Pattern Matching and Abstract Data Types

Tom Murphy VII

3 Dec 2002

Outline

- Problem Setup
- Views (“Views: A Way For Pattern Matching To Cohabit With Data Abstraction”, Wadler, 1986)
- Active Patterns (“A New Look at Pattern Matching in Abstract Data Types”, Pedro, Peña, Núñez, 1996)
- Without language extensions (“Programming with Recursion Schemes”, Wang, Murphy, 2002)
- In the context of module languages (some thoughts)
- Conclusion

Pattern Matching

Pattern matching is a way of conveniently manipulating concrete data types.

```
fun cat nil b = b
  | cat (h :: t) b = h :: (cat t b)
```

vs.

```
fun cat a b =
     if List.null a
     then b
     else hd a :: (cat (tl a) b)
```

Pattern Matching

This is great if we're working with types that are actually implemented with the `datatype` mechanism.

Natural numbers can be thought of as a sort of `datatype`:

```
fun fact Zero = Zero
  | fact (m as Succ n) = m * fact n
```

However, implementing natural numbers (for instance) with datatypes is much too inefficient.

Abstraction

Of course, the solution is abstraction. We hide the “dirty” implementation details behind an interface.

But pattern matching and abstraction are at odds: Pattern matching insists that the type be *concrete* while the entire point of abstraction is to hide such details.

One solution: Wadler’s *views*.

Views

Views: Exhibit an isomorphism between an arbitrary type and a "`datatype`" (view).

- Provide *in* and *out* functions (that are inverses)
- Can have many views of the same type
- Can hold the type abstract while publishing views (using the normal mechanisms)

(Note: I've translated Wadler's examples to an SML-like notation.)

Views: Natural Numbers

Here's a sample view of the existing `int` type.

```
view int as Zero
          | Succ int

with in 0 = Zero
   | in n = Succ(n - 1)

and out Zero = 0
  | out (Succ n) = n + 1
```

`Zero : int`, `Succ : int` $\longrightarrow$ `int` (?), and act as SML constructors.

Using Views

```
fun fib Zero = Zero
  | fib (Succ Zero) = Succ Zero
  | fib (Succ (Succ n)) =
      fib n + fib (Succ n)
```

A Second View

We can add a second view:

```
view int as Zero | Even int | Odd int

with in 0 = Zero
   | in n = if n mod 2 = 0
            then Even (n div 2)
            else Odd (n div 2)

and out Zero = 0
  | out (Even n) = 2 * n
  | out (Odd n) = 2 * n + 1
```

Using this View

```
fun power x Zero = 1
  | power x (Even n) = power (x * x) n
  | power x (Odd n) = x * power (x * x) n
```

Views

That's really all there is to it! Wadler gives some other neat examples, like viewing a list backwards:

```
view α as nil | α list Snoc α

with in (x :: nil) = nil Snoc x
   | in (x :: (l Snoc x')) = (x :: l) Snoc x'

and out (nil Snoc x) = (x :: nil)
  | out ((x :: l) Snoc x') = x :: (l Snoc x')
```

...note that *in/out* are literally inverses. Also note that *in* invokes the view recursively by pattern matching against Snoc.

More. . .

There are more similar examples in the paper about lists and trees.

Weird Stuff: &

Let’s code up the ’as’ pattern (call it &).

```
view α as α & α
with in x = x & x
and out (x & y) = if x = y
                  then x
                  else raise Bogus

fun fact Zero = Zero
  | fact (m & Succ n) = m * fact n
```

This makes sense, but what is the meaning of the *expression* `1 & 1`?

Weird Stuff: Guards/Predicates

```
view int as EvenP of int | OddP of int
with in n = if n mod 2 = 0
            then EvenP else OddP
and out (EvenP n) = if n mod 2 = 0
                    then n else raise Bogus
  | out (OddP n)  = if n mod 2 = 1
                    then n else raise Bogus

fun cz (OddP 1) = ()
  | cz (EvenP n) = cz (n div 2)
  | cz (OddP n) = cz (3 * n + 1)
```

. . . again, what use are EvenP and OddP outside of patterns? What if we don’t even want EvenP and OddP to carry arguments?

Views: Summary

- (+) Combines pattern matching, data abstraction
- (+) Views behave like existing datatype constructors
- (-) Need to validate that *in* and *out* are inverses
- (-) Effectful *out* functions force the programmer to understand the pattern compilation algorithm (TILT's is 1,500 lines)

...

Views: Summary

- (-) “Unnecessary” symmetry with &, predicates—we really just want the destructor
- (-) No treatment of typing
- (-) Views are not higher-order
- (-) Not supported in any language

Active Patterns

1. Like views, but higher-order.
2. Introduce a type of patterns.
3. *Expose* the assymetry between constructors and destructors
4. Emphasis is on patterns, since constructors are just functions
5. Strange syntax (examples will be in Haskell)

Active Patterns

```
(* define datatype of
   complex numbers (as r,i) *)
data cplx = Cart real real

(* the Imag pattern extracts the i field *)
Imag i  match  Cart _ i

(* .. and we can use it like any pattern. *)
isReal (Imag i) = (i == 0.0)
```

Computation, Multiple Arguments

Active Patterns can do computation.

`Magnitude (sqrt (r*r + i*i))`

match `Cart r i`

And APs can also be of any arity.

`SelfAndNeg x (˜ x)` match `x`

Active Patterns: @

"`as`" is written `@` and can have a pattern on each side.

```
Real r   match   Cart r _
```

```
add (Real r1) @ (Imag i1)
    (Real r2) @ (Imag i2) =
      Cart (r1 + r2) (i1 + i2)
```

Active Patterns: Asymmetry

The assymetry of APs allows us to make sense of patterns that are just predicates.

```
Even   match   n, if n mod 2 = 0
Odd    match   n, if n mod 2 = 1


cz 1 = ()
cz n @ Even = cz (n div 2)
cz n @ Odd = cz (n * 3 + 1)
```

To accomplish this with views, we needed to provide canonical representatives of “Even” and “Odd” integers for use as constructors. That didn’t make sense.

Active Patterns: Typing

If we want to pass around APs, we need to assign them types.

If an AP matches values of type τ and has n arguments of types σ_1 through σ_n, then its type is:

$$\langle \sigma_1, \ldots, \sigma_n, \tau \rangle$$

(Not a tuple. Just a crazy notation.)

```
Even, Odd : < int >
Real, Imag : < real, cplx >
SelfAndNeg : < cplx, cplx, cplx >
```

Higher-order APs

Now let's write functions that map between predicates and nullary APs.

```
pred2ap : (α -> Bool) → < α >
ap2pred : < α > → (α -> Bool)
pred2ap p = let C  match  x, if p x
            in C end
ap2pred C = λ x => case x of
                        C => true
                        _ => false
```

Higher-order APs: @

We can code up @ itself, too:

$\texttt{@ : <}\ \alpha\texttt{,}\ \alpha\texttt{,}\ \alpha\ \texttt{>}$

`(x @ x)` match `x`

Active Patterns: Summary

The authors formalize their system, and provide a method for compiling them efficiently, but the details are not interesting in the context of this class.

1. (+) First class patterns
2. (+) At least as powerful as views
3. (-) Requires separate syntactic classes for active patterns / variables
4. (-) Location of effects in patterns still obscured

Who needs language extensions?

We can get much of the functionality of views by simply coding them up in existing functional languages.

```
signature NAT = sig
   type t
   datatype ’a front = Zero | Succ of ’a
   val inj : t front -> t
   val prj : t -> t front
end
```

Implementing NAT

```
structure Nat :> NAT =
struct
   type t = int
   datatype ’a front = Zero | Succ of ’a

   fun inj Zero = 0
     | inj (Succ n) = n + 1

   fun prj 0 = Zero
     | prj n = Succ (n - 1)
end
```

Using Recursion Schemes

```
open Nat
val one = inj (Succ (inj Zero))
fun fact m = case prj m of
         Zero => one
        | Succ n => m * fact n
```

The important things here are the call to `prj` in the `case` object, and the calls to `inj` in `one`.

Recursion Schemes: Nested Patterns

Nested patterns require a little rewriting:

```
fun fib n = case prj n of
              Zero => inj Zero
            | Succ n =>
                 case prj n of
                   Zero => one
                 | Succ m => fib m + fib n
```

. . . but we can define a function

`prj2 : t -> t front front`, and then do a two-deep pattern easily.

Recursion Schemes: Saving Typing

To save some more typing, we can provide iSucc (= inj o Succ) and iZero (= inj Zero), since we always inject after calling one of these constructors.

The paper then goes on to describe how to program generically (fold, unfold) using these recursion schemes.

Recursion Schemes: Summary

1. Use *polymorphic data types*
2. *explicit in/out functions*.
3. (-) Harder to do nested patterns
4. (+) On the other hand, location of effects is explicit
5. (-) Performance penalty
6. (+) No language extension needed

Views as signatures

Harper-Stone-like view of datatypes in signatures:

```
datatype nat =
     Zero
   | Succ nat
```

is really

```
structure nat =
struct
    type t
    val construct : (unit + t) -> t
    val destruct : t -> (unit + t)
end
```

Views as signatures

…that is, `datatype`s are merely an abstract type with coercions into and out of the underlying recursive sum.

(Recursion schemes are really just making this explicit. The sum type is the non-recursive polymorphic datatype.)

The `datatype` declaration gives us one way to match this signature (by generating the coercions automatically). Why not others?

Views as signatures

```
structure Nat :>
      datatype nat = Zero | Succ of nat =
struct

      type t = int

      fun construct (Inl ()) = 0
        | construct (Inr n) = n + 1

      fun destruct 0 = Inl ()
        | destruct n = Inr (n - 1)
end
```

Problems with this approach

In SML, these coercions are known to be *pure* and *total*.

- Value restriction
- Most compilers attempt to avoid function calls when doing separate compilation (see Vanderwaart, et al.'s TLDI '03 paper)
- Same problems with pattern compilation: When do effects happen? How many times?

Conclusion

- Making data abstraction cohabit with other language features can be tricky
- Views and Active Patterns combine abstraction, pattern matching
- Much of this functionality can be coded up in existing languages
- . . . but making new language extensions is tantalizing!

- Have a good winter break!