Patterns as Objects in Grace

Michael Homer
Victoria University of Wellington
mwh@ecs.vuw.ac.nz

James Noble
Victoria University of Wellington
kjx@ecs.vuw.ac.nz

Kim B. Bruce
Pomona College, CA
kim@cs.pomona.edu

Andrew P. Black
Portland State University
black@cs.pdx.edu

David J. Pearce
Victoria University of Wellington
djp@ecs.vuw.ac.nz

Abstract

Object orientation and pattern matching are often seen as conflicting approaches to program design. Object oriented programs place type-dependent behaviour inside objects and invoke it via dynamic dispatch, while pattern matching programs place type-dependent behaviour outside data structures and invoke it via multiway conditionals (case statements). Grace is a new, dynamic, object-oriented language designed to support teaching: to this end, Grace needs to support both styles. In this paper we explain how this conflict can be resolved *gracefully*: by modelling patterns and cases as partial functions, reifying those functions as first-class objects, and then building up complex patterns from simpler ones using pattern combinators. We describe our design for pattern matching in Grace, and its implementation as an object-oriented framework.

1. Introduction

Grace [1] is a dynamic, imperative, object-oriented language that we are designing to support the teaching of programming, typically in the traditional CS1 and CS2 contexts. Grace is a dynamic language with block structure, single dispatch, curly-bracket syntax and optional, gradual type declarations. Grace programs look like an amalgam of Java, Scala, Ruby, Python and Go, but Grace's underlying semantic model is purely object-oriented, and closer to Self, Newspeak, or gBETA.

One important aim of Grace's design is to give instructors and text-book authors the freedom to choose their own teaching sequence, by reducing the dependencies between features in the language and its libraries. (We also hope that reducing feature dependencies will make Grace easier to learn in its own right [20]) Thus, in Grace it is possible to ignore static types completely, introduce them in the middle of the course, mix dynamically and statically typed code, or to begin with strict static type checking on day one. It is also possible to start with objects, or start with classes, or ignore both and simply write structured, procedural programs using only top-level method and variable declarations, vectors, and records represented by degenerate objects.

All these options stay roughly within Grace's pure object-oriented paradigm. We would also like Grace to support programs organized in a "functional programming style" — as collections of functions or procedures that make decisions based on the types or values of their arguments.

We don't expect that Grace will ever do as good a job supporting this "functional programming style" as ML [15] or F# [22], but we do think that it is possible to do a good enough job to give instructors and students the chance to compare and contrast these different ways of organizing programs, without having to learn an entirely new language syntax, semantics, environment, and library. The strawman ACM CS2013 curriculum, for example, regards both object-oriented and function-oriented programming as core topics, and requires programmes to cover both dynamic dispatch and pattern matching [24]. With support for pattern matching they may do so within the same language.

Support for the functional style in Grace is enhanced if the language provides some form of pattern matching to scrutinize and de-structure the arguments of a function, and some form of conditional statement to select behaviour depending on the results of the pattern-match. While this goes against the object-oriented style in some ways, we follow Odersky [16] in believing there are circumstances in which pattern matching, or something similar, is useful in object-oriented languages. Even Smalltalk programmers sometimes find it necessary to ask an object if it understands a particular message. The obvious approach is to extend the core language with one or more special-purpose constructs to support pattern-matching, as in F#'s active patterns, or Scala or Machete's "whole cloth" approach [4, 9, 22]. Although some language support seems unavoidable, we would like to minimize it: we want to create a small, simple, object-oriented language, not a large hybrid multi-paradigm language. Our problem, then is how to support pattern matching in a pure object-oriented language, with minimal extensions.

This paper explains how we solved this problem *gracefully*. Our approach is to model patterns and cases as partial functions, reify those functions as first-class objects, and then build-up more complex patterns from simpler ones using pattern combinators. This results in flexible pattern-matching and case statements that incorporate a programmer-extensible range of pattern matches, including matching against constants, matching against an object's type (that is, its method interface), and also binding a variable of the new type, matching against the value of a variable or expression, and "destructuring" an object to extract its components, which requires the cooperation of the object in determining what those components should be. All of this is presented in a conventional pattern-matching

syntax and implemented using three localized language extensions: treating blocks (lambda expressions) as *partial* functions, binding variables in nested patterns, and match-case expressions with an indeterminate number of case branches.

This paper makes the following contributions.

- A language design that includes an object-oriented form of pattern matching, but which uses a conventional pattern-matching syntax; this is described in Section 3.
- A model for that design using objects and methods, specifically partial function objects, pattern objects, and pattern combinator methods, which is described in Section 4.
- The integration of pattern-matching with Grace's type system, so that the bodies of case blocks can be statically type-checked using information obtained in the pattern match, described in Section 5.

We start in Section 2 by setting the context with a high-level overview of Grace. In Section 7 we will discuss the particular design decisions we made, and roads not taken, set in the context of pattern matching support in other languages. We also present a case study of the use of pattern matching in Grace, which demonstrates the completeness of our design and implementation (Section 6).

2. An Overview of Grace

Grace can be regarded as either a class-based or an object-based language, with single inheritance and gradual typing. A Grace class is an object with a single factory method that returns an object:

```
class aCat.named(n) {
        def name = n
        method meow { print "Meow" }
}
var theFirstCat := aCat.named "Timothy"
```

Here the class is called aCat and the factory method is called named(). After executing this code sequence, theFirstCat is bound to an object with two attributes: a constant field (name), and a method meow. The expression c.name answers the string object "Timothy", and c.meow has the effect of printing *Meow*.

An object can also be constructed using an object literal — a particular form of Grace expression that creates a new object when it is executed. In addition to fields and methods, an object literal can also contain code, which is executed when the object literal is evaluated. For example:

```
var theSecondCat := object {
        def name = "Timothy"
        method meow { print "Meow" }
        print "Timothy now exists!"
}
```

This code has the effect of printing "Timothy now exists!", and binding the variable theSecondCat to a newly-created object, which happens to be operationally equivalent to theFirstCat.

It is important to note that, in Grace, classes are completely separate from types: the class aCat is not a type and does not implicitly declare a type. Grace programs need not have any type declarations whatsoever. A type in Grace is *structural*: it specifies an interface or protocol that an object can support, and any object that supports the required methods will belong to the type. If the programmer wishes to specify types, she may easily do so:

```
type Cat {
        name -> String
        meow -> None
}
```

```
class aCat.named(n : String) {
        def name = n
        method meow { print "Meow" }
}
var theFirstCat : Cat := aCat.named "Timothy"
```

When types are not mentioned, code is dynamically-typed.

Mutable and immutable bindings are distinguished by keyword: **var** defines a name with a variable binding, which can be changed using the := operator, whereas **def** defines a constant binding, initialized using =, as shown here.

```
var currentWord := "hello"
def world = "world"
...
currentWord := "new"
```

The keywords **var** and **def** are used to declare both local bindings and fields inside objects.

An object's methods are immutable, in the sense that once an object is created, the code of its methods cannot be changed. A field that is declared with **def** is constant; the binding between the field name and the object cannot be changed, although the object, if mutable, may change its state. Each constant field declaration creates an accessor method on the object. For example, the object club defined by

```
def club = object {
        def members = MutableSet.empty
}
```

has a *method* called members that returns the current set of members. The value of this set may change over time, for example, after executing club.members.add(anApplicant).

Grace supports visibility annotations that allow the programmer to restrict access to fields and methods from outside an object by marking them as *public* or *confidential*. For simplicity, we do not discuss this further here, and omit visibility annotations in all the sample code in this paper.

In Grace we say that a method is invoked using a "method request". We introduce this terminology to distinguish the operation — fundamental to object-orientation — of *requesting* an object to do something, where the choice of *what* to do is made by the object itself, from procedure or function call, where the choice of operation is made by the caller. This distinction is also conveyed by Smalltalk's "message send" terminology, but now that networks and distributed systems are ubiquitous, "sending a message" has become an ambiguous term. Methods are requested using the now-standard "dot" notation, in which the receiver **self** and the following dot may be omitted, or by using operator symbols like + and <.

Because **self**. can be omitted, field access is syntactically identical to a self method request. As in Eiffel [14] and Self [27], this is deliberate: it makes it easy for the implementor of an object to override a field with a custom method, which can be very useful when there is a need to make a change to the implementation of an object without affecting its interface.

Grace method names may consist of multiple parts ("mixfix" notation), as in Smalltalk [7]. Separate lists of arguments are interleaved between the parts of the name, allowing them to be clearly labelled with their purpose. Thus, we might define on Number objects

```
method between (l:Number) and (u:Number) {
    return (l < self) && (self < u)
}
```

The above method is named between()and(), and we could request it on the object 7 by writing

```
7.between(5) and(9)
```

Single arguments that are literals do not require parentheses, so alternatively we could write

```
7.between 5 and 9
```

String literals, written between double quotes, support interpolation, using a syntax similar to that of Ruby. Code inside braces within a literal is evaluated when the string is; the asString method is requested on the resulting object, and the answer is inserted into the string literal at that point.

```
print "1 + 2 = {1 + 2}" // Prints "1 + 2 = 3"
```

Grace includes blocks, also known as lambda expressions, like Smalltalk, Self, and Ruby. A block is written between curly braces, and contains a piece of code for deferred execution. For example,

```
{ score := score+10;  print "score = {score}" }
```

is a block that adds 10 to the (lexically scoped) variable score and prints score's new value. A block may have parameters, which are separated from the code by −>; a block returns the value of the last-evaluated expression in its body. Thus, for example, the successor function is written {x −> 1+x}.

Because Grace is object-oriented, blocks are represented as objects. Evaluating a block literal results in a block-closure object, so called because it may close over lexically-scoped variables, like scope in the example above. Block-closure objects can be executed by requesting the apply method with arguments that match the block's parameters, so

```
print( {x −> 1+x}.apply(5) )
```

creates a block representing the successor function, and immediately applies it to the argument 5, printing 6.

Control structures in Grace are methods. The built-in structures are defined in the basic library, but an instructor or library designer may replace or add to them. Control structures are designed to look familiar to users of other languages:

```
if (x > 5) then {
    print "Greater than five"
} else {
    print "Too small"
}
for (node.children) do { child −>
    process(child)
}
while {countdown > 0} do {
    countdown := countdown − 1
    print "{countdown}..."
}
```

Notice that the use of braces and parentheses is not arbitrary: parenthesized expressions will always be evaluated exactly once, whereas expressions in braces are blocks, and may thus be evaluated zero, one, or many times. Because a **return** statement inside a block terminates the *method* that lexically encloses the block, it is possible to program *quick exits* from a method by returning from the *then* block of an if()then() or the *do* block of a while()do().

While Grace uses braces to delimit blocks and other literals, it also enforces correct indentation. Braces and indentation may not be inconsistent with one another: the body of a method, for example, must be indented. Enforcing this in the language ensures that students will learn good practice, and avoids the common problem of not being able to find a mismatched brace because of the tendency of one's eye to believe the indentation rather than the braces.

3. Graceful Patterns

This section describes how patterns appear to the Grace programmer. The syntax is explicitly conventional, familiar to programmers of Scala, F#, Haskell, and other languages, and involved only minor extensions to Grace.

The programmer's interface to pattern matching is the match() case()... case() method, with the same form as other Grace control structures. It takes as its first argument the target of the match, and as succeeding arguments several case blocks. These case blocks are almost identical to the blocks used ubiquitously in Grace code, but are treated a little differently. Specifically, the parameter list of a normal block is replaced by a pattern literal; the body of the block contains the code to be executed when the pattern matches. This is best explained using an example.

```
match(expr)
  case { 0 −> "zero" }
  case { n:Number −> "Number less than {n+1}" }
  case { s:String −> "String \"{s}\"" }
  case { x −> error "Unexpected value {x}" }
```

This match expression first evaluates expr to obtain an object obj, and then executes the first case block whose pattern matches obj. The patterns are written before the −>, in the same position as block parameters. The syntax for patterns is a strict superset of that for the parameter list of a single-parameter block; this means that all single-parameter blocks are usable as case blocks.

In the example, the first pattern is the literal 0, which matches the number object 0. All numeric, string, and boolean literals can be used as patterns, and match themselves. The second pattern n:Number matches when obj conforms to the type Number, but also has the effect of binding n to obj within the body of the block. Note that this case block has the same syntax as an ordinary single-parameter block. The third pattern is similar, but matches only when obj conforms to type String. Recall that Grace is gradually typed, so static types like these may be used in code that is otherwise dynamically typed.

The fourth and final pattern introduces a new parameter named x; the pattern always matches and has the effect of binding x to obj. To write a pattern that always matches but does not bind a parameter, the wildcard identifier _ may be used:

```
  case { _ −> error "Unexpected value"}
```

As the second and third cases illustrate, Grace types are usable as patterns. Grace types are structural and simply assert that particular methods exist, with particular argument and result types. A type pattern may thus be used either to test for a particular method, or to distinguish between several objects with different types.

```
match (val)
  case { n:Number −> "Number with value {n}" }
  case { s:String −> "String: {s}" }
  case { p:Pair −> "Pair ({p.left}, {p.right})" }
```

Patterns can be combined using the pattern combinators & and |. a & b is a pattern that matches when patterns a and b both match:

```
type X = { x } // the type with method x
type Y = { y } // the type with method y
match (val)
    case { o:X & Y −> "Point ({o.x}, {o.y})" }
```

while a | b matches when either pattern a or pattern b matches:

```
match (val)
    case { _:Number | String | Boolean −>
        "A value of a built-in type"
    }
```

Patterns can also be used to extract data from the matched object for binding to names, or for further matching. We call this a "destructuring match"; it requires that the matched object cooperate by providing a method that exposes the necessary data.

```
match (astNode)
  case { nd:ASTString("") -> "Empty string" }
  case { nd:ASTNumber(n) -> "The number {n}" }
  case { nd:Operator("+", ASTNumber(0), y) ->
      "Just {y}" }
  case { nd:Operator("+",
         m:ASTMember(name : String,
                Identifier ("self"), y) ->
      "self.{name} + {y}" }
  case { nd:Operator("+", x, y) ->
      "Adding {x} and {y}" }
```

Destructuring matches can be nested arbitrarily deeply; each sub-pattern can use the full pattern syntax.

There is a potential ambiguity in this pattern syntax. If a bare identifier such as d is used as a pattern, in a context where the identifier d is already bound, does it indicate a variable match (which always succeeds and binds d to the object being matched), or does it indicate that the object already bound to d should be used as a pattern? We avoid this ambiguity by requiring that the latter case be written with parenthesis: (d). This feature may also be used to match against the result of a method request.

4. Patterns as Objects

As we mentioned in the introduction, our aim in adding pattern-matching to Grace was to provide the — in most ways quite conventional — facilities described in the previous section by leveraging the existing features of the language, making only minimal extensions. Here we describe the conceptual model that enabled us to achieve this aim, and the way that it is reified as Grace objects in the implementation.

4.1 Conceptual Model

In common with a number of other languages, Grace treats a case statement as combination of partial functions — functions that are defined on a restricted domain of inputs. A request to apply a partial function must supply an argument for the function. If the argument is in the domain of the function then the function executes using that argument and returns a result, but if the argument is outside the domain, the function fails and is not executed.

Grace represents both the partial function itself and its domain as objects. A sequence of method requests ascertains whether the argument is in the domain, applies the function, and returns the result. This representation of partial functions allows Grace pattern matching to be added with minimal disruption to the language.

As Grace is gradually typed, it was already possible to write a type restriction on a block parameter, and to attempt to apply the block to an argument of a different type. If such an invalid application occurred at runtime, the program would report a type error and terminate. The match method extends this functionality to also allow for a *non-fatal* indication of a type mismatch.

The main difference between match and apply is that apply either returns or raises a type error, while match returns a result indicating that the application failed because the argument did not belong to the domain of the function, as shown in Figure 1.

Because of Grace's support for gradual typing, type annotations on block parameters are normally checked at compile time if that is possible, and otherwise at runtime. When a block is applied using match, the type test is *always* dynamic: if the test fails, an appropriate FailedMatch object is returned, but no error is raised.

```
def blk = { x:Number -> x * 2 }
blk.apply 2 // -> 4
blk.match 2 // -> SuccessfulMatch(4)
blk.match "text" // -> FailedMatch("text")
blk.apply "text" // Error: wanted Number, got String
```

Figure 1. A partial function block applied in different ways. match does not raise any errors, but returns a FailedMatch result indicating that it was unable to apply the function.

To permit matching on values, such as numbers and strings, we generalize the annotation of a parameter's *type* to a *pattern*: all types are patterns, but patterns can also represent individual objects or values, sets or ranges of values, bit patterns, or any other criteria that can be defined in code. Patterns are permitted as annotation only on the single argument of a matching block, and on the components of destructuring matches. Patterns also allow the declaration of additional parameters for destructuring matching, which are bound to results returned from the pattern match. These parameters become new identifiers available in the body of the function, just like other parameters. These parameters may themselves have patterns applied, which will also be matched recursively as components of the overall pattern in a composite structure.

This representation of partial functions allows the the match() case()... case() method to take a series of single-parameter blocks and request each in turn to match, until one succeeds. The only thing special about match()case() is that Grace does not currently provide a way of defining multipart-variadic methods, that is, a series of methods named match()case(), match()case()case(), match()case()case()case(), etc. We could simply define variants of this method in the standard prelude with up to, say, ten case branches, but the current prototype implementation handles match()case()...case() specially, with no arbitrary limit.

The *only* other language-level impact that pattern matching and case statements places on Grace are those just described: the extension of type annotations into pattern annotations, and the ability of patterns to bind additional parameters. There are no macros, no rewriting, no additional control structures, and nothing "special" about destructor methods.

To arrive at this model of matching we needed to eliminate a great many conceivable models with superficial attraction, which turned out to be flawed after deeper consideration. Many approaches that can work in a purely dynamically-typed language do not have a viable static typing, for example, while some that work with pure static typing cannot work for dynamically-typed code, but any approach in gradually-typed Grace needed to support both. Other approaches lead to matching protocols that are difficult to follow or understand, such as those relying on nested blocks. We designed and even implemented some of these models before we were able to eliminate them. The final model, of reified partial functions, addresses the shortcomings of the alternatives while being readily embeddable into the language.

4.2 Patterns as an Object Framework

We next describe how we implemented the conceptual model as an object-oriented framework. Most Grace programmers won't need to know about this implementation; the exception is when a library implementor wants to provide new kinds of patterns not supported by the pattern syntax.

A pattern object is nothing more than an object that has a match() method: match takes as an argument the target of the match and returns an object of type MatchResult. MatchResult is implemented by two classes: SuccessfulMatch, which inherits from true, and FailedMatch, which inherits from false. Because

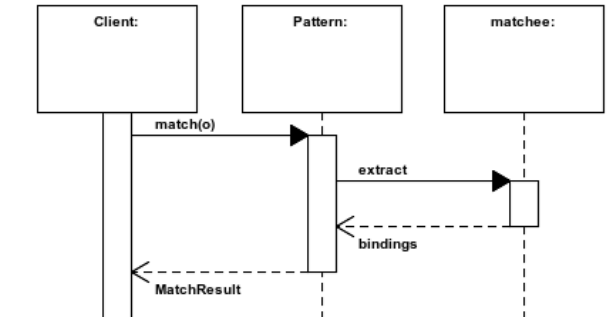

Figure 2. Matching sequence in Grace

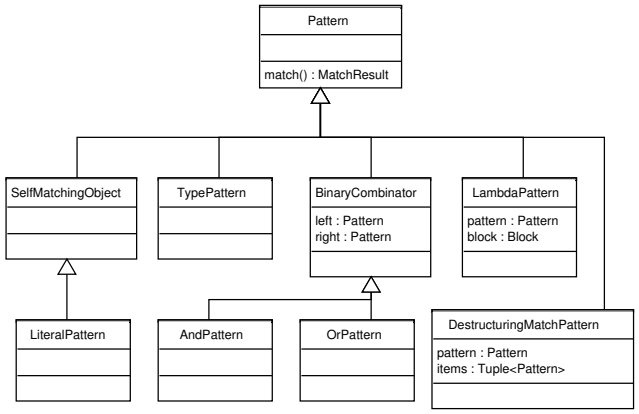

Figure 3. Hierarchy of built-in pattern objects

.

MatchResult objects inherit from the Booleans, the match method of a pattern may be used as a condition:

```
if (pattern.match(obj)) then {
    ...
}
```

MatchResult objects have two methods, in addition to the methods of Booleans. The result method returns the object against which the match was attempted, whether or not it succeeded. In a top-level pattern the result will be the original target of the match, but in a nested pattern it may be more specific. The bindings method returns a list of values that are bound to the variables of the pattern as an effect of a successful match; the bindings of a FailedMatch are always empty. The matching protocol is shown in Figure 2.

A hierarchy of pattern objects, seen in Figure 3, represent patterns at run-time. The simplest pattern is the wildcard pattern, which corresponds to _ in the pattern syntax, always matches, and does not bind anything: the matched value is simply discarded.

```
def wildcardPattern = object {
    method match(o) {
        SuccessfulMatch.new(o, [])
    }
}
```

The variable pattern also always matches, but does bind its argument:

```
class VariablePattern.new(name) {
    method match(o) {
        SuccessfulMatch.new(o, aTuple.new(o))
    }
}
```

The variable pattern corresponds to a variable name in the pattern syntax: { a -> a * 2 } constructs the pattern object VariablePattern.new("a").

Pattern combinators are represented by pattern objects that contain the argument patterns. An AndPattern conjoins two patterns, ensuring that they both match:

```
class AndPattern.new(pattern1, pattern2) {
    method match(o) {
        def match1 = pattern1.match(o)
        if (!match1) then { return match1 }
        def match2 = pattern2.match(o)
        if (!match2) then { return match2 }
        def b = match1.bindings ++ match2.bindings
        SuccessfulMatch.new(o, b)
    }
}
```

Notice that AndPattern is an instance of the *composite* structural pattern [5, p.163], and is itself a pattern. An AndPattern contains other patterns as components and uses the components recursively for matching, without knowing anything about what they are. Here, the component patterns are both applied to the target of the match: if either fails, the AndPattern terminates by returning the failure. When both component patterns successfully match, the AndPattern also returns a SuccessfulMatch, the bindings of which are the concatenation of the bindings of the component matches. The AndPattern corresponds to the & combinator in the pattern syntax.

The other obvious combinator on patterns is disjunction, represented by the OrPattern, which combines two patterns, and succeeds if either of them succeeds.

```
class OrPattern.new(pattern1, pattern2) {
    method match(o) {
        if (pattern1.match(o)) then {
            return SuccessfulMatch.new(o, aTuple.new)
        }
        if (pattern2.match(o)) then {
            return SuccessfulMatch.new(o, aTuple.new)
        }
        FailedMatch.new(o)
    }
}
```

The OrPattern has the dual structure to the AndPattern, but cannot return bindings. This is because the caller cannot know which of the component patterns succeeded. The OrPattern corresponds to the | combinator in the pattern syntax.

Grace types are represented by objects with a method match(o) that returns a SuccessfulMatch if the argument o has a conforming type, and a FailedMatch otherwise. In both cases, bindings is empty. Thus, types are also patterns. In the next example we use a type as a pattern to ensure that o has a value method, and then request it.

```
type Valuable = { value -> Number }
if (Valuable.match(o)) then {
    total := total + o.value
}
```

As a consequence, types can be used as patterns in match-case expressions. A pattern like z:Valuable combines a type-match with a variable pattern that binds a value. This is represented using the AndPattern, so a case of the form:

```
case { z:Valuable -> ... z.value ... }
```

results in the construction of the pattern

```
AndPattern.new(VariablePattern.new("z"), Valuable)
```

This pattern will succeed when the target of the match has the methods defined in the Valuable type, and will result in the variable z being bound to the target in the body of the block.

Like types, the matchable literals of type Number, String, and Boolean are also patterns. As patterns, these objects match exactly those objects to which they are equal. Such patterns do more than duplicate the equality test; they are a useful part of Grace's concise pattern syntax, and fit nicely within the composite structure of patterns.

More complicated patterns are possible: in particular, destructuring matches can be used to match against some of the exposed (conceptual) state of an object. For example, given some Point objects, we may want to match those with a subset of coordinates, or extract and bind the coordinates. We could write a pattern ourselves to do so, but Grace also provides a built-in destructuring pattern for any type with an extract method. For example, given the type

```
type Point = {
  x -> Number
  y -> Number
  extract -> Tuple<Number,Number>
}
```

and the class

```
class aCartesianPoint.at(x':Number,y':Number) {
  def x = x'
  def y = y'
  method extract { aTuple.new(x,y) }
}
```

we can perform a destructuring match using the Point type pattern:

```
match (pt)
  case { p:Point(x, 0) -> "The point ({x}, 0)" }
```

This translates to a use of the DestructuringMatchPattern.

```
class DestructuringMatchPattern.new(pat, items) {
  method match(o) {
    def m = pat.match(o)
    if (!m) then { return FailedMatch.new(o) }
    var matchbindings := m.bindings
    if (matchbindings.size == 0) then {
      matchbindings := pat.extract(m.result)
    }
    var bindings := aTuple.new
    for (items) and (matchbindings) do { it, mb ->
      def b = it.match(mb)
      if (!b) then { return FailedMatch.new(o) }
      bindings := bindings ++ b.bindings
    }
    SuccessfulMatch.new(o, bindings)
  }
}
```

The DestructuringMatchPattern ensures two things: that the parameter pat successfully matches, and that the sub-patterns items match the bindings returned in that SuccessfulMatch. The pattern syntax Point(x, 0) corresponds to the pattern

```
DestructuringMatchPattern.new(
    Point, aTuple.new(VariablePattern.new "x", 0))
```

In this example, the destructuring pattern would ensure that the pattern Point matches the target of the match, and that the bindings returned by requesting extract match the sub-patterns x and 0. The variable pattern x always matches, so it will succeed and accrue one binding, the first extracted value, while the constant pattern matches only when the second extracted value is zero. The DestructuringMatchPattern then has just one binding, to x, and will succeed only when the target is a Point with y-coordinate zero. A programmer can also define their own destructuring patterns with customized bindings. An example is given in Section 7.

Pattern objects may be nested to represent nested patterns. A nested pattern like

```
case { p : Pair(Pair(0, y:Number),
           Pair(w, 1)) -> ... }
```

results in the pattern object

```
DestructuringMatchPattern.new(Pair,
  aTuple.new(DestructuringMatchPattern.new(Pair,
    aTuple.new(0, AndPattern.new(
      VariablePattern.new "y", Number))),
  DestructuringMatchPattern.new(
    Pair, aTuple.new(VariablePattern.new "w", 1))))
```

A destructuring match may match recursively on any of its destructured values. The values may also be bound to variables or ignored entirely.

All of the patterns expressible in the pattern syntax described in Section 3 are represented as objects. Programmers can also construct pattern objects directly, and mix them with the pattern objects generated from the pattern syntax.

Now that we understand how patterns are represented as objects, we can explain how the match()case() method is implemented. The case parameters are LambdaPatterns, which combine a pattern, representing the domain of the function, with a block of executable code, representing the body. The match() method of a LambdaPattern first attempts to match the pattern. Only if the match succeeds does it attempt to execute the block with the accrued bindings. The return value of the block is used as the result of the SuccessfulMatch.

```
class LambdaPattern.new(pattern, block) {
    method match(obj) {
        def result = pattern.match(obj)
        if (!result) then {
            return FailedMatch.new(obj)
        }
        def returnValue = block.applyWithArguments(result
            .bindings)
        SuccessfulMatch.new(returnValue, aTuple.new)
    }
}
```

A lambda pattern may also be used as a sub-pattern when side effects are desirable during matching, or when a simple way of computing the result of a match is required. Lambda patterns are automatically created from every one-parameter block in the source code.

The match-case method tries to apply each LambdaPattern in turn until one succeeds. This behaviour is equivalent to that of the | combinator — try to match each pattern in turn, returning the first success. The two-clause match()case()case() method would look like this:

```
method match(val) case(b1) case(b2) {
  (b1 | b2 | { _ -> error "Failed match-case" })
      .match(val).result
}
```

5. Types and Patterns

How does pattern-matching mesh with Grace's optional, gradual type system? All of the patterns that we have seen so far can be

statically typed. Most importantly, when a variable is bound in a match, either at the top level or via destructuring, that variable can be given a valid static type.

The Pattern and MatchResult types are generic, parameterized over the types of the result and the bindings:

```
type Pattern<R,T> = {
  match(o:Object) -> MatchResult<R,T>
}
type MatchResult<R,T> = {
  result -> R
  bindings -> T
}
```

In untyped code, these parameters are instantiated with type Dynamic, but patterns may declare types for themselves if desired.

The simplest pattern that assigns a type is a type pattern itself. A variable associated with a type pattern has the corresponding type, so the variable n in the pattern n:Number is given type Number. The pattern object would instantiate the parameter R to Number in this case.

In the case:

```
case { p:Pair -> "Pair ({p.left}, {p.right})" }
```

the new variable p has static type Pair, making the operations p.left and p.right statically type safe. The pattern object for Pair would have the type:

```
type PairPattern = {
  match(target:Object) -> MatchResult<Pair,Tuple<>>
}
```

Destructuring matches are handled similarly to Scala [4], though without any need for special "case classes." The values to match must be returned as a tuple from a method named extract. Thus, to destructure a Point, the type Point must have an extract method. As shown previously, this method has signature

```
extract -> Tuple<Number,Number>
```

The parameters of the Tuple type determine the types of the new identifiers introduced by the destructuring.

Note that the signature of the extract method is taken from the type rather than the target of the match. Thus, any object conforming to type Point will work properly with the destructuring match expression `{ _ : Point(x : Number, y : Number)-> x * y }`.

The Point type pattern object will have type:

```
type PointPattern = {
  match(target:Object) -> MatchResult<Point,
                                      Tuple<Number,Number>>
}
```

The correct type can then be statically associated with both the matched object and the destructured values.

When combinators are used, the types of variables become more complex. The & combinator gives the variable the types from both patterns. In the following example, o has a type conforming to both X and Y, so both x and y methods may be requested:

```
type X = { x }  // type with x method
type Y = { y }  // type with y method
match (val)
  case { o : X & Y -> "Point ({o.x}, {o.y})" }
```

Grace's type system uses the notation X & Y for the type that conforms to both X and Y, so this is all consistent, and we can say that o has type X & Y.

By contrast, | matches when either pattern does. In this case, only methods that are shared by all objects matching *either one* of the patterns may be requested in statically type-safe code. This type corresponds to Grace's untagged variant types, also written with |. Any object that conforms to X and any object that conforms to Y will conform to the untagged variant type X | Y — and these are exactly the objects that the pattern X | Y will match.

Untagged variant types also serve another role. A match-case expression can be statically determined to be exhaustive when the target of the match has a variant type, and all branches of the variant have associated cases. A warning can be given both for non-exhaustive matches, which may have unintended behaviour, and for unreachable branches of the match:

```
var x : Number | String | Boolean
...
match (x)
  // Doesn't execute anything if x is a String
  case { n : Number -> ... }
  case { b : Boolean -> ... }
...
match (x)
  case { n : Number -> ... }
  case { s : String -> ... }
  case { b : Boolean -> ... }
  case { _ -> // Unreachable! }
```

Particularly in student code, it can be useful to report errors for missed or impossible cases. The ability to do so is a natural consequence of variant types and the structure of pattern objects. More stringent restrictions would be possible, and the language is designed so that an instructor who wanted stricter static typing could construct a dialect with additional checking.

6. Case study

We implemented an existing system, a pretty-printer for Grace code, using match-case instead of methods on AST nodes. The pretty-printer is a part of the existing Minigrace compiler, and was extended to include the new implementation alongside the existing version.

Both implementations output code that is semantically identical to the input, but with possible changes to layout or low-level structure. The implementations are almost identical in length and substantive content. One uses a "toGrace" method defined inside each AST node, and the other a single method defined outside, containing a match expression with a case for each kind of node. The method and case bodies are the same up to the necessary changes.

Both implementations perform correctly for all of the cases in the Minigrace test suite, producing output semantically identical to their input, and producing identical output on all available files. Each comes to approximately 400 lines of code.

We measured the time taken to pretty-print files of varying sizes and complexities, all from the Minigrace distribution; the results are reported in Table 1. All measurements were made on an Acer Aspire 5745G machine with an Intel Core i7-720QM 1.60GHz processor and 4GB of memory running Linux 3.4.4 with glibc 2.16.0, in 64-bit mode. Tests were run using the "performance" CPU frequency governor, pegging the clock at 1.60GHz, and with CPU affinity fixed to a single processor. A modified version of Minigrace 0.0.7.1072/3ae003b with match-case pretty printing was used[1], compiled with GCC 4.7.1. Times are CPU time given by clock(3), from immediately before pretty-printing to immediately after, using the arithmetic mean of five runs. Heap memory is the

[1] Mainline Minigrace is available from https://github.com/mwh/minigrace and should build on any POSIX-compatible system. A tarball of the modified version used for these tests is available from http://ecs.vuw.ac.nz/~mwh/dls2012-snapshot.tar.bz2.

total allocated by the entire compiler through parsing and pretty-printing, as reported by the runtime's internal memory tracker by setting the GRACE_STATS environment variable. Objects is the total number of objects allocated by the entire compiler through parsing and pretty-printing.

The match-case version is significantly slower than the method-based version, in part because of inefficiencies in the current prototype compiler. A large amount of the time is in allocating objects to represent the blocks, and garbage-collecting those objects subsequently. An object is allocated for every case block — 26 blocks here — each time the match-case clause is executed. Match-case is also ordering-dependent: because every pattern is tested until one matches, the further down the list the successful pattern is, the more methods must be run and the longer it will take. If the most common cases are moved to the top of the match-case statement some speedup is obtained, but which cases these are varies according to the input. With careful profiling we were able to improve the times an average of 17.8% over the version with the cases in more-or-less random order. By removing cases that would never be reached in practice (representing purely-internal or deprecated nodes), speedups of nearly 30% were reached. For code that will be used in heavy loops or recursively, such profiling will be worthwhile, but many uses of match-case are in less performance-critical pathways and will not notice a significant slowdown.

While the match-case version is currently significantly slower, and likely to remain slower even with optimization, it is much more contained. The original addition of the pretty-printer was quite intrusive, requiring new methods to be added to 26 different classes, scattered throughout the code. Using match-case all of the pretty-printing behaviour is centralized in one place. The Visitor pattern [5] is another approach with this property, but which is also somewhat less clear. Match-case can be slightly more concise than either of the alternative approachs, and has substantially greater cohesion.

7. Discussion and Related Work

While object orientation and pattern matching are often contrasted, even in purely object oriented code there are times when we need to know more precisely the type of an object, as when operating on elements of a heterogeneous collection. This is one reason why Java has an `instanceof` operator, and even Smalltalk programmers must sometimes ask an object if it accepts a particular message. Unfortunately such a construct is awkward to use, often requiring type casts and redundant checks. Odersky also argues that pattern-matching is simpler and clearer in many circumstances that would otherwise require use of the Visitor pattern with its extra overhead. He also argues that pattern matching arises naturally in situations that require handling different kinds of exceptions.

For these reasons, and also because we would like to enable instructors and students to compare programs that achieve the same goals using pattern matching and polymorphic dispatch, we include pattern matching in Grace.

Grace's pattern matching was inspired by, and partly based upon, the designs for pattern matching in Scala [4] and in Newspeak [6]: these references also provide good general coverage of the topic of object-oriented pattern-matching. The overall "look" of our `match()case()` statement is derived from Scala, along with much of Grace's surface syntax. However, in Grace each case is an independent block, whereas one of Scala's blocks encompasses multiple partial functions. Grace's `extract` method is based on (and named after) Scala's extractors, but instead of using an implicit unapply method we simply request a method called `extract`. Where Scala supports multiple different extractions via different unapplys, Grace programmers can build explicit patterns to extract whatever features they require, assuming that the object's interface provides the methods needed to obtain the data. A `PolarPointPattern`, for

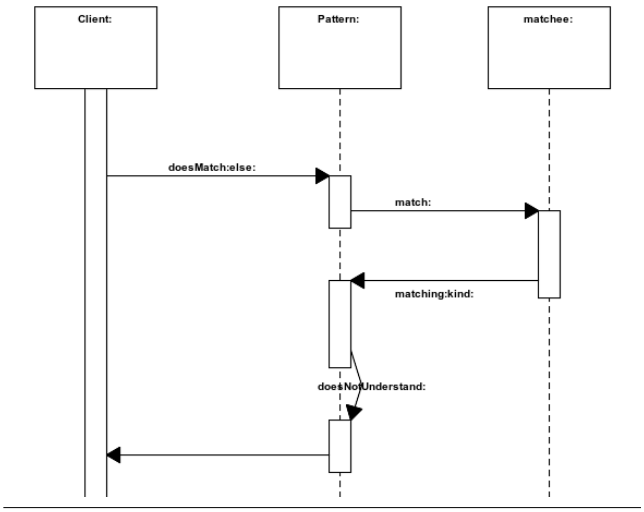

Figure 4. Matching sequence in Newspeak

example, can be defined that matches `Points` but extracts radius and angle:

```
def PolarPointPattern = object {
  method match(o : Object) -> MatchResult {
    match(o)
      case { p:Point(x,y) ->
        SuccessfulMatch.new(p,
          aTuple.new(((x*x)+(y*y)).sqrt, (y/x).arctan))
      }
      case { _ -> FailedMatch.new(o) }
  }
}
```

The notion of first-class patterns and pattern combinators was inspired by Newspeak's design for pattern-matching. Although powerful, and not requiring primitive "type" objects to match, the Newspeak design seems rather more complex than Grace, being based on a quadruple-dispatch between objects, cases, and patterns. Newspeak arguably does the "right" thing (or at least the pure object-oriented thing) in that the target of the match is always in control of the protocol: Newspeak's initial matching message is sent to the subject of the match, passing the pattern as an argument; this is the reverse of Grace's design, in which the subject is passed as argument to the `match` method. In Newspeak, the default response to that initial message is to double-dispatch back, *i.e.*, to send a message to the pattern asking that it match itself against the subject. In theory, this gives the subject complete control of the matching process. However, when we looked at the use-cases for matching, we noted this default method was never overridden: every use-case used the double dispatch. Grace's design benefits from a single match interface implemented by many objects via the composite pattern, and a simple afferent design where match requests are sent to patterns, and extract requests are sent to the objects being matched.

At the bottom of the Newspeak pattern matching protocol, a message characterizing the object is again double-dispatched back to the pattern: a point might send the two-parameter message "x:y:", passing its rectangular coordinates as the arguments to the message, whereas a string may simply send itself as the sole argument to a one-parameter message "string:". Patterns implement the message that will be sent by the object that they match — other messages raise a "does not understand" exception, which is interpreted as a failed match. This design is also similar to Blume et al.'s proposal for matching based on first class *cases*, rather than first class

Source file	LOC	Time (s)			Heap (MB)			Objects		
		Method	Match-1	Match-2	Method	Match-1	Match-2	Method	Match-1	Match-2
lexer.grace	667	0.05	0.88	0.69	42	156	145	290580	1015259	865830
compiler.grace	92	0.01	0.12	0.10	4	22	20	34802	144289	125207
parser.grace	1685	0.12	1.93	1.62	82	325	306	576147	2104009	1842650
genc.grace	1619	0.22	3.17	2.64	99	485	448	726524	3177327	2676071

Table 1. Statistics of method-based pretty-printer and match-case pretty-printer. Match-1 has the match cases in arbitrary order. Match-2 reorders them to place the common cases first.

patterns [3]. Both our design and Newspeak's design need syntactic extensions to support patterns and variable binding: our design also needs the primitive "type pattern" objects to be supplied by the runtime system. (In fact, type patterns could alternatively have been implemented via reflection, but we did not have reflection in Grace when we implemented pattern matching.) In spite of this disadvantage, we consider our pattern matching protocol, shown in Figure 2, significantly more straightforward than Newspeak's quadruple dispatch, shown in Figure 4.

The idea of reifying patterns (rather than cases) as first class partial functions, and then building larger scale structures (*e.g.*, case statements, clausal function definitions) out of pattern combinators goes back at least as far as Tullsen's first-class patterns proposal for Haskell [26]. Tullssen represents patterns as reified partial functions, and pattern combinators are thus function combinators. Reinke's "lambda-match" proposal extends this approach to combine individual cases into whole match statements with combinators [18]. Indeed, Barry Jay has developed an entire calculus of programming based on patterns rather than functions [11]. Our design is similar to these proposals, but reifies partial functions and patterns as objects in an object-oriented language, rather than functions in a functional language.

The idea of supporting programmer defined matching and destructing of abstract types in functional languages goes back at least to Wadler's "Views" proposal [29]. Peyton Jones proposed View Patterns as one way to support views in Haskell: a view pattern is a function that matches an abstract value and returns a concrete data type that can be further matched-against [17]. While Haskell (and most other functional languages) provide excellent syntactic and semantic support for patterns — this support is generally built in to the language — Haskell's patterns are neither first class nor extensible.

The Racket Scheme dialect also includes an extensible pattern matching facility [25]. Uniquely amongst all the designs presented here, Racket's powerful macros enable the language to be extended without any changes to its core implementation. Certainly the Racket implementation is rather more optimized than our design. In contrast, while our design requires semantic support for partial functions and syntactic support for destructuring, the remainder of our design is built using straightforward, non-reflexive, object-oriented design techniques.

The F# language supports matching against abstract structures via active patterns [23]. An active pattern is a function with a structured name where that name indicates one or more alternative cases. Although they can support partial matching, active patterns support exhaustive matching particularly well, because a single succinct pattern function matches and destructures every alternative, returning a result indicating which case of the pattern is matched. In contrast, our design, like Scala and Newspeak, requires individual matching and extract functions for each case. As in our approach, active patterns support first class patterns (as first-class functions) and support a range of pattern combinators.

Pattern matching has also been supported natively in a variety of recent object-oriented language proposals. Thorn [2], for example, makes heavy use of pattern matching, with extensive syntax supporting matches as part of many of the language constructs, including the control structures and clausal function definitions. Thorn offers a wide array of built in data types, each with corresponding matching destructors, and a set of algebraic pattern combinators. A Thorn class's formal parameters are used to initialize instance objects, and they are also extracted if an object is matched against. Fortress [21], another large and flexible language, has a relatively modest range of pattern matching and destructuring object and tuples. Like Thorn and Scala, but unlike Grace, patterns in Fortress may be used freely in definitions, rather that just in a match statement (in Fortress, a typecase statement).

Of course, pattern matching has been suggested as an extension to the ubiquitous Java in a range of different ways. Machete (a forerunner to Thorn) is the most conventional: introducing a match statement, a rich library of patterns, and a separate type of "deconstructor" declaration to objects that extract values when objects are matched [9]. Again presaging Thorn, Machete includes special patterns (and syntax) for matching with regular expressions, arrays, bitpatterns, and XML. MatchO provides a flexible pattern library but does not need (or provide) any specialized syntax — although syntax can be supplied by invoking a generated parser inline [28]. In MatchO, patterns are necessarily first-class, given that they are implemented as a normal Java library, but MatchO does not generalize patterns to pattern combinators. OOMatch [19] is a more radical language design, similar to Fortress in that it fully integrates pattern matching into Java, providing clausal function definitions and multimethods.

As well as being in an object-oriented language, our design differs from most of these designs in that Grace is gradually and optionally typed, while all these languages are strongly statically typed, typically via some mix of inference and explicit declarations. (The outliers here are Racket, which is also optionally typed, and Newspeak, which is purely dynamic). These languages' pattern-matching facilities are generally tied tightly into their type systems. This is as true for Racket as it is for Scala or Fortress: objects are matched and destructured based on their defining class. This is true even in OCaml, which also has a structurally-typed object system, but which supports pattern matching only on algebraic data types, not on objects [13]. In contrast, Grace matches only on the publicly visible interface of an object — its "duck type" if you will — and this is completely decoupled from that object's implementation.

We have considered a number of further extensions to our pattern-matching design. One relatively straightforward extension is to support matching against regular expressions. The most straightforward implementation is to incorporate an external regexp library such as Perl-compatible regular expressions (PCRE) [8]. Here we provide a simple syntax for regular expressions, a prefix `/` method (since Grace supports prefix operators, but not postfix or matchfix), that converts the string into a regexp object. To fit into the pattern matching framework, all the regexp object needs to do is to support the basic protocol captured by the `Pattern` type.

A more ambitious extension is to incorporate combinator parsing into the matching framework. From one perspective, parsers,

especially combinator parsers [12] are rather similar to matching: parsers either complete successfully and return a representation of the parsed input, or fail: that is, parsers are partial functions. The key difference is that while patterns are matched against whole objects, combinator parsers typically parse an *input sequence*, and a successful parse may consume some, all, or none of the remaining input. For this extension, we extend MatchResult to maintain the representation of the unparsed input: the sequence parser combinator ~ starts the right-hand side parsing when the left-hand side parser finishes. What is interesting about this embedding is that alternation and parallel parser combinators correspond exactly to the "and" and "or" pattern combinators. The resulting language is similar in many ways to OMeta [30] — an object-oriented language for parsing — because the parsers are integrated into the matching facility in the language, rather than simply being a stand-alone library. Lua's text pattern matching library is built on Parsing Expression Grammars in a similar style, without any syntactic support, but Lua matches only against text, not arbitrary objects [10].

8. Conclusion

Pattern matching, of various kinds, once the domain of advanced functional languages, or regular-expression-based scripting languages, is now an expected feature of mainstream programming: indeed, the current draft of the forthcoming ACM Computer Science curriculum [24] requires first year students to understand both object-oriented programming with dynamic dispatch and functional programming with pattern matching as two alternative styles of program organisation.

As a language aimed particularly at teaching computer science, we have found it useful for Grace to support pattern matching. This paper describes how we incorporated pattern matching into Grace, while maintaining (indeed leveraging) Grace's nature as a dynamic, pure, object-oriented language. While the detailed design is no doubt specific to Grace, we consider that the general principles — reifying partial functions, modeling both patterns and cases as first-class objects, and building composite structures via pattern combinators and the Composite pattern — can be used to incorporate pattern-matching into most modern object-oriented languages.

References

[1] A. P. Black, K. B. Bruce, M. Homer, and J. Noble. Grace: the absence of (inessential) difficulty. In *Onward!*, 2012. To appear.

[2] B. Bloom1, J. Field, N. Nystrom, J. Östlund, G. Richards, R. Strniša, J. Vitek, and T. Wrigstad. Thorn—robust, concurrent, extensible scripting on the JVM. In *OOPSLA*, 2009.

[3] M. Blume, U. A. Acar, and W. Chae. Extensible programming with first-class cases. In *ICFP*, 2006.

[4] B. Emir, M. Odersky, and J. Williams. Matching objects with patterns. In *ECOOP*, pages 273–298, 2007.

[5] E. Gamma, R. Helm, R. E. Johnson, and J. Vlissides. *Design Patterns*. Addison-Wesley, 1994.

[6] F. Geller, R. Hirschfeld, and G. Bracha. Pattern matching for an object-oriented and dynamically typed programming language. Technical Report 36, Hasso-Plattner-Instituts für Sofwaresystemtechnik an der Universität Potsdam, 2010.

[7] A. Goldberg and D. Robson. *Smalltalk–80: The language and its implementation*. Addison Wesley, 1983.

[8] P. Hazel. *Perl-compatible regular expressions*. The University of Cambridge, 2012. pcre.org.

[9] M. Hirzel, N. Nystrom, B. Bloom, and J. Vitek. Matchete: Paths through the pattern matching jungle. In *PADL*, 2008.

[10] R. Ierusalimschy. A text pattern-matching tool based on parsing expression grammars. *Softw. Pract. Exper.*, 39(3):221–258, 2009.

[11] B. Jay. *Pattern Calculus: Computing with Functions and Structures*. Springer-Verlag, 2009.

[12] D. Leijen and E. Meijer. Parsec: Direct style monadic parser combinators for the real world. Technical Report UU-CS-2001-27, Universiteit Utrecht, 2001.

[13] X. Leroy, D. Doligez, A. Frisch, J. Garrigue, D. Rémy, and J. Vouillon. The OCaml system release 3.12: Documentation and user's manual. http://caml.inria.fr/pub/docs/manual-ocaml/index.html, July 2011.

[14] B. Meyer. *Touch of Class: Learning to Program Well with Object and Contracts*. Springer-Verlag, 2009.

[15] R. Milner. The Standard ML core language. *Polymorphism*, 2(2), 1985. 28 pages. An earlier version appeared in Proc. 1984 ACM Symp. on Lisp and Functional Programming.

[16] M. Odersky. In defense of pattern matching. http://www.artima.com/weblogs/viewpost.jsp?thread=166742, 2006.

[17] S. Peyton Jones. View patterns: lightweight views for Haskell. http://hackage.haskell.org/trac/ghc/wiki/ViewPatterns, 2007.

[18] C. Reinke. Introduce lambda-match (explicit match failure and fall-through). Haskell-Prime Ticket #144, http://hackage.haskell.org/trac/haskell-prime/ticket/114, Oct. 2006.

[19] A. Richard and O. Lhoták. OOMatch: Pattern matching as dispatch in Java. In *FOOL*, 2008.

[20] A. Robbins. Learning edge momentum. *Computer Science Education*, 20(1):37–71, Mar. 2010.

[21] S. Ryu, C. Park, and G. L. Steele Jr. Adding pattern matching to existing object-oriented languages. In *FOOL*, 2010.

[22] D. Syme, G. Neverov, and J. Margetson. Extensible pattern matching via a lightweight language extension. In *Proceedings of the 12th ACM SIGPLAN international conference on Functional programming*, ICFP '07, pages 29–40, New York, NY, USA, 2007. ACM.

[23] D. Syme, G. Neverov, and J. Margetson. Extensible pattern matching via a lightweight language extension. In *ICFP*, 2007.

[24] The Joint Task Force on Computing Curricula. Computer science curricula 2013 (strawman draft). cs2013.org, Feb. 2012.

[25] S. Tobin-Hochstadt. Extensible pattern matching in an extensible language. arXiv:1106.2578v1[cs.PL], 2010.

[26] M. Tullsen. First-class patterns. In *PADL*, number 1753 in LNCS, 2000.

[27] D. Ungar and R. B. Smith. SELF: the Power of Simplicity. *Lisp and Symbolic Computation*, 4(3), June 1991.

[28] J. Visser. Matching objects without language extension. *JOT*, 5(8), 2006. http://www.jot.fm/issues/issue200611/article2.

[29] P. Wadler. Views: A way for pattern matching to cohabit with data abstraction. In *POPL*, 1987.

[30] A. Warth and I. Piumarta. OMeta: an object-oriented language for pattern matching. In *Dynamic Language Symposium*, 2007.