First Class Patterns*

Mark Tullsen

Department of Computer Science
Yale University
New Haven CT 06520-8285
mark.tullsen@yale.edu

Abstract. Pattern matching is a great convenience in programming. However, pattern matching has its problems: it conflicts with data abstraction; it is complex (at least in Haskell, which has pattern guards, irrefutable patterns, n+k patterns, as patterns, etc.); it is a source of runtime errors; and lastly, one cannot abstract over patterns as they are not a first class language construct. This paper proposes a simplification of pattern matching that makes patterns first class. The key idea is to treat patterns as functions of type "`a→Maybe b`"—i.e., "`a→(Nothing|Just b)`"; thus, patterns and pattern combinators can be written as *functions in the language.*

1 Introduction

A hotly debated issue in the language Haskell [HJW92] has been patterns. What are their semantics? Do we want n+1 patterns? Do we need @-patterns? When do we match lazily and when do we match strictly? Do we need to extend patterns with "pattern guards"? And etc. In this paper I will propose, not another extension, but a simplification to patterns that makes them first class language constructs. I will do so in the context of the language Haskell, although the ideas would apply to any language with higher order functions.

1.1 Patterns are too complex

There is no argument about the elegance of the following simple use of patterns:

```
length []     = 0
length (x:xs) = 1 + length xs
```

Here is another example:

```
zipWith f (a:as) (b:bs) = f a b : zipWith f as bs
zipWith _ _      _      = []
```

* This research was supported in part by NSF under Grant Number CCR-9706747.

Although the definition of zipWith appears nicely symmetric, we know it is not actually so symmetric: patterns match from left to right and thus zipWith is strict on its first argument but not its second. What if we wanted to modify zipWith to be strict on its second argument? Using a case statement we could write zipWith' as follows:

```
zipWith' f as bs =
  case (bs,as) of                    -- switch the evaluation order
    (b:bs', a:as') -> f a b : zipWith' f as' bs'
    _              -> []
```

Or we could write zipWith' using an irrefutable pattern, ~(a:as), and a guard:

```
zipWith' f ~(a:as) (b:bs) | not(null(a:as)) = f a b : zipWith' f as bs
zipWith' _ _       _                        = []
```

Note that a small semantic change requires a large syntactic change. These last two examples show some of Haskell's pattern matching features. In Haskell, there are numerous pattern matching features added to the simplest form of pattern matching (i.e., that used above to define the length function above):

```
f (x1:x2:xs)    = e         -- 1)  nested pattern matching
f (p1,p2)       = e         -- 2)  tuple patterns
f p _           = e         -- 3)  a wild card symbol
f ('a')         = e         -- 4)  literal patterns
f (xs@(x:xs'))  = e         -- 5)  as patterns
f ~(x:xs)       = e         -- 6)  irrefutable patterns
f (x+2)         = e         -- 7)  n+k patterns
f p | g         = e1        -- 8)  guards
    | otherwise = e2        -- 9)  an "abbreviation" for True
    where g = e3            -- 10) an extra binding construct
```

Further extensions, such as views [Wad87b,BC93] and pattern guards, have been suggested. Currently ten pages in the Haskell 98 Report[1] are dedicated to pattern matching.

1.2 Patterns are not first class

Pattern matching is not as expressive as one would wish. For example, in Fig. 1 the two functions getC1s and getC2s are virtually identical but we cannot abstract over the commonality to get a more generic function to which we can pass the two constructors, e.g., getmatches C1. Another example where we see the second class nature of patterns is the following

[1] Sections 3.1.3, 3.1.7, and 4.4.3.

```
data T = C1 T1 | C2 T2

-- getC1s - Extract all the C1's from a list:
getC1s []          = []
getC1s (C1 d : xs) = d : getC1s xs
getC1s (_ : xs)    = getC1s xs

-- getC2s - Extract all the C2's from a list:
getC2s []          = []
getC2s (C2 d : xs) = d : getC2s xs
getC2s (_ : xs)    = getC2s xs
```

Fig. 1. `getC1s` and `getC2s`

```
f x = case x of          g x = case x of
        p1 -> e1                 p1 -> e1
        p2 -> e2                 p2 -> e2
        p3 -> e3                 p3 -> e3
        p4 -> e4
```

in which we would like to abstract out the commonality of `f` and `g`. I.e., we would like to write something like the following (where `f` is defined in terms of `g`):

```
f x = g x EXTEND p4 -> e4
```

1.3 Other Problems with Patterns

In addition, patterns have various other problems:

1. Patterns are not compatible with abstract data types [Wad87b].
2. The semantics of patterns is less than elegant:
   (a) Patterns impose a left-to-right, top-to-bottom evaluation order. As seen when modifying `zipWith` above, if we want a different evaluation order, we must either do without patterns or write far less elegant looking code.
   (b) Pattern syntax gives a declarative appearance that does not exist: the defining equation "`f p1 p2 = e`" is not true in all contexts (unless we ensure patterns are non-overlapping). In Haskell, the order of declarations is insignificant *except* when patterns are used in a sequence of function declarations.
   (c) Patterns engender a multitude of controversies. They beg for extension upon extension: n+k patterns, irrefutable patterns, "as" patterns, guards, and `where` bindings; currently, a new construct, "pattern guards", is being considered for inclusion in Haskell.

3. Pattern matching does not make it syntactically apparent when a partial function is defined: partial functions arise (a) when patterns are not exhaustive in case expressions, (b) in let bindings "`let (x:xs) = e1 in e2`", and (c) in lambda bindings "`λ(x:xs)->e`". Each of these can be the source of an unintentional run-time pattern match failure. (Though the first could be solved simply by disallowing non-exhaustive patterns.)
4. Pattern matching can seem to not be referentially transparent (at least to the uninitiated); the following two expressions are not equal (the second obviously unintended)

```
case e1 of {1 -> e2 ; ...}
let x = 1 in case e1 of {x -> e2; ...}
```

5. Patterns go against the grain of a higher-order programming style: we only get their convenience when the arguments of a function are made explicit.

1.4 Overview

We have seen many problems with pattern matching. This paper presents an alternative to standard pattern matching. The essence of this alternative is presented in Sect. 2 and Sect. 3 provides some examples; Sect. 4 introduces the "pattern binder" construct (simple syntactic sugar) which makes patterns easier to write; Sects. 5 and 6 present various extensions and Sect. 7 presents several laws for patterns; Sect. 8 compares this approach to other approaches and summarizes.

2 An alternative to patterns

The key idea is to use functions of type "`a→Maybe b`" as patterns[2]. A function of this type will be referred to as a pattern-function or sometimes just as a pattern. When a pattern-function is applied, it either fails (returns `Nothing`) or succeeds with some result (returns `Just x`). This section explains the ramifications of using pattern-functions: in Sect. 2.1, I show what we *could* take away from the language, in Sect. 2.2 I show how we can construct patterns using pattern combinators, and in Sect. 2.3 I explain what must be added to the language.

2.1 Simplify the Language

Note that nothing in what follows *requires* that we simplify Haskell as in this section; the point is to explore what we can still do even with such simplicity.

In Haskell, patterns are currently allowed in top-level declarations, lets, lambdas, and case alternatives. We could restrict the language to only allow failure-free

[2] In the Haskell Prelude: `data Maybe a = Nothing | Just a`.

patterns in top-level declarations, lets, and lambdas. We could also restrict the form of case expressions.

The syntax of failure-free patterns is as follows (b is for "binding"):

$$\begin{array}{rll} b ::= & v & \text{variables} \\ \mid & (b, ..., b) & \text{tuples} \end{array}$$

The semantics is given by translation. E.g., `f(x,y) = e` is syntactic sugar for `f z = e{fst z/x,snd z/y}`[3]. Also, `case` expressions are restricted such that they are exhaustive over an algebraic type and alternatives cannot contain literals or nested constructors. E.g.,

```
case e of
 x:xs -> e1
 []   -> e2
```

The objective is to eliminate the implicit left-to-right, top-to-bottom evaluation order. A default alternative is allowed (as "`_->e`") when not all the constructors are given. Obviously, we have not lost any of the expressiveness of the language, but we have lost some convenience. Many programs will not miss the lost features, but other programs will be awkward. For example, the function `zipWith3`, defined in the Prelude thus

```
zipWith3 f (a:as) (b:bs) (c:cs) = f a b c : zipWith3 f as bs cs
zipWith3 _ _      _      _      = []
```

becomes awkward to code with these restrictions:

```
zipWith3 f as bs cs =
 case as of
  a:as' -> case bs of
            b:bs' -> case cs of
                      c:cs' -> f a b c : zipWith3 f as' bs' cs'
                      []    -> []
            []    -> []
  []    -> []
```

The functions `zipWith4`, `zipWith5`, `zipWith6` will be even more awkward. Can we define these less awkwardly in the simplified language?

2.2 Use Pattern Combinators

Patterns are just functions, so we need to define functions to construct and manipulate patterns. I use the following type synonym:

```
type Pat a b = a → Maybe b
```

[3] Due to lifted tuples, this is not always true in Haskell, but I'll ignore this complication.

```
pid x            = Just x
pk x y           = if x == y then Just () else Nothing
pfail _          = Nothing

(p1 .| p2) x     = case p1 x of Just r  -> Just r
                                Nothing -> p2 x
p1 |. p2         = p2 .| p1
(p1 .* p2) (a,b) = case p1 a of
                     Just r1 -> case p2 b of
                                  Just r2 -> Just (r1,r2)
                                  Nothing -> Nothing
                     Nothing -> Nothing
(p1 *. p2) (a,b) = case p2 b of
                     Just r2 -> case p1 a of
                                  Just r1 -> Just (r1,r2)
                                  Nothing -> Nothing
                     Nothing -> Nothing
(p1 .: p2) x     = case p1 x of Just a  -> p2 a
                                Nothing -> Nothing
(p .-> f) x      = case p x of Just r  -> Just (f r)
                               Nothing -> Nothing

(p |> f) x       = case p x of Just r  -> r
                               Nothing -> f x
```

Fig. 2. Pattern Combinators

Here are the signatures for the pattern combinators:

	Pattern Introduction	
`pid`	`:: Pat a a`	always matches
`pk`	`:: Eq a => a → Pat a ()`	"pk x" matches x
`pfail`	`:: Pat a b`	never matches
	Pattern Combination	
`(.|),(|.)`	`:: Pat a b → Pat a b → Pat a b`	or match
`(.:)`	`:: Pat a b → Pat b c → Pat a c`	then match
`(.*),(*.)`	`:: Pat a c → Pat b d → Pat (a,b) (c,d)`	parallel match
`(.->)`	`:: Pat a b → (b→c) → Pat a c`	compose match
	Pattern Elimination	
`(|>)`	`:: Pat a b → (a→b) → (a→b)`	apply pattern

The definitions are in Fig. 2. The difference between `(.|)` and `(|.)` is the order of evaluation, likewise for `(.*)` and `(*.)`.

2.3 Add One Primitive

For a constructor `C` we can get the pattern-function corresponding to it using the # primitive

```
C  :: a → b
C# :: b → Maybe a
```

It satisfies the Pattern Law:

```
C# C     = Just ()  -- when C is a nullary constructor
C# (C x) = Just x   -- otherwise
```

In addition the programmer is allowed to define pairs of functions c and $c^{\#}$, which should satisfy the pattern law (this is unverified). Curried constructors are allowed but the pattern law is more straightforward when constructors are uncurried.

3 Examples

So, what do various functions look like using these pattern combinators? Given the functions `list1`, `list2`, and `list3`, it is useful to have the corresponding pattern-functions `list1#`, `list2#`, and `list3#`[4]:

```
list1 x       = [x]
list2 (x,y)   = [x,y]
list3 (x,y,z) = [x,y,z]

list1# = :# .: (pid .* []#)     .-> \(x,())->x
list2# = :# .: (pid .* list1#)
list3# = :# .: (pid .* list2#)  .-> \(a,(b,c))->(a,b,c)
```

Here is an example where we can structure the code to reflect the types involved very nicely:

```
data Tree a = Leaf a
            | Tree (Tree a, Tree a)

zipTree :: (Tree a,Tree b) → Maybe (Tree (a,b))
zipTree = Leaf# .* Leaf# .-> Leaf
       .| Tree# .* Tree# .:  (zipTree.*zipTree .-> Tree) . zipTuple

zipList :: ([a],[b])  → Maybe [(a,b)]
zipList = []#  .* []#  .-> (\_->[])
      .|  :#   .* :#   .:  (pid .* zipList .-> uncurry (:)) . zipTuple

zipTuple ((x1,x2),(y1,y2)) = ((x1,y1),(x2,y2))
```

[4] The precedences of the pattern operators are from tightest to loosest binding: `(.*)`, `(.:)`, `(.->)`, `(.|)`.

Remember the functions getC1s and getC2s? We can now write code generic over constructors of different data types[5]:

```
getmatches :: (a → Maybe b) → [a] → [b]
getmatches p = catMaybes . map p

getC1s = getmatches C1#
getC2s = getmatches C2#
```

Here is a "cons" view of a join list:

```
data List a = Nil | Unit a | Join (List a) (List a)

cons  x xs          = Join (Unit x) xs

cons# Nil           = Nothing
cons# (Unit a)      = Just (a,Nil)
cons# (Join xs ys) = case cons# xs of
                       Just (x,xs') -> Just (x, Join xs' ys)
                       Nothing      -> cons# ys
```

4 Extension I: Pattern Binders

4.1 The Pattern Binder Construct

At times the higher order approach above seems elegant, but at other times it is clumsy compared to standard pattern matching. The "pattern binder" construct is an attempt to add the minimum of syntactic sugar to make patterns easier to write. Here are some examples:

```
list1#   = {$x:[]}       .-> x
list2#   = {$x:$y:[]}    .-> (x,y)
list3#   = {$x:$y:$z:[]}.-> (x,y,z)

pair x   = {=x:=x:$r} .-> r
```

A pattern binder is "$\{p\} \otimes e$" where e is a Haskell expression, $\otimes$ is a Haskell infix operator, and p is defined as follows:

$$
\begin{array}{rll}
p ::= & \$v & \text{variable} \\
\mid & (p, ..., p) & \text{tuple pattern} \\
\mid & = e & \text{constant} \\
\mid & \%e\, p & \text{apply any pattern-function} \\
\mid & c\, p & \text{apply } c^{\#} \text{ pattern-function} \\
\mid & c & \text{apply } c^{\#} \text{ pattern-function (} c \text{ nullary)}
\end{array}
$$

[5] The library function catMaybes (::[Maybe a]->[a]) extracts the Just's from a list.

Note that c is any constructor or any symbol with a $c^{\#}$ function defined. The pattern binder "$\{p\} \otimes e$" is translated into "$f \otimes (\lambda b \to e)$" as follows (where f is a pattern-function and b is a failure-free pattern):

$$\{p\} \otimes e = f \otimes (\lambda b \to e) \qquad \text{where } (f, b) = cvt(p)$$

$$\begin{aligned}
cvt(\$v) &= (pid,\ v) \\
cvt(p_1, p_2, ..., p_n) &= (\,f_1 .* (f_2 .* \ldots f_n)),\ (b_1, (b_2, \ldots b_n))\,) && \text{where } (f_i, b_i) = cvt(p_i) \\
cvt(= e) &= (pk\, e,\ ()) \\
cvt(\%e\, p) &= (e\ .\!:\ f,\ b) && \text{where } (f, b) = cvt(p) \\
cvt(c\, p) &= (c^{\#}\ .\!:\ f,\ b) && \text{where } (f, b) = cvt(p) \\
cvt(c) &= (c^{\#},\ ())
\end{aligned}$$

On the one hand, a pattern binder is a subset of standard pattern matching:

1. Only one "case alternative" is allowed.

On the other hand, it extends pattern matching:

1. It allows arbitrary patterns to be applied anywhere inside a pattern using the `%` construct.
2. It makes matching constants explicit: `{=e}` gives matching against not just literal constants like `'c'` and `1` but any Haskell expression of `Eq` type. (This construct is not strictly necessary as we could use the `%` construct to match constants.)
3. It allows us to extend the "de-constructors" by defining $c^{\#}$ functions.
4. It allows arbitrary control structures, with the $\otimes$ pattern-combinator as a parameter.

Although standard pattern matching can also be understood as merely syntactic sugar [Wad87a], this approach is much simpler. The syntax is not as clean as standard pattern matching because everything is explicit. There are a couple ways to make things syntactically nicer (closer to regular patterns):

One, we could allow for dropping the '`=`' in the `{=e}` construct: for literals (integer, character, and string constants) this would be unambiguous; if we had a variable, $\{v\}$, it could be converted to either "$pk\,v$" or "$v^{\#}$" but these are equivalent when v is a nullary constructor.

Second, we could allow for dropping the '`$`' in the $\{\$v\}$ construct: Unfortunately, $\{v\}$ again is ambiguous: if $v^{\#}$ is defined, we do not know whether to use the $cvt(\$v)$ rule or the $cvt(c)$ rule. We could get around this by using the $cvt(c)$ rule whenever $v^{\#}$ is defined. This effectively disallows shadowing of variables. In Haskell, using capitals for constructors is how we distinguish these two. But this is not going to work any more once we start writing "pseudo-constructors" (functions c that have a $c^{\#}$ "de-constructor" defined). So, while we are making hypothetical changes to Haskell, we should drop this restriction. This is a good thing, it enables abstract data types to work better with patterns: it does not matter if a function corresponds to the constructor of an algebraic data type, all that matters is whether a function c has a corresponding $c^{\#}$ function.

4.2 Examples of Pattern Binders

The function `take` from the Haskell Prelude uses a pattern guard "`| n>0`":

```
take 0 _            = []
take _ []           = []
take n (x:xs) | n>0  = x : take (n-1) xs
take _ _            = error "take"
```

Using pattern binders `take` can be written as follows:

```
take =
     {(=0,            $_)}     .-> []
   .| {($_,            [])}     .-> []
   .| {(%(is (0<)) $n, $x:$xs)} .-> x : take (n-1, xs)
   |> \_                         -> error "take"

is p x = if p x then Just x else Nothing
```

The function `zipWith3` can be written straightforwardly:

```
zipWith3 f = {($x:$xs,$y:$ys,$z:$zs)}.-> f x y z : zipWith3 f (xs,ys,zs)
           |> \_                        -> []
```

We can create a "view" of the positive integers using the `succ` and `succ`$^{\#}$ functions. (In this case the pattern law is being broken unless `succ` and `succ`$^{\#}$ are restricted to positive integers.)

```
succ  n = n+1
succ# n = if n > 0 then Just (n-1) else Nothing

factorial =   {succ $n} .-> (succ n * factorial n)
            |> \_         -> 1
```

4.3 Pattern Binders and New Control Structures

Here is an example extracted from an interpreter:

```
doprimitive =
   {Eprim (%getenv $op) : Eint $a1 : Eint $a2 : []} .-> op a1 a2
  |> \_ -> (error "bad primitive")

getenv :: String → Maybe (Int → Int → Int)
getenv x = lookup x [("+",(+)), ("*",(*)), ("-",(-))]
```

Note the type of getenv, it is a pattern-function so we can use it inside pattern binders. To add a more informative error message, the code is changed to[6]

```
doprimitive list =
  {$hd:$a1:$a2:[]} `matchelse list  (error "expect 3 elements")`
  {Eprim $name}    `matchelse hd    (error "head not Eprim")`
  {Eint  $a1'}     `matchelse a1    (error "1st not Eint")`
  {Eint  $a2'}     `matchelse a2    (error "2nd not Eint")`
  {%getenv $op}    `matchelse name  (error (name ++ " invalid prim"))`
  op a1' a2'
  where
  matchelse val dflt pat contin = ((pat.->contin) |> \_-> dflt) val
```

(Note that pattern binders are right associative.) This is a common pattern, but in Haskell one would write nested case expressions marching off to the right of the page. It is nice to put all the exception code off to the right[7]. Here we also see that other functions besides (.->) are useful with pattern binders.

4.4 Changing Evaluation Order

Just like patterns, pattern binders have an implicit left-to-right evaluation mechanism for tuples. I.e., {(p1,p2)} is translated with (.*)—left-to-right evaluation. Pattern binders could be extended to allow for tuples of patterns that would evaluate right-to-left (e.g., {/(p1,p2)}). We could also revert to explicit use of the combinators.

5 Extension II: Backtracking

We could extend the pattern combinators to use lists instead of Maybe. Success will be [x], failure will be [], and multiple successes will be [x1,x2,...]; but it would be even nicer to write the combinators to work with either Maybe or lists. We can do this easily because both Maybe and lists are monads, specifically, each is an instance of MonadPlus:

```
class  (Monad m) => MonadPlus m  where
    mzero  :: m a
    mplus  :: m a → m a → m a

instance  MonadPlus Maybe  where
    mzero              = Nothing
    Nothing `mplus` y  = y
    x       `mplus` y  = x
```

[6] In Haskell, x `f` y = f x y. Haskell does not really allow expressions inside the backquotes, though some think it should.

[7] This is similar to Perl's very useful idiom "($x =~ /pat/) || exception" where the exception code is off to the right.

```
pid       :: Monad m => b -> m b
pk        :: (Eq a, MonadPlus m) => a -> a -> m ()
pfail     :: MonadPlus m => b -> m c
(.|),(|.) :: MonadPlus m => (a->m c) -> (a->m c) -> (a->m c)
(.:)      :: Monad m     => (b -> m c) -> (c -> m d) -> b -> m d
(.*),(*.) :: Monad m     => (b -> m c) -> (d -> m e) -> (b,d) -> m (c,e)
(.->)     :: Functor m   => (b -> m c) -> (c -> d) -> b -> m d

pid x            = return x
pk x y           = if x == y then return () else mzero
pfail _          = mzero
(f .| g) x       = f x `mplus` g x
(|.)             = flip (.|)
(p1 .* p2) (a,b) = do r1 <- p1 a
                      r2 <- p2 b
                      return (r1,r2)
(p1 *. p2) (a,b) = do r2 <- p2 b
                      r1 <- p1 a
                      return (r1,r2)
(p1 .: p2) x     = p1 x >>= p2
f .-> g          = fmap g . f
```

Fig. 3. Monadic Pattern Combinators

```
instance  MonadPlus []  where
    mzero            = []
    mplus            = (++)        -- list append
```

The more general definitions are in Fig. 3. We need a different pattern elimination construct to replace `(|>)`:

```
(||>) :: (a→[b]) → (a→b) → (a→b)
(p ||> d) f x = case p x of {[]->f x; r:_-> r}
```

When we use `(|>)` we get standard pattern matching because the pattern combinators are instantiated to the `Maybe` monad, but when we use `(||>)` we get backtracking because the pattern combinators are instantiated to the list monad. This requires lazy lists for efficiency. The method of using lazy lists for backtracking is explained in Wadler [Wad85] and is also used in combinator parsers [HM96].

6 Extension III: "Value Constructors"

To define the $c^{\#}$ equivalents for various c "pseudo constructors" is often tedious: for example, the `list1`$^{\#}$, `list2`$^{\#}$, `list3`$^{\#}$ from above. If c is an instance of the `Eq` class, $c^{\#}$ can simply be a shortcut for "`pk` c", but for *functions* c, the $c^{\#}$

pattern is often trivial (but tedious) to generate. Is there a way to formalize this and automate the construction of the corresponding pattern function for a given function c? One method is Aitken and Reppy's "Abstract Value Constructors" [AR92]. The language can generate $c^{\#}$ for any c that is a value constructor. A value constructor is a function whose definition is of the following form

$$c\, b = v_c$$

where b is a failure-free pattern, all variables are used linearly, and v_c is defined as follows

$$\begin{array}{lll} v_c = x & & (x \text{ a variable}) \\ \quad \mid (v_c, ..., v_c) & & \\ \quad \mid C\, v_c & & (C \text{ a constructor}) \\ \quad \mid k & & (k \text{ a constant}) \\ \quad \mid c\, v_c & & (c \text{ a value constructor}) \end{array}$$

There are two advantages to using value constructors:

1. It saves the tedium of defining many pattern functions.
2. It gives function/pattern-function pairs that are *guaranteed* to satisfy the pattern law.

7 Pattern Laws

Here are several laws for the monadic pattern combinators. (These are true for any instance of `MonadPlus` that satisfies the monad laws.)

```
                    p .| pfail = p

                      p1 .| p2 = p2 |. p1

 (p1 .-> f1) .* (p2 .-> f2) = (p1 .* p2) .-> \(x,y)->(f1 x,f2 y)

         (p1 .-> f)  .: p2  = p1 .: (p2.f)

           (pid .* p) (x,y) = fmap (\y->(x,y)) (p y)

           (p1 .: p2) .-> f = (p1 .: (p2 .-> f)
```

If this is true for an instance of `MonadPlus` (it's true for `Maybe` and lists)

```
m >>= (\a->p1 a `mplus` p2 a) = (m >>= p1) `mplus` (m >>= p2)
```

then `(.:)` distributes over `(.|)`:

```
p .: (p1 .| p2)  =  p .: p1 .| p .: p2
```

8 Conclusion

8.1 Relation to Other Work

I am not aware of other work that proposes other *alternatives* to the standard pattern matching constructs. However, there is much work related to getting around the infelicities of pattern matching:

The title of Wadler's seminal paper says it all: "Views: A way for pattern matching to cohabit with data abstraction" [Wad87b]. Other work along these lines is [BC93] and [PPN96]. The goal of this line of work is to reconcile pattern matching and abstract data types; the solutions have been to keep pattern matching as is and *add* more to the language. My proposal here could be seen as another way to reconcile the two (if you allow me to reconcile two parties by eliminating one); but the result is that we get the syntactic convenience of patterns even when dealing with abstract data types.

Other work has been aimed at the expressiveness of patterns: Fahndrich and Boyland [FB97] propose a means to add pattern abstractions. My approach is far simpler, although I do not attempt as they do to make patterns statically checkable for exhaustiveness and redundancy. Aitken and Reppy [AR92] have a simple approach to making pattern matching more expressive; I've borrowed their notion of a "value constructor".

Parser combinators are related to this work, though they are only applicable to matching lists. As Hutton and Meijer [HM96] have shown, monads are useful for designing the combinators. Like my patterns, parser combinators could be parameterized over which monad (list or `Maybe`) to use (i.e., whether to do backtracking or not).

Hughes [Hug98] generalizes monads to what he calls arrows. As his arrows are so general, it is not surprising that my combinators can be made into an instance of an arrow, but the interesting thing is that he extensively uses certain combinators which are a generalization of mine: his $***$ is a generalization of my `(.*)` and his $>>>$ is a generalization of my `(.:)`.

8.2 Summary

So, to summarize what I have proposed: if we add

1. the `#` primitive for constructing pattern-functions from constructors
2. the pattern binder syntactic construct

then we can use the power of the language to write all the pattern combinators we want, and as a result we could, without much loss

1. restrict patterns in top-level declarations, lets, and lambdas to be failure-free

2. restrict case alternatives to be exhaustive and mutually exclusive over a single algebraic data type

The advantages of this proposal are

1. The extensions needed to the language are trivial.
2. Complicated pattern matching constructs are unnecessary.
3. Pattern matching no longer results in run-time errors.
4. Patterns become first class: we can write our own patterns; we can write our own control structures for sequencing patterns; and we can write programs generic over constructors (e.g., the `getC1s` and `getC2s` above).
5. Pattern matching can use backtracking, when needed.
6. Patterns work with abstract data types.

Acknowledgements. Thanks to Valery Trifonov for pointing out the similarities between my pattern combinators and the arrow combinators of John Hughes and thanks to the anonymous referees for many helpful comments.

References

[AR92] William Aitken and John H. Reppy. Abstract value constructors: Symbolic constants for standard ML. Technical Report CORNELLCS//TR92-1290, Cornell University, Computer Science Department, June 1992.

[BC93] F. W. Burton and R. D. Cameron. Pattern matching with abstract data types. *Journal of Functional Programming*, 3(2):171–190, 1993.

[FB97] Manuel Fähndrich and John Boyland. Statically checkable pattern abstractions. In *Proceedings of the 1997 ACM SIGPLAN International Conference on Functional Programming*, pages 75–84, Amsterdam, The Netherlands, 9–11 June 1997.

[HJW92] P. Hudak, S. P. Jones, and P. Wadler. Report on the programming language Haskell. *SIGPLAN Notices*, 27(5), May 1992.

[HM96] Graham Hutton and Erik Meijer. Monadic parser combinators. Technical Report NOTTCS-TR-96-4, University of Nottingham, December 1996.

[Hug98] John Hughes. Generalizing monads to arrows. Submitted for publication, 1998.

[PPN96] Palao Gostanza Pedro, Ricardo Peña, and Manuel Núñez. A new look at pattern matching in abstract data types. In *Proceedings of the ACM SIGPLAN International Conference on Functional Programming (ICFP '96)*, volume 31(6) of *ACM SIGPLAN Notices*, pages 110–121. ACM, June 1996.

[Wad85] P. L. Wadler. How to replace failure by a list of successes. In Jean-Pierre Jouannaud, editor, *Functional Programming Languages and Computer Architecture*, volume 201 of *Lecture Notes in Computer Science*, pages 113–128. Springer Verlag, September 1985.

[Wad87a] Philip Wadler. Efficient compilation of pattern-matching. In S. L. Peyton Jones, editor, *The Implementation of Functional Programming Languages*, chapter 5. Prentice-Hall International, 1987.

[Wad87b] Philip Wadler. Views: A way for pattern-matching to cohabit with data abstraction. In *Conference Record of the Fourteenth Annual ACM Symposium on Principles of Programming Languages*, pages 307–313. ACM, January 1987.