

Declaring Classes in Pattern Calculus

Barry Jay
University of Technology, Sydney
Barry.Jay@uts.edu.au

ABSTRACT

Pattern calculus is able to represent a fixed system of classes by treating methods as pattern-matching functions, along with a powerful account of both quantified types and of subtypes. This paper adds class, and subclass declarations to support an evolving class hierarchy. Even though method meanings change as new classes are introduced, the properties of existing programs do not change. This provides a common foundation for functional and object-oriented languages in which new functions can be defined on existing data types and also new subclasses can inherit existing methods.

1. INTRODUCTION

Object-orientation is based upon the principle that each object is responsible for its own behaviour. From a foundational viewpoint, this is well understood from several directions, including the λ -calculus [8], the *object calculus* [1] and denotational semantics [7]. In practice, however, it is common for objects to share methods with other objects in the same *class*. When some object u invokes a method m the first step is to determine the class of u and use this to find the code for m . Thus a method name is a function from classes to terms. But what is the theoretical status of a class? The dominant metaphor, among both theorists and practitioners, is that a class is a *type* so that a method is a function from types to terms. This describes *type classes* [18] in functional programming rather well, since referential transparency allows the typecase to be resolved during static type inference. The metaphor is less exact in object-orientation, since dynamic dispatch requires a run-time representation of classes. Unfortunately, this mixing of terms and types creates its own burdens which have increased over time. For example, the comprehensive *nominal theory of objects with dependent types* [19], allows terms built from types and also a distinct type for every single object.

This paper advances a quite different approach to classes and objects, based upon *pattern calculus* [13, 14, 12]. Pat-

tern calculus provides a new foundation for computation that gives data structures the same weight as functions, with their interaction mediated by pattern matching. In particular, the *constructors* used to build data structures play a fundamental role. In this framework, a class can be identified with the constructor used to build its members so that matching against this constructor is the test for class membership. Methods are represented by pattern-matching functions, with one case for each class in which the method is declared. There is no obligation for a run-time notion of class or type as the usual functions upon terms suffice.

A formal account of the static aspects of this approach have already been developed in the monograph [12]. These include the mechanics of method invocation, typing and subtyping with respect to a fixed class hierarchy. It also describes the dynamics of class declaration in the programming language **bondi** [2] that implements pattern calculus. This material will be new to many readers as it never escaped from refereed workshops [15, 16] or technical reports [17] into conferences or journals. Most reviewers requested a more complete story, especially, a formal account of dynamic dispatch. Thus, the *novelty* of this paper concerns the nature of classes and their declarations but its *significance* is that of the overall approach. Not only does it clarify and simplify the foundations of object-orientation, but expressive power is increased. In particular, it allows for the smooth integration of object-orientation with other programming styles supported by pattern calculus and **bondi**, including higher-order functions and generic queries. The standard example of points and coloured points will serve to illustrate the approach, after which a more detailed statement of claims will conclude this introduction.

Suppose that points in the class **Point** have two coordinate fields x and y and that a coloured point in the class **ColourPoint** also has a field for a colour z . A first attempt may represent a mere point by the pattern **Point** (x, y) and a coloured point by **ColourPoint** (z, x, y) but this representation hides the fact that coloured points are also points. The representation of a point adopted here is

$$\mathbf{Point\ rest}\ (x, y)$$

where **rest** matches against any additional fields the point may have (recall *row variables* [21, 20]). In this style, the pattern for coloured points becomes

$$\mathbf{Point}\ (\mathbf{ColourPoint\ rest}\ z)\ (x, y) .$$

Now a coloured point *is* a point so that inheritance of meth-

ods for points by coloured points is automatic. On the other hand, a special case for coloured points will not apply to mere points, since they have no colour. In other words, the natural ordering of patterns can be used to represent the class hierarchy, without any need for types. The approach is equally able to handle method specialisation, by adding cases to existing pattern-matching functions.

Interestingly, pattern-matching techniques can be used to simplify the typing, too. In most class-based approaches, such as Featherweight Java [11], each class also designates a type, with subclassing translating directly to subtyping, so that

$$\text{ColourPoint} < \text{Point}$$

must be added as an axiom when the class of coloured points is declared. However, in pattern calculus, the type of a point is

$$\text{Point}[R] = \text{Point_type } R$$

where R is the type of the additional fields, as represented above by `rest`. Further, the type of a coloured point is

$$\text{ColourPoint}[R] = \text{Point_type } (\text{ColourPoint_type } R) .$$

That is, the type of a coloured point *is* the type of a point, and this fact can be exploited by pattern matching on types. It is convenient to define the type `Point` of all points by

$$\text{Point} = \text{Point}[\text{Top}]$$

where `Top` is the top type, and similarly `ColourPoint` = `ColourPoint[Top]` from which the type inequality above follows directly. Finally, the type of a mere point is

$$\text{Point}[] = \text{Point}[\text{Unit}]$$

where `Unit` does not support any fields. Also, the type of a generic point is `Point[X]` where X is a type variable. There is no need for self-types, as in PolyTOIL [4], just as there is no need for a self parameter for terms.

This approach yields a simple, yet powerful type system that resolves the difficulties that beset the typing of classes, as itemised by Bruce [3, Chap. 3]. Here, in brief, are four highlights.

In most type systems, subtyping and type variables do not interact very well. For example, in `Fsub`, [5] and its successors, such as Generic Java [10], the usual type quantification $\forall X.T$ is generalised to a constrained quantification $\forall X < P.T$ in which the bound type variable X must be instantiated by a type U that is a subtype of the constraint P . In pattern calculus, such constraints are not required. For example, consider the type

$$\forall X < \text{Point}. X \rightarrow X .$$

In our type system, the inequality $X < \text{Point}$ is solved if and only if X is of the form `Point[Y]` for some Y . Hence the inequality above is equivalent to

$$\forall Y. \text{Point}[Y] \rightarrow \text{Point}[Y]$$

which does not require a constraint.

Second, a persistent difficulty when combining type variables and subtyping is that type inequalities, unlike type

equations, do not admit of most general solutions. More precisely, if function types are contravariant in their argument types then the type inequality $X \rightarrow X < S \rightarrow T$ is equivalent to

$$S < X < T$$

so that there is no best solution among the range of possible values for X . The answer is to make function argument types *invariant* under subtyping. This would make no sense when data types are encoded as function types, as in System F [9], but is perfectly natural when data types such as `Point[X]` have an independent existence. The canonical type for a method that acts on points is of the form

$$\text{Point}[X] \rightarrow T$$

When it is invoked by some point of type `Point[R]` then it is enough to substitute R for X ; no subtyping is required.

Third, it is easy to discriminate different sorts of methods by their typing. For example, a point method that returns a point has a variety of possible typings, including

$$\begin{aligned} \text{Point}[X] &\rightarrow \text{Point}[X] \\ \text{Point}[X] &\rightarrow \text{Point} \\ \text{Point}[X] &\rightarrow \text{Point}[] \end{aligned}$$

according to whether it produces a point of the same type as its argument, or an arbitrary point, or a mere point. The possibilities multiply when considering binary methods.

Fourth, there is no need for typecasts. Suppose that a point method has type

$$\text{Point}[X] \rightarrow \text{List } \text{Point}$$

so that it produces a list of arbitrary points. A special case for coloured points may have the type

$$\text{ColourPoint}[Y] \rightarrow \text{List } (\text{ColourPoint}[Y])$$

if it produces a list of coloured points. By contrast, in Java-like languages, the return type would have to remain unchanged, with a type cast being used to convert the resulting points to coloured points. IN the pattern calculus with classes, to add a special case of type $P \rightarrow S$ to a default of type $Q \rightarrow T$ it is enough that any unifier of P and Q provides a solution of $S < T$. No type casts are required.

Having briefly covered the statics, of constructors, pattern matching and typing, it is time to consider the new material on classes and the dynamics of their declarations.

Two related treatments are developed. The *pattern calculus with methods* accounts for the shifting meaning of methods as they are specialised, with classes playing a passive role. The *pattern calculus with classes* emphasises classes over methods, with classes forming a hierarchy, as in Java.

In the pattern calculus with methods, a class is of the form

$$c \{m_i = p_i \rightarrow s_i\}_{i=1..n}$$

where c is the class constructor and each *method declaration* $m_i = p_i \rightarrow s_i$ adds a new case $p_i \rightarrow s_i$ to the meaning of m_i which has pattern p_i and returns the *body* s_i after successful matching. Once the methods have been specialised, there is no need to retain the class. The main reason for having

the class is to ensure *behavioural continuity*, i.e. that new class declarations cannot change the behaviour of existing objects. This is enforced by requiring each pattern p_i to contain the class constructor c .

For example, a class of points may be given by

```
Point {
  distance = Point rest (x, y) → sqrt(x * x + y * y);
  is_good = Point rest (x, y) → x ≥ 0.0}
```

while a class of coloured points may be given by

```
ColourPoint {
  is_good = Point (ColourPoint rest z) (x, y) →
    z == red}
```

Note that fields are not mentioned separately, but appear as the arguments of the constructors `Point` and `ColourPoint`. Also, the class of coloured points is not *defined* to be a subclass of the point class. The inheritance of `distance` and specialisation of `is_good` are handled by pattern matching. This makes for a very simple calculus. It is also quite expressive, since it allows any method to be specialised, irrespective of the class in which it was first declared. That said, the great freedom in pattern construction makes evaluation rather inefficient.

In the *pattern calculus with classes* a class declaration takes the form

$$c < d \{(f_j); m_i = s_i\}_{i=1..n}^{j=1..k}.$$

It declares that the class c is a subclass of d , with fields f_j being variable names, and methods being given by their bodies. For example, the class of points and coloured points above can be re-expressed as

```
Point < Object {
  (x, y);
  distance = sqrt(x * x + y * y);
  is_good = x ≥ 0.0}

ColourPoint < Point {
  z;
  is_good = z == red}.
```

In this calculus, each class has a canonical pattern used in pattern matching. For points and coloured points they are the patterns appearing in the declarations of their methods in the pattern calculus with methods above. Now evaluation can be managed using a store that is organised as a tree of classes, in the more familiar manner.

After this brief overview, the many benefits of the approach can be stated. They fall into two main categories, namely, simplicity and expressiveness that will be addressed in the paper that follows.

Here are six sources of simplicity. First, an object is a data structure whose structure reveals its class, and whose data describes its fields. There is no need for records, or λ -abstractions or a separate concept of self. Second, the internal structure also describes the relationship between classes, in that a coloured point really is a point. Similarly, a type of coloured points really is a type of points. Third, the type system is familiar from standard functional

programming. There is no need for record types, or bounded quantification or self-types. Fourth, subtyping is driven by the nature of the top type. The usual questions about covarience and contravariance [6] are resolved by making function argument types invariant under subtyping, and instead relying upon type unification to align argument types. Fifth, the usual constraints upon the typing of method specialisation have been weakened, so that there is no need for type casts. Sixth, there is clear separation between the rules for pattern matching, derived from the referentially transparent pattern calculus, and the rules for invoking methods, whose dynamic dispatch and implicit self-reference are mediated by the store. The latter support behavioural continuity.

Here are four sources of expressiveness. First, simplicity is itself a source of expressive power. For example, it is easier to express method specialisations under the weaker constraints. Second, the pattern calculi with methods or classes contain the pattern calculus within them, so that it supports the higher-order functions familiar from λ -calculus, and also the new forms of polymorphism supported by pattern calculus. Third, it follows that one may freely combine techniques from functional programming, query programming and object-oriented programming in a single term or program. Fourth, the definitions of new types or classes and of new functions that act upon them may be interleaved at will, thus solving Wadler's *expression problem*. In particular, since objects are data structures, new pattern-matching functions may be defined upon them at any time.

The structure of the paper is as follows. Section 1 is this introduction. Section 2 reviews some basic pattern calculus including path polymorphism. Section 3 uses this to support an adaptive printing program. Section 4 introduces the pattern calculus with methods. Section 5 shows how **bondi**'s algebraic data types can be used to specialise existing functions, by inheriting and adapting arithmetic functions to complex numbers. Section 6 defines behavioural continuity and shows it to be a property of the pattern calculus with methods. Section 7 reviews the type system, proves that evaluation preserves typing, and that extraneous classes do not affect the typing of a term. Section 8 introduces subclasses and shows that the resulting calculus can be translated into earlier one. Section 9 provides examples of object-oriented classes in **bondi**, to represent points and coloured points, circles and coloured circles. Section 10 proves that the pattern calculus with classes is *modular*, in the sense that unrelated class declarations can be permuted. Section 11 reviews the approach to subtyping. Section 12 draws conclusions.

Acknowledgements. Thanks to Thomas Given-Wilson and Jens Palsberg for their constructive comments upon drafts of this paper.

2. PATH POLYMORPHISM

This section introduces a simpler, static version of the *extension calculus* introduced in the monograph [12] with its *path polymorphism*. Path polymorphism supports a generic approach to divide-and-conquer algorithms. In more detail, every data structure is either an *atom* or a *compound*. Further, all compounds can be represented by a single pattern

of the form

$$x y$$

whose variables x and y can be bound to its components, just like `car` and `cdr` in Lisp. This is more general than the standard approach in which pattern-matching is conceived of as a means of querying algebraic data types, since then patterns must be headed by the constructors of the type under consideration.

There are two, disjoint sorts of symbols: the *variables* for the terms (meta-variables x, y, z) which are given values during pattern matching, the *constructors* (meta-variable c) used to build data structures. Particular constructors may be written in typewriter font with a leading uppercase letter. For example, the constructor `Nomatch` will be used to indicate match failure.

The syntax for the *terms* (meta-variables r, s, t, u) and *patterns* (meta-variables p, q) is given by

$$\begin{aligned} t &::= x \mid c \mid tt \mid p \rightarrow t \mid t \\ p &::= x \mid c \mid pp. \end{aligned}$$

The terms are given by *variables*, *constructors*, *applications* and *extensions*, respectively. We may write $p \rightarrow s$ for $p \rightarrow s \mid \text{Nomatch}$. The patterns p are either variables, constructors or applications. Definitions of free variables etc. for patterns are just as for terms.

The sole reduction rule is

$$(p \rightarrow s \mid r) u \longrightarrow \{u/p\} s (r u)$$

which is to be understood as follows. If the match $\{u/p\}$ of p against u is *successful* and produces some substitution σ then the result is σs but if the match above *fails* and yields `none` then the result is $r u$. The delicate point in matching is to determine of an argument that is an application whether it is reducible or not, since reduction may affect its matching. If it is headed by a constructor then it is a compound and ready for matching else matching may have to be suspended until the application is sufficiently reduced. Details can be found in prior work [12, Part I] and will be revisited when considering *values* in Section 4. For example, consider the extension

$$\text{Cons } x y \rightarrow x \mid \text{Nomatch}$$

that produces the head x of a list `Cons $x y$` if it has one, and fails otherwise. When applied to `Cons 3 Nil` it reduces to

$$\begin{aligned} \{\text{Cons } 3 \text{ Nil} / \text{Cons } x y\} x (\text{Nomatch } (\text{Cons } 3 \text{ Nil})) \\ &= \text{some } \{3/x, \text{Nil}/y\} x (\text{Nomatch } (\text{Cons } 3 \text{ Nil})) \\ &= \{3/x, \text{Nil}/y\} x = 3. \end{aligned}$$

When this extension is applied to `Nil` then matching fails and the result is `Nomatch Nil`. That `Nomatch` is a constructor means that match failure can be handled by pattern-matching.

The resulting rewriting system has the usual good properties. In particular it is confluent. Also it is referentially transparent, in that the meaning of a term is determined by the context in which it is declared.

3. PRINTING IN BONDI

In `bondi`, printing of values is mediated by a path polymorphic function `toString`. Here is a first attempt, using syntax that is reminiscent of ML.

```
let rec (toString0: a -> String) =
  | x y -> (toString0 x) ^ " " ^ (toString0 y)
  | x -> prim2string x
```

```
let (print0: a -> Unit) x = printstring (toString0 x)
```

The first declaration asserts that `toString0` is a recursive function of type `a -> String`. That is, it will accept an argument of any type `a` and produce a string. It has two cases: if the argument is a compound then concatenate the strings of the components, separated by a space. If the argument is not a compound then use the primitive operation `prim2string`. Then `print0` prints the string of its argument. For example `print0 3` prints `3` and `print0(Pair 3 4.4)` prints `Pair 3 4.4`.

Now the usual display of pairs employs an infix comma, as in `(3,4.4)`. This can be achieved by adding a new case to the function `toString0`. In a referentially transparent style, this requires a new function `toString1` given by

```
let rec (toString1: a -> String) =
  | Pair x y -> "(" ^ (toString1 x) ^ "," ^
    ^ (toString1 y) ^ ")"
  | x y -> (toString1 x) ^ " " ^ (toString1 y)
  | x -> prim2string x
```

However, the value of `print0(Pair 3 4.4)` will not change since it refers to the original function `toString0`. Rather, one must define a new print function `print1` that uses the new function `toString1`.

This quickly becomes tedious, requiring, in effect, a separate print function for each new type, which must be combined by hand on each occasion. The Haskell solution is to adopt type classes and monads. The effect of this is to make programming dependent upon the type and class system which complicates both the syntax and the evaluation. Note, however, that `print1` operates without reference to types or classes, as the pattern-matching machinery is able to recognise a pair from its internal structure. All that is required is the ability to add a case to an existing function, as follows. Define `toString` and `print` just as for `toString0` and `print0` except that the keyword `rec` has been changed to `ext`, for *dynamic extension*.

```
let ext (toString: a -> String) =
  | x y -> (toString x) ^ " " ^ (toString y)
  | x -> prim2string x
;;
let (print: a -> Unit) x = printstring (toString x);;
```

As before, `print (Pair 3 4)` prints `Pair 3 4`. However, such an extensible function can be extended by a case for pairs

```
toString += | Pair x y -> "(" ^ (toString x) ^ "," ^
  ^ (toString y) ^ ")"
```

after which the program `print (Pair 3 4)` prints (3,4). That is, the meaning of both `toString` and the function `print` have been modified dynamically.

It is harder to reason about programs when their elements may change meaning dynamically but this can be managed in various ways. The main technique developed in this paper is to limit such dynamic extensions to the declaration of new types and classes. In the example above, the special case of `toString` for `Pair` will be introduced at the same time as the constructor `Pair` in a class declaration.

4. CLASSES

This section adds method invocations and class declarations to the static extension calculus. As methods evolve, a *method store* is used to keep track of them, and evaluation is given by a structured operational semantics.

To the variable symbols and constructor symbols are added the *method names* or *methods* (meta-variables m, n). Class declarations will bind constructor names but method names will never be bound.

The syntax for the *terms* (meta-variables r, s, t, u), *patterns* (meta-variables p, q) and *classes* (meta-variable cl) is given by

$$\begin{aligned} t &::= x \mid c \mid tt \mid p \rightarrow t|t \mid t.m \mid cl;t \\ p &::= x \mid c \mid pp \\ cl &::= c \{ \overline{m = p \rightarrow t} \}. \end{aligned}$$

The new term forms are the *invocations* and *class declarations* respectively. A declaration $c\{m_i = p_i \rightarrow s_i\}_{i=1\dots n}$ introduces the constructor c and makes the method declarations $m_i = p_i \rightarrow s_i$. The bar notation $\overline{m = p \rightarrow s}$ is used when the precise indexing is immaterial.

The *free variables* $\text{fv}(-)$ of an expression are a set of variables defined by

$$\begin{aligned} \text{fv}(x) &= \{x\} \\ \text{fv}(c) &= \{\} \\ \text{fv}(r \ u) &= \text{fv}(r) \cup \text{fv}(u) \\ \text{fv}(p \rightarrow s \mid r) &= (\text{fv}(s) \setminus \text{fv}(p)) \cup \text{fv}(r) \\ \text{fv}(t.m) &= \text{fv}(t) \\ \text{fv}(c\{\overline{m = p \rightarrow s}\}; t) &= (\text{fv}(\{\overline{m = p \rightarrow s}\}) \cup \text{fv}(t)) \\ \text{fv}(\{m_i = p_i \rightarrow s_i\}) &= \bigcup_i \text{fv}(s_i) \setminus \text{fv}(p_i). \end{aligned}$$

A term is *closed* if it has no free variables.

The *free constructors* $\text{fc}(-)$ of an expression are a set of constructors defined by

$$\begin{aligned} \text{fc}(x) &= \{\} \\ \text{fc}(c) &= \{c\} \\ \text{fc}(r \ u) &= \text{fc}(r) \cup \text{fc}(u) \\ \text{fc}(p \rightarrow s \mid r) &= \text{fc}(p) \cup \text{fc}(s) \cup \text{fc}(r) \\ \text{fc}(t.m) &= \text{fc}(t) \\ \text{fc}(c\{\overline{m = p \rightarrow s}\}; t) &= (\text{fc}(\{\overline{m = p \rightarrow s}\}) \cup \text{fc}(t)) \setminus \{c\} \\ \text{fc}(\{m_i = p_i \rightarrow s_i\}) &= \bigcup_i \text{fc}(p_i) \cup \text{fc}(s_i). \end{aligned}$$

A term t *avoids* a constructor c if $c \notin \text{fc}(t)$.

Given the notion of free variables and free constructors then α -conversion with respect to bound variables and construc-

tors is defined in the obvious manner. In particular, α -conversion will be used to disambiguate constructors during evaluation. Pattern syntax is *well-formed* if no free variable occurs more than once. A class is *well-formed* if its constructor is free in the pattern of each declared method. This will be crucial to establishing behavioural continuity. We restrict attention to well-formed patterns and classes from now on.

Since the meaning of a method name varies dynamically, a store is required to keep track of the shifting meanings, and an operational semantics must determine the evaluation order. In this calculus, the store will be indexed by method names, so that all the special cases for a single method will be kept together. This is simple but inefficient. Later, we will index by classes.

Evaluation produces *values* (meta-variable v) which are either extensions or *data structures* (meta-variable v_d). They are given by closed terms of the form

$$\begin{aligned} v_d &::= c \mid v_d v \\ v &::= v_d \mid p \rightarrow s|t. \end{aligned}$$

A *method store* (meta-variable M) is given by a function from method names to values. Method names that are not being used will be mapped to the constructor `Nomatch`. The *free constructors* $\text{fc}(M)$ of M are given by the union of all the free constructors of the values in its range. Given a class $c\{m_i = p_i \rightarrow s_i\}_{i=1\dots n}$ where c is not free in M then the store $M, c\{m_i = p_i \rightarrow s_i\}_{i=1\dots n}$ is given by the function that maps m_i to $p_i \rightarrow s_i \mid M(m)$ for $1 \leq i \leq k$ and otherwise behaves like M . Note that α -conversion of a class declaration $cl;t$ can be used to ensure that the class constructor is not free in M .

A *program* is a pair $(M; t)$ of a method store M and a term t . The operational semantics is given by the *evaluation relation*

$$(M; t) \Rightarrow (M'; v)$$

defined in Figure 1, by induction on the structure of the term closed term t .

Note that evaluation does not get stuck since there is an evaluation rule applicable for each sort of closed term. Further properties of evaluation are considered in the Section 6.

5. ALGEBRAIC DATA TYPES IN BONDI

The modification of `toString` to handle pairs in Section 2 was reckless since it changed the existing behaviour of the function. A safer approach is to modify `toString` at the point where pairs are introduced, as follows.

```
datatype Binprod x y = Pair of x and y
with toString +=
| Pair x y -> "(" ^ (toString x) ^ ", "
              ^ (toString y) ^ ")"
```

The effect is the same as before, but now the meaning of `toString` is modified before there is any chance to apply it to some pair: only the declaration of `Pair` has authority to modify the meaning of `toString` on pairs. Further, the

$$\begin{array}{c}
\frac{}{(M; c) \Rightarrow (M; c)} \\
\frac{(M; r) \Rightarrow (M'; v_d) \quad (M'; u) \Rightarrow (M''; v_2)}{(M; r \ u) \Rightarrow (M''; v_d \ v_2)} \\
\frac{(M; r) \Rightarrow (M'; p \rightarrow s|t) \quad (M'; u) \Rightarrow (M''; v_2) \quad (M''; \{v_2/p\}s \ (t \ v_2)) \Rightarrow (M'''; v_3)}{(M; r \ u) \Rightarrow (M'''; v_3)} \\
\frac{}{(M; p \rightarrow s|r) \Rightarrow (M; p \rightarrow s|r)} \\
\frac{(M; u) \Rightarrow (M'; v_1) \quad (M'; M'(m) \ v_1) \Rightarrow (M''; v_2)}{(M; u.m) \Rightarrow (M''; v_2)} \\
\frac{(M, c; t) \Rightarrow (M'; v_1)}{(M; c; t) \Rightarrow (M'; v_1)}
\end{array}$$

Figure 1: Evaluation

bondi interpreter is able to detect unsafe modifications. For example the unsafe declaration

```
datatype Foo = Bar
with toString += (* unsafe extension! *)
| Pair x y -> "(,)"
```

produces the following error message

```
Bar: Foo
error: added cases must use a new constructor .
```

Commonly, there is no need to write special cases for path polymorphic functions since their default behaviour is correct. However, it is possible to do so if required. This can be illustrated by considering the arithmetic of complex numbers, whose addition can be inherited, but whose multiplication requires a special case.

Suppose given operations **plusint** and **plusfloat** for adding integers and floats. Then a generic addition can be defined by

```
let ext (plus : a * b -> a) =
| ((x:Int), (y:Int)) -> x plusint y
| ((x:Float), (y:Float)) -> x plusfloat y
| (x1 x2, y1 y2) -> plus (x1, y1) (plus(x2, y2))
| (x, y) -> if x eqcons y then x else Exception "plus"
;;
```

It adds pairs of integers or of floats in the usual way, and uses path polymorphism to add corresponding components of compounds. The last case checks constructors for equality. Note that the arguments must be allowed to have different types since compounds of the same type may have differently typed components. However, this can be repaired by defining the infix operation **+** by

```
let ((+): a -> a -> a) x y = plus(x, y)
```

The same process can be used to define a generic **minus** and its infix form **-** and **times** and its infix version *****.

Now complex numbers can be declared using

```
datatype Complex = Cart of Float and Float
with toString += | Cart x1 x2 -> (toString x1) ^
" +i" ^ (toString x2)
and times += | (Cart x1 x2, Cart y1 y2) ->
Cart ((x1 * y1) - (x2 * y2))
((x1 * y2) + (x2 * y1))
```

For example, if z is **Cart** 0.0 2.0 then **z+z** prints **0.+i4** and **z*z** prints **-4.+i0**. Thus, addition is inherited but the printing and multiplication have been specialised.

6. BEHAVIOURAL CONTINUITY

The use of a store during evaluation makes it harder to reason about programs. One of the goals of a class-based approach is to support reasoning that operates at the level of a single class, or several related classes. That is, method specialisation should not impact upon the behaviour of objects that belong to pre-existing classes; their behaviour should exhibit a form of continuity. The first step is to formalise behaviour, and then consider how to localise it to a particular class.

A *context* $C[-]$ is a closed term with a hole in it, as usual. Define a *property* to consist of a store M and a context $C[-]$. A closed term t has a property $(M; C[-])$ if $(M; C[t])$ is a program and there is a store M' such that

$$(M; C[t]) \Rightarrow (M'; \mathbf{True})$$

where **True** is a given constructor representing truth. The *behaviour* of a program is the collection of its properties.

Given closed terms s and t , say s is *refined* by t , written

$$s < t$$

if every property of s is also a property of t . Also s and t are *behaviourally equivalent*, written $s \approx t$ if s is refined by t and t is refined by s . While this is a natural approach, it is too strong for our purposes as s may have default properties that happen to apply to all constructors, whether appearing in s or not. For example, if some method m is bound to the constant function $x \rightarrow \mathbf{True}$ and a class c specialises m to $c \ x \rightarrow \mathbf{False}$ then the new value of m is not a refinement of the old value since the latter has a default behaviour which happened to apply to c . So the order must be modified to exclude such possibilities. Define a property $(M; C[-])$ to *avoid* a constructor c if c is not free in either the store M or the context $C[-]$. Now define t to *refine* s *apart from* c , which is written $s <_c t$ if $c \notin \text{fc}(s)$ and any property of s that avoids c is also a property of t .

The goal is to show that the calculus is *behaviourally continuous*, i.e. that whenever a class with constructor c adds a special case for a method, that the new method value refine the old value apart from c .

LEMMA 6.1. *If a pattern p matches against some term u then $\text{fc}(p) \subseteq \text{fc}(u)$.*

PROOF. Straightforward induction. \square

LEMMA 6.2. *Let c be a constructor and $p \rightarrow s \mid r$ be a term. If c is free in p but not free in r then*

$$r <_c p \rightarrow s \mid r .$$

PROOF. Consider a property $(M; C[-])$ of r which avoids c . Without loss of generality, the context merely applies its argument to some term u . Hence $(M; (p \rightarrow s \mid r) u)$ has the same value as $(M; C[r])$ by Lemma 6.1. \square

THEOREM 6.3 (BEHAVIOURAL CONTINUITY). *Let t be a term which avoids the constructor c of some class cl . Then*

$$t <_c cl; t .$$

PROOF. It is enough to consider a context which merely invokes some method m . Now apply Lemma 6.2.

\square

7. SPECIALISING QUANTIFIED TYPES

The technique for typing method invocations and classes is based on that for typing extensions [12, Part II] which will be reviewed first.

The type system itself is fairly standard. There are two classes of type symbols, the *type variables* (meta-variables X, Y and Z) and the *type constants* (meta-variables C and D). To each constructor c is associated a distinct type constant C_c which may be denoted c when there is no risk of confusion. The types syntax is

$$T ::= X \mid C \mid T T \mid T \rightarrow T \mid \forall X.T$$

consisting of *variables*, *constants*, *applications*, *function types* and *quantified types*, respectively. For example, the function for computing the tail of a list has type

$$\forall X. \text{List } X \rightarrow \text{List } X .$$

if List is the type constant for lists. The usual machinery of free type variables $\text{fv}(T)$, α -conversion, etc. hold. The *most general unifier* of types S and T is denoted $\{S = T\}$.

Consider an extension of the form $p \rightarrow s \mid r$ where $p : P$ and $s : S$ and $r : R$. In conventional treatments, every case of a pattern-matching function must have the same type, enforcing that $P \rightarrow S = R$. However, many of the examples introduced earlier do not satisfy this requirement. For example, addition has types $\text{Int} * \text{Int} \rightarrow \text{Int}$ and $\text{Float} * \text{Float} \rightarrow \text{Float}$ as well as types for adding pairs, lists etc. Of course, there is no chance of confusion here, since the type of the argument will disambiguate the alternatives. The central principle is that a special case of type $P \rightarrow S$ can be added to a default of type R if no type ambiguity can result. For example, if R is $Q \rightarrow T$ then it is enough that any unifier of P and Q also unifies S and T . That is, if they have a most general unifier $\{P = Q\}$ then

$$\{P = Q\}S = \{P = Q\}T$$

while if they have no unifier at all then no ambiguity is possible. When this property holds then $P \rightarrow S$ and $Q \rightarrow T$ are *similar*, denoted

$$P \rightarrow S \approx Q \rightarrow T .$$

A further constraint is to ensure that term matching guarantees type matching. This requires that patterns take their most general types. However, the typing of the body s of the case may exploit knowledge that P and Q are the same. Leaving out the contextual information, the rule for typing extensions is thus

$$\frac{p : P \quad s : vT \quad r : Q \rightarrow T}{p \rightarrow s \mid r : Q \rightarrow T} \quad v = \{P = Q\}$$

A fuller account of this is in the monograph [12][Part II, Chap. 12]. For the example of addition, there is a single type $X * Y \rightarrow X$ which relates to the types of all the special cases in our examples.

When classes are introduced, it can easily happen that the same method name is used in totally unrelated ways. In this case, there is no most general type R . Rather, the types of the different cases must be combined into a *choice type* as follows. First, each alternative has a type of the form $\forall X_1. \forall X_2. \dots \forall X_n. P \rightarrow S$ where X_1, \dots, X_n consists of the free type variables of P . Such a *basic method type* may be written $\forall \Delta. P \rightarrow S$ where Δ is X_1, \dots, X_n or be written $P \Rightarrow S$. A general *method type* is of the form

$$M = P_1 \Rightarrow S_1 \ \& \ P_2 \Rightarrow S_2 \ \& \ \dots \ \& \ P_n \Rightarrow S_n$$

where $P_i \rightarrow S_i \approx P_j \rightarrow S_j$ for all $1 \leq i < j \leq n$. The type constant $\&$ is written infix for convenience. Now the rule for typing an invocation is

$$\frac{u : U \quad m : M}{u.m : U.M}$$

where the *type invocation* $U.M$ is defined as follows

$$\begin{aligned} U.(P \Rightarrow S) &= \{U/P\}S \\ U.(P \Rightarrow S \ \& \ M) &= U.(P \Rightarrow S) \quad \text{if defined} \\ U.(P \Rightarrow S \ \& \ M) &= U.M \quad \text{otherwise} \end{aligned}$$

where $\{U/P\}$ is the *type match* of P against U that, if defined, produces a substitution mapping P to U as usual.

The rule for typing a class declaration $cl; t$ will use the class to modify the context in which t is typed, by specialising the types of the methods. Thus, a *type context* Γ is a mapping from variables and method names to types. Given two such contexts Γ and B with disjoint domains, define $\Gamma + B$ to be their union as functions. Also, define

$$\Gamma, m : P \Rightarrow S$$

as follows. If m is not in the domain of Γ then simply add the mapping of m to $P \Rightarrow S$ to Γ . Otherwise, overwrite the existing mapping of m to yield $P \Rightarrow S \ \& \ \Gamma(m)$. The typing rule for a method specialisation is then

$$\frac{\Gamma, m : P \Rightarrow S \vdash p \rightarrow s : P \Rightarrow S}{\Gamma \vdash m = p \rightarrow s : P \Rightarrow S} \quad P \Rightarrow S \approx \Gamma(m)$$

More generally, the typing rule for a class will add all specialised types to the context simultaneously, to support mutual recursion between method declarations.

$$\frac{}{x : X \vdash_0 x : X}$$

$$\frac{c : \forall \Delta. T_c}{\vdash c : T_c}$$

$$\frac{B_1 \vdash_0 p_1 : P_1 \quad B_2 \vdash_0 p_2 : P_2 \quad v = \{P_1 = P_2 \rightarrow X\} \quad \text{fv}(B_1) \cap \text{fv}(B_2) = \{\} \quad X \text{ fresh}}{v(B_1, B_2) \vdash_0 p_1 p_2 : vX}$$

Figure 2: Typing Patterns

The syntax of the *typed pattern calculus with methods* is the same as that for the pattern calculus with methods except for two things. There is now a fixpoint operator **fix** with the usual semantics. Also, the class syntax, which now has the form

$$c\{m_i = p_i \rightarrow s_i : P_i \Rightarrow S_i\}_{i=1\dots n}$$

where the method declarations are required to have explicit types. Further, the type P_i must contain the type constant C_c associated to c . These types do not play any role in evaluation.

The type derivation rules for the patterns and terms of the typed pattern calculus with methods are given in Figures 2 and 3. Each constructor symbol has a given type $\forall \Delta. T_c$ where T_c is not a quantified type and avoids **Top**. [The latter restriction is not strictly necessary, but it does minimise the use of subtyping]. The judgement $\Gamma \vdash_0 p : P$ asserts that the pattern p has type P in the context Γ . The rules are set up so that every pattern takes its *most general type*. This ensures that term matching implies type matching.

Just as method specialisation runs the risk of altering program properties during evaluation, so there is a risk of corrupting the types of existing programs. However, the restrictions on the possible types of specialisation ensure that this cannot happen. Note that evaluation of a class declaration $cl; t$ shifts the class information to the store but this is necessary for the typing of t . To show that evaluation preserves typing, it is convenient to modify the store so that it keeps the types of method declarations, even though these will play no role in the evaluation. To each such method store M is associated a type context $\Gamma = |M|$ which keeps this type information while discarding the terms. Also, define a derivation $\Gamma \vdash t : T$ to *avoid* a constructor c if c is not free in t and C_c does not appear anywhere in the derivation.

THEOREM 7.1. *Consider an evaluation $(M; t) \Rightarrow (M'; v)$. If there is a type derivation $|M| \vdash t : T$ then there is a type derivation*

$$|M'| \vdash v : T.$$

PROOF. The proof is by induction upon the structure of the evaluation. It adapts the proof of [12][Theorem 12.4] to handle method invocations and class declarations. The former are similar to applications of extensions. For the latter, it is enough to observe that

$$|M, c\{m = p \rightarrow s : P \Rightarrow S\}| = |M|, \{m : P \Rightarrow S\}.$$

$$\frac{}{\Gamma, x : T \vdash x : T}$$

$$\frac{c : \forall \Delta. T_c}{\Gamma \vdash c : \forall \Delta. T_c}$$

$$\frac{\Gamma \vdash r : U \rightarrow S \quad \Gamma \vdash u : U}{\Gamma \vdash r u : S}$$

$$\frac{\Gamma \vdash r : Q \rightarrow T \quad B \vdash_0 p : P \quad v(\Gamma+B) \vdash s : vT \quad v = \{P = Q\}}{\Gamma \vdash p \rightarrow s \mid r : Q \rightarrow T}$$

$$\frac{\Gamma \vdash t : \forall X. T}{\Gamma \vdash t : \{U/X\}T}$$

$$\frac{\Gamma \vdash t : T}{\Gamma \vdash t : \forall X. T} \quad X \notin \text{fv}(\Gamma)$$

$$\frac{}{\Gamma \vdash \mathbf{fix} : (T \rightarrow T) \rightarrow T}$$

$$\frac{\Gamma \vdash u : U}{\Gamma \vdash u.m : U.\Gamma(m)}$$

$$\frac{\Gamma, \{m_i : P_i \Rightarrow S_i\} \vdash p_i \rightarrow s_i : P_i \Rightarrow S_i \vdash \quad P_i \Rightarrow S_i \approx \Gamma(m_i)}{\Gamma \vdash c\{m_i = p_i \rightarrow s_i : P_i \Rightarrow S_i\}}$$

$$\frac{\Gamma \vdash c\{m_i = p_i \rightarrow s_i : P_i \Rightarrow S_i\} \quad \Gamma, \{m_i : P_i \Rightarrow S_i\} \vdash t : T}{\Gamma \vdash c\{m_i = p_i \rightarrow s_i : P_i \Rightarrow S_i\}; t : T}$$

Figure 3: Typing

□

THEOREM 7.2. *If there is a derivation of $\Gamma \vdash cl$ and also a derivation of $\Gamma \vdash t : T$ that avoids c then there is a derivation of $\Gamma \vdash cl; t : T$.*

PROOF. The proof is by induction upon the structure of the typing of t . The only case of interest is when t is an invocation $u.m$ where $u : U$ and m has a special case $p \rightarrow s : P \Rightarrow S$ in cl . Now C_c is free in P but not free in U so $U.(P \Rightarrow S \ \& \ \Gamma(m)) = U.\Gamma(m)$ which yields the result. □

8. SUBCLASSES

Although the pattern calculus with methods has behavioural continuity, it has some serious practical limitations. First, it is not *modular*, as the order in which classes are declared may affect behaviour. Second, evaluation is rather inefficient as method invocation will check many useless cases. Third, the store does not “compile” classes independently, but modifies the value of each method that a new class specialises.

The primary limitation is illustrated by an example. Suppose that method m is specialised by $(c, x) \rightarrow \mathbf{True}$ in class c and by $(y, d) \rightarrow \mathbf{False}$ in class d . Now the value of $(c, d).m$ depends upon the order in which the cases are considered.

This problem can be avoided by constraining the patterns of methods so that the class constructor appears in a specified position, so that $(y, d) \rightarrow \text{False}$ is not well-formed as a method declaration. This is reminiscent of the problems associated with binary methods.

The solution is to create a class hierarchy. Classes are replaced by *subclasses* of the form

$$cl ::= c < d \{(f_j); m_i = s_i\}_{i=1 \dots n}^{j=1 \dots k}.$$

Now a class c is introduced as a subclass of some existing class d . Note that while c is bound by the declaration, the constructor d is free here. The top of the hierarchy is a given constructor `Object`. Also, (f_j) is a tuple of variable names that represent the *fields* of the object, while the sequence $m_i = s_i$ is of method declarations. The subclassing $c < d$ and the fields are used to build the canonical pattern p_c for an object in the class, so that a method specialisation $m_i = s_i$ here behaves like the special case $m_i = p_c \rightarrow s_i$ in the pattern calculus with methods. For example, the canonical pattern for the point class in Section 1 is

$$\text{Point rest } (x, y) .$$

The general approach to building the canonical pattern p_c for a class c is as follows. To each class c is associated a list of tuples of the field names, listing all of the fields required to build an object. It is given by

$$\begin{aligned} fdss(\text{Object}) &= \text{Nil} \\ fdss(c < d\{(f_j); \dots\}) &= \text{Cons } (f_j) \text{ } fdss(d) . \end{aligned}$$

Then a *new object* of class c is given by

$$\begin{aligned} \text{new Object rest Nil} &= \text{Object rest Un} \\ \text{new } c \text{ rest } (\text{Cons } fds \text{ } fdss) &= \text{new } d \text{ } (c \text{ rest } fds) \text{ } fdss \end{aligned}$$

where d is the parent class of c . Now the canonical pattern for a class c is given by

$$p_c = \text{new } c \text{ rest } (fdss \text{ } c)$$

where the variable `rest` is reserved for this purpose.

Conversely, the class `class`(v) of a data structure v is defined by

$$\begin{aligned} \text{class}(c \ v_1 \ v_2) &= \text{class}(v_1) \text{ if this is defined} \\ \text{class}(c \ v_1 \ v_2) &= c \text{ otherwise} \\ \text{class}(v) &= \text{undefined, otherwise} \end{aligned}$$

Also, the *classes* $cl(t)$ of a term t is a set of class constructors required as parents for class declarations in t . It is defined by

$$\begin{aligned} cl(x) &= \{\} \\ cl(c) &= \{\} \\ cl(r \ u) &= cl(r) \cup cl(u) \\ cl(p \rightarrow s \mid r) &= cl(s) \cup cl(r) \\ cl(u.m) &= cl(u) \\ cl(c < d\{\dots\}; t) &= \{d\} \cup (cl(t) \setminus \{c\}) . \end{aligned}$$

The store is now given by a tree of class declarations, which may be denoted as a sequence. The *classes* $cl(M)$ of the store is given by adding `Object` to the set of classes declared in M . A *program* is a pair $(M; t)$ in which $cl(t) \subseteq cl(M)$.

The operational semantics are based on those of the pattern calculus with methods, except that the rule for method invocation is now

$$\frac{(M; u) \Rightarrow (M'; v) \quad M'(v.m) = s \quad (M'; s) \Rightarrow (M''; v_2)}{(M; u.m) \Rightarrow (M''; v_2)}$$

Here $M'(v.m)$ identifies the class c of v in M' and then finds the smallest superclass d of c in which the method m is specialised by some $m = s$. The result is given by $\{v/p_d\}s$.

THEOREM 8.1. *There is a translation of the pattern calculus with classes into the pattern calculus with methods that preserves evaluation.*

PROOF. The translation is as described above. The proof that it preserves evaluation is by induction. \square

Thus, the raw expressive power of the class hierarchy is contained within the pattern calculus with methods. However, the class hierarchy does support modularity, as will be shown in Section 10.

9. OBJECT-ORIENTATION IN BONDI

Classes in `bondi` are essentially the same as those of the pattern calculus with classes but there are several small differences. The fields are represented as a sequence separated by semi-colons. The method bodies are enclosed in braces. Also, fields are *stateful*, i.e. given by assignable references which are usually accessed by get- and set-methods. Further, the result of invoking a method is required to be a function. The unit value `Un` may be represented by `()`. The variable `this` is bound to the canonical pattern. Similarly, a pattern such as $(z : \text{Point})$ may be read as a type restriction, that z be a point but is actually shorthand for binding z to the canonical pattern for a point. Let us revisit the canonical examples of points and coloured points.

```
class Point {
x_coord : Float;
y_coord : Float;
get_x_coord = { |() -> !this.x_coord }
set_x_coord = { fun d -> this.x_coord = d }
get_y_coord = { |() -> !this.y_coord }
set_y_coord = { fun d -> this.y_coord = d }
is_good = { this.get_x_coord() >= 0.0 }
with toString += | (z:Point) ->
  "{x=" ^ (toString (z.get_x_coord())) ^
  ",y=" ^ (toString (z.get_y_coord())) ^
  "}"
}
```

It has the usual get and set methods for its fields, the x- and y-coordinates, and a simple predicate `is_good` which will be specialised later. It also adds a case for the extensible function `toString`. Note that `toString` is a path polymorphic function, not a method; there is no difficulty in combining the functional and object-oriented styles. Here is an example of its use.

```
~~ let pt = new Point;;
pt: Point[]
pt = {x=_void,y=_void}
~~ pt.set_x_coord 5.5;
```

```
pt.set_y_coord 2.2;
pt;;
it: Point[]
it = {x=5.5,y=2.2}
```

A new point has type `Point []` being the type of a *mere* point with no additional fields.

It is an elementary matter to define new functions that act on points. For example, here is a new function for computing the distance between points:

```
distanceto = | (pt1:Point) -> | (pt2:Point) ->
  let x_diff = pt1.get_x_coord() - pt2.x_coord() in
  let y_diff = pt1.get_y_coord() - pt2.y_coord() in
  sqrt (x*x + y*y)
```

This illustrates the solution of the expression problem.

Now consider some colours

```
datatype Colour = Colour of Int
with toString += | Colour x ->
(| 1 -> "red"
 | 2 -> "green"
 | 3 -> "blue"
 | y -> "Colour "^(toString y))
x
;;
let red = Colour 1;;
let green = Colour 2 ;;
let blue = Colour 3;;
```

and a class of coloured points

```
class ColourPoint extends Point {
  colour : Colour;
  get_colour = { |() -> !this.colour }
  set_colour = { fun x -> this.colour = x }
  is_good = { |() -> this.get_colour() == red;;
with toString += | (z:ColourPoint) ->
  "{x=" ^ (toString (z.get_x_coord())) ^
  ",y=" ^ (toString (z.get_y_coord())) ^
  ",colour=" ^ (toString (z.get_colour())) ^
  "}"
}}
```

For example,

```
~~ let cp1 = new ColourPoint;;
cp1: ColourPoint[]
cp1 = {x=_void,y=_void,colour=_void}
~~ cp1.set_x_coord(-2.2);
cp1.set_y_coord(3.3);
cp1.set_colour(red);
cp1;;
it: ColourPoint[]
it = {x=-2.2,y=3.3,colour=red}
~~ cp1.is_good;;
it: Bool
it = True
```

The type `ColourPoint []` is the type of a mere coloured point. It is *not* a subtype of `Point []` since a coloured point is not a mere point, but it is a subtype of `Point` which will be the type of arbitrary points.

Coloured circles will be used to illustrate the expressive power of method specialisation: the method for getting the central point will be specialised to coloured circles, so that the centre is made to take the colour of the circle, even though the point at the centre may have already had a colour. Define a class of circles by

```
class Circle {
  centre : Point;
  radius :Float;
  get_centre = { |() -> !this.centre }
  set_centre = { fun x -> this.centre = x }
  get_radius = { |() -> !this.radius }
  set_radius = { fun x -> this.radius = x }
}
```

and a class of coloured circles

```
class ColourCircle extends Circle {
  col : Colour ;
  get_col = { |() -> !this.col }
  set_col = { fun x -> this.col = x }
  get_centre = { |() ->
let res = new ColourPoint in
res.set_x_coord ((super.get_centre()).get_x_coord());
res.set_y_coord ((super.get_centre()).get_y_coord());
res.set_colour (this.get_col());
res }
}
```

In this sub-class, the method `get_centre` now produces a coloured point, irrespective of whether the point at the centre has a colour. For example,

```
let x = new ColourCircle in
x.centre = cp1;
x.col = blue;
(!x.centre,x.get_centre());;
```

shows that `get_centre` generates a fresh (coloured) point. The type of the default for `get_centre` is

`Circle[a] -> Unit -> Point`

while the type for its specialisation is

`ColourCircle [b] -> Unit -> ColourPoint`

Note that the type of the specialisation is *not* similar to that of the default, since `ColourPoint` is a proper sub-type of `Point`. Rather, the latter method type is a *specialisation* of the former since any substitution that relates the types `ColourCircle [b]` and `Circle[a]` solves the inequality

`Unit -> ColourPoint < Unit -> Point .`

The underlying theory will be recalled in Section 11.

10. MODULARITY

As the canonical patterns associated to classes are completely determined, the order in which classes are declared makes no difference to program properties, provided that super-classes are declared before subclasses. This is formalised in Theorem 10.3 after establishing some basic properties about the patterns involved.

Define the *compatibility* $p \parallel q$ of patterns p and q by

$$\begin{aligned} x \parallel q &= \text{true} \\ p \parallel x &= \text{true} \\ c \parallel c &= \text{true} \\ p_1 \ p_2 \parallel q_1 \ q_2 &= p_1 \parallel q_1 \text{ and } p_2 \parallel q_2 \\ p \parallel q &= \text{false, otherwise.} \end{aligned}$$

LEMMA 10.1. *Patterns p and q are compatible if and only if there is a term u that p and q both match.*

PROOF. Straightforward. \square

LEMMA 10.2. *If patterns p and q are incompatible then*

$$\begin{array}{c|c} q \rightarrow t & p \rightarrow s \\ \hline p \rightarrow s & \approx & q \rightarrow t \\ \hline r & & r \end{array}$$

for any such pair of extensions.

PROOF. The proof is by induction on the length of the evaluation. Without loss of generality, the context is an application of its hole to a value. Now the result follows from Lemma 10.1. \square

THEOREM 10.3 (MODULARITY). *Consider two classes cl_1 and cl_2 such that neither is the superclass of the other. Then for every term t , the terms $cl_1; cl_2; t$ and $cl_2; cl_1; t$ have the same properties.*

PROOF. Let $(M; C[-])$ be a property of $cl_1; cl_2; t$. The proof that it is a property of $cl_2; cl_1; t$ is by induction on the structure of the context $C[-]$. The only case of interest is when the context is empty, in which case apply Lemma 10.2. \square

11. SUBTYPING

The subtyping relation is driven by a single axiom, that every type is smaller than the *top type* Top , as given in Figure 4. At first glance, these rules appear to be too simple to be both sound and effective. In particular, function types are *invariant* in their argument types. The current consensus is that function types should be contravariant in their arguments, so that $P \rightarrow S < Q \rightarrow T$ if $S < T$ and $Q < P$. However, consider the type inequality

$$X \rightarrow X < S \rightarrow T.$$

Using contravariance, this reduces to the constraint

$$S < X < T$$

which may fail to have a most general solution. Using invariance, it becomes

$$S = X < T$$

which has a most general solution, if it has one at all.

Even more radical is the treatment of data types such as products. Here $P * S < Q * T$ if and only if $P = Q$ and $S < T$

$$\begin{array}{c} \frac{}{T < \text{Top}} \qquad \frac{S < T}{F \ S < F \ T} \qquad \frac{}{T < T} \\ \\ \frac{S < T}{P \rightarrow S < P \rightarrow T} \qquad \frac{S < T}{\forall X. S < \forall X. T} \end{array}$$

Figure 4: Subtyping

since $P * S$ is syntactic sugar for $\text{Binprod } P \ S$. Even though this offends every algebraic principle of symmetry it works very well here, being just enough to model subclassing in the type system while keeping the algebra manageable. For example, consider the type inequality

$$Y * Z < X * X.$$

Using covariance, it requires X to be an upper bound of Y and Z . If Y and Z are unrelated then X must be Top but if Y and Z become equal then X can take the same value. More generally, if Y and Z both turn out to be types of points then X might be Point . Using invariance then the inequality becomes simplified to $Z < X = Y$.

Summarising, the main benefit of this approach is that the supertypes of any type form a finite linear order, just like the class hierarchy. In turn, this simplifies the solution of type inequalities. In particular, it clarifies the nature of binary methods, since subtyping can only apply to one of the arguments.

When declaring a (sub)class c then the canonical pattern p_c has a canonical type P_c and a declared method m has a type of the form $P_c \Rightarrow S$. Further, it is safe to add this specialisation if, for every existing type $P_d \Rightarrow T$ of m , any unifier ν of P_c and P_d also solves $S < T$. Since type substitutions preserve subtyping, it is enough to check the most general unifier of P_c and P_d , if it exists. In this situation we write

$$P_c \Rightarrow S \ll P_d \Rightarrow T$$

In most cases, the type inequality will turn out to be an equation. For example, one may add a special case of type $\text{ColourPoint}[b] \Rightarrow \text{ColourPoint}[b]$ to a default method of type $\text{Point}[a] \Rightarrow \text{Point}[a]$. However, suppose given a method m of type $\text{Point}[a] \Rightarrow \text{List Point}$ where the list may contain a mix of mere points, coloured points etc. If equations were required then any specialisation to coloured points must have type $\text{ColourPoint}[Y] \Rightarrow \text{List Point}$ but inequalities allow the specialisation to produce a list of coloured points since

$$\begin{array}{l} \text{ColourPoint}[b] \Rightarrow \text{List (ColourPoint}[b]) \ll \\ \text{Point}[a] \Rightarrow \text{List Point} \end{array}$$

Thus, there is no need for type casts to repair weaknesses in the typing rules. The typing rules are derived from those of the pattern calculus with methods by: adding a subsumption rule; and replacing the type similarity condition $P_i \Rightarrow S_i \approx \Gamma(m_i)$ in the rule for class declarations, by

$$P_i \Rightarrow S_i \ll \Gamma(m_i).$$

12. CONCLUSION

The pattern calculus with methods shows that the essential elements of a class are its constructor and its method declarations. The constructor is used to build the data structures that contain the fields of an object, while the methods support dynamic dispatch. This account does not require a privileged notion of self, or a type system to operate. Rather, method invocation is controlled by pattern matching, since the pattern language is already rich enough to capture class information. The resulting calculus contains as a subsystem the usual, referentially transparent, pattern calculus. Also, even though the meanings of methods change dynamically, the presence of extraneous classes cannot change the properties of existing programs. The creation of a class hierarchy makes it easier to reason about classes and programs, and is better able to support efficient implementation but does not actually increase expressive power compared to the pattern calculus with methods.

The typing and subtyping of these calculi follows the approach developed in [12]. Most surprising is that function argument types are *invariant* under subtyping. This is possible since type variables may be instantiated to achieve the desired type equalities. As a result there is no need for type casts when specialising methods.

All of the theory and examples in the paper have been implemented in **bondi**, using very little more machinery than that described here. The next step is to put all this expressive power to work.

This paper has illustrated the ten claims made in the introduction about the simplicity and expressive power of this approach to object-orientation. Many of these claims were already made in the pattern calculus monograph but their full significance only becomes apparent when combined with the new account of the dynamics of class declarations and method invocation, and indeed of the nature of methods and classes. Perhaps surprisingly, most of the difficulties melt away once the distinction between functions and data structures is recognised as being fundamental.

13. REFERENCES

- [1] M. Abadi and L. Cardelli. *A Theory of Objects*. Monographs in Computer Science. Springer, 1997.
- [2] bondi programming language. www-staff.it.uts.edu.au/~cbj/bondi.
- [3] Kim Bruce. *Foundations of Object-Oriented Languages: Types and Semantics*. The MIT Press, 2002.
- [4] Kim B. Bruce, Adrian Fiech, Angela Schuett, and Robert van Gent. Polytoil: A type-safe polymorphic object-oriented language. *ACM Trans. on Progr. Lang. and Sys.*, 25(2):225–290, 2003.
- [5] Luca Cardelli, John C. Mitchell, Simone Martini, and Andre Scedrov. An extension of system F with subtyping. In Takayasu Ito and Albert R. Meyer, editors, *Proc. of 1st Int. Symp. on Theor. Aspects of Computer Software, TACS'91, Sendai, Japan, 24–27 Sept 1991*, volume 526, pages 750–770. Springer-Verlag, Berlin, 1991.
- [6] Giuseppe Castagna. Covariance and contravariance: conflict without a cause. *ACM Trans. Program. Lang. Syst.*, 17(3):431–447, 1995.
- [7] W. Cook and J. Palsberg. A denotational semantics of inheritance and its correctness. *SIGPLAN Not.*, 24(10):433–443, 1989.
- [8] Kathleen Fisher, Furio Honsell, and John C. Mitchell. A lambda calculus of objects and method specialization. *Nordic J. of Computing*, 1(1):3–37, 1994.
- [9] J-Y. Girard, Y. Lafont, and P. Taylor. *Proofs and Types*. Tracts in Theoretical Computer Science. Cambridge University Press, 1989.
- [10] Generic java. <http://www.cis.unisa.edu.au/~pizza/gj/>.
- [11] Atsushi Igarashi, Benjamin C. Pierce, and Philip Wadler. Featherweight java: a minimal core calculus for java and gj. *ACM Trans. Program. Lang. Syst.*, 23(3):396–450, 2001.
- [12] Barry Jay. *Pattern Calculus: Computing with Functions and Structures*. Springer, 2009.
- [13] Barry Jay and Delia Kesner. Pure pattern calculus. In *Programming Languages and Systems, 15th European Symposium on Programming, ESOP 2006, Held as Part of the Joint European Conferences on Theory and Practice of Software, ETAPS 2006, Vienna, Austria, March 27–28, 2006, Proceedings (ed: P. Sestoft)*, pages 100–114, 2006. Revised version at www-staff.it.uts.edu.au/~cbj/Publications/purepatterns.pdf.
- [14] Barry Jay and Delia Kesner. First-class patterns. *Journal of Functional Programming*, 19(2):191–225, 2009.
- [15] C.B. Jay. Methods as pattern-matching functions. In *Foundations of Object-Oriented Languages, 2004*, page 16 pp, 2004. <http://www.doc.ic.ac.uk/~scd/F00L11/patterns.pdf>.
- [16] C.B. Jay. Type variables simplify sub-typing. In D. Kesner, M-O. Stehr, and F. van Raamsdonk, editors, *Higher-Order Rewriting, electronic proceedings*, 2006. <http://hor.pps.jussieu.fr/06/proc/jay2.pdf>.
- [17] C.B. Jay. Objects not subjects! <http://www-staff.it.uts.edu.au/~cbj/Publications/ons.pdf>, 2009.
- [18] M.P. Jones. A system of constructor classes: overloading and implicit higher-order polymorphism. *J. of Functional Programming*, 5(1), 1995.
- [19] Martin Odersky, Vincent Cremet, Christine Röck, and Matthias Zenger. A nominal theory of objects with dependent types. In *ECOOP 2003 – Object-Oriented Programming*, volume 2743 of *Lecture Notes in Computer Science*, pages 201–204. Springer, 2003.
- [20] Didier Rémy and Jérôme Vouillon. Objective ML: An effective object-oriented extension to ML. *Theory and Practice of Object Systems*, 4(1):27–50, 1998.
- [21] Mitchell Wand. Type inference for objects with instance variables and inheritance. In Carl A Gunter and John C Mitchell, editors, *Theoretical Aspects of Object-Oriented Programming : Types, Semantics, and Language Design*, pages 97–120. Massachusetts Institute of Technology, 1994.