

Objects not Subjects!

Barry Jay

University of Technology, Sydney

Abstract. Pattern calculus is a new foundation for computation in which the tension between functions and data structures is resolved within pattern-matching functions. This removes one of the main drivers of programming language design, so that the **bondi** programming language is able to represent all of the main programming styles. The basic theory is in a forthcoming monograph on pattern calculus. This paper will use **bondi** to demonstrate some consequences. First, function extensions allow the addition of both new functions to old data types (in functional programming style) and also new data types to old functions (method specialisation). Second, the basic concepts of object-orientation: object, class, self, inheritance, specialisation and dynamic dispatch can all be expressed using the more fundamental concepts of data structure, algebraic type, recursion, pattern-matching, function extension and state.

Keywords: pattern calculus, object-orientation, **bondi**

1 Introduction

Pattern calculus [9, 8, 6, 11, 10, 5, 7, 4] is a new foundation for computing that combines the expressive power of functions and data structures within pattern-matching functions. This combination is able to express all of the main programming styles: its implementation as the programming language **bondi** is able to support functional, imperative, relational and object-oriented styles. The story is told in the forthcoming book [4] on pattern calculus (untyped, typed and sub-typed) and **bondi**.

The main purpose of this paper is to address the transition from calculus to programming language in more detail, by breaking down the distinctions between declarations of classes and of algebraic data types. This has two main benefits.

First, the declarations of types and of functions that act on them can be interleaved at will: new functions can act on old data types (just as in functional programming) and old functions (methods) can be specialised for new types. This solves the *expression problem* [13].

Second, it shows that the object-oriented programming style does not require a separate foundation: the concepts of self, object, class and dynamic dispatch can all be handled within the pre-existing framework by careful coding. Now objects are mere data structures without any ability to act: they are objects, not subjects.

To give a formal account of these results would entail a recapitulation of many concepts, including: pattern matching; quantified function types $[\Delta] P \rightarrow S$;

type-matching; a novel structural sub-typing; and implicit typing before getting to the *sub-typed extension calculus*. Then type and class declarations must be added. Of course, this is beyond the scope of the current paper, but it turns out that **bondi** programming does not require mastery of many new concepts, unless one looks under the hood. That is, the usual things that one wants to do are easy, but pushing the boundaries may generate error messages whose meaning is clear even though the rationale is not. For this reason, the issues will all be addressed informally through examples and counter-examples written in **bondi**.

The expression problem will be addressed through consideration of the generic arithmetic operations in **bondi**, and their extension to a new type of complex numbers.

The standard functional programming solution defines an algebraic data type of complex numbers and separate functions for their addition and multiplication. This creates duplicate functions for one concept, which is bad, but new functions, such as squaring, can be defined at any time.

The standard object-oriented approach defines a class of numbers, with subclasses for floating point numbers and complex numbers. Now each object has its own notion of multiplication, but squaring will require a separate sub-class, so now classes are being duplicated.

In **bondi**, addition and multiplication are both generic operations, of type $a \rightarrow a \rightarrow a$ where a is a type variable. For example, the addition of pairs $(1,2.2) + (3,4.4)$ evaluates to $(4,6.6)$ without the need for a special rule. This *path polymorphism* is one of the new forms of polymorphism supported by pattern calculus. When an algebraic data type of complex numbers is defined, the generic addition yields the desired result without any further effort, but multiplication remains an issue. This is resolved by *adding a case* to the existing multiplication function. Although the syntax is straightforward, a modicum of care was required to add the new case within the existing recursion.

To preserve referential transparency, this must produce a new generic multiplication function separate from the original. While this simplifies reasoning, it is a little tedious, since functions defined using multiplication, such as squaring, must be re-defined.

Alternatively, one may use imperative features to *update* the case analysis. Now existing functions are updated simultaneously.

The final solution defines an object-oriented class of complex numbers that “inherits” addition from the global function, but adds a case to the generic multiplication.

This approach addresses some of the foundational questions associated with object-oriented programming. The examples above show that objects and classes can be represented by data structures and data types. Self-reference is handled by standard accounts of recursion. Type variables can represent unknown information about the type of “self” (rather like *row variables* [14, 12] so there is no need for “self types”). State and dynamic dispatch are handled by standard imperative features. Inheritance and method specialisation are handled by pattern matching.

Of course, these examples do not address the usual questions about sub-typing (see, e.g. [2, Chap. 3]). Should sub-typing be structural or declarative? When is sub-typing covariant, contravariant or invariant? Given the presence of type variables, how should type inequalities be solved? How are binary methods handled?

It turns out that they have simple answers, if one starts from pattern calculus. Central to its approach is that functions and data structures are of equal status. Now, objects are data structures, so that sub-typing is fundamentally a property of data types, not of function types, so that subsumption can be limited to the former. In particular, function argument types are *invariant* under sub-typing,

In brief, these questions have also been answered as follows. Sub-typing is given by some simple structural rules. Type constructions are either covariant or invariant, *never* contravariant. In particular, function argument types are invariant under sub-typing. Some care is taken to ensure that the super-types of a type always form a finite linear order, much as the super-classes of a Java class do. This makes it possible to solve the type inequalities that arise. A binary method has a type of the form $P \rightarrow Q \rightarrow S$ or, equivalently $Q * P \rightarrow S$. In either case, Q is invariant with respect to sub-typing, so cannot create any difficulties. Such invariance solves some problems but introduces others, which are solved by restricting the possible types of methods to *method types*.

In this way, the differences between functions and methods appear as subtle differences in their typing: a function may be typed by any function type, but its extensions with special cases may not exploit sub-typing; a method must take a method type but then its special cases may exploit sub-typing.

Of course, this approach is radically different from the most popular approaches to object-orientation, so it is natural to ask how they compare. In the absence of types, there is a simple translation from the *object calculus* [1] to the pattern calculus given in [4, Chap. 6]. The same is probably true of other approaches. However, there are no translations of sub-typed calculi that support contravariance, for the obvious reason that here sub-typing is never contravariant. In spirit the sub-typing is quite close to that of Java, but no detailed analysis has been performed.

The emphasis in this paper will be on examples in **bondi** (version 1.8) that illustrate the issues, but a brief account of some of the issues is given in the appendix. The sections of the paper are as follows: Introduction; Standard Functional Programming Style; Standard Object-Oriented Style; Path Polymorphism; Adapting existing functions; Updating existing functions; Updating existing functions within classes; Coloured Circles; Linked Lists; Counter-examples in **bondi**; Conclusions; and an appendix, Types in **bondi**.

2 Standard Functional Programming Style

bondi supports familiar machinery for declaring data types and object-oriented classes. Consider a data type of complex numbers in **bondi**, given by

```
~~ datatype Complex = Cart of Float and Float;;
```

```
Cart: Float -> Float -> Complex
```

The prompt `~~` represents the waves on Bondi Beach. The double semi-colon `;;` completes an input. This declares `Complex` to be a type constant, and `Cart` to be a term constructor, of the type shown in the response. For example

```
~~ let z = Cart 3.0 1.0;;
z: Complex
z = Cart 3. 1.
```

Following the functional programming style, one may declare functions that act on complex numbers at any time. For example

```
~~ let plus_complex =
| (Cart x1 y1, Cart x2 y2) -> Cart (x1 + x2) (y1 + y2);;
plus_complex: Complex * Complex -> Complex
```

declares addition of complex numbers as a pattern-matching function, as indicated by the vertical bar. Similarly, one may define multiplication and squaring by

```
~~ let times_complex =
| (Cart x1 y1, Cart x2 y2) ->
    Cart ((x1 timesfloat x2) minusfloat (y1 timesfloat y2))
        ((x1 timesfloat y2) plusfloat (x2 timesfloat y1))
times_complex: Complex * Complex -> Complex
~~ let square_complex z = times_complex (z,z);;
square_complex: Complex -> Complex
~~ square_complex z;;
it: Complex
it = Cart 8. 6.
```

This approach is flexible in that one may define new complex number functions at any time, but it is annoying to have to define new functions separate to the existing ones, e.g. `times_complex` is unrelated to `timesfloat`.

3 Standard Object-Oriented Style

Object-orientation allows a single method for addition, or for multiplication, to be specialised on a new sub-class of complex numbers.

```
~~ class Number {
plusnumber = { (Exception : Number -> Number)}
timesnumber = { (Exception : Number -> Number)}
}
class Number {
plus: Number[a] -> Number -> Number
times: Number[a] -> Number -> Number
}
```

The class `Number` behaves like an interface, specifying the types of the methods `plus` and `times` but they are given exceptions as values. A sub-class of complex numbers can be defined by

```

~~class ComplexNumber extends Number {
real : Float;
imag: Float;
get_real = { | () -> !this.real }
set_real = { | (x:Float) -> this.real = x }
get_imag = { | () -> !this.imag }
set_imag = { | (x:Float) -> this.imag = x }
plusnumber = { fun (x: Number) -> match x with
  | (y: ComplexNumber) ->
    let res = new ComplexNumber in
    res.set_real (this.get_real() + y.get_real());
    res.set_imag (this.get_imag() + y.get_imag());
    res
  | (y: Number) -> (Exception : Number)
  }
timesnumber = { fun (x: Number) -> match x with
  | (y: ComplexNumber) ->
    let res = new ComplexNumber in
    res.set_real (this.get_real() * y.get_real());
    res.set_imag (this.get_imag() * y.get_imag());
    res
  | (y: Number) -> (Exception : Number)
  }
}
class ComplexNumber {
real: ComplexNumber[a] -> ref Float
imag: ComplexNumber[a] -> ref Float
get_real: ComplexNumber[a] -> Unit -> Float
set_real: ComplexNumber[a] -> Float -> Unit
get_imag: ComplexNumber[a] -> Unit -> Float
set_imag: ComplexNumber[a] -> Float -> Unit
plusnumber: ComplexNumber[a] -> Number -> Number
timesnumber: ComplexNumber[a] -> Number -> Number
}

```

Note how the second argument of the methods `plus` and `times` must be of type `Number` to respect the typing. Fortunately, it is still possible to specialise with respect to the second argument using yet more pattern matching.

This approach is flexible in that one may add new cases to the existing methods for `plus` and `times` by sub-classing. One the other hand, to support squaring requires new sub-classes of both `Number` and of `ComplexNumber` which is tedious.

4 Path Polymorphism

Path polymorphic functions are able to traverse arbitrary data structures by recursively matching against a pattern of the form $x\ y$ where x and y are both matchable symbols. When such a pattern is matched against a compound data structure then it binds x and y to the structures components, When matched against the singleton list `Cons 3 Nil` it binds x to `Cons 3` and y to `Nil`.

For example, a generic addition can be defined by

```
let rec (plus0 : a * b -> a) =
| ((x:Int),(y:Int)) -> x plusint y
| ((x:Float),(y:Float)) -> x plusfloat y
| (x1 x2,y1 y2) -> plus0 (x1,y1) (plus0(x2,y2))
| (x,y) -> if x eqcons y then x else Exception "plus0"
;;
plus0: a * b -> a
```

As well as special cases for integers and floats, it has a case for compounds and one for constructors. Corresponding components are added, and equal constructors are merged. For example,

```
~~ plus0 (z,z);;
it: Complex
it = Cart 6. 2.
```

`adds` adds `1.0` to itself, and adds `Cart 3.0` to itself (in two more steps). In object-oriented terminology, the inheritance is automatic, without the need for re-coding.

There are two main points worth remarking upon. First, in a pattern of the form $x\ y$ the components x and y are analogous to the `car` and `cdr` of Lisp. However, prior accounts of pattern-matching have generally excluded such patterns, or only admitted them after giving up on confluence of reduction. One reason for treating such patterns with suspicion is that it is not immediately clear how to type them, since the type of the second component y cannot be inferred from that of the compound $x\ y$. In brief, the “hidden” type of y cannot be allowed to “escape”, in the sense that it behaves as if it were existentially quantified. The other obvious question is how to type collections of cases that have different types.

When a special case of type $P \rightarrow S$ is to be added to a default of type $Q \rightarrow T$ there is no risk unless the types P and Q are the same, or become so upon substitution. Then S and T must be the same too. That is, any solution of $P = Q$ must also solve $S = T$. Then $P \rightarrow S$ is *similar* to $Q \rightarrow T$.

5 Adapting existing functions

However, the situation is not so simple for multiplication. To be sure, `times0` can be defined just like `plus0` in

```

~~ let rec (times0 : a * b -> a) =
| ((x:Int),(y:Int)) -> x timesint y
| ((x:Float),(y:Float)) -> x timesfloat y
| (x1 x2,y1 y2) -> times0 (x1,y1) (times0(x2,y2))
| (x,y) -> if x eqcons y then x else Exception "times0"
;;
times0: a * b -> a

```

but then `times0(z,z)` yields `Cart 9.0 1.0` instead of `Cart 8.0 6.0`.

A first attempt at a solution might be

```

let (times1 : a * b -> a) =
| (Cart x1 y1, Cart x2 y2) ->
    Cart ((x1 timesfloat x2) minusfloat (y1 timesfloat y2))
          ((x1 timesfloat y2) plusfloat (x2 timesfloat y1))
| p -> times0 p
;;

```

Although this multiplies complex numbers correctly, it does not handle them within larger structures, such as singleton lists, where `times1` reverts to being `times0`. For example

```

~~ times1(z,z);;
it: Complex
it = Cart 8. 6.
~~ times1([z],[z]);;
it: List Complex
it = [Cart 9. 1.]

```

A correct solution is to define a new multiplication function by

```

~~ let rec (times2 : a * b -> a) =
| (Cart x1 y1, Cart x2 y2) ->
    Cart ((x1 timesfloat x2) minusfloat (y1 timesfloat y2))
          ((x1 timesfloat y2) plusfloat (x2 timesfloat y1))
| ((x:Int),(y:Int)) -> x timesint y
| ((x:Float),(y:Float)) -> x timesfloat y
| (x1 x2,y1 y2) -> times2 (x1,y1) (times2(x2,y2))
| (x,y) -> if x eqcons y then x else Exception "times2"
;;
times2: a * b -> a
~~ times2([z],[z]);;
it: List Complex
it = [Cart 8. 6.]

```

Of course, it is inconvenient to repeat all of the old cases just to add one more. This can be avoided by separating the case analysis from the recursion, as follows.

```

~~ let (times3_body : (all a. a*b -> a) -> c * d -> c) times =
| ((x:Int),(y:Int)) -> x timesint y
| ((x:Float),(y:Float)) -> x timesfloat y
| (x1 x2,y1 y2) -> times (x1,y1) (times(x2,y2))
| (x,y) -> if x eqcons y then x else Exception "times"
;;
times3_body: (all a.a * b -> a) -> c * d -> c

```

The case analysis expressed in the pattern-matching function above is identical to that of `times0` except that recursive references to `times0` have been replaced by the formal parameter `times`. Then the recursion is added using

```
let rec (times3: a * b -> a) = times3_body times3;;
```

The resulting function `times3` is behaviourally equivalent to `times0` but its internal structure is now accessible.

A case for complex numbers can be added as follows

```

~~ let (times4_body : (all a. a*b -> a) -> c * d -> c) times =
| (Cart x1 y1, Cart x2 y2) ->
    Cart ((x1 timesfloat x2) minusfloat (y1 timesfloat y2))
        ((x1 timesfloat y2) plusfloat (x2 timesfloat y1))
| x -> times4_body times x;;
times4_body: (all a.a * b -> a) -> c * d -> c
~~ let rec (times4: a * b -> a) = times4_body times4;;
times4: a * b -> a

```

Now there is no need to repeat the case analysis.

Summarising, the re-use of addition is quite convenient, and there is no need to repeat all of the cases for `times4_body`, but re-coding of functions for squaring, etc. is still painful.

6 Updating existing functions

The need to re-code multiplication and its dependents can be avoided by updating the existing multiplication rather than defining a new one. More precisely, it is enough to make `times3_body` into an assignable reference, so that it can be updated to `times4_body`. Of course, this requires references to polymorphic functions. This is problematic when type quantification is to be suppressed, but here it can be made explicit, so that such assignments must update with equally polymorphic values.

Rather than work with `times_body` directly, it is convenient to adapt the strategy used for recursive functions, by replacing the keyword `rec` by `ext`, to indicate that a recursive function can be *extended* with new cases. For example, an extensible generic multiplication is given by

```

~~ let ext (times : a * b -> a) =
| ((x:Int),(y:Int)) -> x timesint y
| ((x:Float),(y:Float)) -> x timesfloat y
| (x1 x2,y1 y2) -> times (x1,y1) (times(x2,y2))
| (x,y) -> if x eqcons y then x else Exception "times"
;;
times: a * b -> a

```

times behaves just like times0 except that new cases can be added to it. It can be used to define (*) as before. Similar remarks apply to plus and +. Now cases can be added at will. For example

```

~~ add_case times(Cart x1 y1,Cart x2 y2) =
    Cart ((x1 * x2) - (y1 * y2)) ((x1 * y2) + (x2 * y1));;
times: Complex * Complex -> Complex
~~ [z] * [z];;
it: List Complex
it = [Cart 8. 6.]

```

Summarising, in this example, when the new type of complex numbers is introduced, the generic addition works without any changes, while multiplications, and all of its dependents, can be modified by adding a single new case.

Of course, this ability to change the nature of multiplication, even of integers and floats, brings dangers as well as opportunities. If desired, it can be bundled with data type declarations, to ensure that new cases cannot change the behaviour of existing terms

7 Updating existing functions within classes

Now let us consider how the functional style can impact upon class declarations. Since sub-classing will soon become the focus, It is convenient now to shift to the canonical example of points and coloured points.

Consider a class of 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
add_case toStringOpen (z:Point) =
    ( "{x=" ^ (toString (z.get_x_coord())) ^
      ",y=" ^ (toString (z.get_y_coord())) ^
      "}" ,

```

```

        False)
    }
class Point {
x_coord: Point[a] -> ref Float
y_coord: Point[a] -> ref Float
get_x_coord: Point[a] -> Unit -> Float
set_x_coord: Point[a] -> Float -> Unit
get_y_coord: Point[a] -> Unit -> Float
set_y_coord: Point[a] -> Float -> Unit
is_good: Point[a] -> Bool
toStringOpen: Point[a] -> String * Bool
}

```

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 `toStringOpen` which is used in pretty printing. For example,

```

~~ 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}

```

The modifications to `toStringOpen` propagate to the `print` function, so that points are displayed like records.

Now consider some colours.

```

datatype Colour = Colour of Int
with add_case toStringOpen (Colour x) =
((| 1 -> "red"
 | 2 -> "green"
 | 3 -> "blue"
 | y -> "Colour "^(toString y))
x, False)
;;
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 }
with
add_case toStringOpen (z:ColourPoint) =
  ( "{x=" ^ (toString (z.get_x_coord())) ^
    ",y=" ^ (toString (z.get_y_coord())) ^
    ",colour=" ^ (toString (z.get_colour())) ^
    "}" ,
    False)
}

```

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 = False

```

Note that `cp1.is_good` is false. Now suppose that, red points are to be good, too.

```

~~ add_method is_good (z:ColourPoint) = z.get_colour() == red;;
is_good: ColourPoint[a] -> Bool
cp1.is_good;;

```

The `add_method` declaration is another way of describing the addition of a special case that is normally done inside a class declaration. Of course, this can be easily abused, just like any imperative feature. For example,

```

~~ add_case toStringOpen (z:Point) =
  ("Someone has overwritten my printer!",False) ;;
toStringOpen: Point[a] -> String * Bool
~~ new Point;;
it: Point[]
it = Someone has overwritten my printer!

```

bondi tries to compromise between allowing programmers freedom and avoiding disaster by restricting `add_case` when it appears within a type or class declaration (its normal abode) but not restricting it in general. For example,

```

class BadPoint extends Point {
colour: Colour ;

```

```

with
add\_case toStringOpen (z:Point) =
  ("Someone has overwritten my printer!",False)
}
class BadPoint {
colour: BadPoint[a] -> ref Colour
error: add\_case must use a new constructor

```

That is, if additional cases of functions and methods are restricted to type and class declarations, then old behaviours can never be changed, but only new behaviours for the new types. This is called *behavioural continuity*.

This completes the examples devoted to the expression problem. Further examples will consider polymorphism by sub-typing and by type variables, and their interaction.

8 Coloured Circles

Sub-typing has several uses. One arises from conditionals: the result may be a mere point or a coloured point, so a common super-type is required. Another is the ability to create collections containing a mixture of different sorts of points. Some care is required when using algebraic data types to create collections. For example

```

~~ [pt,cp1];;
type error: ColourPoint_cty Unit and Unit don't unify

```

The difficulty is that the type of `pt` is used to determine the type of the list entries. The solution is to coerce its type explicitly, to

```

~~ [(pt:Point),cp1];;
it: List Point
it = [{x=5.5,y=2.2},{x=-2.2,y=3.3,colour=red}]

```

Note that it is not necessary to give a type for `cp1` as the nature of the list has already been established. The asymmetry of the treatment of the entries is not fundamental. One could equally write

```

(Cons : Point -> List Point -> List Point) pt [cp1];;
it: List Point
it = [{x=5.5,y=2.2},{x=-2.2,y=3.3,colour=red}]

```

Similarly, one may write

```

~~ if True then (pt: Point) else cp1;;
it: Point
it = {x=5.5,y=2.2}

```

Perhaps a more clever type inference algorithm could remove the need for this assistance with the typing.

Now let us consider how sub-typing may appear in specialised methods. Here is a class of circles.

```
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

$$\text{Circle}[a] \rightarrow \text{Unit} \rightarrow \text{Point}$$

while the type for its specialisation is

$$\text{ColourCircle } [b] \rightarrow \text{Unit} \rightarrow \text{ColourPoint}$$

Note that the type of the specialisation is *not* similar to that of the default, since `ColourPoint` is a sub-type of `Point` so they can never be equal. Rather, the latter method type is a *specialisation* of the former since any substitution that relates `ColourCircle [b]` and `Circle[a]` solves the inequality

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

9 Linked Lists

So far, the examples have used type variables solely to manage the class relationships, but they can also be used as parameters in the usual fashion, e.g. to represent the type of data stored within a structure.

For example,

```
~~class Node <a> {
value : a ;
next : Node <a>;
getValue = { |() -> !this.value }
setValue = { fun v -> this.value = v }
getNext = { |() -> !this.next }
setNext = { fun (n: Node <a>) -> this.next = n }
insert = { fun (n: Node<a>) ->
    n.setNext (this.getNext()); this.setNext n }
with
add_case toStringOpen (x: Node<a>) =
    let rec (aux: c -> String) =
        | Node _ (v,Ref (Exception _)) -> toString !v
        | Node _ (v,n) -> (toString !v) ^ "," ^ (aux !n)
        | _ -> ""
    in
    ("[" ^ (aux x) ^"]" , False)
}
class Node {
value: Node<a>[b] -> ref a
next: Node<a>[b] -> ref Node<a>
getValue: Node<a>[b] -> Unit -> a
setValue: Node<a>[b] -> a -> Unit
getNext: Node<a>[b] -> Unit -> Node<a>
setNext: Node<a>[b] -> Node<a> -> Unit
insert: Node<a>[b] -> Node<a> -> Unit
toStringOpen: Node<a>[b] -> String * Bool
}
```

declares a class of nodes of linked lists. Each node has a value and a next node. The type `a` of the node value is a type variable, enclosed in angle brackets for emphasis. Here is an example.

```
~~ let x0 = new Node <Int> ;;
x0: Node<Int>[]
x0 = [_void,]
~~ let x1 = new Node<Int> ;;
x1: Node<Int>[]
x1 = [_void,]
~~ let x2 = new Node<Int> ;;
```

```

x2: Node<Int>[]
x2 = [_void,]
~~ x0.setValue 0;
x1.setValue 1;
x2.setValue 2;
x0.setNext x2;
x0;;
it: Node<Int>[]
it = [0,2,]
~~ x0.insert x1;
x0;;
it: Node<Int>[]
it = [0,1,2,]
~~ let r4 = new Node <Float>;;
r4: Node<Float>[]
r4 = [_void,]
~~ r4.setValue 4.4;
r4;;
it: Node<Float>[]
it = [4.4,]

```

Now create a sub-class of doubly-linked lists.

```

class DNode <a> extends Node {
previous : ref (DNode <a>);
getPrev = { |()-> !(this.previous) }
setPrev = { fun (p: DNode <a>) -> this.previous = Ref p }
setNext = { | (n:DNode<a>[c]) ->
            super.setNext n;
            n.setPrev this
            | (n: Node<a>) -> super.setNext n
}
}
}
class DNode {
previous: DNode<a>[b] -> ref ref DNode<a>
getPrev: DNode<a>[b] -> Unit -> DNode<a>
setPrev: DNode<a>[b] -> DNode<a> -> Unit
setNext: DNode<a>[b] -> Node<a> -> Unit
}
}

```

Note that the field `previous` has type `ref DNode<a>` and so introduces another level of indirection. Without it, any path polymorphic function would go into an infinite loop, following `previous` and `next` links ad infinitum. The problem is resolved by modifying the traversal to ignore links to references, so that these “backward” links are not followed.

Another point is that when a double node invokes the binary method `setnext` it must be able to act on any node, not just a double node. This requirement is

not onerous as it suffices to write a pattern-matching function with two cases: a special case for double nodes and a default for arbitrary nodes.

Here is an example.

```
~~ let d0 = new DNode <Int>  ;;
d0: DNode<Int>[]
d0 = [_void,]
~~ let d1 = new DNode<Int>  ;;
d1: DNode<Int>[]
d1 = [_void,]
~~ let d2 = new DNode<Int>  ;;
d2: DNode<Int>[]
d2 = [_void,]
~~ d0.setValue 0;
d1.setValue 1;
d2.setValue 2;
d0.setNext d2;
d0;;
it: DNode<Int>[]
it = [0,2,]
~~ d2.getPrev();;
it: DNode<Int>
it = [0,2,]
~~ d0.insert d1;
d0;;
it: DNode<Int>[]
it = [0,1,2,]
~~ d2.getPrev();;
it: DNode<Int>
it = [1,2,]
```

Note the smooth interplay of sub-typing and type parameters.

10 Counter-examples in bondi

The class syntax in **bondi** makes it hard to generate type errors in methods, since the type variables are not usually explicit in the declaration. For example, the declaration of the `Point` class does not actually mention any type symbols. The typing is accessible, however, as the following example shows.

```
~~ class ShowType {
  cell :Int ;
  identity = {this}
}
class ShowType {
  cell: ShowType[a] -> ref Int
```

```
identity: ShowType[a] -> ShowType[a]
}
```

This can be modified to produce some examples of type errors.

```
~~ class Fragile {
cell :Int ;
double_up = { (this,this)}
}
class Fragile {
cell: Fragile[a] -> ref Int
type error: Fragile[ty_900] -> Fragile[ty_900] * Fragile[ty_900]
  is not a method type
```

Of course, it is perfectly natural to define a *function* that duplicates its argument

```
~~ let ext double_up x = (x,x);;
double_up: a -> a * a
```

Conversely, a function cannot use sub-typing its extensions

```
~~ let ext lose_info x = (x:Top) ;;
lose_info: a -> Top
~~ add_method lose_info Nil = Nil;;
term error: lose_info is not a method
~~ add_case lose_info Nil = Nil;;
lose_info: List a -> Top
```

The system will not allow this function to be treated as a method, but when a case is added then the special type `List a -> List b` is a sub-type of an instance `List a -> Top` of the default type.

```
~~ class Fragile2 {
cell :Int ;
good_argument = { fun (x:Fragile2) -> this }
bad_argument = { fun (x:Fragile2[a]) -> (this:Fragile2[a]) }
}
class Fragile2 {
cell: Fragile2[a] -> ref Int
good_argument: Fragile2[a] -> Fragile2 -> Fragile2[a]
type error:
  Fragile2[ty_913] -> Fragile2[ty_913] -> Fragile2[ty_913]
  is not a method type
```

In this example, the method `good_argument` is well-typed but `bad_argument` is too specialised, since `a` is not covariant in `Fragile2[a] -> Fragile2[a]`. These two methods illustrate how **bondi** copes with binary methods.

11 Conclusions

bondi provides a successful account of object-orientation through a sophisticated use of pattern-matching, as introduced in the pattern calculus. Objects are just data structures, and class declarations are glorified type declarations, augmented by attribute declarations. The ability to add new cases to existing functions, as well as new functions for existing data structures means that **bondi** solves the expression problem, and the difference between data types and classes is essentially one of syntactic convenience.

A pattern for an object of a super-class will match any object in a sub-class, so that inheritance is automatic. Method specialisation is achieved by adding new cases to the existing method. Such additions must be made to the body of a recursive function, rather than to the function itself, but that is the only concern. Hence, there is no need for any special concept of object, or “self”.

Typing presents some challenges for the pattern calculus, a special cases may have special types. These difficulties are magnified in the presence of sub-typing, but the same general principles apply. The type of “self” is handled by conventional type variables. The deeper challenges arise from the need to solve type inequalities, but this can be handled by making the sub-typing rules as simple as possible. Notably, the super-type of a type form a finite linear order: function argument types are *invariant*, as are the first components of product types. In turn, this determines the treatment of binary methods, since the type of the second argument must be invariant.

State requires no special treatment, except that one must allow references to polymorphic functions.

References

1. M. Abadi and L Cardelli. *A Theory of Objects*. Monographs in Computer Science. Springer-Verlag, New York, 1997.
2. Kim Bruce. *Foundations of Object-Oriented Languages: Types and Semantics*. The MIT Press, 2002.
3. G. Castagna, G. Ghelli, and G. Longo. A calculus for overloaded functions with subtyping. *Information and Computation*, 117(1):115–135, 1995.
4. Barry Jay. *Pattern Calculus: Computing with Functions and Data Structures*. Springer, to appear 2009.
5. Barry Jay and Simon Peyton Jones. Scrap your type applications. In Philippe Audebaud and Christine Paulin-Mohring, editors, *Mathematics of Program Construction, 9th International Conference, MPC 2008, Marseille, France, July 2008*, volume 5133 of *Lecture Notes in Computer Science*, pages 2–27. Springer, 2008.
6. 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.

7. Barry Jay and Delia Kesner. First-class patterns. *Journal of Functional Programming*, page 37 pages, 2009. to appear.
8. 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>.
9. C.B. Jay. The pattern calculus. *ACM Transactions on Programming Languages and Systems (TOPLAS)*, 26(6):911–937, November 2004.
10. 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>.
11. C.B. Jay. Typing first-class patterns. In D. Kesner, M-O. Stehr, and F. van Raamsdonk, editors, *Higher-Order Rewriting, electronic proceedings*, 2006. <http://hor.pps.jussieu.fr/06/proc/jay1.pdf>.
12. Didier Rmy and Jrme Vouillon. Objective ML: An effective object-oriented extension to ML. *Theory and Practice of Object Systems*, 4(1):27–50, 1998.
13. Philip Wadler. The expression problem, November 1998.
14. 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.

12 Types in bondi

Like λ -calculus, pattern calculus has many variants. The book [4] includes untyped, typed, sub-typed and implicitly typed calculi with different emphases. Of these, the closest to **bondi** is the sub-typed extension calculus, but then **bondi** also supports imperative features.

For this paper, a conventional understanding of pattern-matching will suffice for most purposes, since patterns for objects are always headed by a term constructor, in the familiar manner. The main exceptions are the generic arithmetic operations, which use patterns of the form $x\ y$ to support path polymorphism. When applied to a data structure u then x corresponds to `car u` and y to `cdr u` in the style of Lisp. **bondi** also supports pattern polymorphism, but this has not been used.

Similarly, the **bondi** types are fairly conventional. Sufficient for this paper are the types given by

$$T ::= a \mid \text{Constant} \mid T T \mid T \rightarrow T \mid \text{all } a. T \mid \text{ref } T$$

consisting of variables, constants, applications, function types, quantified types and reference types, respectively. These are similar to those of functional languages such as ML or Haskell. Note, however, that `List` and other structures are given by type constants, so that the type `List Int` of lists of integers is an application. By contrast, `ref` is a type-forming operator, not a type, used to support references. Also, quantified types may appear anywhere within a term, so that type inference will sometimes require some explicit typing in the program. In **bondi**, type variables are given by words commencing with a lower-case letter, such as `a` or `b` and constants commence with an upper-case letter, such as `List` or `Point_ty`.

When a class `C` is declared then a typical object `o` in this class has type `C[rest]` where `rest` is some type for the fields of `o` that are not a consequence of being in class `C`. When `rest` is the *top type* `Top` then the object's type may be written as `C`. When `rest` is the *unit type* `Unit` then the object's type may be written `C[]`. The other common situation is when `rest` is a type variable `a` yielding `C[a]`. The actual meaning of `C[a]` depends on the position of `C` in the class hierarchy. If `C` is a root class then `C[a]` is merely `C_ty a` where `C_ty` is a type constants introduced with the declaration of `C`. However, if `C` is an immediate sub-class of `D` then `C[a]` is `D[C_ty a]`. In this manner, the type of an object in class `C` is also the type of an object in class `D` so that inheritance is automatic. The notation `C[a]` avoids explicit mention of all the super-classes of `C`, as is natural.

bondi typically elides leading quantifiers. For example, the type of `times` is `a * b -> a` rather than `all a.all b. a * b -> a`. However, quantified types are first-class and may type function arguments. For example, the type of the generic iterator is

$$\text{iter} : (\text{all } a.a \rightarrow \text{Unit}) \rightarrow b \rightarrow \text{Unit}$$

$$\begin{array}{ccc}
\frac{}{T < \text{Top}} & \frac{S < T}{F S < F T} & \frac{}{T < T} \\
\\
\frac{S < T}{P \rightarrow S < P \rightarrow T} & & \frac{S < T}{\text{all } a. S < \text{all } a. T}
\end{array}$$

Fig. 1. Sub-typing

since its argument must be able to act on any component of a data structure.

Similarly, it is important to allow references of quantified type, since these are used when defining extensible functions. Of course, some well-known difficulties arise when combining type quantification and assignment, but these only arise when suppressing quantifiers. Here, they are first class, and are only suppressed when convenient.

Now let us briefly consider the types of functions in which special cases have special types. When adding a special case of type $P \rightarrow S$ to a function of type $Q \rightarrow T$ type safety requires that any solution of $P = Q$ is also a solution of $S = T$. For example, a default of type $X * Y \rightarrow X$ may be specialised by a function of type $\text{Int} * \text{Int} \rightarrow \text{Int}$. In this manner, type unification becomes integral to the type derivation rules as well as those for type inference.

The main concern for this paper lies in the treatment of sub-typing, which will be discussed in a little more detail. The *sub-typing rules* given in Figure 1 are rather unusual. The main rules are in the first line of the figure. The type constant `Top` is the top type, a super-type of every other such. Type applications are covariant on the right and invariant on the left. For example,

`ComplexNumber[a] = Number (ComplexNumber a) < Number Top`

but `Number a` is *not* a sub-type of `Top a`. In this way, the sub-type hierarchy for the types of classes exactly reflects the class hierarchy. Similarly,

`a * b = BinProd a b < BinProd a Top = a * Top`

but `a*b` is *not* a sub-type of `Top*b`. This may seem a little strange but removes problems associated with binary methods, since only one argument can be in a covariant position.

The rules for function types and quantified types are not strictly necessary but they are harmless, and quite convenient in practice. By contrast, a reference type `ref T` is invariant in T . This is because assignment takes an argument of type T , and function argument types must be invariant.

No harm comes from the invariance of function argument types. Suppose that $V < U$ and $f : U \rightarrow S$ is a function. If $v : V$ is a term then subsumption implies that $v : U$ too so that $f v$ is well-typed without any need to change the type of f . The only possible reason to want $f : V \rightarrow S$ is so that it can be an argument of some other, higher-order function. In this case, simply η -expand f to $\lambda(x : V).fx : V \rightarrow S$. Such explicit coercions are not normal for objects, but

then f is a function and objects are data structures, so this treatment of f is quite acceptable.

These structural sub-typing rules are very simple. In particular, the super-types of a type form a finite linear order, which greatly simplifies the challenge of solving type inequalities. For example, consider the inequality

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

In most approaches to sub-typing, function argument types are contravariant so that this is equivalent to $S < X < T$ which has no best solution. Here, by contrast, this amounts to $S = X < T$ which is easily solved.

Similarly, if product types are covariant in both arguments then the inequality

$$S * T < X * X$$

requires a least upper bound for S and T . Suppose that S and T are variables Y and Z , respectively. Then one solution is to identify X, Y and Z but a different solution is to map X to Top . Neither is to be preferred. Here, linearity implies that either $S < T$ or $T < S$ so it is enough to first solve the problem of relating S and T and then map X to the larger of them.

Now let us consider the types of methods and the typing of method invocation. Roughly speaking, when a method of type $P \rightarrow S$ is invoked by an object of type U then the inequality $U < P$ must be solved. If P is a closed type (having no free type variables) then it is a simple matter to check if U is a sub-type of P but in general, it may be necessary to solve the inequality and apply any resulting substitution ρ to S . Now it may happen that the object reduces to something of type $V < U$ in which case the invocation will have type given by solving $V < P$ to get some ρ' . In this case, it is essential for type safety that $\rho'S < \rho S$ holds.

A moment's thought shows that this is not true in general. For example, if $P \rightarrow S$ is $\mathbf{a} \rightarrow \mathbf{a} * \mathbf{a}$ then $V * V$ is *not* a sub-type of $U \delta U$ since products are invariant on the left. To control this, define the *covariant type symbols* of a type by

$$\begin{aligned} \text{CTV}(\mathbf{a}) &= \{\mathbf{a}\} \\ \text{CTV}(F U) &= \text{CTV}(U) \setminus \text{FTV}(F) \\ \text{CTV}(U \rightarrow S) &= \text{CTV}(S) \setminus \text{FTV}(U) \\ \text{CTV}(\mathbf{all} \ \mathbf{a}. \ S) &= \text{CTV}(S) \setminus \{\mathbf{a}\} \\ \text{CTV}(U) &= \{\} \text{ otherwise.} \end{aligned}$$

The *invariant type symbols* of a type are its free type symbols (defined in the usual manner) that are not covariant.

A *simple method type*

$$P \Rightarrow S$$

is defined as follows. If $\text{CTV}(P) = \{\}$ then it is $P \rightarrow S$. If $\text{CTV}(P) = \{\mathbf{a}\}$ and \mathbf{a} is into invariant in S then it is $\mathbf{all} \ \mathbf{a}. \ P \rightarrow S$. For example $\text{Point}[\mathbf{a}] \Rightarrow \text{Point}[\mathbf{a}]$ is $\mathbf{all} \ \mathbf{a}. \ \text{Point}[\mathbf{a}] \rightarrow \text{Point}[\mathbf{a}]$. but $\text{Point}[\mathbf{a}] \Rightarrow \mathbf{a} * \text{Point}[\mathbf{a}]$ is undefined.

Now the *type invocation* $U.(P \Rightarrow S)$ is defined by finding a substitution on the covariant type symbols of P that solves $U < P$ and applying this to S .

When a method m of type $Q \Rightarrow T$ acquires a special case of type $P \Rightarrow S$ this is well-typed if any type substitution that relates P and Q (solves $P < Q$ or $Q < P$) also solves $S < T$. Under these circumstances, $P \Rightarrow S$ is a *specialisation* of $Q \Rightarrow T$ written

$$P \Rightarrow S \ll Q \Rightarrow T.$$

After this, the method has type

$$P \Rightarrow S \ \& \ Q \Rightarrow T$$

where $\&$ is a type constant for choice (written infix), reminiscent of the $\lambda\&$ -calculus [3].

Now $P \Rightarrow S \ \& \ Q \Rightarrow T$ is *not* a sub-type of $Q \Rightarrow T$ so that the type of m has changed dynamically. This would be a problem if m were a program, so methods are *not* first-class, even though functions are. Within a program, a method must be invoked by some object, to fix the typing. Under evaluation, the invocation will reduce to a function application.