsto x, b \mapsto y, c \mapsto z, \ldots\}$ “a maps to x, b maps to y, c maps to z, ...”

and can be viewed variously as a lookup table, in which each maplet relates a key (on the left) to a corresponding value (on the right), or alternatively as a function[1] from the domain $\{a, b, c, \ldots \}$ to the range $\{x, y, z, \ldots \}$. Where a map models a function, all the domain values are unique (but the range could contain duplicates).

An advantage in modelling objects as records is that they can also be thought of as maps: each field is a maplet consisting of a label (the domain) that maps to a value (the range). In object maps, the labels always have the same type (viz: Label), but the values

[1] The operator $\oplus$ is sometimes called *function override* in formal methods like Z, precisely because functions in Z are modelled as maps.

could be of many different types. For this reason, we give the union with override operator the following definition:

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

The top line is a polymorphic type signature [2], saying that $\oplus$ takes two maps with the individual type signatures $(\alpha\rightarrow\beta)$ and $(\alpha\rightarrow\gamma)$, and returns a map with the signature $(\alpha\rightarrow\beta \cup\gamma)$. Notice how the type of the domain α is the same in each case (for labels), but the types of the ranges β, γ are possibly different. The range in the result is a union type $\beta\cup\gamma$, formed by merging the types of the ranges of each argument. This is consistent with the type unions used in the previous article [1].

The full definition follows. This says that $\oplus$ takes two argument maps, *f* and *g* (with the given types) and produces a result map (the whole expression in braces). This result is the set of all those maplets $k \mapsto v$ that satisfy the following conditions (after the vertical bar |). The domain values *k* are obtained by taking the union of the domains of each argument map. This ensures that the domain of the result contains all the unique domain values from both maps. The range values *v* are obtained according to the asymmetric rule: if *k* is in the domain of the right-hand map *g*, use the corresponding range value *g(k)* from the right-hand map; otherwise use the corresponding range value *f(k)* from the left-hand map. This works, because every *k* must either be in the domain of the left-, or right-hand maps, or both. Note that *f(k), g(k)* denote range values by appealing to the functional interpretation of maps: *f(k)* "applies" the map *f* to the domain value *k*, yielding the corresponding range value.

When used with records, the union with override operator has the effect of merging two records, but preferring the fields from the right-hand side in the case of conflicting labels. Where there are no label-conflicts, it takes the union of the fields. If there are label-conflicts, it discards fields on the left-hand side and chooses fields from the right-hand side. This is exactly the behaviour we need to model record extension with overriding.

Another use for this operator is to model field updates in an object. We shall look at this first, as a simple way of illustrating the behaviour of $\oplus$. Consider updating the x position of the simple co-ordinate from above:

$$
\begin{aligned}
&\text{aCoord2D} = \{\ x \mapsto 3, y \mapsto 5\ \} && \text{- starting state values} \\
&\text{newCoord2D} = \text{aCoord2D} \oplus \{x \mapsto 7\} && \text{- override the x value} \\
&\qquad = \{\ x \mapsto 7, y \mapsto 5\ \} && \text{- modified state values}
\end{aligned}
$$

This can be used to model a primitive notion of field reassignment, although in the calculus we always create and return new objects (the λ-calculus is purely functional, after all).

6 SCHIZOPHRENIC SELF-REFERENCE

Following this example, it would seem natural to define the extended object *aPoint3D* by combining the record for *aPoint2D* with a record of the extra methods (*z* and the modified *equal*) that we want it to have. The overriding behaviour of ⊕ will ensure that the result gains just one copy of the *equal* method, from the right-hand record of extra methods, which has the following structure:

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

Notice how this record also implies a recursion somewhere, because the *equal* method contains free references to *self*. To which object should this *self* refer? Looking at the body of the *equal* method, it is clear that we intend it to refer to *aPoint3D*, because we want to invoke the nested method *self.z*, which is not valid for *aPoint2D*. The only way to make *self* refer to the resulting object is to bind it outside the record combination (see bold highlight):

$$\begin{aligned}\text{aPoint3D} \;=\; & \boldsymbol{\mu\,\text{self}\,.\,(}\ \text{aPoint2D} \oplus \{ z \mapsto 2, \\ & \quad \text{equal} \mapsto \lambda p.(\text{self}.x = p.x \wedge \text{self}.y = p.y \wedge \text{self}.z = p.z) \}\ \boldsymbol{)}\end{aligned}$$

This says that *aPoint3D* is a recursive object, formed by taking the union-with-override of *aPoint2D* and a record of extra methods, in which *self* refers to *aPoint3D*. Unrolling the definition of *aPoint2D* (see bold highlight) this gives:

$$\begin{aligned}\text{aPoint3D} \;=\; & \mu\,\text{self}\,.\,(\ \boldsymbol{\{ x \mapsto 3, y \mapsto 5, \text{identity} \mapsto \text{aPoint2D},} \\ & \qquad \boldsymbol{\text{equal} \mapsto \lambda p.(\ \text{aPoint2D}.x = p.x \wedge \text{aPoint2D}.y = p.y) \}} \\ & \quad \oplus \{ z \mapsto 2, \text{equal} \mapsto \lambda p.(\text{self}.x = p.x \wedge \text{self}.y = p.y \wedge \text{self}.z = p.z) \}\ )\end{aligned}$$

We can now judge how ⊕ will combine the fields of these two records. The result should contain fields having all the labels *{x, y, z, identity, equal}* and the right-hand version of *equal* should be preferred, overriding the left-hand version. After record combination, this simplifies to:

$$\begin{aligned}\text{aPoint3D} \;=\; & \mu\,\text{self}\,.\,\{ x \mapsto 3, y \mapsto 5, z \mapsto 2, \text{identity} \mapsto \text{aPoint2D}, \\ & \qquad \text{equal} \mapsto \lambda p.(\text{self}.x = p.x \wedge \text{self}.y = p.y \wedge \text{self}.z = p.z) \}\end{aligned}$$

This is the correct result, but on closer inspection, it may not be quite what was expected. Unrolling the definition of *aPoint3D* reveals what has become of all the references to *self*:

$$\begin{aligned}& \{ x \mapsto 3, y \mapsto 5, z \mapsto 2, \text{identity} \mapsto \textbf{aPoint2D}, \\ & \quad \text{equal} \mapsto \lambda p.(\textbf{aPoint3D}.x = p.x \wedge \textbf{aPoint3D}.y = p.y \wedge \textbf{aPoint3D}.z = p.z) \}\end{aligned}$$

From this it is apparent that *aPoint3D* is a schizophrenic object, in two minds about itself! The inherited method *identity* thinks that *self* ← *aPoint2D*, while the locally added method *equal* thinks that *self* ← *aPoint3D*. How did this confusion arise? Recall that the recursive object *aPoint2D* was defined in isolation, such that *self* was bound over a different record:

$$\begin{aligned}\text{aPoint2D} = \mu\ \text{self}\ .\ \{\ & x \mapsto 3,\ y \mapsto 5,\ \text{identity} \mapsto \text{self},\\ & \text{equal} \mapsto \lambda p.(\text{self}.x = p.x \wedge \text{self}.y = p.y)\ \}\end{aligned}$$

All *self*-reference in this object inevitably refers to *aPoint2D*. So, when we inherit any methods from *aPoint2D* that contain *self*, this always refers to *aPoint2D*. In an earlier article [2], we described the problem of type-loss in languages such as C++ and Java, when methods referring to the self-type are inherited. The current example explains in more detail why this type-loss occurs: it is because inherited *self* always refers to the old object.

7 REDIRECTING SELF-REFERENCE

It was Cook and Palsberg who first described the more flexible behaviour of *self* in languages like Smalltalk and Eiffel [6]. They drew analogies between object self-reference and recursive function derivations . Figure 2 shows the two contrasting cases.

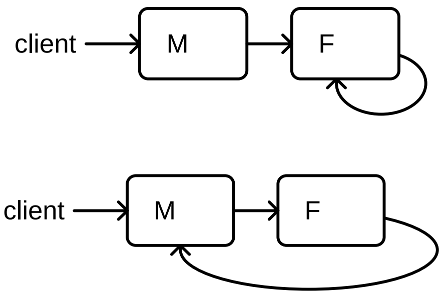


Figure 2: Naïve, and mutually recursive derivations

In the first example, a simple recursive function F calls itself, indicated by the loop. A modification to F is modelled as a derived function M which calls F. Recursive calls in F are unaffected, so the encapsulation of F is preserved. This is like the treatment of *self* in languages such as Java and C++. In these languages, a derived object does not modify the self-reference of the base object.

In the second example, the derived function M is *mutually recursive* with F. Recursive calls in F *are* affected by the modification, in the sense that they refer back to M, instead of to F. This is like the treatment of *self* in languages such as Smalltalk and Eiffel. In these languages, a derived object implicitly modifies the inherited *self*-references of the base object, such that these refer instead to the derived object.

To model inheritance in Smalltalk and Eiffel, we must redirect inherited *self*-references to refer to the derived object. In the formal model, this is accomplished by using generators instead of records. A generator is a function of *self*, in which the argument *self* is unbound (see section 3 above). The generator for 2D point objects is given by:

$$\begin{aligned}\text{genAPoint2D} = \lambda\, \text{self}_{2D} \,.\, \{\, & x \mapsto 3,\ y \mapsto 5,\ \text{identity} \mapsto \text{self}_{2D},\\ & \text{equal} \mapsto \lambda p.(\text{self}_{2D}.x = p.x \wedge \text{self}_{2D}.y = p.y)\ \}\end{aligned}$$

and now we seek to derive a generator for 3D point objects, by inheritance. The key strategy is to make sure that the old *self*$_{2D}$ is replaced by the new *self* before any record fields are combined. This is achieved by applying *genAPoint2D* to the new self-argument for 3D points (see bold highlight):

$$\begin{aligned}\text{genAPoint3D} = \lambda\, \text{self} \,.\, (\ & \mathbf{genAPoint2D(self)} \oplus \{\, z \mapsto 2,\\ & \text{equal} \mapsto \lambda p.(\ \text{self}.x = p.x \wedge \text{self}.y = p.y \wedge \text{self}.z = p.z)\ \}\ )\end{aligned}$$

This creates an instance of the generator body in which the substitution {*self/self*$_{2D}$ } has taken place. This body has the form of a record (see bold highlight):

$$\begin{aligned}\text{genAPoint3D} = \lambda\, \text{self} \,.\, (\ & \mathbf{\{\ x \mapsto 3,\ y \mapsto 5,\ identity \mapsto self,}\\ & \mathbf{equal \mapsto \lambda p.(\ self.x = p.x \wedge self.y = p.y)\ \}}\\ & \oplus\ \{\ z \mapsto 2,\ \text{equal} \mapsto \lambda p.(\text{self}.x = p.x \wedge \text{self}.y = p.y \wedge \text{self}.z = p.z)\ \}\ )\end{aligned}$$

Both records on the left- and right-hand sides now refer to exactly the same *self*. We may combine them using the union with override operator, yielding the final simplified expression:

$$\begin{aligned}\text{genAPoint3D} = \lambda\, \mathbf{self} \,.\, \{\ & x \mapsto 3,\ y \mapsto 5,\ z \mapsto 2,\ \text{identity} \mapsto \mathbf{self},\\ & \text{equal} \mapsto \lambda p.(\mathbf{self}.x = p.x \wedge \mathbf{self}.y = p.y \wedge \mathbf{self}.z = p.z)\ \}\end{aligned}$$

This produces the generator that we desired, in which all self-reference is uniform (see bold highlight). To create the derived recursive object, all we need to do is take the fixpoint of the generator, which has the effect of binding *self* to the resulting record:

$$\text{aPoint3D} = \mathbf{Y}\ (\text{genAPoint3D})$$

$$\begin{aligned}= \mu\, \text{self} \,.\, \{\ & x \mapsto 3,\ y \mapsto 5,\ z \mapsto 2,\ \text{identity} \mapsto \text{self},\\ & \text{equal} \mapsto \lambda p.(\text{self}.x = p.x \wedge \text{self}.y = p.y \wedge \text{self}.z = p.z)\ \}\end{aligned}$$

We have now succeeded in our challenge, for this object has the desired structure and uniformity that we anticipated in section 4, but it required a more sophisticated model of inheritance. This model of implementation inheritance is somehow more satisfying, in that it allows inherited code to adapt to the derived object. For example, the inherited *identity* method will now return the derived object *aPoint3D*, rather than the old object *aPoint2D*.

8 CONCLUSION

We have constructed two different models of implementation inheritance and found that these correspond broadly to the mechanisms used in the major object-oriented languages. The core idea of extending and modifying object templates can be modelled using records and the union with override operator ⊕, for which we provided a satisfying typed definition. In Cook's earlier work [4, 3, 6], no general typed definition was ever given for ⊕, so this aspect is novel in our Theory of Classification. The operator captures the basic mechanism for record combination with overriding for all languages. After this, object-oriented languages diverge, falling into two groups.

In the first group, which includes Java and C++, self-reference is fixed as soon as the object template is defined. When such an object is extended, although self-reference in the additional methods refers to the *derived object*, self-reference in the inherited methods always refers to the *base object*. In a suitably deep inheritance hierarchy, this means that objects may contain many versions of self, each referring to a different embedded ancestor-object! We described this as a kind of schizophrenia in self-reference. It is also linked to the problem of type-loss when methods passing values of the self-type are inherited [2]. Nonetheless, this model is the one used in all languages based on *types and subtyping*.

In the second group, which includes Smalltalk and Eiffel, self-reference is open to modification, *even after* the object template is defined. When such an object is extended, self-reference in the inherited methods is always redirected to refer to the derived object. All references to self are uniform, which is good for consistency, but they are interpreted locally in the object concerned. This is a novel feature of object-oriented languages, anticipating other kinds of adaptive programming. It is interesting to note how this flexibility came about. In Smalltalk, all methods are dynamically interpreted, so references to *self* are only bound at runtime to mean the current receiver. In Eiffel, the recursion variable *current* was originally conceived as a macro, to be expanded locally to refer to the new class. This had the effect of building implicit type redefinition into the language. The flexible model of inheritance is used in all languages based on *classes and subclassing* [1].

Formally, the difference between the two models of inheritance is very slight: it depends only on *when* fixpoints are taken. In the subtyping model, the recursion in the base object is fixed *before* record combination. In the subclassing model, the recursion is only fixed *after* record combination. For this reason, we can claim that the theory is both elegant and economical.

REFERENCES

[1] 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_04/column4

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

[3] 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.

[4] 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), pp. 273-280.

[5] 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

[6] W Cook and J Palsberg, "A denotational semantics of inheritance and its correctness", *Proc. 4th ACM Conf. Obj.-Oriented Prog. Sys. Lang. and Appl.*, pub. *Sigplan Notices, 24(10)*, (ACM Sigplan, 1989), pp. 433-443.

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.

JOURNAL OF OBJECT TECHNOLOGY

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

Vol. 3, No. 1, January-February 2004

The Theory of Classification Part 10: Method Combination and Super-Reference

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

1 INTRODUCTION

This is the tenth article in a regular series on object-oriented type theory, aimed specifically at non-theoreticians. Recent articles have presented a formal model of classification that differs from the conventional model of types and subtyping [1, 2]. The popular object-oriented languages fall into two groups – those based on simple types and subtyping, such as C++ and Java, and those based on polymorphic classes and subclassing, such as Smalltalk and Eiffel [2, 3]. A programmer's class has a formal interpretation at the typeful level [2] and a concrete interpretation the implementation level [3]. In the theory, we have been careful to distinguish *classification* from *inheritance*.

Classification is the hierarchical relationship between classes, whereby one class is judged to be a subclass of another, according to type rules governing subclassing [2]. Inheritance, on the other hand, is a short-hand mechanism for defining a new class in relation to an existing class[1], specifying what is new or different, and otherwise *inheriting* all existing features [3]. This presents an interesting formal challenge. If inheritance is indeed merely a short-hand, we should be able to prove in our theory that a class constructed by inheritance is equivalent to a class defined as a whole, from first principles [3]. This challenge is made more complicated by the possibility of *method combination*, the merging of local and inherited versions of a method. Furthermore, Smalltalk and Java can invoke inherited versions of methods through a pseudo-variable called *super*. So, our

[1] Popular books on object-oriented programming sometimes confuse these notions, referring to the hierarchical relationship as "inheritance". Strictly speaking, inheritance is just the extension mechanism. However, an inheriting class will typically be a subclass, but only by virtue of obeying the rules about classification [2].

theory must be able to explain the meaning of *super*, and show how super-method invocations are eventually equivalent to ordinary method invocations.

2 METHOD COMBINATION

So far, our treatment of inheritance and method overriding has assumed that individual methods are always replaced as a whole. For example, in the previous article [3] we replaced an *equal* method of a two-dimensional point:

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

in which *self* refers to a Point2D instance, by an *equal* method of a three-dimensional point:

$$\text{equal} \mapsto \lambda p.(\text{self.x} = \text{p.x} \wedge \text{self.y} = \text{p.y} \wedge \text{self.z} = \text{p.z})$$

in which *self* refers to a Point3D instance. On the surface, the overriding method appears to be syntactically quite similar to the method that was replaced. In most object-oriented languages, programmers don't have to write out such replacement methods long-hand; instead the languages offer a short-hand mechanism for adapting the inherited version of the method. The idea is that the overriding method may somehow reuse the code body of the inherited method and need only specify what additional computations take place. This is known as *method combination*.

In C++ and Eiffel, method combination is achieved simply by naming the local and inherited versions of the method differently. In C++ it is always possible to qualify method names globally by their owning class, in the style: *ParentClass::method* and *ChildClass::method*. So, the redefined body of the *Child*'s *method* may invoke *ParentClass::method* explicitly, by referring to its global name. In Eiffel, the name of a method may be locally renamed when it is inherited, in the style: ***class*** *Child* ***inherit*** *Parent* ***rename*** *method* ***as*** *old_method …* ***end***. Then, the body of *Child*'s *method* may invoke *old_method*. In both of these cases, invoking an inherited method inside a redefined version is no different from invoking any other differently-named method.

Smalltalk and Java achieve method combination in a much more interesting way. These languages have a pseudo-variable called *super*, which somehow allows a programmer to invoke an inherited version of a method inside the redefined version of the same method, without renaming the method. In the theoretical model, we can express Point3D's *equal* method more simply as:

$$\text{equal} \mapsto \lambda p.(\text{super.equal(p)} \wedge \text{self.z} = \text{p.z})$$

In this, *super* somehow stands for the current object in the context of the parent class. Whereas an invocation *self.equal(…)* would call the local version, *super.equal(…)* is deemed to call the inherited version of the same method. It is quite difficult to understand exactly what object *super* refers to! By the end of this article, we hope to answer this important question. However, one consideration is that the sub-expression: *super.equal(p)* must eventually be shown to be equivalent to the portion of code in the long-hand version of the method: $self.x = p.x \wedge self.y = p.y$.

3 RENAMING METHODS

To handle method combination in C++ and Eiffel, we must first consider how to rename methods in the theoretical model. In C++, we could either decide that the global names exist permanently as alternative, duplicate names for the same methods, or we could introduce them on demand, when we want to combine methods. In Eiffel, however, we must allow methods to be renamed locally, on demand, during inheritance. Recall that inheritance is modelled as record combination in the theoretical model [3], in which a *base* record is extended by combining it with a record of *extra* methods, to yield a *derived* record:

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

Intuitively, we now want to rename methods in the base record, then combine this with the extra record. For this, we need a renaming operator ®, and expect to use it in the following way:

$$\text{derived} = (\text{base} \circledR \text{renamings}) \oplus \text{extra}$$

in which *renamings* is a map from original labels to revised labels. In the same style as the union with override operator [3], we may define the renaming operator to accept an object record (a map from labels to methods) and a record of renamings (a map from labels to labels) and return a new object record in which some labels have been replaced:

$$\begin{aligned}
&\forall\alpha, \beta \,.\, \circledR : (\alpha\rightarrow\beta) \times (\alpha\rightarrow\alpha) \rightarrow (\alpha\rightarrow\beta)\\
&\circledR = \lambda(f{:}\alpha\rightarrow\beta).\lambda(g{:}\alpha\rightarrow \alpha).\\
&\qquad\qquad \{\ k \mapsto v \mid \forall h \in \mathrm{dom}(f) \,.\, v = f(h) \wedge\\
&\qquad\qquad\qquad\quad (h \in \mathrm{dom}(g) \Rightarrow k = g(h)) \wedge\\
&\qquad\qquad\qquad\quad (h \notin \mathrm{dom}(g) \Rightarrow k = h)\ \ \}
\end{aligned}$$

The top line is a polymorphic type signature [1], saying that ® takes two maps with the individual type signatures $(\alpha\rightarrow\beta)$ and $(\alpha\rightarrow\alpha)$, and returns a map with the signature $(\alpha\rightarrow\beta)$. The type α is the label-type, and β is the method-type in the object record. Note how the object-type of the result is unchanged, reflecting that methods have merely been renamed.

The full definition follows. This says that ® takes two argument maps, *f* and *g* (with the given types) and produces a result map (the whole expression in braces). This result is the set of all those maplets $k \mapsto v$ that satisfy the following conditions (after the vertical bar |). For all original labels *h* in the domain of the base object *f*, if *h* is also listed in the domain of the renamings *g*, the resulting label *k* is equal to the translation *g(h)*, otherwise *k* is equal to the original label *h*. The corresponding values *v* are always equal to *f(h)*, as in the original object map. Recall that *f(h)*, *g(h)* denote range values by appealing to the functional interpretation of maps: *f(h)* "applies" the map *f* to the domain value *h*, yielding the corresponding range value.

We can use this operator to rename methods of an object. Here is a simple Point2D instance:

$$\begin{aligned}\text{aPoint2D} = \mu\,\text{self} . \{ & \text{x} \mapsto 3,\ \text{y} \mapsto 5,\ \text{identity} \mapsto \text{self},\\ & \text{equal} \mapsto \lambda\text{p}.(\text{self.x} = \text{p.x} \wedge \text{self.y} = \text{p.y})\ \}\end{aligned}$$

in which we wish to rename the *equal* method with a longer, qualified name, *old_equal*:

$$\begin{aligned}&\text{aPoint2D} ® \mathbf{\{equal \mapsto old\_equal\}}\\ &\quad = \mu\,\text{self} . \{ \text{x} \mapsto 3,\ \text{y} \mapsto 5,\ \text{identity} \mapsto \text{self},\\ &\qquad\quad \text{old\_equal} \mapsto \lambda\text{p}.(\text{self.x} = \text{p.x} \wedge \text{self.y} = \text{p.y})\ \}\end{aligned}$$

The renaming operator ® takes, as its operands, the object and a map of renamings (see bold highlight), yielding an object in which the *equal* method has been renamed.

4 METHOD COMBINATION WITH RENAMING

This allows us to construct a model of inheritance with method combination supported through renaming. The following is an expression to derive *aPoint3D* from *aPoint2D* by renaming its *equal* method and then supplying a redefined version, which refers back to the old version:

$$\begin{aligned}\text{aPoint3D} \ = \ \mu\,\text{self} . (\ & \mathbf{(aPoint2D ® \{equal \mapsto old\_equal\})} \oplus\\ & \{ \text{z} \mapsto 2,\ \text{equal} \mapsto \lambda\text{q}.(\mathbf{self.old\_equal(q)} \wedge \text{self.z} = \text{q.z})\ \}\ )\end{aligned}$$

The left-hand operand of the record combination operator ⊕ is a renaming expression (see first bold highlight), in which the *equal* method is renamed before combination. The right-hand operand of ⊕ is a record of extra methods, in which the redefined *equal* method invokes the renamed method *old_equal* in its body (see second bold highlight), thereby benefiting from the more succinct syntax offered by method combination.

We want to show that this method combination syntax is equivalent to a regular, wholesale method replacement. Simplifying the renaming expression yields a base record (see bold highlight):

$$\begin{aligned}\text{aPoint3D} \ = \ \mu\,\text{self} . (\ & \mathbf{\{ x \mapsto 3,\ y \mapsto 5,\ identity \mapsto aPoint2D,}\\ & \quad \mathbf{old\_equal \mapsto \lambda p.(\ aPoint2D.x = p.x \wedge aPoint2D.y = p.y)\ \}}\\ & \oplus \{ \text{z} \mapsto 2,\ \text{equal} \mapsto \lambda\text{q}.(\text{self.old\_equal(q)} \wedge \text{self.z} = \text{q.z})\ \}\ )\end{aligned}$$

which in turn may be combined with the record of extra methods (see [3]) to yield the following unusual record, in which inherited self-references are resolved to *aPoint2D*, while the local self is bound over *aPoint3D*:

$$\begin{aligned} \text{aPoint3D} = \mu\,\text{self} . \{ & x \mapsto 3, y \mapsto 5, z \mapsto 2, \text{identity} \mapsto \text{aPoint2D}, \\ & \text{old\_equal} \mapsto \lambda p.(\ \text{aPoint2D}.x = p.x \wedge \text{aPoint2D}.y = p.y), \\ & \text{equal} \mapsto \lambda q.(\mathbf{self.old\_equal(q)} \wedge \text{self}.z = q.z)\ \} \end{aligned}$$

The resulting object has both the renamed method *old_equal* and the redefined version *equal*. This is quite normal in languages with renaming. Note that we have deliberately given these two functions different formal argument names, *p* and *q*, in order to observe what happens to object references in the next stage. We now want to understand the precise meaning of the redefined *equal* in detail. To do this, we may simplify the embedded call to the old version of the method (see bold highlight above). This simplification is equivalent to replacing the call by the inlined body of the *old_equal* method, after substituting the argument {*q/p*}, *viz* the evaluation:

$$\text{self.old\_equal}(q) \Rightarrow (\text{aPoint2D}.x = q.x \wedge \text{aPoint2D}.y = q.y)$$

This internal simplification is performed exactly like any other function simplification; and may be thought of as an evaluation in which the call is replaced by the body after arguments have been substituted. In this case, the actual argument *q* (a Point3D instance) is substituted in place of the formal argument *p* (a Point2D variable) yielding the simplified form:

$$\begin{aligned} \text{aPoint3D} = \mu\,\text{self} . \{ & x \mapsto 3, y \mapsto 5, z \mapsto 2, \text{identity} \mapsto \text{aPoint2D}, \\ & \text{old\_equal} \mapsto \lambda p.(\ \text{aPoint2D}.x = p.x \wedge \text{aPoint2D}.y = p.y), \\ & \text{equal} \mapsto \lambda q.(\mathbf{aPoint2D.x = q.x} \wedge \mathbf{aPoint2D.y = q.y} \wedge \text{self}.z = q.z) \\ \} \end{aligned}$$

This demonstrates formally how the derived *equal* method does indeed compare all of the x, y and z dimensions of the Point3D instance *q*. However, note how the body of this method suffers from the same kind of schizophrenia that we have noted before [2, 3] when dealing with simple object types. The local binding is *self* ← *aPoint3D*, whereas the inherited *self* ← *aPoint2D*. After the internal simplification, the combined method is comparing mixtures of 2D and 3D points for the equality of their x and y fields! For this reason, we cannot yet regard this kind of method combination as wholly equivalent to defining a replacement method wholesale.

5 FLEXIBLE METHOD COMBINATION WITH RENAMING

The problem is one of ensuring uniform self-reference in both local and inherited methods. Recall that in Eiffel, self-reference is flexible, such that inherited occurrences of self (known as *current* in Eiffel) are redirected to refer to the derived object. In the formal model, this requires the use of an object generator to express the object definition [3]:

$$\begin{aligned}\text{genAPoint2D} = \lambda\ \text{self}\ .\ \{\ & \text{x} \mapsto 3,\ \text{y} \mapsto 5,\ \text{identity} \mapsto \text{self},\\ & \text{equal} \mapsto \lambda\text{p}.(\text{self.x} = \text{p.x} \wedge \text{self.y} = \text{p.y})\ \}\end{aligned}$$

This is different from a plain object in that *self* is a formal argument of the generator. As such, we can bind it to different values, representing the different objects that *self* may range over. Inheritance is modelled as an adaptation on generators [3]. We now add the renaming scheme to this model:

$$\begin{aligned}\text{genAPoint3D}\ =\ \lambda\ \text{self}\ .\ (\ & \mathbf{(genAPoint2D(self)\ ®\ \{equal \mapsto old\_equal\})}\\ & \oplus\ \{\ \text{z} \mapsto 2,\ \text{equal} \mapsto \lambda\text{q}.(\text{self.old\_equal(q)} \wedge \text{self.z} = \text{q.z})\ \}\ )\end{aligned}$$

The critical difference is in the way self-reference is modified, through the generator application: *genAPoint2D(self),* yielding an adapted record in which *self* refers to the new object. After simplifying the renaming-expression (see bold highlights above and below), this yields:

$$\begin{aligned}\text{genAPoint3D}\ =\ \lambda\ \text{self}\ .\ (\ & \mathbf{\{\ x \mapsto 3,\ y \mapsto 5,\ identity \mapsto self,}\\ & \mathbf{old\_equal \mapsto \lambda p.(self.x = p.x \wedge self.y = p.y)\ \}}\\ & \oplus\ \{\ \text{z} \mapsto 2,\ \text{equal} \mapsto \lambda\text{q}.(\text{self.old\_equal(q)} \wedge \text{self.z} = \text{q.z})\ \}\ )\end{aligned}$$

and after record combination, this yields a result in which both *equal* and *old_equal* methods co-exist, but all self-reference is now uniform:

$$\begin{aligned}\text{genAPoint3D}\ =\ \lambda\ \text{self}\ .\ \{\ & \text{x} \mapsto 3,\ \text{y} \mapsto 5,\ \text{z} \mapsto 2,\ \text{identity} \mapsto \text{self},\\ & \text{old\_equal} \mapsto \lambda\text{p}.(\text{self.x} = \text{p.x} \wedge \text{self.y} = \text{p.y})\\ & \text{equal} \mapsto \lambda\text{q}.(\mathbf{self.old\_equal(q)} \wedge \text{self.z} = \text{q.z})\ \}\end{aligned}$$

We now simplify the body of the *equal* method, by expanding inline the call to *old_equal*:

$$\begin{aligned}\text{genAPoint3D}\ =\ \lambda\ \text{self}\ .\ \{\ & \text{x} \mapsto 3,\ \text{y} \mapsto 5,\ \text{z} \mapsto 2,\ \text{identity} \mapsto \text{self},\\ & \text{old\_equal} \mapsto \lambda\text{p}.(\text{self.x} = \text{p.x} \wedge \text{self.y} = \text{p.y})\\ & \text{equal} \mapsto \lambda\text{q}.(\mathbf{self.x = q.x \wedge self.y = q.y} \wedge \text{self.z} = \text{q.z})\ \}\end{aligned}$$

This yields the final result (see highlights) in which *self* refers uniformly to the current object. This is a satisfactory outcome, in that it meets our requirement that method combination should be provably equivalent to redefining the method wholesale. Because of its special treatment of *current*, Eiffel's method combination with renaming follows this semantics.

6 THE MEANING OF SUPER

However, it may be considered inelegant for objects to keep both the old and new versions of a method. Having to rename methods is irksome and keeping both versions is redundant, especially if you only want the old version once, during method combination.

For this reason, languages like Smalltalk and Java provide “one time access” to the inherited versions of methods, when redefining the same methods, through a special pseudo-variable called *super*:

$$\text{equal} \mapsto \lambda p.(\mathbf{super.equal(p)} \wedge \text{self.z} = \text{p.z})$$

It is clear that *super* must refer in some sense to the current object, somewhat like *self*, yet different from the point of view of method lookup. The operational explanation of *super*-method invocation is typically that “the search for a method starts in the immediate superclass of the class of *self*” [4]. Other attempts at describing *super* sometimes say that it is “the inherited object” or “the embedded parent-part of the current object”. In fact, the meaning of *super* is subtle and varies from language to language, as we shall show below.

We may construct a model of *super* from the operational description of method lookup. The most obvious way of invoking the “next most general” version of a method in our theory is to ensure that we select it from the “next most general” version of the object-instance with which we are dealing. In other words, if our most recently derived object is expressed as:

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

such that we would expect *derived.method* to invoke the latest version of some *method*, then *base.method* should be the expression that invokes the next most general version of the same *method*, skipping over any redefinition supplied in the *extra* record. From this analysis, it seems clear that *super* is equivalent to the *base* object record, in a record combination expression.

In the theory, this object is always supplied as the left-hand operand to the record combination operator ⊕ (see bold highlights below). Recall that in record combination with simple object records (see section 6 in [3]) we have:

$$\begin{aligned}\text{aPoint3D} = \mu\, \text{self} . (\mathbf{aPoint2D} \oplus \{ & \text{z} \mapsto 2,\\ & \text{equal} \mapsto \lambda p.(\text{self.x} = \text{p.x} \wedge \text{self.y} = \text{p.y} \wedge \text{self.z} = \text{p.z}) \})\end{aligned}$$

which indicates that this operand is in fact equivalent to an instance of the base type. This gives the meaning of *super* in Java, a language based on simple types and subtyping in our theory. The variable *super* corresponds to the embedded parent-part of the current object, *self*, or more exactly to the inherited parent-part before any fields are replaced during record combination. It has the type *super : Point2D* and behaves exactly like an instance of the base type, in that self-references in super-methods refer back to *super*.

On the other hand, in flexible record combination with object generators (see section 7 in [3]) we have the completely different left-hand operand:

$$\begin{aligned}\text{genAPoint3D} = \lambda\, \text{self} . (\mathbf{genAPoint2D(self)} \oplus \{ & \text{z} \mapsto 2,\\ & \text{equal} \mapsto \lambda p.(\text{self.x} = \text{p.x} \wedge \text{self.y} = \text{p.y} \wedge \text{self.z} = \text{p.z}) \})\end{aligned}$$

which is an adapted record, obtained by applying the base generator to the new self reference (of the derived generator). This gives the exact meaning of *super* in Smalltalk, a language based on classes and subclassing in our theory. The variable *super* is like an adapted instance of the parent type, in which self-reference is redirected to refer to the current, derived object. It has an adapted type given by the application of a type generator *super : GenPoint3D[Point2D]* and behaves differently from an instance of the base type, in that self-references in super-methods refer to the whole of the current, derived object, rather than to an embedded parent instance.

7 METHOD COMBINATION WITH SUPER-REFERENCE

In order to model method combination with super-reference in our theory, we need to introduce the *super* variable, and bind it to a value standing for the (possibly adapted) parent instance, such that we may refer to *super* throughout the record combination expression, in the style:

$$\mathbf{super} \oplus \{ z \mapsto 2, \text{equal} \mapsto \lambda q.(\mathbf{super}.\text{equal}(q) \wedge \text{self}.z = q.z) \}$$

The variable may be introduced as the formal argument of a function: λ*super.(…)* which is later bound to a suitable value. The order of variable introduction is dictated by the need for *super* to be bound outside the scope of record combination, but inside the scope of *self*:

$$\begin{aligned} \text{aPoint3D} = \mu\, \text{self} . \,&\mathbf{(\lambda super . (}\text{super} \oplus \{ z \mapsto 2, \\ &\quad \text{equal} \mapsto \lambda q.(\text{super.equal}(q) \wedge \text{self}.z = q.z) \}\ \mathbf{)} \\ &\mathbf{aPoint2D)} \end{aligned}$$

Here, the expression λ*super.(…)* is a function, binding *super*, whose body is a record combination expression that contains free references to *super* as intended. This super-function is then applied to the value: *aPoint2D*. To verify that this is equivalent to regular record combination, we may simplify the super-function application internally, with the binding *super* ← *aPoint2D*, to obtain:

$$\begin{aligned} \text{aPoint3D} = \mu\, \text{self} . \,&\mathbf{(\ aPoint2D} \oplus \{ z \mapsto 2, \\ &\text{equal} \mapsto \lambda q.(\mathbf{aPoint2D}.\text{equal}(q) \wedge \text{self}.z = q.z) \}\ \mathbf{)} \end{aligned}$$

Firstly, we see that *super* is replaced by the desired parent object on the left-hand side of the combination operator. Secondly, we find that *super.equal(…)* translates exactly into an expression invoking the *equal* method of a parent instance, which is promising, since it clearly skips the current version. To simplify this internally, we replace the call by the inlined body of the parent's method, after substituting the argument {*q/p*} as before. After record combination:

$$\begin{aligned} \text{aPoint3D} = \mu\, \text{self} . \{\, &x \mapsto 3, y \mapsto 5, z \mapsto 2, \text{identity} \mapsto \text{aPoint2D}, \\ &\text{equal} \mapsto \lambda q.(\mathbf{aPoint2D.x = q.x} \wedge \mathbf{aPoint2D.y = q.y} \wedge \text{self}.z = q.z) \\ \} \end{aligned}$$

this yields a non-uniform solution similar to that in section 4 above, but without duplicate versions of the *equal* method. This model of method combination explains the behaviour of languages like Java, in which *super* always resolves to an instance of the immediate parent type.

The more flexible kind of method combination with object generators may be modelled as:

$$\begin{aligned}\text{genAPoint3D} \;=\; \lambda\,\text{self}\,.\,\mathbf{(\lambda super\,.\,(}&\text{super} \oplus \{\, z \mapsto 2,\\ &\quad \text{equal} \mapsto \lambda q.(\text{super.equal}(q) \wedge \text{self.z} = q.z)\ \}\ \mathbf{)}\\ &\mathbf{genAPoint2D(self))}\end{aligned}$$

in which the super-function is bound differently with *super* ← *genAPoint2D(self)*. Note in passing how *self* must be bound before we can bind *super*. This time, *super* does not denote a parent instance, but rather an adapted object, the result of applying the parent generator to *self*. To explore further what this means, we may simplify the super-function application internally, yielding:

$$\begin{aligned}\text{genAPoint3D} \;=\; \lambda\,\text{self}.\ &\mathbf{(genAPoint2D(self)} \oplus \{\, z \mapsto 2,\\ &\text{equal} \mapsto \lambda q.(\mathbf{genAPoint2D(self)}.\text{equal}(q) \wedge \text{self.z} = q.z)\ \}\ \mathbf{)}\end{aligned}$$

From this, it appears that the *super.equal(…)* invocation is equivalent to invoking *equal* on an adapted parent object. This is quite subtle, because self-reference is redirected in this object onto the new *self* of 3D points. We can illustrate this more graphically by expanding *super*:

$$\begin{aligned}\text{super} &= \text{genAPoint2D}(\mathbf{self_{3D}})\\ &= \{\, x \mapsto 3,\ y \mapsto 5,\ \text{identity} \mapsto \mathbf{self_{3D}},\\ &\qquad\qquad \text{equal} \mapsto \lambda p.(\ \mathbf{self_{3D}}.x = p.x \wedge \mathbf{self_{3D}}.y = p.y)\ \}\end{aligned}$$

From this, it is clear that *super.equal* will select the inherited *equal* method body, in which *self* refers back to the local Point3D instance. Furthermore, *super.equal(q)* will produce the argument substitution {*q/p*}, where *q* is implicitly a variable of the type Point3D:

$$\text{super.equal}(\mathbf{q}) \;\Rightarrow\; (\text{self}_{3D}.x = \mathbf{q}.x \wedge \text{self}_{3D}.y = \mathbf{q}.y)$$

When this subexpression is replaced in the body of the local *equal* method, we obtain a combined *equal* method, which has consistent self-reference and an argument in the same type:

$$\text{equal} \mapsto \lambda q.(\ \text{self}_{3D}.x = q.x \wedge \text{self}_{3D}.y = q.y \wedge \text{self}_{3D}.z = q.z)$$

Method combination using super-reference is thereby proved to be equivalent to redefining the method wholesale. Note how the more subtle semantics of *super* is needed for this to work out fully. Cook et al. were the first to identify this formal interpretation of *super* in Smalltalk [5, 6], from the operational description of inheritance in that language.

8 CONCLUSION

We have constructed four different models for inheritance with method combination. Two involve the renaming of methods, in the style of C++ and Eiffel. Two involve the use of the pseudo-variable *super*, in the style of Java and Smalltalk. While C++ and Java have a simple model of inheritance, based on the extension of object records, Smalltalk and Eiffel have a more subtle model of inheritance, based on the extension of object generators. Hereafter in our theory, we shall refer to the simple extension model as *derivation*, to distinguish it from "genuine" *inheritance*, in which self-references are redirected to refer to more specific objects [3, 5].

Considering each approach to method combination individually, the first uses derivation and renaming. The second uses inheritance and renaming, of which Eiffel is the exemplar. The third uses derivation and super-reference, of which Java is the exemplar. The fourth uses inheritance and super-reference, of which Smalltalk is the exemplar. An even simpler model exists for C++, if we allow objects to have two names for each method, one local and one global:

$$\begin{aligned}&\text{equal} \mapsto \lambda p.(\text{self}.x = p.x \wedge \text{self}.y = p.y)\\ &\text{Point2D\_equal} \mapsto \lambda p.(\text{self}.x = p.x \wedge \text{self}.y = p.y)\end{aligned}$$

and adopt the convention that only the local names are ever overridden. In this way, we can express the combined *equal* method for Point3D as:

$$\text{equal} \mapsto \lambda q.(\text{self.Point2D\_equal}(q) \wedge \text{self}.z = q.z)$$

which simplifies in accordance with our first model, above. The method combination strategies using inheritance were shown to be equivalent to wholesale method replacement, demonstrating again the usefulness of the theoretical model. The model also provided the pseudo-variable *super* with two semantic interpretations, corresponding to the meanings of this variable in Java and Smalltalk. We also provided an original renaming operator ® to account for Eiffel's behaviour.

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. 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 Goldberg and D Robson, *Smalltalk 80: The Language and its Implementation*, Addison-Wesley, 1983.

[5] 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.

[6] W Cook and J Palsberg, “A denotational semantics of inheritance and its correctness”, *Proc. 4th ACM Conf. Obj.-Oriented Prog. Sys. Lang. and Appl.*, pub. *Sigplan Notices, 24(10)*, (ACM Sigplan, 1989), pp. 433-443.

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.

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 $\oplus$ 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 $\oplus$, 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.

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:

negate = λx.(-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:

negate : Integer $\rightarrow$ Integer
= λ(x : 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:

negate : $\forall\tau.\ \tau \rightarrow \tau$
= $\lambda\tau.\lambda$(x : τ).(-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: $\lambda\tau$. 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) $\Rightarrow$ -3
negate [Real] (2.1 : Real) $\Rightarrow$ -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 $\lambda\tau$. So, the value returned after type-application is identical to a simply-typed version of the function (like that shown above):

negate [Integer] $\Rightarrow$ λ(x : Integer).(-x)
negate [Real] $\Rightarrow$ 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: $\lambda(\tau$ :: 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:

$$\begin{aligned}&\text{negate} : \forall(\tau :: \text{SIGNED}).\ \tau \to \tau\\&= \lambda(\tau :: \text{SIGNED}).\lambda(x : \tau).(\text{-}x)\end{aligned}$$

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:

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

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:

$$\begin{aligned}&\forall(\tau <: \text{GenPoint}[\tau]).\ \tau.x : \to \text{Integer}\\&\forall(\tau <: \text{GenPoint}[\tau]).\ \tau.y : \to \text{Integer}\\&\forall(\tau <: \text{GenPoint}[\tau]).\ \tau.\text{equal} : \tau \to \text{Boolean}\end{aligned}$$

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:

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

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:

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

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:

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

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:

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

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: $\lambda(\tau <: \mathrm{GenPoint}[\tau])$, then the *self*-argument: $\lambda(\mathit{self} : \tau)$, 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*:

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

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, λ(*super* : GenPoint[σ]).(…) 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: $\forall(\tau <:$ GenPoint[τ]). $\tau \rightarrow$ GenPoint[τ], 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 $\tau <:$ GenPoint[τ]. Technically, the application *genAPoint*[σ] is simply a matter of substituting one type parameter for another: $\{\sigma/\tau\}$. All type parameters have the same kind, TYPE, so this should not be a problem. However, a more subtle thing is happening. By substituting $\{\sigma/\tau\}$, 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:

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

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:

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

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:

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

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:

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

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:

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

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

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

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:

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

Looking at the pooled method types: {Integer, $\sigma \to$ Boolean, Boolean} in the result, this does seem to be a union of {Integer, $\sigma \to$ Boolean} $\cup$ {$\sigma \to$ 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:

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

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 $\oplus$, in order to restrict the legitimate types of extra methods added to an object during inheritance, but our liberally-typed version of $\oplus$ works as intended with the object implementations shown. Technically, the definition given for $\oplus$ 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.

JOURNAL OF OBJECT TECHNOLOGY

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

Vol. 3, No. 5, May-June 2004

The Theory of Classification Part 12: Building the Class Hierarchy

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

1 INTRODUCTION

This is the twelfth article in a regular series on object-oriented type theory for non-specialists. Readers following the series will by now have gained some experience in theoretical models and their uses. Previous articles have gradually built up models of objects [1], types [2] and classes [3] in the λ-calculus. Inheritance has been shown to extend both type schemes [4] and implementations [5]. The most recent article combined the type and implementation aspects of inheritance [6], introducing the typed notation that will be used in the rest of this series. The theoretical model has already revealed some interesting differences between the notions of *type* and *class*. We have shown that one group of languages, exemplified by C++ and Java, uses *subtyping* [2, 3, 5] as the basis for type compatibility, whereas another group, exemplified by Smalltalk and Eiffel, uses *subclassing* [3, 4, 5], which is a distinct formal relationship in the *Theory of Classification*.

In this article, we review the whole model so far, to demonstrate more of its formal modelling power. The theoretical model is able to represent the whole spread of object-oriented concepts, such as objects, types, classes, abstract classes and interfaces. It can handle the notion of object creation, class extension through inheritance, type compatibility and interface matching. We shall build up a simple model of a class hierarchy, demonstrating all of these concepts. First, we shall briefly review the elements of the model.

2 SIMPLE OBJECTS AND TYPES

Objects are modelled as simple records, whose fields are values, representing attributes, or functions, representing methods. Types are modelled as record types, whose fields are the corresponding type signatures of the attributes or methods. The following introduces a simple co-ordinate type and an instance of the type:

$$\text{Coord} = \{x : \text{Integer},\ y : \text{Integer}\}$$

$$\text{coord} : \text{Coord} = \{x \mapsto 2,\ y \mapsto 3\}$$

We use capitalisation to indicate type names, and lower case for object names. The above coordinate is a specific instance of Coord at the location (2, 3). To construct such an instance, we can define the constructor-function *makeCoord*:

$$\text{makeCoord} : \text{Integer} \times \text{Integer} \rightarrow \text{Coord}$$
$$= \lambda(a, b : \text{Integer} \times \text{Integer}).\{x \mapsto a,\ y \mapsto b\}$$

This accepts a pair of arguments *a*, *b*, and returns a record, corresponding to a Coord instance. We can then create *coord* from first principles by the constructor-call:

$$\text{coord} = \text{makeCoord}(2, 3) \ \Rightarrow\ \{x \mapsto 2,\ y \mapsto 3\}$$

We access its fields using the "dot" syntax: $coord.x \Rightarrow 2$, $coord.y \Rightarrow 3$. The fields *x*, *y* have fixed values once the object is constructed. We can think of these fields either as constant public attributes, or else we can think of them as "unary methods", functions that accept no argument and access an encapsulated value[1]. We typically take the second view, so that all record fields can be thought of as functions.

3 RECURSIVE OBJECTS AND TYPES

Objects are frequently recursive, since methods may invoke further methods on *self*, the current object. Types are also recursive, since methods may accept or return arguments of the same type as the *self*-type. These two kinds of recursion are related, but independent. We use a standard technique in the λ-calculus to introduce and bind the recursion, called *fixpoint finding* [1] and separate fixpoints are needed at the object-level and the type-level. The technique involves the use of *generators* (also called *functionals*), which are functions of the self-value (or self-type). The following introduces type- and object-generators for a more sophisticated Cartesian point type, which has a recursive *equal* method:

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

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

[1] It is possible to think of simple fields as functions accepting an empty argument, eg: $x : Empty \rightarrow Integer$, and think of all method invocations as supplying the empty value ε automatically, eg: $coord.x(\varepsilon) \Rightarrow 2$.

The type generator *GenPoint* is not yet a record type, but rather a function of σ (the self-type parameter) that returns a record type. We construct the recursive *Point* type by binding σ recursively to the function body, using the fixpoint finder **Y** (see [1] for full details):

$$\begin{aligned}&\text{Point} = \mathbf{Y}[\text{GenPoint}]\\&\Rightarrow \{\text{x} : \text{Integer, y} : \text{Integer, equal} : \text{Point} \rightarrow \text{Boolean}\}\end{aligned}$$

In this way, we build the recursive *Point* type from first principles. Note how, once the definition is complete, we may *unroll Point* (denoted by the evaluation step $\Rightarrow$) to a record type containing recursive reference to the *Point* type.

Likewise, the object generator *genPoint* is not yet an object instance, but rather a function that returns this instance. It accepts two arguments: τ stands for the self-type and *self* for the self-value. To create a point-instance from this generator, we need to supply a suitable type for τ and then bind *self* recursively over the rest of the function body. In the first step, we choose to supply the type *Point*, and the returned result is a generator, a simple function of *self*:

$$\begin{aligned}&\text{genPoint[Point]} = \lambda(\text{self} : \text{Point}).\{\text{x} \mapsto 2,\ \text{y} \mapsto 3,\\&\qquad\text{equal} \mapsto \lambda(\text{p} : \text{Point}).(\text{self.x} = \text{p.x} \wedge \text{self.y} = \text{p.y})\}\end{aligned}$$

In the second step, this simple generator may be recursively fixed using **Y**:

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

to bind *self* over the rest of the body. Note again how, after unrolling, the defined *point* instance contains recursive references to *point*. If we consider that the type *Point* was itself the result of a fixpoint operation, we could equally define *point* in the style:

$$\text{point} = \mathbf{Y}(\text{genPoint}[\mathbf{Y}\ \text{GenPoint}])$$

illustrating how it takes two fixpoints to bind the recursions at object- and type-level in each instance. From a theoretician's point of view, this presents some challenges. Bruce and Mitchell [7] were the first to show that **Y** existed both at both the type- and object-levels, and prove that fixpoint operations converged at infinity, provided that objects were records of functions. (Convergence fails if an object has a field which directly returns *self*; however, we can always convert the *identity* method into a function accepting an empty argument – see the earlier footnote[1]).

4 CONSTRUCTING OBJECT INSTANCES

In the previous article [6], we settled on a style for defining all classes using type- and object-generators, as illustrated above. It turns out that this is most useful, because we can create subclasses by adapting generators [4, 5], in a style that mimics inheritance.

However, the exact process of constructing *distinct* object instances was not fully described. In fact, *genPoint* is a generator for a *specific* instance, with *point.x* $\Rightarrow$ 2 and *point.y* $\Rightarrow$ 3. To make this function more general-purpose, we should force it to accept extra initialisation arguments:

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

Here, *initPoint* is an extended version of *genPoint*, which accepts a type τ, then a pair of Integer arguments *a*, *b*, then the usual value for *self*. This function can be used to construct objects having distinct *x*, *y* field values in the following style:

$$\begin{aligned}&p1 = \mathbf{Y}(\text{initPoint}[\text{Point}](4, 5))\\ &\Rightarrow \{x \mapsto 4,\ y \mapsto 5,\ \text{equal} \mapsto \lambda(p : \text{Point}).(p1.x = p.x \wedge p1.y = p.y)\}\end{aligned}$$

$$\begin{aligned}&p2 = \mathbf{Y}(\text{initPoint}[\text{Point}](6, 7))\\ &\Rightarrow \{x \mapsto 6,\ y \mapsto 7,\ \text{equal} \mapsto \lambda(p : \text{Point}).(p2.x = p.x \wedge p2.y = p.y)\}\end{aligned}$$

Note the order of application in the creation of *p1*: first we supply the precise type, *Point*; then we supply the initialisation values (4, 5). This returns a simple object generator, having the form: λ(*self* : Point).{ … }, which we can fix using **Y** to bind *self* recursively over the rest of the generator body.

If we wanted, we could define a simple object constructor, *makePoint*, in the same style as *makeCoord* (see section 2), which uses the extended function *initPoint* internally:

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

$$\begin{aligned}&p1 = \text{makePoint}(4, 5)\\ &p2 = \text{makePoint}(6, 7)\end{aligned}$$

In this, you can think of **Y** as taking a fixed snap-shot of the flexible type-structure and object-structure represented by the two generators. Constructors in practical object-oriented languages always create a specific instance of a specific type.

5 INVOKING OBJECT METHODS

Having created two distinct instances of *Point*, we can simulate their behaviour in the theory, by evaluating expressions representing method invocations, such as:

$$\begin{aligned}&p1.\text{equal}(p2)\\ &\Rightarrow \lambda(p : \text{Point}).(p1.x = p.x \wedge p1.y = p.y)\ (p2) && \text{- by selecting } \textit{equal} \text{ from } \textit{p1}\\ &\Rightarrow p1.x = p2.x \wedge p1.y = p2.y && \text{- by substituting } \{\textit{p2/p:Point}\}\end{aligned}$$

$\Rightarrow$ 4 = p2.x $\wedge$ p1.y = p2.y	- by selecting *x* from *p1*
$\Rightarrow$ 4 = 6 $\wedge$ p1.y = p2.y	- by selecting *x* from *p2*
$\Rightarrow$ false $\wedge$ p1.y = p2.y	- by *Integer*.=
$\Rightarrow$ false	- by *Boolean*.$\wedge$

The steps shown above are mostly single evaluation steps in the calculus, showing on the right which rule applied in each case. In particular, we use the record selection rule to access the methods for *equal* and *x*, and the function application rule when applying the result of *p1.equal* to the argument *p2*. The details of primitive *Integer* and *Boolean* operations are omitted here. We assume that suitable definitions exist for these types.

6 ROOTING A CLASS HIERARCHY

Within the theory, we can model the notion of a class hierarchy, starting with a root class of all objects. We shall later derive other classes by extending the root class. In this example, we shall assume that the root class only defines a single method, *equal*, and that the default implementation of this method is identity of reference, represented by ==.

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

$$\text{genObject} : \forall(\tau <: \text{GenObject}[\tau]).\ \tau \rightarrow \text{GenObject}[\tau]$$
$$= \lambda(\tau <: \text{GenObject}[\tau]).\ \lambda(\text{self} : \tau).\ \{\ \text{equal} \mapsto \lambda(o : \tau).(\text{self} == o)\}$$

The type generator *GenObject* describes the type-shape of the most general kind of object, saying that it has an *equal* method. The object generator *genObject* provides the default implementation of the *equal* method, and restricts the applicability of this method to arguments of the type λ(o : τ), where τ ranges over the family of types expressed by the constraint: ∀(τ <: *GenObject*[τ]). In earlier articles [3, 4] we described how this expresses exactly the notion of a class, a family of related types. This constraint [8] accepts all record types that have at least an *equal* method, with a type signature σ → *Boolean*.

The *Point* from section 3 above appears to match this pattern. It is possible to derive *Point*'s generators from *Object*'s generators, in the style that mimics the operation of inheritance in object-oriented programming. Recall that ⊕ is union with override, creating an extended record by combining a base record with a record of extra methods [5]:

$$\text{GenPoint} = \lambda\tau.(\textbf{GenObject}[\tau] \cup \{x : \text{Integer},\ y : \text{Integer}\})$$
$$\Rightarrow \lambda\tau.\{\text{equal} : \tau \rightarrow \text{Boolean},\ x : \text{Integer},\ y : \text{Integer}\}$$

$$\text{initPoint} : \forall(\sigma <: \text{GenPoint}[\sigma]).\ (\text{Integer} \times \text{Integer}) \rightarrow \sigma \rightarrow \text{GenPoint}[\sigma]$$
$$= \lambda(\sigma <: \text{GenPoint}[\sigma]).\lambda(a, b : \text{Integer} \times \text{Integer}).\lambda(\text{self} : \sigma).$$

$$\mathbf{genObject[\sigma](self)} \oplus \{x \mapsto a, y \mapsto b, equal \mapsto \lambda(p : \sigma).(self.x = p.x \wedge self.y = p.y)\}$$

$$\Rightarrow \lambda(\sigma <: GenPoint[\sigma]).\lambda(a, b : Integer \times Integer).\lambda(self : \sigma). \{x \mapsto a, y \mapsto b, equal \mapsto \lambda(p : \sigma).(self.x = p.x \wedge self.y = p.y)\}$$

The bold highlights indicate how the old type-generator and object-generator are reused in the definition of the new, extended functions. At the type-level, we added the types of the *x, y* fields. At the object-level, we added implementations for these, and also redefined *equal* to compare the *x, y* field values, rather than test for reference identity. The right-handed preference of ⊕ causes the new version of *equal* to override the default version provided in the root class, when the extra record is combined with the base record [5].

To establish that the *Point*- and *Object*-classes are in a proper hierarchical relationship, we need to show that the *Point*-interface is compatible with the *Object*-interface. This is done formally using the *Classify* rule [4], in which we have to compare the two type-generators for a pointwise subtyping relationship:

$$\forall\tau \,.\, GenPoint[\tau] <: GenObject[\tau]$$

We can show that the relationship holds for a single dummy exemplar type *t*:

$$\begin{array}{ll} \text{for some } t, \quad GenPoint[t] <: GenObject[t] & \\ \Rightarrow \{x : Integer, y : Integer, equal : t \rightarrow Boolean\} & \\ \qquad <: \{equal : t \rightarrow Boolean\} & \\ \Rightarrow true & \text{- by record subtyping [2].} \end{array}$$

and by generalising *t* to all types ∀τ, assert that the relationship holds everywhere. As a result, we maintain that all types belonging to the *Point*-class will also belong to the *Object*-class.

7 INTERMEDIATE ABSTRACT CLASSES

We can also define the notion of an abstract class. This is expressed by a pair of generators in which full type information is given in the type-generator, but some of the implementation information is left undefined in the object-generator. Earlier, we used ⊥ to represent the *undefined* value [1]. In λ-calculus, any expression which simplifies to ⊥ is the formal equivalent of raising an exception in a practical programming language. This is also quite suitable for representing abstract methods, since it is an error to invoke them (instead, you would expect the abstract methods to be overridden in a concrete subclass).

We seek to define a *Shape*-class, the abstract ancestor of geometric shapes, such as *Circle* and *Rectangle*, which provides a concrete *origin* method, indicating its screen coordinates, but an abstract *area* method, for which no implementation can yet be given. Furthermore, we define this class by extension from the root *Object*-class above.

$$\begin{aligned}&\text{GenShape} = \lambda\tau.(\text{GenObject}[\tau] \cup \{\ \textbf{origin : Point, area : Integer}\ \})\\&\Rightarrow \lambda\tau.\{\text{equal} : \tau \to \text{Boolean}, \textbf{origin : Point, area : Integer}\ \}\end{aligned}$$

$$\begin{aligned}&\text{initShape} : \forall(\sigma <: \text{GenShape}[\sigma]).\ \text{Point} \to \sigma \to \text{GenShape}[\sigma]\\&= \lambda(\sigma <: \text{GenShape}[\sigma]).\ \lambda(p : \text{Point}).\ \lambda(\text{self} : \sigma).\\&\qquad \text{genObject}[\sigma](\text{self}) \ \oplus\ \{\ \textbf{equal} \mapsto \boldsymbol{\lambda}\textbf{(s : }\boldsymbol{\sigma}\textbf{).(self.origin.equal(s.origin)),}\\&\qquad\qquad \textbf{origin} \mapsto \textbf{p, area} \mapsto \bot\ \}\end{aligned}$$

$$\begin{aligned}&\Rightarrow \lambda(\sigma <: \text{GenShape}[\sigma]).\ \lambda(p : \text{Point}).\ \lambda(\text{self} : \sigma).\\&\qquad \{\ \textbf{equal} \mapsto \boldsymbol{\lambda}\textbf{(s : }\boldsymbol{\sigma}\textbf{).(self.origin.equal(s.origin)), origin} \mapsto \textbf{p, area} \mapsto \bot\ \}\end{aligned}$$

In the type-generator *GenPoint*, the extra type-information is highlighted in bold. The object-function *initShape* is written in the same style as *initPoint*, with extra initialisation arguments, since we want to be able to supply the initial *Point* coordinate *p* at which a *Shape* is to be located. Given this argument *p*, we can define *origin* to return *p*, and can redefine *equal* to compare the *origins* of two *Shapes* – this in turn uses the *equal* method defined earlier for *Points*. Note the use of ⊥ in the body of the *area* method, indicating that this is so far undefined.

To demonstrate what happens in the model when you try to invoke an abstract method, we exceptionally provide a simple constructor for abstract Shapes, so that we can create an instance and invoke the abstract method (normally, no constructor would be provided, since the class is abstract):

$$\begin{aligned}&\text{makeShape} : \text{Point} \to \text{Shape}\\&= \lambda(p : \text{Point}).\ \mathbf{Y}\ (\text{initShape}\ [\mathbf{Y}\ \text{GenShape}]\ (p))\end{aligned}$$

$$\begin{aligned}&\text{p1} = \text{makePoint}(4, 5) \Rightarrow \ldots && \text{- detail omitted for clarity}\\&\text{s1} = \text{makeShape}(\text{p1}) \Rightarrow &&\\&\ \ \{\ \text{equal} \mapsto \lambda(s : \text{Shape}).(\text{s1.origin.equal(s.origin)}),\ \text{origin} \mapsto \text{p1},\ \text{area} \mapsto \bot\ \} &&\\&\text{s1.origin} \ \Rightarrow\ \text{p1} && \text{- concrete, by selecting } \textit{origin} \text{ in } \textit{s1}\\&\text{s1.area} \ \Rightarrow\ \bot && \text{- abstract, by selecting } \textit{area} \text{ in } \textit{s1}\end{aligned}$$

The result of attempting to select an abstract method is the undefined value ⊥, equivalent to raising an exception.

8 FINAL CONCRETE CLASSES

We now seek to derive a concrete *Rectangle*-class as a subclass of the abstract *Shape*-class, with an implementation of its *area* method. The type-generator declares the extra signatures for the *width* and *height* fields, while the object-function supplies implementations for these, and also provides a concrete implementation for the *area* method, and re-implements the *equal* method to compare *width* and *height*, in addition to *origin*:

$$\begin{aligned}
&\text{GenRectangle} = \lambda\sigma.(\text{GenShape}[\sigma] \cup \{\ \mathbf{width : Integer, height : Integer}\ \})\\
&\Rightarrow \lambda\sigma.\{\text{equal} : \sigma \to \text{Boolean, origin : Point, area : Integer, } \mathbf{width : Integer,}\\
&\qquad\qquad \mathbf{height : Integer}\ \}
\end{aligned}$$

$$\begin{aligned}
&\text{initRectangle} : \forall(\tau <: \text{GenRectangle}[\tau]).\\
&\qquad (\text{Point} \times \text{Integer} \times \text{Integer}) \to \tau \to \text{GenInteger}[\tau]\\
&= \lambda(\tau <: \text{GenRectangle}[\tau]).\ \lambda(\text{p, w, h} : \text{Point} \times \text{Integer} \times \text{Integer}).\ \lambda(\text{self} : \tau).\\
&\qquad \text{initShape}[\tau]\ (\text{p})\ (\text{self})\ \oplus\ \{\ \mathbf{area} \mapsto \mathbf{(self.width \times self.height),}\\
&\qquad\ \ \mathbf{equal} \mapsto \boldsymbol{\lambda}\mathbf{(r : \tau).(self.origin.equal(r.origin)\ \wedge\ self.width = r.width}\\
&\qquad\qquad\qquad\qquad\qquad\qquad \mathbf{\wedge\ self.height = r.height),}\\
&\qquad\ \ \mathbf{width} \mapsto \mathbf{w,\ height} \mapsto \mathbf{h}\ \}
\end{aligned}$$

$$\begin{aligned}
&\Rightarrow \lambda(\tau <: \text{GenRectangle}[\tau]).\ \lambda(\text{p, w, h} : \text{Point} \times \text{Integer} \times \text{Integer}).\ \lambda(\text{self} : \tau).\\
&\qquad \{\ \text{origin} \mapsto \text{p, } \mathbf{area} \mapsto \mathbf{(self.width \times self.height),}\\
&\qquad\ \ \mathbf{equal} \mapsto \boldsymbol{\lambda}\mathbf{(r : \tau).(self.origin.equal(r.origin)\ \wedge\ self.width = r.width}\\
&\qquad\qquad\qquad\qquad\qquad\qquad \mathbf{\wedge\ self.height = r.height),}\\
&\qquad\ \ \mathbf{width} \mapsto \mathbf{w,\ height} \mapsto \mathbf{h}\ \}
\end{aligned}$$

By the usual operation of ⊕, the right-hand, concrete version of *area* is preferred, overriding the abstract version inherited from the *Shape*-class; likewise, the new version of *equal* is preferred, replacing the inherited version. The *Rectangle*-class is now fully defined, with suitable implementations for all of its methods.

In the object-function *initRectangle*, the initialisation-value is of a different type than the argument supplied to *initShape*, since we need to initialise a rectangle with (all of) its *origin, width* and *height* values. Formally, this initialiser is a single value, a tuple of the product type Point × Integer × Integer, whose projections we access implicitly. Notice how the initialiser passed back to *initShape* in the inheritance-expression (on the left of ⊕) is just the *p : Point* value, the first projection from the current initialiser. This models the notion of passing back some construction arguments to a superclass.

We can provide a simple constructor for *Rectangle* objects, in the usual way, which fixes the object- and type-level recursions to construct an instance of this exact type:

$$\begin{aligned}
&\text{makeRectangle} : \text{Point} \times \text{Integer} \times \text{Integer} \to \text{Rectangle}\\
&= \lambda(\text{p, w, h} : \text{Point} \times \text{Integer} \times \text{Integer}).\\
&\qquad \mathbf{Y}\ (\text{initRectangle}\ [\mathbf{Y}\ \text{GenRectangle}]\ (\text{p, w, h}))
\end{aligned}$$

and with this, we may construct a number of distinct *Rectangle* instances:

```
p1 = makePoint(4, 5)  ⇒  …                         - detail omitted for clarity
r1 = makeRectangle(p1, 2, 3) ⇒
  { origin ↦ p1, area ↦ (r1.width × r1.height),
        equal ↦ λ(r : Rectangle).(r1.origin.equals(r.origin) ∧
                        r1.width = r.width ∧ r1.height = r.height),
        width ↦ 2, height ↦ 3 }
r2 = makeRectangle(p1, 6, 7) ⇒
  { origin ↦ p1, area ↦ (r2.width × r2.height),
        equal ↦ λ(r : Rectangle).(r2.origin.equals(r.origin) ∧
                        r2.width = r.width ∧ r2.height = r.height),
        width ↦ 6, height ↦ 7 }
```

The instances *r1, r2 : Rectangle* have different values for their *width* and *height* fields, although they happen to share their *origin* in this example. Also, their *area* and *equal* methods invoke further methods (recursively) on the same instance, as intended.

9 TYPE COMPATIBILITY AND INTERFACE MATCHING

The *Rectangle* instances should be type-compatible with the *Shape*-class interface, and transitively with the *Object*-class interface. Methods defined for these general classes should be type-correct when applied to instances of the specific *Rectangle* type. One way of checking this is to see whether *Rectangle* is included in the type-family of each class [4]. First, we want to see if *Rectangle* is in the class of *Shapes*:

```
Rectangle <: GenShape[Rectangle]
⇒ {equal : Rectangle → Boolean, origin : Point, area : Integer,
          width : Integer, height : Integer}
   <:  {equal : Rectangle → Boolean, origin : Point, area : Integer}
⇒ true,                          - by record subtyping [2];
```

and secondly, whether *Rectangle* is in the class of *Objects*:

```
Rectangle <: GenObject[Rectangle]
⇒ {equal : Rectangle → Boolean, origin : Point, area : Integer,
          width : Integer, height : Integer}
   <:  {equal : Rectangle → Boolean}
⇒ true,                          - by record subtyping [2].
```

In both cases, the answer is yes, because *Rectangle* extends the interface provided by each superclass. In particular, the *equal* method was originally declared in the *Object*-class, with the type: $\forall(\tau <: \mathit{GenObject}[\tau]).\ \tau.\mathit{equal} : \tau \rightarrow \mathit{Boolean}$, so applying this to a *Rectangle* instance simply produces the substitution: $\{\mathit{Rectangle}/\tau\}$ and yields the specifically-typed version: $\mathit{Rectangle.equal} : \mathit{Rectangle} \rightarrow \mathit{Boolean}$. Similarly, the *area* method was originally declared in the *Shape*-class with the type: $\forall(\tau <: \mathit{GenShape}[\tau])$.

$\tau.area$: *Integer*, so applying this to a *Rectangle* instance produces the same substitution: $\{Rectangle/\tau\}$ and yields the specifically-typed version: $Rectangle.area : Rectangle \rightarrow Integer$.

Looking at the unrolled version of the *Rectangle* type above, we can see that the redefined versions of these methods have exactly the same types. So, *Rectangle* matches all the expected superclass interfaces, as intended. These interfaces were defined as part of the type-information associated with a class. However, some practical programming languages allow the definition of interfaces that are not associated with any class.

In the theory, the concept of an independent *interface* may be represented exactly by a type-generator, without a corresponding object-generator. If we wanted to specify (for example) a *Locatable* interface for all object types providing an *origin* method, we could do so using just the type generator:

$$\text{GenLocatable} = \lambda\sigma.\{\text{origin} : \text{Point}\}$$

and then give *Locatable* variables the polymorphic type: $\forall(\tau <: GenLocatable[\tau])$. It is clear that both the *Shape* and *Rectangle* classes satisfy this interface, by the pointwise subtyping condition expected in the Classify rule [4]:

$$\forall\tau\ .\ \text{GenShape}[\tau] <: \text{GenLocatable}[\tau]$$
$$\forall\tau\ .\ \text{GenRectangle}[\tau] <: \text{GenLocatable}[\tau]$$

and the interested reader is encouraged to prove this, using the same kind of strategy as demonstrated in section 6 above.

10 CONCLUSION

This article has focused on two main areas: how to construct specific object instances, and the development of a simple class hierarchy. The overall aim was to demonstrate how the *Theory of Classification* can model a variety of object-oriented concepts, including objects, types, classes, abstract classes and interfaces.

Regarding object construction, we showed how generators can be extended into flexible object-creation functions with initialisation arguments. Furthermore, initialisation values may be passed back to the superclass functions, mimicking the behaviour of real object-oriented languages. From a formal perspective, creating a unique instance of an exact type always involves taking a double fixpoint, one for the type-generator and one for the object-generator.

Regarding class hierarchy development, we gave examples of recognisable classes, with mixtures of default, abstract and concrete methods. Note especially how the type- and implementation-aspects were able to evolve independently, according to need. A class may introduce a new method, may declare an abstract method, or may re-implement an existing method, replacing it with a more appropriate version. All of this is handled uniformly within the theory. Furthermore, we have shown that classes derived by inheritance (using generators) give rise to object constructors, which create objects of exact types (after fixpoints are taken). These exact types match the expected interfaces of their superclasses, and also of separately-declared interfaces, where appropriate. The matching condition (*Classify* [4]) is exactly the same in both cases, demonstrating the economy of the theory.

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_08/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 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

[7] K Bruce and J Mitchell, "PER models of subtyping, recursive types and higher-order polymorphism", *Proc. 19th ACM Symp. Principles of Prog. Langs.*, (1992), 316-327.

[8] 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.

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.

JOURNAL OF OBJECT TECHNOLOGY

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

Vol. 3, No. 7, July-August 2004

The Theory of Classification Part 13: Template Classes and Genericity

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

1 INTRODUCTION

This is the thirteenth article in a regular series on object-oriented type theory for non-specialists. Previous articles have gradually built up models of objects [1], types [2] and classes [3] in the λ-calculus. Inheritance has been shown to extend both type schemes [4] and implementations [5]. The most recent article [6] presented a model of a simple class hierarchy, with a root *Object* class, and various subclasses modelling geometric concepts, including a Cartesian *Point*, an abstract *Shape* class and a concrete *Rectangle* class. The aim was to demonstrate how natural intuitions about generalisation and specialisation could be expressed in the theoretical model, both at the type and implementation levels. Methods were written for abstract classes which also applied in a type-correct way to all classes beneath them in the class hierarchy, such as the *origin* method for *Shapes* [6].

However, abstract classes are not the only way in which generality can be expressed. Some object-oriented languages allow the introduction of type parameters, standing in place of actual types. These are known as *templates* in C++, or *generic parameters* in Ada or Eiffel[1]. The idea is that algorithms may be written without knowing full type information about all the elements involved. The actual types are supplied later, in a process known as *instantiating* the type parameters. In this article, we explore the consequences of adding generic classes to the Theory of Classification. Firstly, we look at some historical notions of polymorphism and type parameters. Secondly, we examine how to incorporate these into the type-level of the theory. Finally, we look at how introducing or instantiating type parameters can be combined with the process of deriving subclasses by inheritance.

[1] At the time of writing, several proposals exist for adding generic types to Java. One is actively being pursued for inclusion in the next revision of the language.

2 TYPE ABSTRACTION AND POLYMORPHISM

It is tempting to think that the object-oriented family of languages was the first to generalise the notion of type. This is incorrect, although it is fair to say that the object-oriented family is the only group of languages to suppose that *systematic* sets of relationships exist between all the types (chiefly through the type hierarchy induced by the *subtype* [2] or *subclass* [4] relationships). The term used to describe generalisation over types is *polymorphism*, coming from the Greek *poly* (many) and *morphe* (form). The earliest strongly-typed programming languages were *monomorphic,* that is, variables were given a single type and could only be bound to values of this type. By contrast, a *polymorphic* language is one in which type constraints are systematically generalised and variables may be bound to values of more than one type. This opens the way to generic styles of programming, in which generic algorithms accept arguments of many different types.

As long ago as the mid-1960s, Strachey and others [7, 8, 9] identified families of types that were sufficiently similar in structure that one could write polymorphic functions acting over them. These were typically the container types, such as *List* and *Stack*, for which functions like *cons*, *append, push* and *pop* could be written irrespective of the type of element they contained. Tennent [10] first proposed the use of type parameters to abstract over the unknown parts of these types, giving rise to the declaration style: *Stack[T]* representing a *Stack* of any element type *T*. So, it was possible to write a polymorphic *push* function that acted upon many different types of *Stack*, by giving it the parameterised type signature:

push (elem : T, stk : Stack[T]) : Stack[T]

Elsewhere, Strachey noted a tendency in programming languages to provide polymorphic functions in another way, simply by adding extra overloaded definitions to existing function names. The operator + might be used in one place to add *Integers* and *Reals*, but then also in another place to concatenate *Strings* and append *Lists*. Strachey therefore distinguished between:

- parametric polymorphism – provided by parameterised functions acting in a systematic way over a variety of types; and
- *ad hoc* polymorphism – provided by defining extra meanings for existing function names in an undisciplined way.

Today, these two forms of polymorphism are respectively known as *genericity* (or *templates*) and *overloading*. Strachey rejected *ad hoc* polymorphism on the grounds that it was not amenable to formal analysis. No semantic correspondence need exist between the different definitions overloaded on a single function name, for example: $x + y == y + x$ is true if $x, y : Integer$, but false if $x, y : String$. On the other hand, systematic parametric polymorphic mechanisms later entered into the designs of functional programming languages, such as ML [11]. In ML, sophisticated type inference is used at runtime to propagate actual type information into type parameters. Ada was the first modular language to introduce generic packages, which had to be instantiated explicitly before use [12], generating a separate compiled image for each instantiation. However, parametric polymorphism existed even before these languages used it systematically. For example, in Pascal, the declaration:

myArray : ARRAY 1..10 OF Integer;

uses the ARRAY OF… special type constructor to build arrays. Such a type constructor can be readily explained as a Tennent-style parameterised polymorphic type:

Array[SubrangeType, ElementType]

in which the *SubrangeType* and *ElementType* are type parameters. Likewise, Pascal's SET OF… constructor can be considered a polymorphic type. Today, parametric polymorphism exists in all the strongly-typed functional languages, including ML, Hope, Miranda, Clean and Haskell. It is present in many object-oriented languages, such as Ada-95, Eiffel and C++, which have explicit parametric typing mechanisms.

3 A FORMAL MODEL OF POLYMORPHISM

Girard [13] and Reynolds [14] are independently credited with having provided the first formal model of polymorphism. They extended the simply-typed λ-calculus to include arguments standing for types, as well as for values. This is the (second-order) polymorphic typed λ-calculus, which we first introduced in the earlier article [3]. The differences between the simply-typed and polymorphic λ-calculus are here explained in more detail.

In the simply-typed λ-calculus, one can write functions whose arguments accept values that have types. For example, a function for constructing a coordinate object can be written[2]:

$\text{makeIntegerCoord} : \text{Integer} \rightarrow \text{Integer} \rightarrow \text{IntegerCoord}$
$= \lambda(a : \text{Integer}).\lambda(b : \text{Integer}).\{x \mapsto a, y \mapsto b\}$

This function accepts two arguments *a* and *b*, both values of the *Integer* type, and returns a record, whose *x* and *y* fields map to these *Integer* values. So, for example, we can create an *IntegerCoord* object at the location (2, 3) by constructing it:

makeIntegerCoord(2)(3) - *ie apply to value 2, then apply to value 3*
$\Rightarrow \{x \mapsto 2, y \mapsto 3\}$

The type of the result is a record type, called *IntegerCoord* in the type signature of the function above. Technically, we should have defined this record type, before using it in the function's type signature, in the style:

IntegerCoord = {x : Integer, y : Integer}

[2] This style is slightly different from the previous article [6]. Here, we introduce each argument separately. Previously, we introduced the pair of Integers as a single argument.

Let us assume now that we want to generalise coordinates so that we can construct real-valued coordinates as well as integral-valued coordinates. Intuitively, we want to abstract over the type of the fields, and replace the hard-wired *Integer* type by a type parameter. The definition of *Coord* must therefore be turned into a type constructor function:

$$\text{Coord} = \lambda\tau.\{x : \tau,\ y : \tau\}$$

which accepts one type parameter, τ. We can create actual coordinate-types by applying this function to different arguments representing the type we desire for the *x* and *y* fields, for example:

$$\text{IntegerCoord} = \text{Coord}[\text{Integer}] = \{x : \text{Integer},\ y : \text{Integer}\}$$
$$\text{RealCoord} = \text{Coord}[\text{Real}] = \{x : \text{Real},\ y : \text{Real}\}$$

It is clear, therefore, that a type constructor function in the λ-calculus is the formal equivalent of a generic type in Ada or Eiffel, and the process of instantiating a generic type is modelled by applying the type constructor function to an actual type argument.

In the polymorphic typed λ-calculus, one may write functions that accept *both* type-arguments *and* value-arguments. The convention is for the type-arguments to be introduced before the value-arguments, mainly because the values might be of one of the introduced types. The polymorphic function for constructing a generic coordinate is written:

$$\forall\tau\ .\ \text{makeCoord} : \tau \to \tau \to \text{Coord}[\tau]$$
$$= \lambda\tau\ .\lambda(a : \tau).\lambda(b : \tau).\{x \mapsto a,\ y \mapsto b\}$$

Notice how the type declaration (the first line, above) is prefixed by the universal quantification ∀τ, meaning "for all types τ". Then, the rest of the declaration says that *makeCoord* accepts two arguments of the τ type and constructs a Coord[τ] from this. Notice also how the implementation (the second line, above) expects the first argument to be a type, and binds this to the type variable τ. Thereafter, the subsequent arguments *a* and *b* are expected to be values of this same τ type, and the result is a record whose *x* and *y* fields map to these values, so the type of the coordinate is clearly dependent on the type of the arguments. In the type signature of *makeCoord*, this type-dependency was expressed in the result-type as: *Coord*[τ], because the record-type of the resulting coordinate is actually generated by applying the type-function *Coord* to whatever type τ was supplied as the first argument. We can create coordinate instances of different types in the following way:

makeCoord[Integer](2)(3) - *ie apply to the Integer type, then to 2, then to 3*
$\Rightarrow \{x \mapsto 2,\ y \mapsto 3\}$
makeCoord[Real](2.1)(3.4) - *ie apply to the Real type, then to 2.1, then to 3.4*
$\Rightarrow \{x \mapsto 2.1,\ y \mapsto 3.4\}$

This demonstrates that *makeCoord* is a polymorphic function in Strachey's original sense, in that it can be applied uniformly to values of different types. It is a parametric-polymorphic function in Tennent's sense, since the unknown part of the coordinate type is modelled using a type parameter.

4 GENERIC OBJECT TYPES

In a similar way, any kind of generic type can be constructed by replacing some parts of a simple type by type parameters. In previous articles [1, 3] we have seen that object types are often recursive, because their methods may accept or return objects of the same type. A recursive, simply-typed *IntegerStack* type can be written:

$$\begin{aligned}\text{IntegerStack} = \mu\sigma.\{&\text{push} : \text{Integer} \rightarrow \sigma,\ \text{pop} : \rightarrow \sigma,\ \text{top} : \rightarrow \text{Integer},\\ &\text{empty} : \rightarrow \text{Boolean},\ \text{size} : \rightarrow \text{Integer}\}\end{aligned}$$

In this, $\mu\sigma$ introduces the recursion in the type and σ stands for the eventual *IntegerStack*, in the body. We may generalise this definition to create a generic *Stack* type constructor if we replace occurrences of *Integer* by a type parameter τ:

$$\begin{aligned}\text{Stack} = \lambda\tau.\mu\sigma.\{&\text{push} : \tau \rightarrow \sigma,\ \text{pop} : \rightarrow \sigma,\ \text{top} : \rightarrow \tau,\\ &\text{empty} : \rightarrow \text{Boolean},\ \text{size} : \rightarrow \text{Integer}\}\end{aligned}$$

Here, $\lambda\tau$ introduces the parameter τ, standing for the element-type, ahead of $\mu\sigma$, which binds the recursion. This generic *Stack* definition has the form of a type function, which expects a type argument: τ and then returns a result, a recursive record type in which τ will be bound to some actual type. To see how this works, we can apply *Stack* to the *Integer* type (*ie* call *Stack* with *Integer* as its actual type argument):

$$\begin{aligned}\text{Stack[Integer]} = \mu\sigma.\{&\text{push} : \text{Integer} \rightarrow \sigma,\ \text{pop} : \rightarrow \sigma,\ \text{top} : \rightarrow \text{Integer},\\ &\text{empty} : \rightarrow \text{Boolean},\ \text{size} : \rightarrow \text{Integer}\}\end{aligned}$$

to see how this yields a recursive record type exactly like *IntegerStack*, above. We could also construct *Stack[Real]*, *Stack[Boolean]* and other types of *Stack*, each with different substitutions for the element-type τ.

Readers who have been following this series will know that the notation $\mu\sigma$ is actually a short-hand for constructing a recursive type from first principles, using a type generator [1]. To define a generic *Stack* from first principles, we need a type generator *GenStack*, which introduces the self-type argument σ as well as the element-type argument τ:

$$\begin{aligned}\text{GenStack} = \lambda\tau.\lambda\sigma.\{&\text{push} : \tau \rightarrow \sigma,\ \text{pop} : \rightarrow \sigma,\ \text{top} : \rightarrow \tau,\\ &\text{empty} : \rightarrow \text{Boolean},\ \text{size} : \rightarrow \text{Integer}\}\end{aligned}$$

GenStack is a type function accepting *two* type arguments. The order of introduction is significant: it is important to introduce the element-type τ before the self-type σ. This is because we want the element-type τ to be in scope when the self-type σ is declared. As a consequence, σ stands for the "whole of the self-type".

The relationship between *GenStack* and the generic *Stack* above is straightforward, but difficult to see at first. The order of parameters expects you to supply an element type first, then to take the fixpoint of the resulting generator. For example we can create a fully-instantiated, recursive *RealStack* type by supplying {*Real*/τ} and then taking the fixpoint:

$$\begin{aligned}\text{RealStack} &= (\mathbf{Y}\ \text{GenStack[Real]})\\ &\Rightarrow \{\text{push} : \text{Real} \to \text{RealStack},\ \text{pop} : \to \text{RealStack},\ \text{top} : \to \text{Real},\\ &\qquad \text{empty} : \to \text{Boolean},\ \text{size} : \to \text{Integer}\}\end{aligned}$$

This works because *GenStack[Real]* yields a generator of the form: λσ.{…} whose fixpoint can then be taken with **Y**, so binding σ recursively over the rest of the record. To create the generic *Stack* type, we somehow need to fix the recursion of σ without replacing the element-type parameter τ with any actual type. The trick is to re-introduce the parameter on the outside of the fixpoint:

$$\begin{aligned}\text{Stack} &= \lambda\tau'\ .\ (\mathbf{Y}\ \text{GenStack}[\tau'])\\ &= \lambda\tau'.\mu\sigma.\{\text{push} : \tau' \to \sigma,\ \text{pop} : \to \sigma,\ \text{top} : \to \tau',\\ &\qquad \text{empty} : \to \text{Boolean},\ \text{size} : \to \text{Integer}\}\end{aligned}$$

and this yields a type constructor function exactly like the *Stack* constructor above. The only difference here is that we supplied the new parameter {τ'/ τ} before taking the fixpoint, instead of some actual type, as in the *RealStack* example.

5 GENERIC CLASSES

The generic *Stack* above may best be described as a *generic type*, but not as a *generic class*. It is only a generic type, because the recursion of the self-type is fixed and the self-type cannot therefore evolve further under inheritance. A generic class may be defined by keeping the self-type open to extension. In this and the following sections, we shall develop a family of *List* classes, looking at how the typeful aspects evolve, but we will skip over the details of their implementations, for simplicity's sake.

Recall that a class is a family of types which all share some common structure, a minimum set of common methods [3, 4]. The class constraint is expressed using a bounded parameter, a type parameter with a restriction on the types which can replace it. For example, if all *Numbers* have at least a *plus* method, we can define a type generator for this record type:

$$\text{GenNumber} = \lambda\sigma.\{\text{plus} : \sigma \to \sigma\}$$

and then express the class of *Numbers* using the generator function in the constraint, which is known as a *function bound*, or *F-bound* [15]:

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

This says, "for all types σ that have at least as many methods as *GenNumber*[σ], σ is the entire class of numbers". This is how to express the membership of an ordinary class.

A generic class can be defined in the same way, using an F-bound. To define the base *List* class in the hierarchy, we first need to declare a type generator:

$$GenList = \lambda\tau\,.\lambda\sigma.\{cons : \tau \to \sigma,\ head : \to \tau,\ tail : \to \sigma,\ equal : \sigma \to Boolean\}$$

This is a type function with two type arguments: τ is the element-type and σ is the self-type, introduced within the scope of τ. The F-bound is constructed in a slightly more elaborate way, which takes the element-type into account:

$$\forall\tau\,.\forall(\sigma <: GenList[\tau][\sigma])\ .\ \sigma$$

This says, "for all element-types τ, and for all list-types σ that have at least as many methods as the type *GenList*[τ][σ], σ is that entire class of lists". The new aspect here is that the F-bound is expressed in terms of both τ and σ. This is because we must apply *GenList* to two type-arguments in order to release the record type in its body.

To validate this new kind of F-bound, describing the membership of a generic class, we shall define an actual list type that we expect to be in the class. To make things a little more difficult, this list type will have an extra *size* method, and a particular (instantiated) element type *Integer*. We shall call this type *IntSzList*, in recognition of the above. Its full type definition is given by:

$$\begin{aligned}IntSzList = \mu\sigma.\{&cons : Integer \to \sigma,\ head : \to Integer,\ tail : \to \sigma,\\ &equal : \sigma \to Boolean,\ size : \to Integer\}\\ \Rightarrow \{&cons : Integer \to IntSzList,\ head : \to Integer,\ tail : \to IntSzList,\\ &equal : IntSzList \to Boolean,\ size : \to Integer\}\end{aligned}$$

The real question is whether *IntSzList* is a member of the generic *List* class. To test this conjecture, we substitute {*Integer*/τ} and {*IntSzList*/σ} in the formula given above. This simplifies to the comparison:

$$\begin{aligned}&IntSzList <: GenList[Integer][IntSzList]\\ \Rightarrow\ &\{cons : Integer \to IntSzList,\ head : \to Integer,\ tail : \to IntSzList,\\ &\qquad equal : IntSzList \to Boolean,\ size : \to Integer\}\\ &\quad <:\ \{cons : Integer \to IntSzList,\ head : \to Integer,\ tail : \to IntSzList\\ &\qquad equal : IntSzList \to Boolean\}\\ \Rightarrow\ &\text{true, by record subtyping.}\end{aligned}$$

thereby demonstrating that *IntSzList* is a member of the class of generic *Lists*.

6 GENERIC INHERITANCE

Is it possible to introduce and adapt generic classes during inheritance? In practical object-oriented languages that combine generic polymorphism with subclassing, you can:

- introduce a subclass with extra type parameters, especially when the need to express genericity first arises in the hierarchy; and
- introduce a subclass with fewer type parameters, by instantiating some of the parent's parameters in the subclass.

The first property is necessary to allow generic classes to exist within the same class hierarchy as ordinary classes. The second property is necessary to allow specific subclass instantiations of generic classes. We shall seek to demonstrate both these properties in the model, by seeing if we can adapt generators for generic classes from other generators.

First, we shall model the introduction of a generic class which inherits from a non-generic parent class. Let us assume that the class hierarchy has a root *Object* class with an *equal* method, as defined by the generator:

$$\text{GenObject} = \lambda\sigma.\{\text{equal} : \sigma \rightarrow \text{Boolean}\}$$
$$\forall(\sigma <: \text{GenObject}[\sigma])\ .\ \sigma \qquad \text{-- \textit{is the class of all Objects}}$$

We wish to introduce our *List* subclass, that is, a family of generic lists with equality. It is relatively easy to define the generator *GenList* for this class by adapting *GenObject*:

$$\text{GenList} = \lambda\tau.\lambda\sigma.(\mathbf{GenObject[\sigma]} \cup \{\ \text{cons} : \tau \rightarrow \sigma,\ \text{head} : \rightarrow \tau,\ \text{tail} : \rightarrow \sigma\})$$
$$\Rightarrow\ \lambda\tau.\lambda\sigma.(\text{equal} : \sigma \rightarrow \text{Boolean},\ \text{cons} : \tau \rightarrow \sigma,\ \text{head} : \rightarrow \tau,\ \text{tail} : \rightarrow \sigma\}$$
$$\forall\tau\ .\forall(\sigma <: \text{GenList}[\tau][\sigma])\ .\ \sigma \qquad \text{-- \textit{is the class of all Lists}}$$

because the new self-type of the list, σ, can be passed back as an argument to the *GenObject* generator (see bold highlight), such that the inherited equal method's self-type is adapted to the new self-type. Because we introduced σ inside the scope of τ, the new self-type implicitly stands for the "whole of the self-type" of the list, including the fact that it contains elements of the τ type. So, we have successfully demonstrated the introduction of a generic class.

To demonstrate the second property, we need to be able to define a subclass (with possibly extra methods) that also instantiates the generic element-type during inheritance. For this, we will introduce the generator *GenIntSzList* for a class of lists of *Integers*, with an additional *size* method. This generator will be defined by adapting the *GenList* generator, which has the extra element-type parameter τ, but the subclass generator will not have this, since it will have been instantiated by the *Integer* type.

$$\text{GenIntSzList} = \lambda\sigma.(\mathbf{GenList[Integer][\sigma]} \cup \{\ \text{size} : \rightarrow \text{Integer}\})$$
$$\Rightarrow\ \lambda\sigma.(\text{equal} : \sigma \rightarrow \text{Boolean},\ \text{cons} : \text{Integer} \rightarrow \sigma,\ \text{head} : \rightarrow \text{Integer},$$
$$\text{tail} : \rightarrow \sigma,\ \text{size} : \rightarrow \text{Integer}\}$$
$$\forall(\sigma <: \text{GenIntSzList}[\sigma])\ .\ \sigma \qquad \text{-- \textit{is the class of all IntSzLists}}$$

The *GenIntSzList* generator clearly only has the self-type parameter, so it is no longer a generator for a generic class. The generic parameter τ was instantiated when *Integer* was supplied as one of the type arguments passed back to the *GenList* generator (see bold highlight), such that the inherited part of the record type has {*Integer*/τ} substituted everywhere. So, we have successfully demonstrated the removal of genericity during the operation of inheritance. This close integration of generic classes with inheritance and with old-fashioned type constructors, like Pascal's SET OF... was first demonstrated by Simons [16, 17], who also showed the important formal property of *confluence*. This property allows the same type to be derived either by instantiating, then inheriting; or by inheriting, then instantiating the parameters, and is an important symmetry property.

7 CONSTRAINED GENERICITY

The template types of C++ are exactly modelled by the universally-quantified type parameters provided in the Girard-Reynolds approach to polymorphism. This is because no restriction is placed on the possible types that might instantiate the parameters: the quantification $\forall\tau$ literally means "for all types τ". In practice, if you supply an unsuitable type for a type parameter in C++, this is not detected until the compiler generates a separate image for the instantiated code, because the compiler cannot check template class declarations.

In Eiffel, it is possible to check at the point of type-substitution whether suitable types are being supplied for a type parameter. This is because Eiffel also allows the expression of constraints on the type parameter, of the form: *SortedList [T → Comparable]*, meaning a *SortedList* of any element type *T* that conforms to the *Comparable* class. This is a more expressive kind of parametric polymorphism, since it allows a compiler to check the code for a generic class, before it is instantiated. All the calls made on variables of parametric type *T* can be checked, because we know that *T* is at least of the *Comparable* type.

Fortunately, the concept of restricting a type parameter to a certain family of types is captured exactly by an F-bound, which we have used so far to constrain the family of types in a class. It is particularly satisfying to find that F-bounds can also be used to model constrained generic types [17, 18]. To define the *SortedList* above, we first need to define a generator for the *Comparable* class, assuming that this supplies the methods *lessThan* and *equal*:

$$\text{GenComparable} = \lambda\sigma.\{\text{lessThan} : \sigma \to \text{Boolean}, \text{equal} : \sigma \to \text{Boolean}\}$$

The generator for a *SortedList* defines the operations that you would expect in such a list, such as an (ordered) *insert* operation, and *first* to extract the element at the head of the list.

$$\text{GenSortedList} = \lambda\tau\,.\lambda\sigma.\{\text{insert} : \tau \to \sigma, \text{first} : \to \tau, \text{rest} : \to \sigma\}$$

Finally, the F-bound can be constructed, to express the family of all those types that belong in the class of *SortedLists*:

$$\forall(\tau <: \text{GenComparable}[\tau]).\forall(\sigma <: \text{GenSortedList}[\tau][\sigma])\ .\ \sigma$$

This says, "for all those element-types τ which have at least the methods of *GenComparable*[τ], and then for all those list-types that have at least the methods of *GenSortedList*[τ][σ], σ is the entire class of sorted lists". This captures exactly Eiffel's notion of a generic class which has a *constrained generic type* parameter.

8 CONCLUSION

We have shown how *parametric polymorphism*, also known as *templates* in C++ and *genericity* in Ada and Eiffel, can be added to the Theory of Classification. We demonstrated how generic types could be created by abstracting over parts of simple types. A generic type is modelled as a type function expecting an actual type argument. We then extended this to model generic classes. A generic class is modelled by first creating a special type function, called a type generator, which has both element-type and self-type parameters. The notion of a generic class is formally all those types which satisfy the F-bound, expressed using the generator. We then showed how the generators for generic classes are well-behaved under inheritance, and can be extended at the same time as introducing, or instantiating the generic type parameters.

F-bounds have been especially useful in this aspect of the Theory of Classification. Cook originally used F-bounds just to model the self-types of classes and explain how these were modified under inheritance [15]. Simons integrated this use of F-bounds with generic classes in his Theory of Classification [16, 17], finding that the same modelling concept could be used everywhere. In a later paper, he also showed how all three of Eiffel's typing mechanisms (conformance, type anchors and constrained genericity) could be modelled by F-bounds, demonstrating the economy and power of the theory [18].

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://ww.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] C Strachey, *Fundamental Concepts of Programming Languages*, Oxford University: Programming Research Group, 1967.

[8] C Strachey, *Varieties of Programming Languages*, Oxford University: Programming Research Group, 1973.

[9] R Milne and C Strachey, *A Theory of Programming Language Semantics*, London: Chapman and Hall, 1976.

[10] R D Tennent, *Principles of Programming Languages*, Prentice Hall, 1981.

[11] R Milner, “A theory of type polymorphism in programming”, in *J. Computer and System Sciences, 17,* (1978), pp. 48-375.

[12] J Ichbiah, J Barnes, J Heliard, B Krieg-Bruckner, O Roubine and B Wichmann, “Rationale and design of the programming language Ada”, in *ACM Sigplan Notices, 14(6),* (1979).

[13] J-Y Girard, “Interprétation fonctionelle et élimination des coupures de l'arithmétique d'ordre supérieur“, *PhD Thesis,* Université Paris VII, (1972).

[14] J Reynolds, “Towards a theory of type structure”, *Proc. Coll. Prog., New York, LNCS 19* (Berlin: Springer Verlag, 1974), pp. 408-425.

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

[16] A J H Simons, “A Language with Class: The Theory of Classification Exemplified in an Object-Oriented Programming Language”, *PhD Thesis*, Department of Computer Science, University of Sheffield (1995).

[17] A J H Simons, “A theory of class”, in *Proc. 3rd Int. Conf. Object-Oriented Info. Sys.*, eds. D Patel, Y Sun and S Patel, (London: Springer Verlag, 1996), pp. 44-56.

[18] A J H Simons, “Rationalising Eiffel's type system”, in *Proc. 18th Conf. Technology of Object- Oriented Languages and Systems (TOOLS Pacific)*, eds. C. Mingins, R. Duke and B. Meyer (Melbourne, 1995), pp. 365-377.

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.

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".

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:

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

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 $\rightarrow$ Label $\rightarrow$ Value $\rightarrow$ Multimap
env-remove : Multimap $\rightarrow$ Label $\rightarrow$ Multimap
env-replace : Multimap $\rightarrow$ Label $\rightarrow$ Value $\rightarrow$ Multimap
env-lookup : Multimap $\rightarrow$ Label $\rightarrow$ 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:

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

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:

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

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

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

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):

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

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:

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

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]:

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

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:

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

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:

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

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

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:

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

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):

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

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* : τ → τ, 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:

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

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:

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

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:

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

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*:

genInitPoint[Point] -- *step 1, supply the type argument*

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

This yields a typed function in which {Point/τ} 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:

$\mathbf{Y}$ (genInitPoint[Point]) -- *step 2, fix the constructor recursion*

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

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:

$\mathbf{Y}$ (genInitPoint[Point]) (2, 3) -- *step 3, supply instance coordinates*

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

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**:

$\mathbf{Y}$ ($\mathbf{Y}$ (genInitPoint[Point]) (2, 3)) -- *step 4, fix the object recursion*

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

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:

$\mathbf{Y}$ ($\mathbf{Y}$ (genInitPoint[$\mathbf{Y}$ 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:

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

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

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*:

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

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:

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

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

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

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

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

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.

JOURNAL OF OBJECT TECHNOLOGY

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

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

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

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

1 INTRODUCTION

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

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

2 FLAVORS AND MIXINS

Flavors [6, 7] was an important early object-oriented language, developed at MIT in the late 1970s. It was the first to introduce *multiple inheritance*, the idea that a child class may have more than one parent class and combine all the inherited features in some principled way. As legend has it, this idea was inspired by the presence of several famous ice-cream parlours in the vicinity of MIT. When visiting these emporia, you could choose your basic vanilla ice-cream and then mix in one of several other flavours[1], such as pistachio or strawberry. By analogy, the root class in the new language was called "Vanilla Flavor" and other classes were extensions of this. The idea of a "mixin" was inspired by the extra sauces and toppings you could add to your ice-cream. A mixin is not itself a whole class, but rather a package of optional features that you can choose to add to a class. It is "mixed in" in the sense that, through the mechanism of inheritance, the mixin's features may become interwoven with the features of the class with which it is combined. Mixins were the first attempt to provide flexible solutions to some of the same problems that are currently addressed using *aspects* in aspect-oriented programming, which are woven together in a similar way.

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

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

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

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

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

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

3 TEMPLATES AND ABSTRACT SUBCLASSES

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

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

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

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

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

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

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

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

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

4 MIXINS VERSUS ABSTRACT SUBCLASSES

The difference between this “abstract subclass” approach and the earlier “mixin” approach is that Flavors defines each mixin component independently, as a freestanding extension. Combining mixins is more like multiple inheritance in C++ (with virtual base classes), in which the programmer derives a new subclass which "mixes" all the components. The abstract subclass approach is more like a wrapper function, which expects to be applied to some base object denoting *super*, and then combines the additional fields with the base fields yielding the subtype directly.

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

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

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

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

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

5 MIXINS AS EXTENSION TYPES

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

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

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

Note that the extension record has an *equal* method accepting the self-type σ, which after unrolling the recursion, is equivalent to Point. This is because σ is recursively bound *outside* the union of base fields and extra fields, to the result of the union. The self-type in the extension record is therefore not independent, but refers to the subtype.

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

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

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

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

6 MIXINS AS EXTENSION GENERATORS

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

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

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

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

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

$$\begin{aligned}&\text{Point} = (\mathbf{Y}\ \text{GenPoint})\\ &\quad\Rightarrow \{\text{identity} : \to \text{Point},\ x :\to \text{Integer},\ y :\to \text{Integer},\\ &\qquad \text{equal} : \text{Point} \to \text{Boolean}\}, \quad \text{after unrolling.}\end{aligned}$$

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

7 BOUND AND FREE MIXINS

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

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

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

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

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

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

$$\begin{aligned}&\text{genLocSquare} = \lambda\text{self}.(\ \text{genSquare(self)} \oplus \text{freePointMixin(self)}\ )\\[6pt] &= \lambda\text{self}.(\ \{\text{side} \mapsto 5,\ \text{equal} \mapsto \lambda s.(\text{self.side} = s.\text{side})\}\ \oplus\\ &\qquad\qquad \{x \mapsto 2,\ y \mapsto 3,\ \text{equal} \mapsto \lambda p.(\text{self}.x = p.x \wedge \text{self}.y = p.y)\}\ )\\[6pt] &= \lambda\text{self}.\{\text{side} \mapsto 5,\ x \mapsto 2,\ y \mapsto 3,\\ &\qquad\qquad \text{equal} \mapsto \lambda p.(\text{self}.x = p.x \wedge \text{self}.y = p.y)\}\end{aligned}$$

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

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

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

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

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

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

$$\begin{aligned}\text{genLocSquare} &= \lambda\text{self}.(\ \text{genSquare(self)} \oplus \\ &\qquad \text{boundPointMixin(self, genSquare(self))}\ ) \\ &= \lambda\text{self}.(\ \{\text{side} \mapsto 5,\ \text{equal} \mapsto \lambda\text{s}.(\text{self.side} = \text{s.side})\} \oplus \{\text{x} \mapsto 2,\ \text{y} \mapsto 3, \\ &\qquad \text{equal} \mapsto \lambda\text{p}.((\ \lambda\text{s}.(\text{self.side} = \text{s.side})\ \text{p}) \wedge \text{self.x} = \text{p.x} \wedge \text{self.y} = \text{p.y})\}\ ) \\ &= \lambda\text{self}.\{\text{side} \mapsto 5,\ \text{x} \mapsto 2,\ \text{y} \mapsto 3, \\ &\qquad \text{equal} \mapsto \lambda\text{p}.(\text{self.side} = \text{p.side} \wedge \text{self.x} = \text{p.x} \wedge \text{self.y} = \text{p.y})\}\end{aligned}$$

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

8 THE SUPERCLASS INTERFACE

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

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

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

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

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

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

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

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

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

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

and from this create the upper bound condition:

$$\forall(\sigma <: \mathrm{GenPointMixin}[\sigma]) \,.\, \forall(\tau <: \mathrm{GenEqual}[\sigma]) \,.\, \mathrm{super} : \tau$$

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

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

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

9 TYPED MIXIN COMBINATION

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

then if $\sigma <: \text{GenLocSquare}[\sigma]$ then it follows that $\sigma <: \text{GenPointMixin}[\sigma]$.

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

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

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

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

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

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

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

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

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

10 CONCLUSION

We started this article by presenting the concept of a mixin. A mixin is a freestanding record of extra fields, intended to be combined with any other object. In some sense, mixins are the primitive building blocks in languages with inheritance-like mechanisms. Bracha and Cook revived interest in mixins when they demonstrated how the models of inheritance in languages as diverse as Smalltalk, Beta and CLOS could all be mapped onto a simpler model based on the composition of mixins [8]. However, the typing they gave was based on simple first-order types, resulting in fragmented self-types after record combination. This is why mixins don't receive so much attention in Java and C++, and one reason why multiple inheritance is less useful in languages like C++ which are based on simple subtyping.

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

REFERENCES

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

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

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

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

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

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

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

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

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

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

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

About the author

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

JOURNAL OF OBJECT TECHNOLOGY

Online at http://www.jot.fm. Published by ETH Zurich, Chair of Software Engineering ©**JOT, 2005**

Vol. 4, No. 1, January-February 2005

The Theory of Classification Part 16: Rules of Extension and the Typing of Inheritance

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

1 INTRODUCTION

This is the sixteenth article in a regular series on object-oriented type theory for non-specialists. Earlier articles have built up λ-calculus models of objects [1], classes [2], inheritance [3, 4] and generic template types [5]. *Classification* describes the way in which typed objects fit into a hierarchy of classes, which nest inside each other [3]. *Inheritance* is the short-hand mechanism for definining a subclass by extension from another class, specifying only the additions and modifications to the base class [4]. Previously, we have modelled the inheritance of *type* [3] and *implementation* [4] and combined both of these in a model of *typed inheritance* [6]. We showed how short-hand inheritance expressions can be simplified to yield a canonical subclass definition that is type compatible with the base superclass from which it was derived. Further aspects of inheritance have included method combination [7], mixin inheritance [8] and inheritance among generic classes [5].

The current article examines the mechanism of inheritance in more detail, looking at the constraints on what may or may not be added to a class during inheritance. Most object-oriented languages have restrictions on the types of overriding methods, to ensure that the resulting subclass is still type compatible with the superclass. This requires more precise rules about the typing of $\oplus$, the inheritance operator. Previously, we thought of $\oplus$ as a polymorphic map override operator that could combine two maps of any types, irrespective of the types of the fields. We now require inheritance to be properly typed, in the second-order F-bounded λ-calculus, so that we can restrict the kinds of extension that are deemed legal. The extended type resulting from inheritance is shown to be an *intersection type* [9].

2 MERGING RECORDS AND RECORD TYPES

In the *Theory of Classification*, we model *objects* as simple records of functions, representing their methods, and *object types* as the corresponding record types, containing the signatures of these methods. This leads to a model of inheritance based on record union with overriding, denoted by the operator $\oplus$. Intuitively, this creates a longer record by combining two shorter records:

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

in which the *derived* result contains the union of all the fields of *base* and *extra*, but fields from *extra* are preferred over any identically-labelled fields from *base* in the result [3]. This right-handed preference of $\oplus$ models the notion of *overriding* in object-oriented languages, in which the *derived* class may replace some of the methods present in the *base* class with redefined versions supplied in the *extra* extension.

So far, we have always used $\oplus$ in a context where the replacement fields have the same types as the original versions. As a consequence, we have been able to construct *Derived* record types using simple set union $\cup$ to merge the two sets of type signatures in the corresponding *Base* and *Extra* type-records:

$$\text{Derived} = \text{Base} \cup \text{Extra}$$

We think of objects as sets of maplets from labels to functions, and object types as corresponding sets of maplets from labels to function types. It is reasonable to think of the merger of two record types as the set union of their respective sets of maplets, since any fields with identical labels will also have identical types.

Merging in the subtyping model

In the subtyping model [10], we must consider the possibility that fields of different types may be merged. This is because the record subtyping rule allows fields to be replaced by subtype fields. In this case, we cannot use $\cup$ to merge the record types. The following is a plausible definition of a record subtype by extension:

$$\text{Vehicle} = \{\text{owner} : \rightarrow \text{Person}, \text{home} : \rightarrow \text{Location}\}$$

$$\begin{aligned}\text{Car} &= \text{Vehicle} \oplus \{\text{home} : \rightarrow \text{Garage}, \text{range} : \text{Litres} \rightarrow \text{Kilometres}\}\\ &= \{\text{owner} : \rightarrow \text{Person}, \text{home} : \rightarrow \text{Garage}, \text{range} : \text{Litres} \rightarrow \text{Kilometres}\}\end{aligned}$$

That is, we wish to obtain the subtyping relationship *Car* <: *Vehicle*. According to the record subtyping rule [10], this requires *Car* to have at least as many fields as *Vehicle* (it has one more) and requires any replacement fields to be subtypes. The field *home* : $\rightarrow$ *Location* is replaced by *home* : $\rightarrow$ *Garage*, so the subtyping condition is only met if

Garage <: *Location*, which is plausible. We use $\oplus$, rather than $\cup$, to combine the two record types, because we wish the right-hand types of any common fields to be retained in the result.

The above model could be used in languages like Java and C++, which are based on types and subtyping. However, in these languages, a replacement method is typically expected to have *exactly* the same type as the method it replaces. In more recent versions of C++, the type of *this* is allowed to be more specific in the result of a replacement method. The overriding rules of Trellis are closest to the subtyping model, allowing both method argument and result types to change in accordance with the function subtyping rule [10].

Merging in the subclassing model

In the subclassing model, we merge generators and type generators, rather than objects and object types [3, 4]. A curious thing happens when merging records according to the (F-bounded) parametric model: the parameters are instantiated or replaced before any record combination occurs. This means that references to different parametric types on the left and right hand sides become unified before record combination. As a result, record combination always merges records whose common fields have the same types. The following illustrates a simple type generator example, in which a (somewhat simplified) *Integer*-class generator is defined by extension from a *Number*-class generator:

$$\text{GenNumber} = \lambda\sigma.\{\text{plus} : \sigma \to \sigma,\ \text{equal} : \sigma \to \text{Boolean}\}$$

$$\begin{aligned}\text{GenInteger} &= \lambda\tau.(\text{GenNumber}[\tau] \cup \{\text{minus} : \tau \to \tau,\ \text{equal} : \tau \to \text{Boolean}\})\\ &= \lambda\tau.\{\text{plus} : \tau \to \tau,\ \text{minus} : \tau \to \tau,\ \text{equal} : \tau \to \text{Boolean}\}\end{aligned}$$

We obtain the subclass relationship: $\forall\tau.\ \mathit{GenInteger}[\tau] <: \mathit{GenNumber}[\tau]$. This is achieved by making sure that *GenInteger* has more fields than *GenNumber* and that the common fields are typed in terms of parameters which can be unified before record combination occurs. In the inner type-record combination expression, *GenNumber*[τ] causes the substitution of $\{\tau/\sigma\}$ in the body of *GenNumber*, such that the record types on both sides of the union $\cup$ have field types which refer to the self-type uniformly as τ; and, in particular, the common field *equal* has the same type: $\tau \to$ *Boolean* on each side of the union.

In fact, it is always the case, in the subclassing model, that self-type parameters are unified before record combination. Likewise, generic type parameters may be unified before combination [5] (see also 5.4 below). So, the simpler union $\cup$ of type-records appears to be all that we need in the subclassing model.

3 SUBSETS, SUBTYPES AND TYPE INTERSECTIONS

Throughout this series, we have been careful to distinguish the notation for the subset $\subseteq$ and subtype <: relationships. This is because the relationship between the two depends on whether we are thinking about the set of values in a concrete type, or the set of type signatures in a record type. These two set-theoretic constructions are different, and they correspond to different subtyping relationships.

Concrete versus abstract representation

The fundamental relationship is that types may be modelled as sets. When we assert that an element is of a particular type, this is equivalent (in the model) to asserting that the element is a member of a particular set:

$x : T \equiv x \in T$ “x is of type T means that x is in the set T”

From this it follows that a subtype may be modelled as a subset:

$S <: T \equiv S \subseteq T$ “S is a subtype of T means that S is a subset of T”

In the universe of types, we want to show that if x : S and S <: T, then x : T also. In the universe of sets, $x \in T$ follows from $x \in S$ and $S \subseteq T$, by the definition of the subset relationship:

$S \subseteq T \equiv \forall x . x \in S \Rightarrow S \in T$ “S is a subset of T means that if x is in S, then x is also in T”

This is the fundamental relationship, which applies to types defined *concretely* as sets. When we move to defining types *abstractly*, in terms of their syntactic signatures, then the relationship is different. A record type with more signatures denotes a subtype of a record type with fewer signatures. For example, if the following record types are defined:

$S = \{plus : Integer \rightarrow Integer, minus : Integer \rightarrow Integer\}$

$T = \{plus : Integer \rightarrow Integer\}$

then it is clear that S is the larger record type and contains the signatures of T, which we express as $T \subseteq S$ in the universe of signature-based types. However, it is also clear that S denotes a subtype of T, because every object that satisfies the interface S will also satisfy the interface T. The record subtyping rule (see [10]) expresses this fact.

Intensional versus extensional definition

There are grounds for confusion here: in one model, we say: $S \subseteq T$; but in the other model, we say: $T \subseteq S$. The difference is that, in the first model, we are comparing sets of values, but in the second model, we are comparing sets of type signatures. To see how these both ultimately reflect the same subtyping relationship, we have to distinguish the *intensional* and *extensional* definitions of a type.

- The *extension* of a type is the enumeration of the set of elements that it contains, for example, the *Boolean* type has the extension: {*false*, *true*}
- The *intension* of a type is the enumeration of the set of properties that characterise the type, for example, the (existentially defined [1]) *Boolean* type has the intension:

$$\begin{aligned}\text{Boolean} \equiv \exists \text{b}.\{&\text{not} : \text{b} \to \text{b},\ \text{and} : \text{b} \times \text{b} \to \text{b},\ \text{or} : \text{b} \times \text{b} \to \text{b},\\ &\text{implies} : \text{b} \times \text{b} \to \text{b}\}\end{aligned}$$

  followed by a set of axioms defining the meanings of these operations.

To unify the concrete and abstract views of a type, it is easiest to imagine the extension of the type, that is, the set of values (or objects) populating the type. This is the usual view adopted in type theoretical treatments. In object-oriented programming, we usually characterise a class intensionally, that is, by its properties (type signatures). From this, we have to imagine the extension of the class, that is, the possible set of objects which could populate it.

Intersection types

Here, we try to establish the relationship between intensional (signature-based) types and extensional (value-based) types. Earlier, we modelled type extension as the union of type-records: *Derived* = *Base* ∪ *Extra*. In terms of sets of signatures, this means that *Base* ⊆ *Derived* and *Extra* ⊆ *Derived*, that is, both *Base* and *Extra* contain a subset of the signatures of *Derived*, which is a longer record type. By the record subtyping rule [10], a longer record type with more field signatures is a subtype. According to this, the direction of the subtyping relationship is *contravariant* with the direction of the signature subsets: *Derived* <: *Base* and also *Derived* <: *Extra*. This is a fundamental property of type hierarchies: the larger the interface, the smaller the set of objects which may satisfy it.

From this, we may reason about the extensions of each type. Instances of the *Derived* type may also be considered instances of the *Base* type (and instances of the *Extra* type), by the subtyping rule of subsumption. So, the extension set of the *Base* type is larger than that of the *Derived* type; likewise the extension set of the *Extra* type is larger than that of the *Derived* type. Since elements of the *Derived* extension are also members of the *Base* and *Extra* extensions, the membership of the *Derived* extension is precisely the *intersection* of the memberships of the *Base* and *Extra* extensions.

For this reason, the kinds of types created by merging signature-based types are sometimes known as *intersection types*. Instead of writing: *Base* ∪ *Extra* (in the world of signatures), we write: *Base* ∧ *Extra* (in the world of sets), to denote the intersection of the *Base* and *Extra* types. Much of the fundamental research on this was done by Compagnoni and Pierce in the mid-1990s [9, 11]. They developed a type system called

"System $F^{\omega}\wedge$", pronounced "System F-omega-meet", a higher-order type system with intersection types.

4 CONSTRAINING THE INHERITANCE FUNCTION

Many object-oriented languages have strict rules about method overriding, during inheritance, because they wish to preserve type compatibility (either subtyping, or subclassing) in the derived type. In C++ or Java, any replacement method must have *exactly* the same type as the original method it replaces. This imposes a constraint on the inheritance function, which we should like to capture in the model. We shall try to capture this constraint in a general enough way that it will apply both to the first-order *subtyping* model of inheritance, as found in Java, and also in the second-order *subclassing* model of inheritance, which is a more appropriate general model for object-oriented programming, in which polymorphic *classes* and simple *types* are actually distinct notions.

The *extend* inheritance function

Inheritance is only well-defined if the *Extra* record provides fields whose types "merge" with the types of the *Base* record. This "merge" condition is expressed as a constraint M between the two record-types in the following F-bounded, second-order definition of the inheritance function *extend*, which we shall now use in place of the earlier unconstrained $\oplus$ map override operator:

$$\begin{aligned}
&\text{extend} : \forall\text{Base}.\ \forall(\text{Extra M Base}).\ \text{Base} \rightarrow \text{Extra} \rightarrow (\text{Base} \wedge \text{Extra})\\
&= \lambda\text{Base}.\ \lambda(\text{Extra M Base}).\ \lambda(\text{base} : \text{Base}).\ \lambda(\text{extra} : \text{Extra}).\\
&\qquad\{\ \text{label} \mapsto \text{value} \mid (\text{label} \in \text{dom(base)} \cup \text{dom(extra)}) \wedge\\
&\qquad\qquad(\text{label} \in \text{dom(extra)} \Rightarrow \text{value} = \text{extra(label)}) \wedge\\
&\qquad\qquad(\text{label} \notin \text{dom(extra)} \Rightarrow \text{value} = \text{base(label)})\ \}
\end{aligned}$$

This definition says that: "*extend* takes two type arguments, *Base* and *Extra*, where *Extra* must satisfy the type-merge condition with *Base*, then two record arguments, *base : Base* and *extra : Extra*, and constructs a result by merging the two records, which has the intersection type (*Base* ∧ *Extra*). The result is a map of label-value pairs, such that the set of labels is the union of the domains of *base* and *extra*, and the values are preferentially taken from *extra*, if the label is present in *extra*, otherwise taken from *base*." (Note that *base(label)* maps to the *value* opposite *label* in the *base* map [4]).

Readers will note that the body of *extend* is identical to the body of ⊕ in earlier articles [4]. These two functions are essentially the same, except that *extend* is now properly-defined in the second-order λ-calculus, with type arguments (*Base* and *Extra*) as well as value arguments (*base* and *extra*). The type arguments were conveniently omitted from the earlier definition of ⊕, which we imagined could be applied directly to two record values. We can retrospectively define the operator ⊕ in terms of *extend*:

$$\forall\beta.\ \forall\varepsilon.\ \oplus_{\beta,\,\varepsilon} = \text{extend}\,[\beta,\varepsilon]$$

This creates a simply-typed version of ⊕ for each pair of records we wish to combine. Really, ⊕ is just a short-hand for *extend* with two types already supplied.

The M type-merge condition

The all-important type-merge condition M is a constraint that restricts the record-types that are allowed to be substituted for the *Extra* type argument. Although this is a rather special condition, constructed for the purpose of typing inheritance, it is syntactically no different from other kinds of restriction, such as the F-bound: $\forall(\tau <: F[\tau])$, which restricts a type τ to be a subtype of some generator expression. Here, we restrict *Extra* to range over those record types whose field-types enter into a particular relationship with the types of the *Base* fields. The constraint M is defined as follows:

$$\begin{aligned}&\forall\text{Base}.\ \forall\text{Extra}.\ \text{Extra M Base} ::= \\ &\qquad \forall\text{label} \in \text{dom(Base)} \cap \text{dom(Extra)}.\ \text{Base(label)} = \text{Extra(label)}\end{aligned}$$

This says: "For all types *Base* and *Extra*, the type-merger condition *Extra* M *Base* is defined as being satisfied if, for all common fields in *Base* and *Extra* with identical labels, the corresponding types are also equal".

For this, we must assume that the notion of "type equality" is well-defined. In full, this might be expressed by a whole set of rules. For the model of inheritance used in the *Theory of Classification*, we require the following kinds of type equality:

$t = t$	-- identity of simple types
$\tau = \tau$	-- identity of type parameters
$(S \times T) = (S \times T)$	-- equality of product types, where $S, T ::= t \mid \tau$
$(S \to T) = (S \to T)$	-- equality of function types, where $S ::= t \mid \tau \mid T \times T$ and $T ::= t \mid \tau$

where t is a simple type, τ is a type parameter, and S, T are metavariables ranging over simple types and type parameters. (Type rules sometimes use metavariables like this to save having to repeat the same rule for simple types and parametric types).

Constrained typed inheritance

The result of *extend* is well-defined if *Extra* M *Base* ("*Extra* merges with *Base*"). This rule constrains inheritance just enough to behave exactly like typed inheritance in Java, but disallows certain other kinds of inheritance For example, the Trellis-style of inheritance in section 2.1 is now ruled out by the type-merge condition, because a field is replaced by a field which has a *different* type. The *Base* and *Extra* records have the types:

$$\text{Base} = \{\text{owner} : \to \text{Person}, \text{home} : \to \text{Location}\}$$
$$\text{Extra} = \{\text{home} : \to \text{Garage}, \text{range} : \text{Litres} \to \text{Kilometres}\}$$

and the common labels in *dom(Base)* $\cap$ *dom(Extra)* $\Rightarrow$ *{home}*. However, when we compare the corresponding types, we find that: *Base(home)* $\Rightarrow$ *Location* and *Extra(home)* $\Rightarrow$ *Garage*. So, we establish that common field-types are not identical: *Base(home)* $\neq$ *Extra(home)*, and therefore that M is not satisfied. To pass the type-merge condition, the *Extra* record would have to redefine the *home* field with the same type: *home* : $\to$ *Location*, as in Java.

By deliberate design, the same type-merger rule allows the kind of unions of type-records we require for the merger of parameterised record types, which are used in the subclassing model of inheritance. Repeating the example from section 2.2:

$$\text{GenNumber} = \lambda\sigma.\{\text{plus} : \sigma \to \sigma, \text{equal} : \sigma \to \text{Boolean}\}$$

$$\text{GenInteger} = \lambda\tau.(\text{GenNumber}[\tau] \cup \{\text{minus} : \tau \to \tau, \text{equal} : \tau \to \text{Boolean}\})$$
$$= \lambda\tau.\{\text{plus} : \tau \to \tau, \text{minus} : \tau \to \tau, \text{equal} : \tau \to \text{Boolean}\}$$

The *Base* and *Extra* records have the following types, after the parameter substitution $\{\tau/\sigma\}$:

$$\text{Base} = \{\text{plus} : \tau \to \tau, \text{equal} : \tau \to \text{Boolean}\}$$
$$\text{Extra} = \{\text{minus} : \tau \to \tau, \text{equal} : \tau \to \text{Boolean}\}$$

and the common labels in *dom(Base)* $\cap$ *dom(Extra)* $\Rightarrow$ *{equal}*. When we compare the corresponding types, we find that: *Base(equal)* $\Rightarrow \tau \to$ *Boolean*, and: *Extra(equal)* $\Rightarrow \tau \to$ *Boolean*. Intuitively, these two types are identical; formally we would need to appeal to the equality of two function-types (see 4.2) based on the identity of the two argument type parameters τ and the identity of the two simple *Boolean* result types. Ultimately, the condition M is satisfied, so this is a legal extension.

5 VARIATIONS ON TYPED INHERITANCE

The standard "reference" model of inheritance consists of the *extend* inheritance function and the M type-merger constraint. This allows a record to be extended only if overriding fields have the same types as in the original fields they replace. The resulting intersection type is always a record-type consisting of the union of the signatures of the *Base* and *Extra* record types, since common fields have the same types. We now consider a number of object-oriented languages and examine how their models of typed inheritance differ from this reference model.

Inheritance in Smalltalk

Smalltalk is not strongly typed. However, certain rules are still observed about inheritance. A method can only override another method if its untyped "signature" is structurally similar, for example, the method *at:put:* always has the structural form:

at: anIndex put: anItem

Any method in a descendant class must have the same name and structural form in order to override this method. So, the "arity" of method arguments and results is always preserved, although nothing can be said about the individual types of each argument or result. Smalltalk can distinguish product types $\sigma \times \tau$ from basic types τ, but apart from this, all basic types (and parameters, considering that *self* has an F-bounded parametric type) are indistinguishable, and so must be considered equivalent. So, for Smalltalk, we should have to redefine the notion of type equality to allow $s = t = \sigma = \tau$ for all simple types s, t and all parameters σ, τ.

Inheritance in Trellis

The type-merger condition above is too restricting to describe inheritance in Trellis. Trellis allows full subtyping in its overriding rules, that is, methods may be replaced by other methods whose arguments have more general types and whose results have more specific types, according to the contravariant and covariant parts of the function subtyping rule. To handle Trellis, we should modify our definition of M:

$$\begin{array}{l}\forall \text{Base}.\ \forall \text{Extra}.\ \text{Extra}\ \text{M}_{\text{Trellis}}\ \text{Base} ::= \\ \qquad \forall \text{label} \in \text{dom(Base)} \cap \text{dom(Extra)}.\ \text{Extra(label)} <: \text{Base(label)}\end{array}$$

This now allows field types in the *Extra* record to be subtypes of common fields in the *Base* record. The resulting intersection type *Base* $\wedge$ *Extra* may contain finer intersections of field types, for example, the extension of *Vehicle* in 2.1:

$$\begin{array}{l}\{\text{owner} : \rightarrow \text{Person, home} : \rightarrow \text{Location}\} \wedge \\ \quad \{\text{home} : \rightarrow \text{Garage, range} : \text{Litres} \rightarrow \text{Kilometres}\} \\ \Rightarrow \{\text{owner} : \rightarrow \text{Person, home} : \rightarrow (\text{Location} \wedge \text{Garage}), \\ \qquad \text{range} : \text{Litres} \rightarrow \text{Kilometres}\} \\ \Rightarrow \{\text{owner} : \rightarrow \text{Person, home} : \rightarrow \text{Garage, range} : \text{Litres} \rightarrow \text{Kilometres}\}\end{array}$$

requires the nested intersection: *Location* $\wedge$ *Garage* = *Garage*. (Constructively, *Garage* is the largest type which is a subtype of both *Garage* and *Location*).

Inheritance in Java and C++

The original type-merger condition describes exactly the constraint on inheritance in Java, in which all replacement methods must have exactly the same types as the methods they replace. This strict equality *nearly* describes the condition in C++, apart from the relaxation that applies to returned self-types. We can express this relaxation as:

$$\begin{array}{l} \forall \text{Base}.\ \forall \text{Extra}.\ \text{Extra}\ \text{M}_{\text{C++}}\ \text{Base} ::= \\ \quad \forall \text{label} \in \text{dom(Base)} \cap \text{dom(Extra)}. \\ \qquad \forall \sigma.\ \text{Base(label)} \neq (\sigma \rightarrow \text{Base}) \Rightarrow \text{Base(label)} = \text{Extra(label)} \wedge \\ \qquad \forall \sigma.\ \text{Base(label)} = (\sigma \rightarrow \text{Base}) \Rightarrow \text{Extra(label)} = (\sigma \rightarrow \text{Base} \wedge \text{Extra}) \end{array}$$

saying that replacement methods must have the identical types *unless* they return the self-type, in which case methods of the function type: $\sigma \rightarrow$ *Base* must be replaced by methods of the function type $\sigma \rightarrow$ (*Base* $\wedge$ *Extra*). The resulting intersection type *Base* $\wedge$ *Extra* will be a subtype of *Base*.

C++ may also have type parameters in its method signatures, if the *template class* mechanism is used (and so will Java, from version 1.5 onward). The notion of type equality must therefore allow for the comparison both of exact types and type parameters (see 4.2).

Inheritance in Eiffel

The overriding rules of Eiffel allow methods to be replaced by methods whose arguments and results are both uniformly specialised. This is not legal within a simple subtyping regime; but Eiffel is not based on the subtyping model of inheritance. Elsewhere, Eiffel implicitly evolves the self-type (the type of *current*) under inheritance and anchors other types to the self-type, especially in binary methods[1] such as the infix “+” method in the *Numeric* class:

infix “+” (arg : **like** *current*) : **like** *current*

Because of this, it is tempting to think of Eiffel as following the F-bounded subclassing model of inheritance, in which “like current” is actually a parametric type σ of the kind: $\forall(\sigma <: \text{GenNumeric}[\sigma])$. Eiffel also has generic and *constrained* generic parameters:

class SortedList [T $\rightarrow$ Comparable] … **end**

which are *exactly* the same notion as F-bounds. Think of the constrained type parameter T as a parametric type: $\forall(\tau <: \text{GenComparable}[\tau])$. So, it makes most sense to think of Eiffel as belonging to the second-order family of languages, along with Smalltalk and Flavors.

This being the case, the reference definition of type-merge is adequate to capture Eiffel’s model of inheritance. You simply have to imagine that *all* Eiffel class-types are in fact parametric types, which are only fixed when object instances are created. The model of inheritance unifies all the type parameters before combining the records. We illustrate this with a parametric version of the example from 2.1 above:

[1] A binary method is one which accepts an argument of the same type as self. It is binary in the sense that it deals with two objects of the same type.

$$\begin{aligned}
&\text{GenVehicle} = \lambda(\pi <: \text{GenPerson}[\pi]).\lambda(\theta <: \text{GenLocation}[\theta]).\\
&\qquad \lambda\sigma.\{\text{owner} : \rightarrow \pi,\ \text{home} : \rightarrow \theta\}
\end{aligned}$$

$$\begin{aligned}
&\text{GenCar} = \lambda(\pi <: \text{GenPerson}[\pi]).\lambda(\theta <: \text{GenGarage}[\theta]).\\
&\qquad \lambda(\rho <: \text{GenLitres}[\rho]).\lambda(\kappa <: \text{GenKilometres}[\kappa]).\\
&\qquad\qquad \lambda\sigma.(\text{GenVehicle}[\pi,\theta] \cup \{\text{home} : \rightarrow \theta,\ \text{range} : \rho \rightarrow \kappa\})
\end{aligned}$$

$$\begin{aligned}
&= \lambda(\pi <: \text{GenPerson}[\pi]).\lambda(\theta <: \text{GenGarage}[\theta]).\\
&\qquad \lambda(\rho <: \text{GenLitres}[\rho]).\lambda(\kappa <: \text{GenKilometres}[\kappa]).\\
&\qquad\qquad \lambda\sigma.\{\text{owner} : \rightarrow \pi,\ \text{home} : \rightarrow \theta,\ \text{range} : \rho \rightarrow \kappa\}
\end{aligned}$$

The subclass generator *GenCar* reintroduces all the parametric types used within the class, and substitutes these new parameters inside the body of the parent generator, through the application: GenVehicle[π,θ] before merging this adapted record with the record of extra methods. So, all common fields have the same types before record combination is computed, and the simple union of signatures is all that is required. The notion of type equality must allow for equality of simple types (such as Eiffel's *Integer* and *Real* types) and equality of type parameters for all class-types (see 4.2). Simons first proposed a unified parametric model of Eiffel's type system in 1995 [12], in which the self-type, anchored types, constrained generic types and ordinary class-types were all modelled using F-bounded parameters.

6 CONCLUSION

In this article, we have revisited the notion of typed inheritance. *The Theory of Classification* describes two models of inheritance, one a first-order model based on subtyping (Java, C++) and the other a second-order model based on subclassing (Smalltalk, Eiffel). Objects are modelled as records, or maps from labels to methods, so inheritance may be modelled as map union with override. Previously, the classical function override operator $\oplus$ was used without any constraints on the types of the records being combined. Here, we have introduced an F-bounded second-order definition of the inheritance function, called *extend*, with a constraint M on the type of extension that may legally be combined with any record.

We showed how, in the reference model, the constraint merely has to ensure that replacement fields have the same types as the fields they replace. This works for Java-style inheritance (first order) and also for Eiffel-style inheritance (second-order) in which field types may be parametric as well as simply-typed. Variations on this allow replacement fields to be subtypes (Trellis), or a mixture of type-equal and subtype fields (C++). One observation emerging from this is that the ability to replace fields with subtype fields is not a frequent requirement in object-oriented languages. The subclassing model of inheritance only requires type-equality, because all the field types are unified prior to combination, whether by parameter unification [3], or instantiation [5]. Simons and Bruce were the first to note the poor match between simple subtyping and natural models of inheritance [13, 14]. This is what originally motivated the *Theory of Classification*.

REFERENCES

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

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

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

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

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

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

[7] 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

[8] A J H Simons, "The theory of classification, Part 15: Mixins and the superclass interface", in *Journal of Object Technology*, vol. 3, no. 10, November-December 2004, pp. 7-18. http://www.jot.fm/issues/issue_2004_11/column1

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

[10] 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

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

[12] A J H Simons, “Rationalising Eiffel’s type system”, *Proc. 18th Conf. Tech. Object-Oriented Lang. and Sys.*, eds. R Duke, C Mingins and B Meyer, (Melbourne : Prentice Hall, 1995), 365-377.

[13] A J H Simons, “A language with class: The theory of classification exemplified in an object-oriented language”, PhD Thesis, University of Sheffield (Sheffield, Department of Computer Science, 1995).

[14] K B Bruce, A Fiech and L Petersen, “Subtyping is not a good “match” for object-oriented languages”, *Proc. European Conf. Obj-Oriented Prog. 1997, pub. LNCS 1241*, (Jyväskylä: Springer Verlag, 1997) 104-127.

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.

JOURNAL OF OBJECT TECHNOLOGY

Online at http://www.jot.fm. Published by ETH Zurich, Chair of Software Engineering ©**JOT, 2005**

Vol. 4, No.2, March-April 2005

The Theory of Classification Part 17: Multiple Inheritance and the Resolution of Inheritance Conflicts

Anthony J H Simons, Department of Computer Science, University of Sheffield, Regent Court, 211 Portobello Street, Sheffield S1 4DP, UK

1 INTRODUCTION

This is the seventeenth article in a regular series on object-oriented theory for non-specialists. Using a second-order λ-calculus model, we have previously modelled the notion of *inheritance* as a short-hand mechanism for defining subclasses by extending superclass definitions. Initially, we considered the inheritance of *type* [1] and *implementation* [2] separately, but later combined both of these in a model of *typed inheritance* [3]. By simplifying the short-hand inheritance expressions, we showed how these are equivalent to canonical class definitions. We also showed how classes derived by inheritance are type compatible with their superclass. Further aspects of inheritance have included method combination [4], mixin inheritance [5] and inheritance among generic classes [6].

Most recently, we re-examined the ⊕ inheritance operator [7], to show how extending a class definition (the *intension* of a class) has the effect of restricting the set of objects that belong to the class (the *extension* of the class). We also added a type constraint to the ⊕ operator, to restrict the types of fields that may legally be combined with a base class to yield a subclass. By varying the form of this constraint, we modelled the typing of inheritance in Java, Eiffel, C++ and Smalltalk. Object-oriented languages vary widely in their policies on inheritance. Some, like Smalltalk, Objective C and Java, only support *single inheritance*, whereby a class may have at most one parent class. Others, like Flavors, Eiffel, CLOS and C++, support multiple inheritance, whereby a class may have possibly many parent classes. In this article, we consider first the theoretical issues raised by combining multiple implementations. Then, we consider what it means for an object to belong to multiple parent classes, defining the notion of *multiple classification*.

2 MULTIPLE RECORD COMBINATION

In the *Theory of Classification*, we model objects as simple records, whose fields map from labels to functions, representing their methods. Inheritance is modelled as a kind of record combination, in which *extra* fields are added to the fields of a *parent* object to yield the desired union of fields in the *child* object:

$$\text{child} = \text{parent} \oplus \text{extra}$$

In the resulting *child*, fields obtained from the *extra* extension may replace similarly-labelled fields obtained from the *parent*, modelling the notion of method overriding. This is assured by the right-handed preference of the $\oplus$ union with override operator [2, 7].

In single inheritance, the *child* obtains all the fields of its *parent*, adding the *extra* fields to these. Intuitively, in multiple inheritance, the *child* must somehow obtain all the fields of multiple parents, adding the *extra* fields to these. We can think of this initially as a kind of multiple record combination, in which the operator $\oplus$ is used many times to combine the fields of the parents and then combine this result with the *extra* fields:

$$\text{child} = \text{father} \oplus \text{mother} \oplus \text{extra}$$

However, this establishes a particular kind of multiple inheritance, in which records are combined in a strict left-to-right order, with fields in the later records overriding similarly-labelled fields of the earlier records. The above expression is evaluated in the order:

$$(\text{father} \oplus \text{mother}) \oplus \text{extra}$$

such that fields from *mother* will possibly override fields in *father*; and then fields in *extra* will possibly override fields in the result of the first combination. This rule is similar to the multiple mixin combination of Flavors [8, 5], and is most like the recency-based superclass ordering rule from LOOPS [9]. The fields that you get in the result of multiple inheritance expressions using $\oplus$ are critically dependent on the order in which you list the parent classes. But, as we shall see below, this is not the only strategy that could be proposed; nor is it perhaps the best strategy.

3 MULTIPLE INHERITANCE POLICIES

Theoretically, we start from the premise that the *child* should obtain at least the union of the fields of its *mother* and *father*. If there is no limit on the number of parent classes, this is a *distributed* union, of the form:

$$\text{parent1} \cup \text{parent2} \cup \text{parent3}$$

The first design choice in any programming language is whether this should be a simple union, or a disjoint union of fields. In a *simple union*, fields with the same labels on both sides are merged, so that only one copy of a field is retained. In a *disjoint union*, fields with the same labels on both sides are considered distinct, so they are not merged.

The simple union policy is usually adopted on the grounds that unique names should be chosen for fields everywhere, since the same name should always refer to the same property. It is perhaps the most theoretically challenging option, since it requires an automatic rule for merging fields. Every time two versions of a field with the same label are encountered, one must be preferred over the other, usually by determining the order of precedence among ancestor classes. Object-oriented languages have proposed many strategies for this, ranging from simple class pre-order [8, 9] to sophisticated topological sorting algorithms for linearising the multiple inheritance graph [10, 11]. Languages with automatic inheritance policies include: Flavors [8], LOOPS [9] and CLOS [11].

The disjoint union policy is usually adopted on the grounds that field-names were ill-chosen and therefore both inherited versions of the field are required in the child. This is the default policy in C++ [12] and for identically-labelled fields introduced at multiple points in Eiffel [13]. This policy gives rise to ambiguity when fields are accessed in the scope of the child. In C++, the two fields must be qualified explicitly by the name of each parent, to resolve the ambiguity, whereas in Eiffel, the programmer must rename one or other of the conflicting fields in the child's inheritance clause. A benefit of the disjoint union policy is that the fields of a child may always be computed locally from the fields of its immediate parents, without worrying about the order in which its more distant ancestors were declared. A disadvantage is that inheritance graphs with fork-join patterns will give rise to unnecessary duplications of fields that were declared in a common ancestor.

For this reason, Eiffel distinguishes the "repeated inheritance" of the same field, via multiple paths from a common ancestor, from the "multiple inheritance" of two distinct fields with the same names [13]. The default rule is to merge the common "repeatedly inherited" field. (The same effect can be achieved in C++ by using "virtual base classes", although this is a less elegant mechanism, required by the language's implementation strategy).

Merging or recombining fields is a complex issue. Over the years, object-oriented languages have proposed many different kinds of automatic rule for combining multiple parent classes. We shall consider a few here, to uncover their advantages and disadvantages.

LOOPS computes a local precedence order for the immediate parent classes. The child's fields take priority over the first parent's fields, which take priority over the second parent's fields, etc., on the basis that a child is "more like" its nearer parents than its distant ones [9]. Similarly, Flavors computes a global order for all classes, by merging all the local precedence orders (as defined above) and any duplicated classes in this list are eliminated, retaining the most recent copy nearest to the front. However, if the local orders are found to be globally inconsistent, for example by requiring class A to precede class B and, at the same time, class B to precede class A, the definition is rejected [8]. Both of these adopt recency-based criteria for choosing which field to retain, rather like Touretzky's "inferential distance" algorithm [14], according to which a class retains the version of the field that was defined closest to it in the inheritance graph. Where the graph forks and joins, all paths leading up to the joins are explored before the path beyond the join is considered.

In CLOS [11], all possible ordered pairs of classes are computed, such that the child precedes each of its parents separately, and parents precede each other pairwise according to their local ordering declared in the child. A topological sorting algorithm then computes a global order for all classes, preserving the pairwise constraints. Again, if no globally consistent order can be found, the definition is rejected [10]. The aim of this precedence-based scheme is to preserve, as much as possible, the local ordering of parents in a class, whether or not the class is included as the ancestor of another class. However, even this sophisticated algorithm for linearising the multiple inheritance graph delivers counter-intuitive results for certain lattices [15]. If inheritance is only a local short-hand for a canonical class definition, the way a child class inherits from its parents should not depend on unexpected interactions between the ordering of its more distant ancestors (see arguments in [13], especially pages 246-250).

4 INHERITANCE CONFLICT RESOLUTION

The *Theory of Classification* adopts a particular "reference model" of multiple inheritance, which captures aspects of both the automatic and explicitly-specified policies on multiple inheritance. The compromise is motivated by examining the nature of inheritance conflicts. An *inheritance conflict* arises when a class obtains the same named method from more than one parent – the resolution of this conflict determines which of these methods (perhaps both) should be incorporated in the child. We distinguish *accidental* and *recombinant* inheritance conflicts.

- An *accidental* conflict arises from inheriting two semantically distinct methods, introduced in two places, which accidentally have the same name.
- A *recombinant* conflict arises from inheriting two semantically related methods, which were originally introduced at a single point in the multiple inheritance graph.

As discussed above, an object-oriented language may prefer to merge conflicting definitions (simple union) or preserve both (disjoint union). Disjoint union really exists to support the resolution of accidental conflicts. We consider it inappropriate for the fundamental theoretical model to have to rectify poor naming conventions! Instead, we assume that accidental conflicts can be resolved by renaming one of the conflicting methods throughout, in the class hierarchy (here, perhaps better called a *heterarchy*, since it is a full directed, acyclic graph). By removing accidental conflicts, our model only has to provide for the resolution of recombinant conflicts, by simple union. But this is harder than at first it seems. We do not simply want to achieve Eiffel's "repeated inheritance", but also want to recognise that the common field may have been redefined in either, or both branches leading to the parents. Semantically, this is still the "same" operation (albeit in a refined form).

Ideally, a child class should be a deterministic synthesis of the most specific aspects of its parent classes, without undue prejudice to either parent. This is what is wrong with existing automatic inheritance schemes: they force the programmer to prioritise one whole parent class before the other, whereas what we really desire is a scheme that prioritises individual methods, by recency of their definition. However, we think it is inappropriate for objects to have to reason about the points at which methods were introduced in the inheritance graph, since this mitigates against the "locality" of inheritance expressions. The following cases may therefore be distinguished:

- the father and mother classes may have no fields in common; in which case multiple inheritance should lead to the straightforward concatenation of their fields;
- the father and mother classes may have some commonly-labelled fields, which are pairwise identical, since they are inherited from a common ancestor; in which case only one copy of each of these should be retained;
- the father and mother classes may have commonly-labelled fields which are *not* pariwise identical, due to further specialisation in one or other parent since their introduction; in which case automatic selection is impossible.

In the last case, even though the desire is to select the most specific redefinition of a method, a system that only has local knowledge of the immediate parents cannot detect which of the versions of this method was redefined most recently. Instead, the programmer must specify explicitly which method should be retained (sometimes a combination of both).

5 SYMMETRICAL COMBINATION

A variant of the $\oplus$ operator is defined below to allow the construction of multiple inheritance expressions. The new *symmetrical* record combination operator is written $\otimes$ and its specific character is that it treats both its left- and right-hand operands fairly, rather than preferring its right-hand operand, like $\oplus$. It is defined to obey the three principles described above in section 4. Where it can determine which field to select, it does so; and where it cannot, it leaves the result of the combination undefined, so that the programmer may later choose explicitly which field to include in the child class.

The symmetrical operator $\otimes$ is actually a short-hand for a typed second-order polymorphic function called *merge*, designed for merging two parent classes fairly:

```
merge : $\forall$Father. $\forall$(Mother M Father). Father $\rightarrow$ Mother $\rightarrow$ (Father $\wedge$ Mother)
= $\lambda$Father. $\lambda$(Mother M Father). $\lambda$(father : Father). $\lambda$(mother : Mother).
  { label $\mapsto$ value | (label $\in$ dom(father) $\cup$ dom(mother)) $\wedge$
          (label $\in$ dom(father) $\wedge$ label $\in$ dom(mother) $\Rightarrow$
                  (father(label) = mother(label) $\Rightarrow$ value = father(label)) $\wedge$
                  (father(label) $\neq$ mother(label) $\Rightarrow$ value = $\perp$)) $\wedge$
          (label $\in$ dom(father) $\wedge$ label $\notin$ dom(mother) $\Rightarrow$
                  value = father(label)) $\wedge$
          (label $\notin$ dom(father) $\wedge$ label $\in$ dom(mother) $\Rightarrow$
                  value = mother(label))  }
```

This says that "two records *father* and *mother* of the respective types *Father* and *Mother* may be merged, if these types satisfy a merging type-constraint M. The result of the merger is a map containing the union of fields from both parents, in which fields unique to the *father* or to the *mother* are inherited unchanged, but fields common to both *father* and *mother* are tested to see if their values are identical. If they are, then one copy is retained (the father's), but if they are not, then the result of the merger is undefined." In the above definition, $\perp$ denotes the undefined value and function-call expressions like: *father(label)* denote the *value* stored opposite the *label* in the *father* object, which is a map from labels to values. This definition uses the type constraint M that was introduced in the previous article [7], which says that two record types may be merged if their common fields have the same types. This is sufficient for second-order polymorphic typed inheritance, adopted in the *Theory of Classification* [7, 1].

We can now define the symmetrical operator $\otimes$ in terms of *merge*:

$\forall\beta.\ \forall\varepsilon.\ \otimes_{\beta,\varepsilon} =$ merge $[\beta,\varepsilon]$

This creates a simply-typed version of $\otimes$ for each pair of records we wish to combine. Really, $\otimes$ is just a short-hand for *merge* with two types already supplied. To see how $\otimes$ works, we will seek to construct a *point3D* object by combining a two-dimensional *point* object with *zcoord*, a mixin object representing the third dimensional coordinate. Initially, we shall just observe the operation of $\otimes$ in isolation:

point = μself.{x $\mapsto$ 2, y $\mapsto$ 3, equal $\mapsto$ λp.(self.x = p.x $\wedge$ self.y = p.y}

zcoord = μthis.{z $\mapsto$ 5, equal $\mapsto$ λq.(this.z = q.z)}

point3D = point $\otimes$ zcoord $\Rightarrow$ {x $\mapsto$ 2, y $\mapsto$ 3, z $\mapsto$ 5, equal $\mapsto$ $\perp$}

In this initial example, *point3D* is constructed by merging the definitions of the two parent objects, *point* and *zcoord*. The result contains copies of the unique methods inherited from both parents, but the *equal* method was defined in both *point* and *zcoord*, so by the definition of $\otimes$, the body of *equal* is undefined. This is because we cannot automatically determine which version of the method we should inherit (we assume by convention that *equal* was declared at a single point in the inheritance graph, and is intended to stand everywhere for the same semantic operation; although the calculus cannot determine this).

Now, the example was deliberately chosen to illustrate a further aspect of automatic multiple inheritance that is not usually considered in object-oriented languages. It would in fact be a mistake to prefer one version of the *equal* method over the other; what we really desire is to inherit *both* parent methods, suitably combined. We may use method combination [4], of the kind used to explain super-method invocation, to achieve this:

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

$$\text{genZcoord} = \lambda\text{this}.\{z \mapsto 5,\ \text{equal} \mapsto \lambda q.(\text{this}.z = q.z)\}$$

$$\begin{aligned}\text{genPoint3D} = \lambda\text{me}.(&\lambda\text{father}.\ \lambda\text{mother}.\mathbf{(}\text{father} \otimes \text{mother} \\ &\quad \oplus \{\text{equal} \mapsto \lambda r.(\text{father.equal}(r) \wedge \text{mother.equal}(r))\ \}\ \mathbf{)} \\ &\text{genPoint(me) genZcoord(me)}\ ) \\ \Rightarrow\ \lambda\text{me}.\{&x \mapsto 2,\ y \mapsto 3,\ z \mapsto 5, \\ &\text{equal} \mapsto \lambda r.(\text{me}.x = r.x \wedge \text{me}.y = r.y \wedge \text{me}.z = r.z)\}\end{aligned}$$

The revised example uses generators instead of objects, because we wish to unify self-reference in the manner explained in the earlier article [2]. The two parent objects are created using *genPoint(me)* and *genZcoord(me)*, where *me* is the self-reference introduced by *genPoint3D*. These parent objects are bound internally to the variables *father* and *mother*, in exactly the same way that we bound an inherited object to *super* in [2]. There is no reason why we should not have many super-objects in multiple inheritance; and this is what opens the way to novel kinds of method combination. The multiple inheritance expression is resolved by first combining *father* ⊗ *mother*. Since this yields an undefined body for the *equal* method, the child object now specifies explicitly how to combine the two inherited versions: it is the logical-and of the *father's* and *mother's* implementations for *equal*. The new version of *equal* is given in a record of additional methods, to be added to the merger of the parents, using the usual ⊕ operator. After combination, it will override the undefined placeholder for *equal*, yielding the desired version of *equal* for a 3D point.

The notion of multiple *super*-objects has not been used before in any mainstream object-oriented language. It is powerful enough to model any kind of explicit method recombination, whether preferring the left-hand or right-hand version, or, as in this case, creating a fusion of both. In C++, you can get a similar effect by defining a method which explicitly calls both parent methods, qualifying these by their owning classes. In Eiffel, the same effect may be achieved by a combination of renaming and redefining. There is no limit on the number of *super*-objects, since ⊗ may be used to combine any number of parents fairly, whose common methods may be accepted, refused or combined in all possible ways in the child.

6 MULTIPLE CLASSIFICATION

Turning now to the issue of types, intuitively the type of the child object must be compatible with the types of each of its parents. In the first-order subtyping model (c.f. Java, C++), we may express this as a dual subtyping condition, which yields an interesting result in terms of type intersections:

$$(\text{Child} <: \text{Father}) \wedge (\text{Child} <: \text{Mother}) \Rightarrow \text{Child} <: (\text{Father} \wedge \text{Mother})$$

“If the *Child* is a subtype of the *Father* and also of the *Mother*, then the *Child* is a subtype of the intersection[1] of the *Father* and *Mother* types.” Type intersections were introduced in the previous article [7]. We showed how merging two object types results in a new type whose extension-set is the intersection of the extension-sets of the original two types. The extension of the *Child* type is a subset of each of its parents’ extensions, which fits nicely with the idea that all the *Child’s* instances should also be considered as belonging to the *Father’s* and *Mother’s* types.

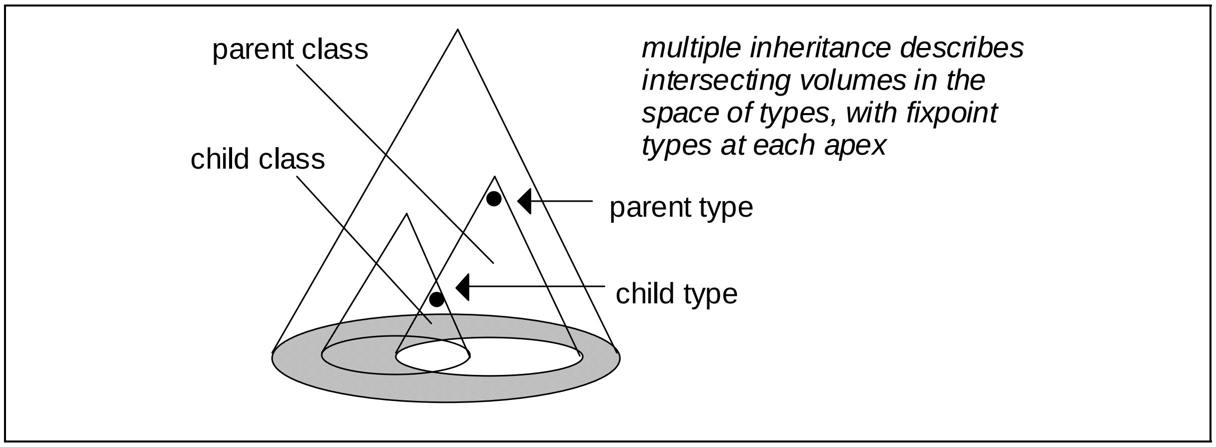

parent class
child class
multiple inheritance describes intersecting volumes in the space of types, with fixpoint types at each apex
parent type
child type

Figure 1: Multiple inheritance describes intersecting volumes

In the more sophisticated second-order parametric typing model [1, 3] (c.f. Eiffel, Smalltalk) there is a similar result which we can express using type generators:

$$\forall\tau\ .\ (\tau <: \text{GenFather}[\tau]) \wedge (\tau <: \text{GenMother}[\tau]) \Rightarrow \\ \tau <: (\text{GenFather}[\tau] \wedge \text{GenMother}[\tau])$$

“If the self-type τ is a pointwise subtype of all types created using the *GenFather* generator and also of all types created using the *GenMother* generator, then it is a pointwise subtype of all intersection types created simultaneously from both generators.”

[1] The symbol ∧ is overloaded here to mean logical-and and type intersection. An intersection type contains the union of the fields of the two record type arguments.

The intersection type in the result is basically like the merger of two record types, in which τ has been replaced in turn by each self-type that satisfies the F-bound of both parents' generators. The constrained type expression: $\forall\tau <: \mathit{GenFather}[\tau] \wedge \mathit{GenMother}[\tau]$ is a new kind of F-bound, describing a family of types that occupy the intersection of the volumes described separately by each of the parent generators. Regular readers will recall that such a "space of types" is equivalent to the notion of a *class* in the *Theory of Classification*.

We can visualise this in figure 1. Here, the two parent classes are illustrated as intersecting cones. The volume where they intersect is the resulting child class you obtain by merging the two parents by multiple inheritance. (If the child were to add further methods, it would form a slightly smaller cone nesting inside the intersection volume). While polymorphic classes are represented by volumes, exact simple types are represented by points. If we recall that a class is a family of structurally-similar types, then all the exact types within a given volume satisfy the F-bound of the related generator and so possess at least the set of methods described in the generator's body. A type which resides inside the intersection volume therefore possesses at least the methods of both generators.

In figure 1, the point at the apex of a cone represents the least type that is still a member of that class. Mathematically, this simple type is the fixpoint of the generator used to describe the class. For multiple parent classes, we may describe this as:

$$\text{Father} = \mathbf{Y}\ \text{GenFather}, \quad \text{Mother} = \mathbf{Y}\ \text{GenMother}$$

but what is the least type that sits just inside the intersection volume (see figure 1)? This is the least type of the child class. Mathematically, we may create this *Child* type by merging the generators of the two parents and then taking the fixpoint of the result:

$$\text{Child} = \mathbf{Y}\ \lambda\tau.(\text{GenFather}[\tau] \wedge \text{GenMother}[\tau])$$

The least *Child* type is exactly the fixpoint of the type generator intersections (extensional definition). In other words, this is the fixpoint of the generator obtained by taking the union of the fields of both parent generators (intensional definition), after unifying the self-type τ. The *Child* type satisfies the F-bounds for each of its parent classes:

$$\text{Child} <: \text{GenFather}[\text{Child}], \ \ \text{Child} <: \text{GenMother}[\text{Child}]$$

and the least *Child* is exactly equivalent to the type intersection:

$$\text{Child} = \text{GenFather}[\text{Child}] \wedge \text{GenMother}[\text{Child}]$$

by the fixpoint theorem. So, all the formal properties that were established in earlier articles for classes nesting in a single classification hierarchy [1, 3] also apply in a uniform way to overlapping classes created by multiple inheritance. It is pertinent to describe this notion as *multiple classification*, the idea that an object may have a type belonging simultaneously to more than one overlapping class.

7 CONCLUSION

In this article, we have developed a flexible formal model for multiple inheritance. By comparing existing programming language models for resolving inheritance conflicts automatically and then contrasting these with models that rely on explicit resolution by the programmer, we came up with a compromise that resolves fork-join "repeated inheritance" automatically, but relies on a form of explicit super-method combination to resolve other cases where the methods to be recombined have been redefined at some point in the inheritance graph. Reasoning globally about the points at which methods are introduced could also resolve this automatically, but we considered this inappropriate, since it conflicts with the view that inheritance should be a local short-hand for defining classes incrementally by extension from its immediate parents. We excluded accidental name conflicts from the formal model, on the grounds that these could be solved simply by renaming one or other method systematically in the inheritance graph.

Multiple inheritance is different from ordinary inheritance, because it involves symmetrical merging as well as asymmetric extension. To merge multiple parent classes fairly, a different combination operator $\otimes$ was defined, which treats both of its operands symmetrically. It constructs records containing the union of distinctly-labelled fields, merges identically-labelled fields that map to identical values, and declares the result of the merger to be undefined otherwise. This is the only sensible automatic choice, given that the programmer might wish to retain the left-hand, right-hand, or possibly a fusion of both versions of the method in the resulting child. The latter option has not been treated before in conflict-resolution schemes. The operator $\otimes$ was defined as a set of simply-typed operators created by instantiating a polymorphic typed function *merge*, in the same way that $\oplus$ was defined out of *extend* in the previous article [7].

Finally, it was shown that objects created by simultaneous extension from several parents have types which belong to multiple classes, in the *Theory of Classification*. The notion of multiple classification was visualised as creating intersections in the space of types, satisfying natural intuitions about belonging to more than one class.

REFERENCES

[1] 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

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

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

[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 15: Mixins and the superclass interface”, in *Journal of Object Technology*, vol. 3, no. 10, November-December 2004, pp. 7-18. http://www.jot.fm/issues/issue_2004_11/column1

[6] 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

[7] A J H Simons, “The theory of classification, part 16: Rules of extension and the typing of inheritance”, in *Journal of Object Technology*, vol. 4, no. 1, January-February 2005, pp. 13-25. http://www.jot.fm/issues/issue_2005_01/column2

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

[9] D Bobrow and M Stefik, *The LOOPS Manual* (Palo Alto: Xerox PARC, 1983).

[10] D Bobrow, L DeMichiel, R Gabriel, S Keene, G Kiczales and D Moon, *Common Lisp Object System Specification, X3J13 Document 88-002R*, June, 1988.

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

[12] B Stroustrup, "Multiple inheritance for C++", *Proc. European Users' Group Conf.*, (Helsinki, 1987).

[13] B Meyer, *Object-Oriented Software Construction, 1st ed.*, (Wokingham: Prentice-Hall, 1988). Note – cited page numbers refer to this earlier edition, rather than the later, greatly enlarged edition.

[14] D Touretzky, *The Mathematics of Inheritance Systems* (Palo Alto: Morgan Kaufmann, 1986).

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.

JOURNAL OF OBJECT TECHNOLOGY

Online at http://www.jot.fm. Published by ETH Zurich, Chair of Software Engineering ©**JOT, 2005**

Vol. 4, No. 4, May–June 2005

The Theory of Classification Part 18: Polymorphism through the Looking Glass

Anthony J H Simons, Department of Computer Science, University of Sheffield

1 INTRODUCTION

In this, the eighteenth article in a regular series on object-oriented type theory, we look at how object-oriented languages might evolve in the future, given that the formal notion of *class* is now better understood than at the outset. Object-oriented languages were the first family to suppose that there might be systematic sets of relationships between all the program data types and use this as the basis for a kind of type compatibility. However, the early formal models chosen were based on simple types and subtyping [1] and struggled in practice to support all the obvious, systematic relationships that programmers intuitively recognised [2]. For a while, objects were thought to have *class* and *type* independently, where *class* was demoted to a mere implementation construct. Later, it was realised that the notion of *class* is also a typeful construct that requires at least a bounded second-order λ-calculus model to explain it [3]. We have developed this model in the *Theory of Classification*, showing how it deals properly with typed inheritance [4, 5] and generic types [6] in a consistent framework.

However, current object-oriented languages fall short of what is actually possible in a language that *really* supports the notion of class. The majority still treat classes for the most part as if they were the same thing as simple types, and it only becomes clear that something more sophisticated is intended when dynamic binding in these languages is examined, showing dispatching behaviour equivalent to higher-order functions [7]. The additional template mechanisms of C++ and Java (from version 1.5) are intended partly to compensate for the lack of expressiveness caused by treating classes as simple types. But do we really need all these separate typing mechanisms? What would a language look like that consistently supported the higher-order notion of *class* throughout?

2 THE HALFWAY HOUSE

In the very first article of this series [8], we described three increasingly more flexible kinds of plug-in type compatibility, in the context of supplying a component to match an interface:

- correspondence: the component is identical in type and its behaviour exactly matches the expectations made of it when calls are made through the interface;
- subtyping: the component is a more specific type, but behaves exactly like the more general expectations when calls are made through the interface;
- subclassing: the component is a more specific type and behaves in ways that exceed the more general expectations when calls are made through the interface.

An example of *correspondence* is the strong monomorphic typing exhibited in languages like Pascal or Modula-2, in which every object is of a single type and may only be passed to variables of exactly the same type. Pascal's strong *name equivalence* rule means that even structurally equivalent types are considered distinct, if their type names are distinct (in contrast to C++'s *typedefs*, which are only aliases for the base type).

An example of *subtyping* is where a subrange object is coerced to a base type variable, so that the base type's function may be executed, such as where two *SmallInt* objects are passed to an *Integer plus* function and the result is returned as an *Integer*. The function originally expected *Integer*s, but could handle subtypes of *Integer* and convert them. Note that no dynamic binding is implied or required. Also, a simply-typed first-order calculus (with subtyping) is adequate to explain this behaviour.

An example of *subclassing* is where the functions of the interface are systematically replaced by functions appropriate to the new type, such as where a *Numeric* type's abstract functions *plus, minus, times* and *divide*, are replaced by retyped versions for a *Complex* type. Rather than coerce a *Complex* object to a *Numeric*, the call to *plus* through *Numeric* executes the replacement *Complex plus* function. This could be achieved through dynamic binding; or alternatively through template instantiation (in which the parameter *Numeric* is replaced throughout by an actual *Complex* type), requiring at least a second-order calculus.

What are the important differences between the simple subtyping and subclassing approaches? In the subtyping approach, the *Integer plus* function treats its *SmallInt* arguments exactly as if they were plain *Integers*. It returns a result of the general type *Integer* and does not know or care whether the result is still in the range of a *SmallInt*. On the other hand, in the subclassing approach, there is an obligation to propagate type information about the actual argument and result types of *Complex's plus* back to the call-site. Whereas the interface expected a *Numeric*, once this was bound to a *Complex* number, the second argument was also forced to be a *Complex* number. Furthermore, the result-type, which was formerly *Numeric*, is now known also to be of the *Complex* type. This means that the caller of *plus* must know how to deal with more specific types than originally specified in the interface. From our point of view, this is exciting stuff, in the true spirit of classification, and something worth exploiting in the design of object-oriented languages.

However, the majority of languages only practise a halfway-house approach, which is *subtyping* with *dynamic binding*. This is similar to subtyping, except that the subtype may provide a replacement function that is executed instead. Recalling the earlier example, this is like the *SmallInt* type providing its own version of the *plus* function which wraps the result back into the *SmallInt* range. Syntactically, the result is acceptable as an *Integer*, but semantically it may yield different results from the original *Integer plus* function (when wrap-around occurs). From the type perspective, we still only know that the result is of the *Integer* type (rather than *SmallInt*) because there is no way of propagating type information about the actual arguments through to the result of the function. So, we have a situation where more specific functions are executed, but externally we cannot see that their type has changed. This gives rise to the phenomenon of “type loss” in C++ and Java, requiring corrective use of type downcasting to recover the most specific types of returned objects [2].

3 POLYMORPHISM REVISITED

Stopping at the halfway house constitutes a failure of nerve in the design of object-oriented languages. At the heart of this problem is the inability to distinguish the notions of *class* and *type* in the syntax of programming languages. If an object-oriented language implemented this distinction properly, a programmer should never have to perform type downcasting, but the language could always recover the most specific types of returned objects for itself. To make this distinction absolutely clear, in the *Theory of Classification*:

- a *type* always refers to a simple monomorphic type, a first-order construct;
- a *class* always refers to a polymorphic type, a second-order (or higher) construct.

As stated previously [8], the term *polymorphism* has less to do with the dynamic binding of methods and properly describes the generalised types of variables that may receive values of more than one type. In conventional programming languages, we consider that type constructors, such as *Stack[T]* or *Map[K,V]* are polymorphic, because they contain type variables standing for possibly many types, and may be adapted to specific types by parameter instantiation. In object-oriented languages, we also consider that a variable of “class-type” is polymorphic and can be made to receive actual objects of possibly many types, where these are restricted by the class hierarchy to be of some “subclass-type” of the target variable. These polymorphic mechanisms seem on the surface to be quite different, but they are fundamentally the same.

In the λ-calculus, polymorphism always requires a type parameter, standing for the generalised type; and when a polymorphic variable binds to a specific type, this type is propagated into the parameter, throughout the whole parameterised expression. The formal model therefore brings together the notions of class-based polymorphism and template-based polymorphism. In earlier articles [2, 3], we deliberately drew out the similarity between classical Girard-Reynolds [9, 10] univeral polymorphism:

$$\forall\tau\ .\ \text{identity} : \tau \to \tau$$

$$\forall\tau\ .\ \text{insert} : \tau \times \text{List}[\tau] \to \text{List}[\tau]$$

in which you could give functions truly generalised types (where τ ranges over absolutely any type) and Cook et al.'s [11, 12] function bounded polymorphism:

$$\forall(\tau <: \text{GenNumeric}[\tau])\ .\ \text{plus} : \tau \times \tau \to \tau$$

$$\forall(\tau <: \text{GenComparable}[\tau])\ .\ \text{insert} : \tau \times \text{SortedList}[\tau] \to \text{SortedList}[\tau]$$

in which you could give functions class-types (where τ ranges over *only* those types which have at least the functions specified in the interface of the bounding generator function). F-bounded polymorphism is more general than universal polymorphism (since you can type more things using F-bounds, for example you can type *SortedLists* of *Comparable* elements with F-bounds, whereas you can only type plain *Lists* of universal elements, without them). This can be shown formally by recasting Girard-Reynolds polymorphism as a special case of F-bounded polymorphism:

$$\text{GenUniversal} = \lambda\sigma.\{\} \qquad \text{// the content-free constraint}$$

$$\forall(\tau <: \text{GenUniversal}[\tau])\ .\ \text{identity} : \tau \to \tau$$

$$\forall(\tau <: \text{GenUniversal}[\tau])\ .\ \text{insert} : \tau \times \text{List}[\tau] \to \text{List}[\tau]$$

That is, we constrain τ to range over those types which have at least the functions of the universal interface, but this interface is trivial (empty), so τ ranges again over any type.

There are two practical consequences of this discussion. The first is that, wherever a polymorphic variable is required in our programming language, we should always model its type using some kind of type parameter in the formal calculus. The fact that object-oriented languages don't make the type parameters explicit for their classes is one of the reasons why the notions of *class* and *type* get so confused. The second is that we do not need separate mechanisms to explain template-based and class-based polymorphism. The class parameters constrained by F-bounds are adequate for both purposes [6], being more general than classical unconstrained parameters.

4 DISTINGUISHING CLASS AND TYPE

In current object-oriented languages, objects and variables are "typed" using the names of the classes like type identifiers. These identifiers are used ambiguously, to describe either an object or value with an exact type (a monomorphic *type* in the theory), or, alternatively, to describe a variable with a flexible type (a polymorphic *class* in the theory). What we should like is for object-oriented languages to indicate a *class*, or a *type* unambiguously.

Informally, it is possible to infer the intended semantics of class identifiers from the program context in which they appear. In a C++ or Java-like language, when we create an object, we usually intend to create something with a fixed implementation and an exact type:

```
    … = new Point;              // exactly typed object creation
```

In this context, we do not expect to obtain some instance of a subclass of *Point*, but rather an exact instance of the exact type *Point*. On the other hand, when we declare a program variable of the *Point* class, it is clear that we intend this to be flexible, capable of receiving values that might be more specific than a *Point*, but which are at least of this class:

```
    Point p = …                 // polymorphic variable declaration

    accept(Point p) { … }       // polymorphic method arguments
```

In this context, we do not expect these variables to be restricted to accepting only objects of the exact *Point* type, but rather any type which is at least a subclass of *Point*. We can model these differences in the λ-calculus, to make them explicit.

Recall that a class is defined essentially as a flexible, open-ended implementation, parameterised by *self*, with a corresponding polymorphic type, parameterised by σ, the self-type [4]. We give the type-shape of the class using a type generator, followed by the implementation-shape using an object generator, which is typed using the type generator as the F-bound, restricting what types may eventually replace the self-type:

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

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

In our C++ or Java-like programming language, when we declare a variable of the *Point* class, what we are really asserting is the polymorphic typing *p0* : τ, where τ is a type parameter constrained to range over any type in the *Point* class:

Programming Language	**Formal Model**
Point p;	$p0 : \forall(\tau <: \text{GenPoint}[\tau])\ .\ \tau$

Table 1: Polymorphic variable declaration

On the other hand, when we create an exact instance of the *Point* type, we must fix both the type and the implementation. In the calculus, this is done by taking the fixpoint of the type generator and of the object generator [4]:

$$
\begin{aligned}
&\text{Point} \;=\; \mathbf{Y}\,\text{GenPoint}\\
&\quad = \;\mu\sigma.\{x : \text{Integer},\, y : \text{Integer},\, \text{equal} : \sigma \to \text{Boolean}\,\}\\
&\quad \Rightarrow \{x : \text{Integer},\, y : \text{Integer},\, \text{equal} : \text{Point} \to \text{Boolean}\,\}, \qquad \text{after unrolling;}
\end{aligned}
$$

$$
\begin{aligned}
&\text{aPoint} \;=\; \mathbf{Y}\,\text{genAPoint}[\mathbf{Y}\,\text{GenPoint}] \;=\; \mathbf{Y}\,\text{genAPoint}[\text{Point}]\\
&\quad = \mathbf{Y}\,\lambda(\text{self} : \text{Point}).\{\,x \mapsto 2,\, y \mapsto 3,\\
&\qquad\qquad\qquad \text{equal} \mapsto \lambda(q : \text{Point}).(\text{self}.x = q.x \wedge \text{self}.y = q.y)\}\\
&\quad = \mu(\text{self} : \text{Point}).\{\,x \mapsto 2,\, y \mapsto 3,\\
&\qquad\qquad\qquad \text{equal} \mapsto \lambda(q : \text{Point}).(\text{self}.x = q.x \wedge \text{self}.y = q.y)\}\\
&\quad \Rightarrow \{\,x \mapsto 2,\, y \mapsto 3,\, \text{equal} \mapsto \lambda(q : \text{Point}).(\text{aPoint}.x = q.x \wedge \text{aPoint}.y = q.y)\},\\
&\qquad\qquad\qquad\qquad\qquad\qquad \text{after unrolling.}
\end{aligned}
$$

This creates the exact instance *aPoint* of the exact *Point* type. We can therefore model the meaning of object creation expressions in our programming language:

Programming Language	**Formal Model**
new Point;	p1 = **Y** genAPoint[**Y** GenPoint]; p1 : Point

Table 2: Exactly-typed object creation

Here, we have taken the liberty of introducing the temporary variable *p1* in the formal model, so that we can initialise this variable to the rather complex object creation expression and then see that it has an exact type, which is the *Point* type we expected. The temporary variable is simply a convenience, to save repeating longer expressions. In section 6 below, we will use a similar approach to analyse program behaviour in step-by-step detail.

5 TYPE CHECKING WITH FIRST-ORDER TYPES

First, we shall introduce a test-case that exemplifies some of the difficulties identified with type systems that check types in the first-order model (with simple types and subtyping). The example code fragment, expressed in our C++ or Java-like language, is a cut-down version of the infamous “Eiffel type failure” problem first identified by Cook [13]:

```
Point p = new Point3D;          // alias a more specific Point3D
Point q = new Point;            // create a standard Point
Boolean b = p.equal(q);         // dynamically invoke the specific equal
```

Programmers expect a *Point3D* instance to be type-compatible with a *Point* variable, but in the first order model, this is not the case. To explain why the above fragment is problematic, we should define the *Point3D* class, which describes a three-dimensional point, whose interface extends that of a standard two-dimensional *Point*:

$$\text{GenPoint3D} = \lambda\sigma.\{\text{x} : \text{Integer}, \text{y} : \text{Integer}, \text{z} : \text{Integer}, \text{equal} : \sigma \to \text{Boolean}\}$$

$$\begin{aligned}
&\text{genAPoint3D} : \forall(\tau <: \text{GenPoint3D}[\tau]).\tau \to \text{GenPoint3D}[\tau] \\
&\text{genAPoint3D} = \lambda(\tau <: \text{GenPoint3D}[\tau]).\lambda(\text{self} : \tau). \\
&\qquad \{\text{x} \to 2, \text{y} \mapsto 3, \text{z} \mapsto 5, \\
&\qquad\qquad \text{equal} \mapsto \lambda(\text{r} : \tau).(\text{self.x} = \text{r.x} \wedge \text{self.y} = \text{r.y} \wedge \text{self.z} = \text{r.z})\}
\end{aligned}$$

In particular, an instance *aPoint3D : Point3D*, created from these generators by taking the fixpoints (see section 4) will have an extra *z* field; and when *aPoint3D* tests itself for equality against another point, it will compare all of its *x, y* and *z* fields.

In the original "type failure" scenario, the programming language expected the subtyping relationship *Point3D* <: *Point* to hold. In fact, we now know that these types are *not* in a subtyping relationship, because the retyping of *Point3D*'s *equal* method violates the function subtyping rule [1]. However, Eiffel allowed subclasses to retype their methods with more specific argument types, since it is unlikely in practice that we should want a *Point3D* to compare itself with more general kinds of point.

An undetected type failure arises as follows. First, we create a specific *Point3D* instance and assign it (by polymorphic aliasing) to the more general variable *p : Point*. This is permitted by the (faulty) assumption that *Point3D* <: *Point*. Then, we create another instance *q : Point*. Finally, we invoke *p.equal(q)*, at which moment the undetected type failure occurs. Statically, the type of *equal* is *Point.equal : Point* → *Boolean*, so it appears to be legal to pass in the given argument *q : Point*. However, *p* currently contains a dynamic instance of *Point3D* and the version of the *equal* method which is actually invoked is *Point3D.equal : Point3D* → *Boolean*. This receives the too-general argument *q : Point*, and during the execution of the method body, an attempt is made to access the *z* field of a plain *Point*, which will cause the program to crash, generating a memory segmentation fault.

Cook originally proposed to fix this problem by forcing Eiffel to conform to strict subtyping rules [13]. Redefined argument types for *equal* would therefore not be allowed. Although this technically satisfies subtyping, we have seen how this results in a strictly less expressive language [2]. In particular, the *equal* method, which is required by every class, may only be typed with the most general kind of argument (usually, the root class *Object*), and it may never be retyped with more restricted types of argument. Instead, redefined versions of *equal* have to accept *Object* arguments and use runtime-checked type downcasting internally, to recover the more specific dynamic type of the argument, before comparison can be made. This merely pushes the type failure problem back into the run-time.

6 TYPE CHECKING WITH SECOND-ORDER CLASSES

Classes are type-recursive, meaning that their methods often accept or return arguments of the *self*-type. So it is natural to want these arguments and results to become uniformly specialised along with the class itself. We want to allow a *Point3D* to specialise the argument type of its *equal* method. However, we still want to avoid unchecked type failures.

Programming Language	**Formal Model**
Point p …	$p0 : \forall(\tau <: GenPoint[\tau]).\tau$
… new Point3D;	$p1 = \mathbf{Y}\ genAPoint3D[\mathbf{Y}\ GenPoint3D];$ $p1 : Point3D$
Point p = new Point3D;	$p2 = (p0 := p1);\ \{Point3D / \tau\}$ $p2 : Point3D$
Point q …	$q0 : \forall(\sigma <: GenPoint[\sigma]).\sigma$
… new Point;	$q1 = \mathbf{Y}\ genAPoint\ [\mathbf{Y}\ GenPoint];$ $q1 : Point$
Point q = new Point;	$q2 = (q0 := q1);\ \{Point / \sigma\}$ $q2 : Point$
Boolean b …	$b0 : Boolean$
… p.equal …	$p2 : Point3D;$ $p2.equal : Point3D \rightarrow Boolean$
… p.equal(q);	$q2 : Point;$ $p2.equal : Point3D \rightarrow Boolean;$ $p2.equal(q2 : Point) :$ ERROR $Point \neq Point3D$

Table 3: Polymorphic checking with type substitution

In the *Theory of Classification*, we take the view that a class is not a first-order concept, but a second-order, polymorphic concept. One of the advantages this brings is the ability to relate closed recursive types to each other, by relating their generators in a (second-order) *pointwise* subtyping relationship [3]. This allows us to specialise argument and result types uniformly, in line with programmers' intuitions about classes. However, the recursive types themselves do not enter into simple subtyping relationships, so we cannot type-check them in the usual first-order system. By properly distinguishing the polymorphic notion of *class* from the monomorphic notion of *type*, we may type-check the same fragment of object-oriented code in a second-order model, showing that polymorphic assignment really involves the *propagation of types* into polymorphic type parameters. This is a very powerful checking mechanism, capable of resolving many of the difficulties formerly identified with object-oriented type systems.

On the left-hand side of table 3, the expressions in the programming language are broken down into small steps, in order to examine the types of these expressions in the formal model on the right-hand side. On the first row, we declare a polymorphic *Point* variable and show this to have a F-bounded parametric type. On the second row, we create an exact *Point3D* object and show this to have the exact *Point3D* type. On the third row, we assign the specific instance to the general variable. This is where the new type-checking principle first comes into play. At the moment of polymorphic aliasing, the exact type of the object is propagated into the type parameter of the variable, shown by the substitution: {Point3D / τ}. As a consequence, we obtain a new context *p2* after the assignment (*p0 := p1*), in which the type of the bound variable expression has been updated.

This is how the type mismatch is eventually detected. When checking the program expression: *p.equal(q)*, the model can predict the type of the *equal* method, and its expected argument type, since statically it knows that this is selected from *p2*. At the same time, the model knows the type of the actual argument, from the context *q2*. The formal and actual argument types are shown to conflict ($Point \neq Point3D$), so the checker can raise a type mismatch at compile time. Not only do we spot the type error at compile time, but we do this without having to restrict the expressiveness of the language. We still allow *Point3D* objects to be passed into polymorphic variables $p : \forall(\tau <: GenPoint[\tau]).\tau$, so long as this does not conflict with other typing requirements further down the line. For example, the following code fragment is readily accepted by this checking algorithm:

```
Point p = new Point3D;         // alias a more specific Point3D
Point3D q = new Point3D;       // create a specific Point3D
Boolean b = p.equal(q);        // dynamically invoke the specific equal
```

since, at the moment of selection, the *equal* method has the type $Point3D.equal : Point3D \rightarrow Boolean$. As a consequence, it can happily accept the actual argument $q : Point3D$. The following code fragment is also acceptable:

```
Point p = new Point3D;         // alias a more specific Point3D
Point q = new Point3D;         // alias another specific Point3D
Boolean b = p.equal(q);        // dynamically invoke the specific equal
```

because the type substitution {*Point3D* / τ <: *GenPoint*[τ]} is made consistently when both *p* and *q* alias values of exact types, before the typing of the *equal* method invocation is considered. Though the variables *p, q* were originally declared with general polymorphic types, new type contexts are established by the polymorphic aliasing.

7 CONCLUSION

Any object-oriented language that truly supports the notion of *class* should be able to distinguish contexts where simple types, or polymorphic classes are intended. The secret to success is to preserve the underlying type parameter in expressions where polymorphism is intended. When polymorphic variables alias each other, this has the effect of substituting one type parameter for another, possibly strengthening the F-bound constraint (this is because unifying two type variables requires that you accept the more restricting of the two type constraints – see the previous discussion on *intersection types* in [5]). When polymorphic variables alias objects with exact types, these types are substituted into the type parameters. As a consequence, it is always clear whether an expression has a polymorphic, or fixed type, in a given context.

Object-oriented lanaguages that adopted this simple rule could remove a lot of clutter from their syntax. To start with, there would be no need to have both this kind of (genuine) polymorphic typing *and* subtyping. So, type checkers that performed parametric substitutions would not also have to perform subtyping coercions. If two simple types turned out not to be the same, the checker could immediately rule them as mutually incompatible! Secondly, there would be no need for separate syntactic treatments of template-based and class-based polymorphism, since both would be handled using the same underlying F-bounded parametric mechanism. However, the type instantiation process might happen at run-time as well as at compile-time (this unifies the notions of dynamic binding and template instantiation). Thirdly, we would have to consider more carefully the scope of type substitutions made when polymorphic aliasing occurs. We would expect, for example, that a polymorphic method would bind type parameters on entry, but release these bindings on exit, so that the method could be applied to an object of some different type on another occasion. What then is the scope of a polymorphic assignment? We saw above that binding one type rules out subsequent assignments to different types. The scope of an assignment would have to be defined carefully, with rules for "undoing" an assignment and recovering the old polymorphic type of the variable.

The advantages of (genuine) polymorphic typing do not stop there. For example, type propagation may have considerably stronger and pervasive effects on the behaviour of a piece of software. The C++ Standard Template Library makes use of this when it defines template *allocators* for handling the memory management aspects of regular data types. Substituting different actual allocators can alter the efficiency of the whole program. In fact, parametric substitution is related to reflective meta-programming and, when properly exploited, can produce most of the pervasive benefits claimed by aspect-oriented programming. By understanding the true polymorphic nature of the *class*, we may yet obtain much simpler, yet more powerful programming languages.

REFERENCES

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

[2] A J H Simons, "The theory of classification, part 7: A class is a type family", *Journal of Object Technology*, vol.2 no. 3, May-June 2003, pp 13-22. http://www.jot.fm/issues/issue_2004_05/column2

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

[4] A J H Simons, "The theory of classification, part 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

[5] A J H Simons, "The theory of classification, part 16: Rules of extension and the typing of inheritance", in *Journal of Object Technology*, vol. 4, no. 1, January-February 2005, pp. 13-25. http://www.jot.fm/issues/issue_2005_01/column2

[6] 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

[7] W Harris, "Contravariance for the rest of us", *Journal of Object-Oriented Programming*, Nov-Dec 1991, 10-18.

[8] A J H Simons, "The theory of classification, part 1: Perspectives on type compatibility", *Journal of Object Technology*, vol. 1 no. 1, May-June 2002, pp 55-61. http://www.jot.fm/issues/issue_2002_05/column5

[9] J-Y Girard, Interpretation fonctionelle et elimination des coupures de l'arithmetique d'ordre superieur, *PhD Thesis*, Universite Paris VII, 1972.

[10] J Reynolds, Towards a theory of type structure, *Proc. Coll. Prog.*, New York, LNCS 19 (Springer Verlag, 1974), 408-425.

[11] 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.

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

[13] W Cook, “A proposal for making Eiffel type safe”, Proc. 3rd European Conf. Object-Oriented Prog., 1989, 57-70; reprinted in *Computer Journal 32(4)*, 1989, 305-311

About the author

Anthony Simons is a Senior Lecturer and Director of Teaching Quality 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.

JOURNAL OF OBJECT TECHNOLOGY

Online at http://www.jot.fm. Published by ETH Zurich, Chair of Software Engineering ©**JOT, 2005**

Vol. 4, No. 5, July - August 2005

The Theory of Classification Part 19: The Proliferation of Parameters

Anthony J H Simons, Department of Computer Science, University of Sheffield, UK

1 INTRODUCTION

The *Theory of Classification* is an informal series of articles considering the formal notions of type and class in object-oriented languages. The series began by constructing models of objects, types, and inheritance, then branched out into interesting areas such as mixins, multiple inheritance and generic classes. Most recently, we returned to the core argument that simple types and subtyping are formally different concepts from polymorphic classes and subclassing. The previous article [1] described how object-oriented languages were unclear about the distinction between simple and polymorphic types, so we described one possible approach in which class names (which are used like type identifiers) were interpreted unambiguously either as simple types, or polymorphic classes, according to context.

In the current article, we consider in more detail the kinds of manipulations performed upon polymorphic class-types. These are expressed using function-bounded type parameters of the form: $\tau <: F[\tau]$, where F is a type function, describing the shape of the interface that the type τ is expected to satisfy [2]. In the following, we motivate the need for parameters and bounded parameters, then examine what happens when we require a language to express all of its polymorphism in this way. The result is a proliferation of parameters and constraints, which has both good and bad consequences. On the positive side, it is clear over what types the polymorphic variables may range. On the negative side, the syntax of such languages becomes inflated with parameters and is therefore somewhat unwieldy.

2 POLYMORPHISM NEEDS PARAMETERS

Throughout this series, we have made it clear that wherever polymorphism is intended, this formally requires the use of a type parameter, to stand for the group of types over which the function is defined [3]. For example, consider how the *identity* function applies to a value of any type and returns an identical value, and the result is of the same type as the argument. To define such a function in C++, we must use the template mechanism:

```
template <class T>
T identity (const T arg)       // argument is not modified
{  return arg;                 // returns a copy of arg
}
```

in which the line "`template <class T>`" introduces the type parameter `T`, which stands for any type. This function can be applied to values of many different types in C++, including primitive types, pointers and structured object types:

```
int x = identity(5);                 // returns an int
double y = identity(3.14);           // returns a double
Point* p = identity(new Point);      // returns a Point* pointer
Point q = identity(Point(2, 3));     // returns a Point value
```

The polymorphic function appears to be "smart" because it somehow detects the type of its argument and returns a value of the exact same type. In C++ this is accomplished by the compiler, which statically detects the argument's type and creates a specific instantiation of the template function at compile time. In fact, for every distinct instantiation of the *identity* function, the compiler must generate a new copy of the *identity* function. In the end, it is as if the above program contained four different *identity* functions, with the overloaded type signatures:

```
int identity(int arg);
double identity(double arg);
Point* identity(Point* arg);
Point identity(Point arg);
```

and then C++'s normal rules for selecting one or other overloaded function were used to determine which function to apply.

In the λ-calculus, which is a very simple symbol-manipulation system, we bind the type parameter explicitly, supplying the intended type. The calculus cannot detect by itself that a value has any particular type unless we tell it so. All polymorphic functions therefore accept type arguments and value arguments, in that order [3]. The identity function is defined:

identity : $\forall\tau \,.\, \tau \rightarrow \tau$ // for all τ, accept a τ and return a τ
identity = $\lambda\tau \,.\, \lambda(\text{arg} : \tau) \,.\, \text{arg}$ // bind τ, then bind *arg* : τ, then return *arg*

and we apply it to values of different types in the following style, in which we supply first the type argument, then a value argument of that type:

$$\text{identity int } 5 \;\Rightarrow\; 5$$
$$\text{identity double } 3.14 \;\Rightarrow\; 3.14$$

Readers will recall that a λ-function simply binds its arguments in a left-to-right fashion and so doesn't need to put the arguments in parentheses. In earlier articles, we have sometimes adopted the convention of wrapping type arguments in brackets [] and value arguments in parentheses () to help the reader visualise what is going on inside the function.

Second-order functions like the above are really two functions, one nested inside the other. The outer function is a type-function and the inner function is a value-function. In the above examples, we supplied both the type argument and then the value argument. If we only supply the first type argument, then the result we get back is the second function, which is the body of the first function, in which the type parameter τ has been replaced:

$$\text{identity int} \;\Rightarrow\; \lambda(\text{arg} : \text{int}) \,.\, \text{arg} \qquad \text{// replace } \tau \text{ by int: } \{\text{int}/\tau\}$$
$$\text{identity double} \;\Rightarrow\; \lambda(\text{arg} : \text{double}) \,.\, \text{arg} \qquad \text{// replace } \tau \text{ by double: } \{\text{double}/\tau\}$$

and this models the effect of C++'s type instantiation, since it returns a specific version of the identity function, ready to be applied to a value of one particular type.

3 CLASSES NEED BOUNDED PARAMETERS

Throughout this series, we have argued that class-types are also polymorphic things and therefore need to be modelled using type parameters [3, 4, 1]. The main difference between a universal polymorphic function, such as *identity*, and the kinds of function that belong to a restricted class, such as *plus*, *minus*, *times* and *divide*, is that we want these functions to apply not just to any old type, but only to those types which are considered to be "at least some kind of number". In earlier articles, we demonstrated that this required the introduction of *constraints*, or *bounds* on the type parameter. We defined the interface of a number class using a type function:

$$\text{GenNumber} = \lambda\sigma.\{\text{plus} : \sigma \to \sigma,\ \text{minus} : \sigma \to \sigma,\ \text{times} : \sigma \to \sigma,\ \text{divide} : \sigma \to \sigma\}$$

and then used this to constrain a type parameter, in the function-bounded style [2, 4]:

$$\forall(\tau <: \text{GenNumber}[\tau]) \,.\, (\ \ldots \text{ some definition involving } \tau \ldots\ )$$

How does this constraint express what we mean by a class? The intention is that τ may only range over certain types in a type family [3]. What the constraint literally says is "all types τ that are subtypes of the type you get when you apply *GenNumber* to the τ type".

To unpack this further, recall that we started with a simple model of objects as records, whose labelled fields are functions, representing the object's methods. The types of objects are therefore represented by record types, whose fields are the corresponding type signatures of the object's methods. The notion of subtyping we need, in order to understand this constraint, is therefore record subtyping [5]. The record subtyping rule permits a type with more fields to be a subtype of a type with fewer fields.

So, returning to the original problem, imagine that we expect the *Integer* type to belong to the class of numbers. Therefore, *Integer* must be one of the types that satisfies the above constraint, in other words, we expect the following to be true:

$$\text{Integer} <: \text{GenNumber[Integer]}, \qquad \text{which we can re-express as:}$$

$$\begin{aligned}\text{Integer} <: \{&\text{plus} : \text{Integer} \to \text{Integer}, \text{minus} : \text{Integer} \to \text{Integer},\\ &\text{times} : \text{Integer} \to \text{Integer}, \text{divide} : \text{Integer} \to \text{Integer}\}\end{aligned}$$

by applying *GenNumber[Integer]* and seeing what kind of record-type we get when we substitute {*Integer* / σ} in the body of the generator. So, what this constraint is really saying is that the *Integer* type must have at least as many methods as the record type on the right-hand side. This is easy enough to satisfy if we give *Integer* all of those methods; and we could possibly give it more methods, such as: *modulo : Integer* → *Integer*, which returns a remainder.

So, this kind of bounded type parameter captures exactly the sort of constraint we need when defining groups of functions that apply to all the types in a given class and only to those types. We may write the polymorphic type of *plus* and *minus* in the style of methods:

$$\forall(\tau <: \text{GenNumber}[\tau]) \,.\, \tau.\text{plus} : \tau \to \tau$$
$$\forall(\tau <: \text{GenNumber}[\tau]) \,.\, \tau.\text{minus} : \tau \to \tau$$

This says that, for all numeric types τ, selecting the *plus* or *minus* methods from an object of type τ will return a function that accepts the remaining argument of the same type τ and this will also return a result of the type τ. Alternatively, we can write the polymorphic type of *plus* and *minus* in the style of regular functions:

$$\text{plus} : \forall(\tau <: \text{GenNumber}[\tau]) \,.\, \tau \to (\tau \to \tau)$$
$$\text{minus} : \forall(\tau <: \text{GenNumber}[\tau]) \,.\, \tau \to (\tau \to \tau)$$

which are now clearly seen to accept two arguments, the first of which is the receiver object, from which the method is to be selected. The remaining argument and result type are as above.

4 BOUNDED PARAMETERS IN FUNCTIONAL LANGUAGES

Our notion of a class is exactly the same as the notion of *type classes* in the strongly-typed functional programming languages. The language Haskell [6] defines polymorphic functions using type parameters and sometimes needs to assert that these parameters range over certain restricted classes of types. In the following example of a polymorphic function to compute the `length` of a list, "`[]`" denotes a list-type and "`a`" denotes a type parameter:

```
length :: [a] -> Int
length nil = 0                          -- empty case
length (head : tail) = 1 + length tail  -- non-empty case
```

The first line is a type signature, saying that `length` takes a list of any polymorphic element type "`[a]`" and returns a result of the `Int` type. This is a universal polymorphic function, rather like *identity* earlier. The `length` can be computed, irrespective of what type of element we choose for the list.

However, the polymorphic function for list membership cannot be defined in quite the same way. To determine list membership, the body of the `elem` function needs to be able to compare the supplied value with successive elements of the supplied list, using the equal function "`==`". So, the polymorphic element type must be one that is *in the class of types possessing an equal function*. In Haskell, this is represented by the constraint "`Eq a`", which has the sense "any polymorphic type `a` in the `Eq` class". Elsewhere, Haskell defines the `Eq` class as the class of all those types possessing "`==`" and "`/=`" (not equal). The definition of `elem` is given by:

```
elem :: Eq a => a -> [a] -> Bool
elem x nil = False              -- empty case
elem x (head : tail)                   -- non-empty case
    | x == head = True          -- found the element
    | otherwise = elem x tail   -- keep searching
```

in which the constraint "`Eq a =>`" on the first line has the sense: "provided that the type `a` is a member of the `Eq` class, then this function takes an element of the type `a` and a list of the type `[a]` and returns a `Bool` result". The effect of this class constraint is to ensure that `elem` can only apply to lists, whose elements have a well-defined equal function. If this constraint were not present, then the Haskell compiler would refuse to compile the definition of `elem`, because it could not guarantee that "`x == head`" was a well-typed expression. This, by the way, contrasts with the approach taken in C++, which allows you to write arbitrary expressions involving variables with parametric types, because C++ doesn't compile or check any of its template definitions. It simply waits for these to be instantiated, and then checks that all the calls are well-typed for each instantiation, separately.

In the λ-calculus, we can express such a class of types with equality, using a type generator to constrain the polymorphic type parameter to accept only types that have at least the two functions *equal* and *notEqual*:

$$\text{GenEqual} = \lambda\sigma.\{\text{equal} : \sigma \rightarrow \text{Boolean}, \text{notEqual} : \sigma \rightarrow \text{Boolean}\}$$

$$\forall(\tau <: \text{GenEqual}[\tau]) \;.\; (\ldots \text{ some definition involving } \tau \ldots)$$

The above F-bound provides the exact meaning of Haskell's "`Eq a =>` …" constraint. It was quite satisfying to find this convergence between the notion of class put forward in the *Theory of Classification*, and the notion of "type classes" in functional programming languages, because it means that we are all probably on the right track!

5 CLASSES WITHIN CLASSES NEED NESTED PARAMETERS

After a while, the realisation comes that everywhere you want polymorphism, you formally need another type parameter. This has interesting consequences when you consider more complex classes that are built up out of a number of other classes as their "elements". Simple examples include the kind of generic classes we considered in an earlier article [7]. For example, if we have a *List* class whose self-type is expressed as the parameter σ, and if this *List* has a method *insert* accepting elements of some other class, whose polymorphic type is expressed as the parameter τ, then in which order should these be declared?

The solution to this quandary is found by thinking about the notion of type dependency. The element-type may exist by itself, but the list-type depends in some way upon the element-type in its definition. That is, for whatever element type τ we imagine, the list self-type σ somehow depends on the type of τ. This is natural, since we would expect an *IntegerList* to depend on its *Integer* elements, whereas a *RealList* would have *Reals* as its elements. We therefore introduce τ before σ, since this keeps within the *second-order* λ-calculus, in which type parameters only range over simple types [3]. That is, we introduce σ in a context in which τ is already bound to some actual type, so σ can range over simple types also.

To see how the type parameters stack up, let us construct the type signature for a generic *List* class with a method *elem* to test whether the inserted elements are members of the list. From section 4 above, we know that this puts a constraint on the element type, which must have an *equal* method to compare itself with other elements. Therefore, we will first require a type generator to express the shape of the element type:

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

The shape of the list's interface is expressed through a second type generator:

$$\begin{aligned}\text{GenList} = \lambda\tau.\lambda\sigma.\{&\text{insert} : \tau \rightarrow \sigma,\ \text{head} : \tau,\ \text{tail} : \sigma,\\ &\text{length} : \text{Integer},\ \text{elem} : \tau \rightarrow \text{Boolean}\}\end{aligned}$$

which now recognises that there are two parameters involved. The first is τ for the element-type. We can see this by applying *GenList* to some actual element type, say *Integer*, to see what kind of type-expression this produces:

$$\begin{aligned}\text{GenList[Integer]} = \lambda\sigma.\{&\text{insert} : \text{Integer} \rightarrow \sigma,\ \text{head} : \text{Integer},\ \text{tail} : \sigma,\\ &\text{length} : \text{Integer},\ \text{elem} : \text{Integer} \rightarrow \text{Boolean}\}\end{aligned}$$

The result substitutes $\{\mathit{Integer}/\tau\}$ in the body of the generator, returning a nested type function beginning: $\lambda\sigma.\{\ldots\}$. This looks exactly like the shape of a regular type generator for a non-generic class, and so it is. The second type parameter σ stands for the self-type of this class. We need this because we are dealing with polymorphic classes, not simple types. That is, the above definition is the minimum common interface for a variety of different list-types, which might include more specialised list-types, such as sorted lists.

What is the form of the F-bound constraint that ensures that our list self-type σ ranges over only those lists which (i) have elements with an *equal*-method and (ii) have a list-interface that includes all of the methods: *insert*, *head*, *tail*, *length* and *elem*? This is a constraint constructed using both of the above two type generators:

$$\begin{aligned}&\forall(\tau <: \text{GenEqual}[\tau]).\forall(\sigma <: \text{GenList}[\tau, \sigma]).\\ &\qquad (\ \ldots \text{ some definition involving } \tau \text{ and } \sigma \ldots\ )\end{aligned}$$

What is interesting here is the double quantification. On the outside, we assert that the element type τ may only range over subtypes of *GenEqual*[t], that is, types with at least the methods: *equal* and *notEqual*. Then, on the inside, we assert that the self-type σ may only range over subtypes of *GenList*[τ, σ], that is, types with at least the methods: *insert*, *head*, *tail*, *length* and *elem*, provided that τ is of the earlier specified type. This is the way in which the constraint for the list-type depends on the element-type. Translating this into object-oriented programming terms, the polymorphic type of a class depends on the polymorphic types of the other classes which it references internally. More precisely, it depends on those classes which affect the shape of its external interface.

6 SUBCLASSING USES PARAMETER SUBSTITUTION

Assume now that we wish to derive a more specialised kind of *List* class, say a *SortedList* of elements which are inserted automatically in ascending order. *SortedList* has two new operations *least* and *greatest*, to return the smallest and largest elements and otherwise has all the operations of a *List*. Below, we define the polymorphic type of the *SortedList* class by inheritance, constructing the new constraint partly from information present in the old one. Afterwards, we show how the new constraint preserves the old constraint.

To permit the sorting of elements when they are *inserted*, elements must be comparable with each other using *lessThan*. This means that the constraint on the element-type will be stronger than the one required for a plain *List* element. The new type generator must at least have the interface of *Equal*, otherwise the old *List* membership method *elem* would not work. The new generator *GenComparable* is therefore defined to extend the interface of *GenEqual*:

$$\begin{aligned}\text{GenComparable} &= \lambda\omega.(\text{GenEqual}[\omega] \cup \{\text{lessThan} : \omega \rightarrow \text{Boolean},\\ &\qquad \text{greaterThan} : \omega \rightarrow \text{Boolean}\})\\ &= \lambda\omega.\{\text{equal} : \omega \rightarrow \text{Boolean}, \text{notEqual} : \omega \rightarrow \text{Boolean},\\ &\qquad \text{lessThan} : \omega \rightarrow \text{Boolean}, \text{greaterThan} : \omega \rightarrow \text{Boolean}\}\end{aligned}$$

In the above, the new element type parameter is ω. This is introduced on the outside, then the result is formed internally as the union of the old interface and the extra methods. Note that the inherited interface is given by: *GenEqual*[ω]. This essentially applies the old generator to the new parameter, and creates a record type in which {ω/τ} has been substituted throughout:

$$\text{GenEqual}[\omega] = \{\text{equal} : \omega \rightarrow \text{Boolean}, \text{notEqual} : \omega \rightarrow \text{Boolean}\}$$

and this inherited record type can be safely unioned with the additional method signatures. A similar trick is used to extend the *List* generator to produce the *SortedList* generator:

$$\begin{aligned}\text{GenSortedList} &= \lambda\omega.\lambda\psi.(\text{GenList}[\omega, \psi] \cup \{\text{least} : \omega, \text{greatest} : \omega\})\\ &= \lambda\omega.\lambda\psi.\{\ \text{insert} : \omega \rightarrow \psi\ \text{head} : \omega, \text{tail} : \psi, \text{length} : \text{Integer},\\ &\qquad \text{elem} : \omega \rightarrow \text{Boolean}, \text{least} : \omega, \text{greatest} : \omega\}\end{aligned}$$

Note again how this creates an inherited version of the old *List* interface by applying the old generator to the new parameters: *GenList*[ω, ψ]. This returns a record type in which {ω/τ, ψ/σ} have been substituted throughout:

$$\begin{aligned}\text{GenList}[\omega, \psi] = \{\ &\text{insert} : \omega \rightarrow \psi\ \text{head} : \omega, \text{tail} : \psi,\\ &\text{length} : \text{Integer}, \text{elem} : \omega \rightarrow \text{Boolean}\}\end{aligned}$$

and this inherited record type can be safely unioned with the new method signatures. Previously, we noted that the reason for substituting parameters like this was to ensure, in the inheriting class, that the self-type was consistently denoted by ψ (rather than a mixture of new ψ and old σ) [3, 4]. Now that we are dealing with nested classes, the rules must be applied recursively for the element type, which must be consistently ω throughout.

The stronger constraint for our new *SortedList* class, which ensures that the type paramter ψ ranges over only those types which are at least *SortedLists* of *Comparable* elements, is given by a nested F-bound that is written in terms of the two new generators, but is otherwise similar to the nested F-bound for a *List*:

$$\forall(\omega <: \text{GenComparable}[\omega]).\forall(\psi <: \text{GenSortedList}[\omega, \psi]).$$
$$(\ \ldots \text{ some definition involving } \omega \text{ and } \psi \ldots\ )$$

7 SUBCLASSING IS THE INCREASING OF TYPE CONSTRAINTS

What is even more sophisticated in this model of inheritance is the preservation of the constraints on all the type parameters. Technically, we are substituting parameters bounded by one set of constraints with new parameters bounded by a different set of constraints. Are the substitutions all legal? To do this, we check the expectations made internally by the type generators.

When we apply: *GenEqual*[ω], we substitute {ω/τ}. Now, the τ-parameter expects to receive a type satisfying: $\tau <: \textit{GenEqual}[\tau]$, in other words, a type with at least the interface of the *Equal* class. It so happens that we are replacing one parameter with another parameter, rather than an actual type. So instead, we have to consider all the types that might be allowed by the new parameter. The new parameter ω expects to receive a type satisfying: $\omega <: \textit{GenComparable}[\omega]$, in other words, any type with at least the interface of the *Comparable* class. So, we have to ensure that all types that we could substitute for ω will also be acceptable types for the parameter τ. Formally, we determine this using the pointwise subtyping rule [4], checking the assertion:

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

By inspection, the *Comparable* interface includes the *Equal* interface, no matter what value we supply for τ, so this substitution is proven legitimate.

A similar process happens when we apply: *GenList*[ω, ψ], causing the substitutions {ω/τ, ψ/σ}. We follow the same argument for ω, and then a similar argument for ψ as it replaces the parameter σ. Eventually, we conclude that any type which we could substitute for ψ will also be an acceptable type for the parameter σ, as a result of the pointwise rule:

$$\forall\tau \,.\, \forall\sigma \,.\, \text{GenSortedList}[\tau, \sigma] <: \text{GenList}[\tau, \sigma]$$

This expresses the assertion that the interfaces of *SortedList* and *List* stand in a pointwise subtyping relationship, no matter what types we substitute for τ and σ. By inspection of the two interfaces, we conclude this holds, so the substitution is proven legitimate.

In general, the derivation of subclasses follows a process of monotonic restriction (steadily increasing, or stable constraint) on all the type parameters involved. The kinds of restriction that are permitted can be modelled as a kind of commuting diagram, shown in figure 1.

$$\begin{tikzcd}
\forall(\tau <: \mathrm{GenEqual}[\tau]).\ \forall(\sigma <: \mathrm{GenList}[\tau, \sigma])\ .\ \sigma \arrow[r, "\{\omega/\tau\} \Rightarrow \{\gamma/\sigma\}"] \arrow[dr, "\{\omega/\tau,\ \psi/\sigma\}"'] & \forall(\omega <: \mathrm{GenComparable}[\omega]).\ \forall(\gamma <: \mathrm{GenList}[\omega, \gamma])\ .\ \gamma \arrow[d, "\{\psi/\gamma\}"'] \\
 & \forall(\omega <: \mathrm{GenComparable}[\omega]).\ \forall(\psi <: \mathrm{GenSortedList}[\omega, \psi])\ .\ \psi
\end{tikzcd}$$

Figure 1: Commuting diagram of increasing constraints

This shows that you can start with a polymorphic list σ of elements τ with equality, and can choose to restrict the element-type, giving a similar list γ of comparable elements ω with less-than ordering, and then constrain the list further to a sorted list ψ with comparable elements ω. Alternatively, both restrictions on the element-type and list-type may be carried out simultaneously, which is illustrated by the diagonal path. A fourth substitution path, changing a list σ to a sorted list ψ *without* changing the element type is not possible, according to the diagram. This is because a sorted list depends on having a comparable element-type.

One new thing that the diagram shows is that substituting one of the "type elements" of the main class, here, the element-type of the list, leads to a change in the self-type also. Merely choosing to substitute {ω/τ} with a more constrained element-type entails the substitution of the self-type {γ/σ} in figure 1. This is something rarely considered in informal treatments of object-oriented type systems. Why has the self-type of the list changed? Well, the result of one access method is typed: *head* : τ for the original list; but after we have substituted {ω/τ}, then we would expect this method to return a different type: *head* : ω. Therefore, this must be the *head* of a different type of list. Jens Palsberg and Michael Schwartzbach were the first to explore the consequential effects of type substitutions to any significant degree in object-oriented type systems [8]. This is still a relatively new area, the subject of continuing research.

8 CONCLUSIONS

We have shown how a proper treatment of object-oriented polymorphism involves the use and manipulation of constrained type parameters. In particular, the polymorphic type intended by a class identifier in object-oriented programs is quite a complex notion. It is a parametric type, restricted to receive only types with a certain interface structure. But more interestingly, the parametric type depends in turn on all the type elements of the class in question. So, the polymorphic aliasing of one of these element-types may also affect the type of the whole class. During the operation of inheritance, many type substitutions are performed, both of the element-types and the class's self-type. This follows a pattern which gradually increases the constraints on all type parameters monotonically, restricting the types which may eventually be valid members of the subclass.

Very few attempts have been made so far to build a practical object-oriented programming language based on entirely parametric treatments of polymorphism. One attempt was by Simons *et al.* in the early 1990s [9, 10]. In the experimental language *Brunel*, simple types were written in the usual way as: `x:Integer; y:Boolean;` and polymorphic class-types were written using explicit type parameters: `p:P`, where `#Point[P]` introduced the parameter and expressed the F-bound constraint that `P` is in the class of *Points*. However, the language eventually proved unwieldy, due to the stacking up of type parameters. Every class that itself contained further class-elements had to declare the element-types up front. For larger classes, this was a considerably high overhead and eventually was judged impractical for a real programming language. It seems that the most practical way ahead should be to design languages that can keep track automatically of all the complex, and interdependent type substitutions. This might eventually lead to a whole new kind of compiler technology.

REFERENCES

[1] A J H Simons, "The theory of classification, part 18: Polymorphism through the looking glass", *Journal of Object Technology, 4(4), May-June 2005*, 7-18. http://www.jot.fm/issues/issue_2005_05/column1

[2] 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.

[3] A J H Simons, "The theory of classification, part 7: A class is a type family", *Journal of Object Technology, 2(3), May-June 2003*, 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", *Journal of Object Technology, 2(4), July-August 2003*, 55-64. http://www.jot.fm/issues/issue_2003_07/index_html

[5] A J H Simons, "The theory of classification, part 4: Object types and subtyping", *Journal of Object Technology, 1(5), November-December 2002*, 27-35 http://www.jot.fm/issues/issue_2002_11/column2

[6] S Peyton-Jones et al., *The Haskell 98 Language and Libraries: the Revised Report* (Cambridge, UK: CUP, 2003), 270pp. Also pub. as special edn. *Journal of Functional Programming, 13(1)*, January, 2003. Online version: http://www.haskell.org/onlinereport/

[7] A J H Simons, "The theory of classification, part 13: Template classes and genericity", *Journal of Object Technology, 3 (7), July-August 2004*,, 15-25. http://www.jot.fm/issues/issue_2004_07/index_html

[8] J Palsberg and M I Schwartzbach, *Object-Oriented Type Systems* (Chichester: John Wiley, 1994).

[8] A J H Simons, Low E-K and Ng Y-M, "An optimising delivery system for object-oriented software", *Object-Oriented Systems, 1 (1)* (1994), 21-44.

[9] A J H Simons, *A Language with Class: The Theory of Classification Exemplified in an Object-Oriented Programming Language*, PhD Thesis, Department of Computer Science, University of Sheffield (Sheffield, 1995), 255pp.

About the author

Anthony Simons is a Senior Lecturer and Director of Teaching Quality 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.

JOURNAL OF OBJECT TECHNOLOGY

Online at http://www.jot.fm. Published by ETH Zurich, Chair of Software Engineering ©**JOT, 2005**

Vol. 4, No. 7, September-October 2005

The Theory of Classification Part 20: Modular Checking of Classtypes

Anthony J H Simons, Department of Computer Science, University of Sheffield, UK

1 INTRODUCTION

This is the final article in an informal series on the *Theory of Classification*, which has considered the theoretical notions of *type* and *class* in object-oriented languages. The series began by constructing models of objects, types, classes and inheritance, then branched out into interesting areas such as mixins, multiple inheritance and generic classes. The core of our argument has been that the notions of *class* and *type* are distinct, but both can be described formally in the λ-calculus. Strongly-typed object-oriented programming languages are largely based on the idea that "a class is a type" and "subclassing is subtyping" [1]. In earlier articles, we demonstrated why this is not really satisfactory as a formal model of classes and classification. A type system based on first-order types and subtyping:

- fails to capture natural relationships between recursive types (whose methods accept or return values of the same type as themselves), since recursive types can have no proper subtypes [2];
- loses type information when methods are inherited [2], requiring the use of *type downcasting* everywhere to recover the most specific type of the object returned by a general method, which is tantamount to breaking the type-system;
- conflicts with the notion of *type classes* adopted elsewhere in functional programming languages like Haskell, which use *type parameters* to express this notion [3].

Instead, we have argued that a class is a *family of related types*, which can only be expressed in a second-order type system with polymorphism [4]. Classical polymorphism is represented using type parameters that range over many different actual types, but object-oriented programming requires a kind of polymorphism where type parameters receive only certain related types that satisfy a particular interface description. Mathematically, this is constructed by placing constraints on type parameters, called *function bounds*, or *F-bounds*, which have form: $\tau <: F[\tau]$, where F is a type function, describing the shape of the interface that the type τ is expected to satisfy [5]. However, while this gives a much more satisfying account of classes and classification, very few programming languages have ventured into this new and exciting territory. In this final article, we try to understand why this is so; and what practical problems remain to be solved in the modular checking of class-types.

2 TRADING MODULARITY AND EXPRESSIVENESS

A first-order type system has two things to commend it. Firstly, it is quite simple to implement a type-checker that can check types for exact correspondence, or for subtype compatibility with a given type. The type of the source object can be compared with that of the target variable to see if the former can be converted up to the latter, using subtyping rules like those we discussed in [1]. Secondly, code that has been checked once need never be checked again, or recompiled in new contexts. This is because the type system can never reveal more specific information about an object that is passed into a more general variable (which we have called the "type loss problem"), so the code need only be checked once over the most general type that it can accept. This, more than any other reason, is why object-oriented languages have been slow to take up the new insights into the nature of classes and classification: the desire to have *modular* and *incremental* compilation. Without this, it would not be possible to build industrial-scale systems.

In the last two articles, we showed that full support for the notion of classes and subclassing requires a distinction to be drawn between simple, monomorphic *types* and polymorphic *classes*, the latter formally expressed using type parameters [6]. However, this means that type checking rules are more complicated. The compiler has to keep track of sets of type parameters, one for each variable with a "class-type", and has to know how to substitute one parameter for another when values are passed, and also check that the various constraints on the parameters will allow the given substitutions [3].

Now, different type substitutions may happen upon different occasions. For example, consider a polymorphic method for moving graphical shapes on a screen, that accepts *Integer* coordinates and returns the moved object:

$$\text{move} : \forall(\tau <: \text{GenShape}[\tau]) \,.\, \tau \rightarrow (\text{Integer} \rightarrow \text{Integer} \rightarrow \tau)$$

This method is defined for a polymorphic class of *Shapes*, expressed by $\tau <: \mathit{GenShape}[\tau]$ in the style described in earlier articles [2, 4]. Below, we assume that *Square* and *Circle* are exact types that satisfy this F-bound constraint. Now, if *move* is legally invoked on different actual shapes on different occasions, say on a *Square* and a *Circle*, this will cause the two different type substitutions $\{\mathit{Square}/\tau\}$ and $\{\mathit{Circle}/\tau\}$, yielding two differently-typed versions of the *move* method. These variants will have have the exact types:

$$\text{move} : \text{Square} \rightarrow (\text{Integer} \rightarrow \text{Integer} \rightarrow \text{Square})$$
$$\text{move} : \text{Circle} \rightarrow (\text{Integer} \rightarrow \text{Integer} \rightarrow \text{Circle})$$

for the duration of the binding of the (polymorphic) argument to the (exactly-typed) objects. Does this mean that we must compile two different versions of the source-code for *move*?

Well, if *Circle* and *Square* were passed by value, then we should need multiple compilations of the source to handle the different physical layout of each type – this would be analogous to the *template mechanism* in C++, in which multiple copies of the template code are compiled, one copy for each distinct type-instantiation. However, it is more likely that the objects will be passed by reference and both will share the same physical layout (in the low-order bytes) for storing information about their screen location. This case therefore has more similarities with Java 1.5's treatment of type parameters. They are used in the typechecker to eliminate the need for explicit type downcasting, but are erased later in the virtual machine, in which objects are treated in the same way as in the usual subtyping approach. So, having flexible typing for *move* does not necessarily require multiple compilations of this method.

Nonetheless, the polymorphic typing situation is quite different from the kind of typing that is possible in a first-order type system. In the latter, the result of *move* can only ever have the general type *Shape*, which is typically not useful, especially if we want to do something else with the moved object (for which we would first have to perform a *type downcast*). But, in the second-order type system, we recover the exact type of the moved object straight away. This is good, from the point of view of expressiveness.

Now, consider the context in which *move* was called. What does this context expect the result-type to be? It knows the result must be some type $\tau <: GenShape[\tau]$, but, on different occasions, it receives back objects of the more specific types *Square* and *Circle*. It is possible to optimise further method invocations on these results, on the basis of this type analysis. For example, imagine that the *Shape* class declares an abstract method *area()*, which has distinct implementations in all subclasses. The method *area()* may be statically bound, if we can tell in advance that the target is definitely a *Square* or a *Circle*, rather than some unknown kind of *Shape*, for which we would have to insert a dynamically-bound call, to detect the exact type at run-time. However, using the more expressive polymorphic type system, we can propagate exact type information back into the calling context and choose to bind the *area()* method statically. The cost of this is that we must compile multiple versions of the context code.

Simons et al. first analysed these kinds of parametric issues as part of a wider optimisation strategy for object-oriented compilers, using the experimental language *Brunel* as an exemplar for the techniques [7]. They discovered that a fully parametric type system gives you the choice of using more or less of the exact type information available, to tailor the optimisation of bindings. The early binding algorithm described in [7] bears some similarities to Chambers and Ungar's notion of pre-emptive type analysis in the untyped language *Self* [8]. However, the trade-off is that the more type information you use, the more copies of the object-code you generate. In *Brunel*, a global compilation approach was adopted, in which a full type analysis of the whole program was performed and compiler switches could be set to control the amount of early binding and code duplication.

The early binding approach does not transfer over to a modular compilation strategy. When compiling modules incrementally, only partial type information is available. For example, we must compile the methods of the *Shape* class, without any knowledge that it will eventually have two subclasses *Circle* and *Square*, which we previously assumed were the "leaves" in the class hierarchy, becoming the exact types of objects used in the program. Obviously, the compiler could not know in advance whether these are in fact leaf-nodes, or whether they too might eventually be specialised further. The best that can be done is to insert a dynamically-bound call for abstract methods like *area()*. No further optimisation can be performed. However, the parametric type information can be retained and used to avoid type downcasting on the result of *move()*.

3 A UNIFICATION APPROACH TO PARAMETRIC TYPING

Java 1.5 is introducing a form of parametric type analysis that captures an aspect of the strategy we describe above. However, parameters will be used only for certain generic classes, like those in the *Java Collections* framework, and will only characterise the element-types of these collections. In our view, this same approach could be used to characterise all class-types, in the manner described in [3]. All variables marked with a "class-type" are polymorphic and so should be treated in a parametric way. The parametric type information could be used everywhere in the type-checker to obtain more exact type information, but erased in the runtime model. This would allow code modules (classes, in Java) to be incrementally compiled, provided that most calls were bound dynamically, as is usual in Java. But it would also allow some optimisations and static binding of methods for simple leaf-classes, like *Integer* and *Boolean*, for which the compiler could be told that no further specialisations were intended. (In fact, the Java simple types *int*, *float*, etc. could be merged with the class-types in a single-rooted class hierarchy).

The most significant challenge to the development of proper, parametric type-checkers is the problem of unifying different polymorphic types. This happens whenever one polymorphic variable is passed as an argument or result to another method, where the formal and actual types of the variables may be distinct. However, there exists at least one programming language, Prolog, which already has a similar algorithm at the heart of its interpreter. This is the *most general unifier* (MGU) algorithm, which calculates the most general term that can result from the unification of two other terms.

For those unfamiliar with Prolog, this is a language in which the programmer constructs logical expressions, in a declarative style, and program execution is then analogous to solving the simultaneous equation expressed by all the terms. Terms are structured as *predicates*, which may contain grounded values (written in lowercase) or variables (written capitalised). So, the following two terms may exist:

loves(john, Loved). *loves(Lover, mary).*

and it is possible to see that these terms may be unified, yielding the MGU:

loves(john, mary). with substitutions: {*john/Lover, mary/Loved*}

In this unification, the variable *Loved* in the left-hand term receives the value *mary* from the right-hand term; and the variable *Lover* in the right-hand term receives the value *john* from the left-hand term in a symmetrical act of merging. In logic, this is equivalent to saying: "these terms can be unified, provided that the *Lover* stands for *john* and the *Loved* person stands for *mary*". The important thing to note is that both the left- and right-hand terms contributed some of the specific values to the resulting unified term.

It is not difficult to see how this kind of substitution mirrors the process a polymorphic type checker must go through when a polymorphic variable receives an object of some exact type. However, this is an even better analogy for when two polymorphic variables have their types merged, for example when a polymorphic variable with the type $\tau <: F[\tau]$ is passed into a method, whose formal argument has the type $\sigma <: G[\sigma]$. For the duration of the binding, $\sigma == \tau$ and therefore the dual constraint must apply: $\tau <: F[\tau] \wedge \tau <: G[\tau]$. In earlier articles dealing with inheritance and multiple inheritance, we called this an *intersection type* [9, 10] because the type variable τ is being constrained to accept two different, overlapping sets of types and therefore accepts the intersection of these sets. Accordingly, we used $\tau <: F[\tau] \wedge G[\tau]$ to denote an intersection on the parameter τ.

The important thing to note is that this merging of type-constraints is even-handed: it doesn't matter whether $\tau <: F[\tau]$ or $\tau <: G[\tau]$ is the more restricting F-bound, since any type replacing τ must satisfy both constraints. So, this mechanism is adequate to handle specialisation (when a polymorphic type is replaced by a more restricted polymorphic type [9]) and also the kind of symmetrical type-merger that happens with multiple inheritance (when the polymorphic types of two parent classes become unified in the child [10]). The latter case also extends to languages with a combination of single inheritance and multiple interface satisfaction (the λ-calculus model treats all class-like and interface-like types in the same way). So, parametric type checkers will in future need to perform unification on type variables and compute the pool of merged constraints; but fortunately this is no more difficult than unification in Prolog.

4 DISGUISING THE TYPE PARAMETERS

Another of the challenges to be faced is how to make genuinely polymorphic[1] languages attractive to programmers. The previous article [3] reported how such languages tend to become swamped by the proliferation of type parameters. If each class requires its own self-type parameter, then a class with "class-typed" polymorphic variables in its attributes and methods needs a distinct parameter for each such variable that could eventually be

[1] By this, we mean languages with second-order type systems; the kind of type aliasing performed in first-order subtyping is not technically polymorphism, but something much weaker.

bound to a distinct type. If the classes describing these elements also contain further "class-typed" variables of their own, then our original class has a declaration which is already three layers of parameters deep! We showed how the order of declaration was significant, in that the type parameters for the inner element classes have to be declared first, on the outside, and the dependent type parameters standing for the outer composite classes have to be declared within their scope (in a second-order type system). Essentially, any polymorphic structure must expose, in its interface, all of the different type parameters which could be bound to a different type at some point during the execution of the program.

Various attempts have therefore been made to disguise the existence of type parameters and the many substitution operations that must be performed on them. Perhaps the most careful and thorough of these treatments is Bruce's *matching*. This is an alternative to F-bounds that establishes flexible type compatibility relationships between class-types. Bruce and his co-workers started building type-safe experimental object-oriented languages in the early 1990s. TOOPL and TOOPLE were functional-style object languages (rather like the λ-calculus models we have used in this series), which supported both simple subtyping and a new treatment of the *self*-type using a distinguished type variable called *MyType* [11]. Originally, the motivation for *MyType* arose from considering the same problems with subtyping in the presence of recursive types that led Cook to devise F-bounded quantification [5]; and Bruce's early treatments relied on an F-bounded explanation. However, in later work, Bruce defined complete and consistent type rules for *MyType* which dispensed with explicit F-bounds altogether. The later languages TOIL and PolyTOIL were styled more like imperative object-oriented languages, with variable reassignment [12].

The first advantage gained through using a distinguished *MyType* is that this type variable is implicitly defined within each class-type. The meaning of *MyType* is rather like the type parameter σ in F-bounded constructions like: $\sigma <: GenMyType[\sigma]$, but for each new class, *MyType* is implicitly rebound to refer to the *self*-type of the new class-type. The implicit declaration of *MyType* can be seen below in the type declaration of the abstract *Comparable*, which defines abstract methods *lessThan* and *equal*; and in the type declaration of the concrete *BoxedInteger*, which is essentially a wrapper for a simple *int* type:

$$\text{Comparable} = \text{ObjectType } \{ \text{ lessThan : MyType} \rightarrow \text{bool; equal : MyType} \rightarrow \text{bool } \}$$

$$\begin{aligned}\text{BoxedInteger} = \text{ObjectType } \{ & \text{ lessThan : MyType} \rightarrow \text{bool; equal : MyType} \rightarrow \text{bool;} \\ & \text{getValue : void} \rightarrow \text{int; setValue : int} \rightarrow \text{void } \}\end{aligned}$$

where *ObjectType* is the keyword introducing the new object types (these examples are adapted from [12]). Notice how *MyType* occurs freely inside both definitions, but stands in each case for a different polymorphic type. In our approach using F-bounds, we would introduce two generators, which declare two self-type parameters σ and τ up-front, and then construct F-bounds to use in type expressions:

$$GenComparable = \lambda\sigma.\{\ lessThan : \sigma \to bool;\ equal : \sigma \to bool\ \}$$
$$GenBoxedInteger = \lambda\tau.\{lessThan : \tau \to bool;\ equal : \tau \to bool;\ getValue : void \to int;\ setValue : int \to void\ \}$$

$$\forall(\sigma <: GenComparable[\sigma]).some\_type\_expr\_using(\sigma)$$
$$\forall(\tau <: GenNumType[\tau]).some\_type\_expr\_using(\tau)$$

Now, in PolyTOIL you can declare variables with object types directly, for example, it is legal to declare myInteger : BoxedInteger, in which all internal occurrences of MyType are eventually resolved (in the type rules) to refer to a BoxedInteger, recursively. The parameter MyType is replaced by the actual type of the object receiver, when a method is invoked upon it. In our approach using F-bounds, this requires taking a fixpoint first:

$$BoxedInteger = \mathbf{Y}\ GenBoxedInteger; \qquad \text{recursively bind } \{BoxedInteger/\tau\}$$
$$myInteger : BoxedInteger$$

The second innovation in Bruce's approach is the way in which subclass relationships can be expressed directly between these object types, using the novel *matching* mechanism. You can assert the usual subclass relationship as: *BoxedInteger* <# *Comparable* (read this as "*BoxedInteger* matches *Comparable*"), where "<#" is the new matching operator. Bruce describes the matching relation as:

"the same as subtyping in the absence of the *MyType* construct, but differs in the presence of *MyType*, because *MyType* implicitly has different meanings in different types." [13].

In fact, matching behaves in a similar manner to F-bounded inclusion, in the presence of *MyType*, but in a similar manner to simple subtyping elsewhere. In our approach, we would have to establish a second-order pointwise subtyping relationship between the two corresponding type generators, to ensure that the two parameters were unified before interfaces were compared, and then that one interface were longer than the other:

$$\forall\tau\ .\ GenBoxedInteger[\tau] <: GenComparable[\tau]$$

Bruce's rules simply compare the structure of object types, in which all occurrences of *MyType* are considered equivalent when determining if one type matches another. This dispenses with some of the fiddly detail of parameters. Matching has the same expressive power as F-bounds, for example, note that while *BoxedInteger* <# *Comparable*, the subtyping relationship *BoxedInteger* <: *Comparable* does not hold, because *MyType* occurs as a method argument type (in contravariant position). The type rules for inheritance ensure that *MyType* evolves smoothly to represent the self-type of inheriting classes, which dispenses with another layer of type substitutions in the explicit approach. Finally, if the programmer so wishes, it is also possible to declare explicit type parameters in PolyTOIL. For example, the element-type of a *SortedList* may be given as *elt : T <# Comparable*, to denote any type which matches *Comparable*. This is the analogue of ∀(τ <: *GenComparable*[τ]). *elt :* τ in our approach. In later work, Bruce developed a version of matching with "hash types" that was sufficiently expressive that subtyping could be dropped altogether [14]. This is closer to our approach, which recognises only exact simple types, or parametric polymorphic types.

5 IMPLICIT CLASS-TYPE SUBSTITUTIONS

Perhaps the trickiest issue for future compilers, with the more thorough kind of type analysis we have been proposing here, is to keep track of all the subtle changes to type descriptions that happen as a result of objects being mutually related to each other. This can lead to some hidden evolution in the types of expressions, of which the programmer may not be aware! Consider a class hierarchy describing the various kinds of *Vehicle* that exist, together with the different kinds of *Location* in which such vehicles are typically kept. The root concepts could be described in Java as:

```
class Vehicle {
  private Person myOwn;
  private Location myLoc;
  public Vehicle(Person p) { myOwn = p; myLoc = null; }
  public Vehicle(Person p, Location c) {
    myOwn = p; myLoc = c; c.keep(this); }
  public Person owner() { return myOwn; }
  public Location keptAt() { return myLoc; }
  public void keepAt(Location c) {
    myLoc = c; if (c.keeps() != this) c.keep(this); }
}

class Location {
  private String myAdr;
  private Vehicle myVeh;
  public Location(String a) { myAddr = a; myVeh = null; }
  public Location(String a, Vehicle v) {
    myAdr = a; myVeh = v; v.keepAt(this); }
  public String address() { return myAdr; }
  public Vehicle keeps() { return myVeh; }
  public void keep(Vehicle v) {
    myVeh = v; if (v.keptAt() != this) v.keepAt(this); }
}
```

We can build a pair of mutually-referencing objects by constructing a *Vehicle* and a *Location* in either order, since their constructors set up the reciprocal references:

```
Person wal = new Person("Wallace");
Location lcn = new Location("42 West Wallaby Street");
Vehicle veh = new Vehicle(wal, lcn);
```

Now, the intention is that these classes should be specialised in pairs, for example, we might create *Car/Garage*, or *Aircraft/Hangar*, or *Ship/Port* pairs. But what happens if the programmer only specialises one half of this mutual relationship?

```
class Car extends Vehicle {
  public Car(Person p, Location c) { super(p, c); }
}
Person wen = new Person("Wendolene");
Car car = new Car(wen);
Location loc = new Location("3 Town Square", car);
```

In Java, the result of enquiring *loc.keeps()* always has the type *Vehicle* (we are suffering from the "type-loss" problem again), but dynamically it contains an instance of *Car*. In a parametric type system, we would expect to be able to recover the exact vehicle-type. This is because, when the *Location* is constructed with a value of the exact type *Car*, this type is propagated into the vehicle-type parameter τ of *Location's* polymorphic variable *myVeh*, which we imagine might have the type:

$\forall(\tau <: \text{GenVehicle}[\tau])\ .\ \text{myVeh} : \tau$	which then becomes…
$\text{myVeh} : \text{Car}$	…after substituting $\{\text{Car}/\tau\}$.

This is exciting from the viewpoint of type analysis; but notice that we have created a new, unforeseen type. We expected eventually to specialise *Vehicle* and *Location* in step with each other, producing *Car* and *Garage*, such that the *Garage.keeps()* method returns a *Car*, and the *Car.keptAt()* method returns a *Garage*. Because we only specialised one half of the mutual relationship, we created a new intermediate type variant, a *Location'* whose *keeps()* method returns a *Car*, rather than a *Vehicle*. This type is neither a *Location*, nor a *Garage*, but something in between.

Palsberg and Schwartzbach were the first to report such intermediate types in object-oriented languages [15]. They were using a *type substitution* mechanism, which has only slightly less expressive power than the full parametric mechanism used in our approach[2]. They discovered that checkers which perform full type analysis will inevitably synthesise many intermediate versions of types, as a result of the evolution of other closely-related types. The consensus nowadays is that a mutually-referring set of types creates another enclosing formal structure, a *closure*, which is specialised as a whole, when any one of the related types is specialised. This, then, is the challenge facing the designers of future object-oriented compilers with smart type analysis, implicit type evolution and incremental compilation.

[2] If you systematically substitute type X by type Y within some scope, then all Xs must change into Ys. But with type parameters, you can declare different parameters P <# X and Q <# X, choosing to substitute {Y/P} and {Z/Q}, so this gives you slightly finer control over which substitutions happen together.

6 CONCLUSIONS

Maybe in the future we will see object-oriented languages that exemplify the Theory of Classification in full. I'd like to think that one day, we could have a programming language that is based on a few simple concepts, which is as type-safe as Pascal and as expressive as Eiffel (or Algol-68, or whatever the last really good programming language was). In my crystal ball, this language has to distinguish the theoretical notions of class and type, to allow programmers to understand clearly when simple, or polymorphic typing is intended. It will relate all built-in and programmer-defined types and support obvious, intuitive notions of classification, for example, that the simple types Integer and Boolean are first-class members of the class/type hierarchy and fit underneath an abstract class of Numbers, whose abstract arithmetic-methods are appropriately specialised when they are implemented in Integer and Real. Multiple classification will be possible, such that both Complex numbers and Sets will be considered PartiallyOrdered types, Complex and Integer numbers will be considered kinds of Number, and Sets and Bags kinds of UnorderedCollection. Interfaces will be the same thing as abstract classes.

Incremental compilation will continue to be supported and dynamic binding will be the norm, with some static optimisations performed on the standard leaf-types. The syntax of these languages may start out using explicit type parameters everywhere (such as the cutting-edge work on Haskell *type classes*), but the parameters may eventually disappear inside the compiler, maybe at the loss of a small amount of flexibility and expressiveness. The compiler's ability to perform early type analysis will improve and I expect that in future, code modules will be compiled, which retain their type parameters, such that when the modules are linked and bound at their call-sites, exact type information will be propagated throughout the web of type-constraints, allowing the call-site to extract precisely-typed results. The binding of such parametric modules will result in a bi-directional flow of type-information, yielding solutions such as the "most general intersecting type", computed using unification algorithms.

Throughout my work in this area, I have been standing on the shoulders of giants. I owe particular thanks to Willam Cook, Kim Bruce and Luca Cardelli for formative conversations in the early 1990s and occasional exchanges since then. If you have been stimulated by this informal series of articles on the typing and semantics of object-oriented languages, the next stage might be to get to grips with the details of the type rules, perhaps in [13, 16]. If you have comments, insights or critiques to make, please feel free to contact me by email. If you would like to help bring about the "language with class", then I have a PhD project in this area that needs a good student.

REFERENCES

[1] A J H Simons, "The theory of classification, part 4: Object types and subtyping", *Journal of Object Technology, 1(5), November-December 2002*, pp 27-35 http://www.jot.fm/jot/issues/issue_2002_11/column2/index_html

[2] A J H Simons, "The theory of classification, part 7: A class is a type family", *Journal of Object Technology, 2(3), May-June 2003*, pp 13-22, http://www.jot.fm/issues/issue_2003_05/column2

[3] A J H Simons, "The theory of classification, part 19: The proliferation of parameters", *Journal of Object Technology, 4(5), July-August 2005*, pp 37-48, http://www.jot.fm/issues/issue_2005_07/column4

[4] A J H Simons, "The theory of classification, part 8: Classification and inheritance", *Journal of Object Technology, 2(4), July-August 2003*, pp 55-64, http://www.jot.fm/jot/issues/issue_2003_07/column4/index_html.

[5] 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.

[6] A J H Simons, "The theory of classification, part 18: Polymorphism through the looking glass", *Journal of Object Technology, 4 (4), May-June 2005*, pp 7-18, http://www.jot.fm/issues/issue_2005_05/column1

[7] A J H Simons, Low E-K and Ng Y-M, "An optimising delivery system for object-oriented software", *Object-Oriented Systems, 1 (1)*, (1994), 21-44.

[8] C Chambers and D Ungar, "Iterative type analysis and extended message splitting: optimizing dynamically-typed object-oriented programs", *Proc. 5th ACM Conf. Prog. Lang. Design and Impl.*, pub. *ACM Sigplan Notices, 25(6)*, (1990), 150-164. Reprinted in: *Lisp and Symbolic Computation, 4(3)*, (1991), 283-310.

[9] A J H Simons, "The theory of classification, part 16: Rules of extension and the typing of inheritance", *Journal of Object Technology, 4 (1), January-February 2005*, pp 13-25, http://www.jot.fm/issues/issue_2005_01/column2

[10] A J H Simons, "The theory of classification, Part 17: Multiple inheritance and the resolution of inheritance conflicts", *Journal of Object Technology, 4 (2), March - April 2005*, pp 15-26, http://www.jot.fm/issues/issue_2005_03/-column2.

[11] K Bruce, J Crabtree, A Dimock, R Muller, T Murtaugh and R van Gent, "Safe and decidable type checking in an object-oriented language", Proc. 8th ACM Conf. Obj.-Oriented. Prog. Sys., Lang. and Appl., (1993), 29-46.

[12] K Bruce, A Schuett and R van Gent, "PolyTOIL: A type-safe, polymorphic object-oriented language," *ACM Trans. Prog. Langs. and Sys., 25(2)*, March (2003), 225-290.

[13] K B Bruce, *Foundations of Object-Oriented Languages: Types and Semantics*, (Cambridge MA: MIT Press, 2002).

[14] K Bruce, A Fiech and L Petersen, "Subtyping is not a good 'match' for object-oriented languages", Proc. European Conf. Obj.-Oriented Prog., pub. LNCS, 1241, (New York: Springer Verlag, 1997), 104-127.

[15] J Palsberg and M I Schwartzbach, *Object-Oriented Type Systems* (Chichester: John Wiley, 1994).

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

About the author

Anthony Simons is a Senior Lecturer and Director of Teaching Quality 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.