Under consideration for publication in J. Functional Programming

FUNCTIONAL PEARL

Type-safe pattern combinators

Morten Rhiger∗
The IT University of Copenhagen, Copenhagen, Denmark
(*e-mail:* mir@itu.dk)

Abstract

Macros still have not made their way into typed higher-order programming languages such as Haskell and Standard ML. Therefore, to extend the expressiveness of Haskell or Standard ML gradually, one must express new linguistic features in terms of functions that fit within the static type systems of these languages. This is particularly challenging when introducing features that span across multiple types and that bind variables. We address this challenge by developing, in a step-by-step manner, mechanisms for encoding patterns and pattern matching in Haskell in a type-safe way.

1 Untyped patterns

Let us tacitly ignore the existence of Haskell's pattern-matching facility and let us attempt to extend Haskell with our own library of patterns and with our own implementation of pattern matching:

```
data Pat = Pcst Int | Ppair Pat Pat | Pvar String
data Val = Vcst Int | Vpair Val Val

match                    :: Pat → Val → [(String, Val)]
match (Pvar x)     v     = [(x, v)]
match (Pcst i)     (Vcst j)     = if i ≡ j then [] else error "match"
match (Ppair p q) (Vpair v w) = match p v ++ match q w
match _            _     = error "type error"
```

This solution defines data types of patterns (consisting of integer constants, pairs, and pattern variables) and of values (consisting of integer constants and pairs), and a function that matches a pattern against a value and produces a list of bindings on success. The solution is *first order* and it is *untyped*: All type checks involving patterns are performed at runtime, by inspecting patterns and values. For example, while an application of Haskell's built-in pattern-matching facility, such as

```
case ((1,2),(3,4)) of
  (5,x) → 2+x
```

∗ At the time of writing, the author is on leave from Roskilde University, Roskilde, Denmark.

results in compile-time type errors (in this particular case since the types of the pattern 5 and the value $(1,2)$ mismatch and since the variable x is used as an integer but bound to a pair), the "type errors" in its encoded counterpart

```
case (match (Ppair (Pcst 5) (Pvar "x"))
            (Vpair (Vpair (Vcst 1) (Vcst 2))
                   (Vpair (Vcst 3) (Vcst 4)))) of
  [("x", Vcst v)] → 2 + v
  _               → error "type error"
```

pass undetected through Haskell's type checker.

The goal of this Functional Pearl is to implement a library of type safe patterns that can be extended with new and (hopefully) interesting patterns. Unlike the first-order solution above, our implementation does not require values to be encoded in terms of a data type and it does not demand the use of Haskell's built-in pattern matching to decode the values of bound variables.

The means are functional programming techniques: We demonstrate that polymorphic higher-order functions alone enable us to develop the mechanisms required to encode type-safe patterns in Haskell and Standard ML in a way that goes beyond their built-in pattern-matching facilities.

Example 1. Using the operations *match*, *pair*, *cst*, *var*, and $\rightarrow\!\!\!\rightarrow$ from the library implemented in Section 4 of this Functional Pearl, the example presented above is encoded as follows.

```
match ((1,2),(3,4)) $
  pair (cst 5) var ↠ λx → 2 + x
```

As required, this expression does not type-check in Haskell.

Example 2. Again using the library from Section 4, the following example does type check, and yields 7.

```
match (5,(3,4)) $
  pair (cst 5) (pair var var) ↠ λx y → x + y
```

2 Patterns exposing their type

We first introduce machinery allowing us to expose to Haskell's type system the types of patterns and the structure of bindings produced on successful matches.

2.1 Exposing types

We want the *value* of the first argument (the pattern) to *match* to determine the *type* of the second argument (the value). For example, if the first argument is *Ppair* (*Pint* 1) (*Pint* 2) then the second argument should be of type (Int, Int).

We expose this relationship by representing patterns of type α as functions whose argument has type α, as follows. (The intricacies of pattern variables have deliberately been postponed to the discussion below.)

$$
\begin{array}{ll}
\mathit{var}\ x & = \lambda v \to \ldots \\
\mathit{cst}\ v' & = \lambda v \to \textbf{if}\ v \equiv v'\ \textbf{then}\ [\,]\ \textbf{else}\ \mathit{error}\ \texttt{"match"} \\
\mathit{pair}\ p\ q & = \lambda v \to p\ (\mathit{fst}\ v) +\!\!+ q\ (\mathit{snd}\ v) \\
\mathit{match}\ p\ v & = p\ v
\end{array}
$$

This implementation is *typed*: All type errors that result from a mismatch between the pattern and the value are detected at compile time. Furthermore, run-time values need no longer be encoded and decoded using a data type. An additional advantage is that the combinator *cst*, unlike its first-order counterpart *Pcst*, is not restricted to integers but may be applied to values of any type that exhibits equality. As an example,

$$\mathit{match}\ (\mathit{pair}\ (\mathit{cst}\ 1)\ (\mathit{cst}\ \mathit{True}))$$

is a function that expects an argument of type $(\mathit{Int}, \mathit{Bool})$ and that matches this argument against the pattern $(1, \mathit{True})$.

The technique applied here has been used by Danvy (1998) to implement *printf* in Standard ML, by Fridlender and Indrika (2000) to implement a generic *zipWith* in Haskell, and by Filinski (1999), Yang (2004), and the present author (Rhiger, 1999) to implement type-directed partial evaluation in Standard ML. It is described in detail by Yang (2004).

2.2 Exposing bindings

In the first-order untyped implementation of patterns, the right-hand side of a clause would be represented by a function accepting a list of bindings and fetching the values of bound variables from this list as follows.

$$\lambda \mathit{env} \to \ldots \mathit{lookup}\ \texttt{"x"}\ \mathit{env} \ldots$$

We must avoid such representations of pattern variables as strings since they hinder reasoning about the number and the types of pattern variables within Haskell's type system.

For the time being, we will be content with fetching values of bound variables from a structure that is isomorphic to the pattern in question, rather than from a list-like structure. For this purpose, we introduce a pattern *var* that marks a place in the pattern where a variable occur, without requiring us to name that variable. This idea results in a notion of *nameless* patterns, implemented as follows.

$$
\begin{array}{ll}
\mathit{var} & = \lambda v \to v \\
\mathit{cst}\ v' & = \lambda v \to \textbf{if}\ v \equiv v'\ \textbf{then}\ ()\ \textbf{else}\ \mathit{error}\ \texttt{"match"} \\
\mathit{pair}\ p\ q & = \lambda v \to (p\ (\mathit{fst}\ v), q\ (\mathit{snd}\ v)) \\
\mathit{match}\ p\ v & = p\ v
\end{array}
$$

Again, this implementation is typed: All type errors that result from mismatches between the definition and the use of variables are detected at compile time. For example, we have the following typings. (The left-hand side of the function arrow is the type of the value to match against and the right-hand side represents the bindings produced on a successful match.)

$$\mathit{match}\ (\mathit{pair}\ \mathit{var}\ (\mathit{pair}\ (\mathit{cst}\ 2)\ \mathit{var})) :: (\alpha, (\mathit{Int}, \beta)) \to (\alpha, ((), \beta))$$

Therefore, the Haskell expression

case $(1,(2,3))$ **of** $(x,(2,y))) \to x+y$

is represented by

case *match* (*pair var* (*pair* (*cst* 2) *var*)) $(1,(2,3))$ **of**
$(x,((),y)) \to x+y$

And indeed, we're not required to encode values in a data type and there's no dynamic lookup of pattern variables.

However, as satisfying as this may seem, we're not quite done yet: The implementation still requires us to fetch the values of pattern variables inside a structure that resembles the original value that the pattern was matched against. Therefore, we must encode the structure of a pattern twice, first using the pattern combinators and then using Haskell's built-in patterns.

3 Patterns producing flat bindings

To improve the implementation presented above, we employ a *flattened* representation of bindings as *heterogeneous sequences.* The structure of such bindings is independent of the structure of the corresponding pattern, just as in the first-order untyped implementation of patterns. But unlike the first-order implementation, the heterogeneous sequences correctly expose their length and the types of their elements within Haskell's type system.

Assuming the existence of operations to build empty sequences (*nil*), to build singleton sequences (*one v*), and to append two sequences (m # n), the implementation of patterns is defined as follows.

```
var       = λv → one v                                  -- bind v
cst v'    = λv → if v ≡ v' then nil else error "match"  -- bind nothing or fail
pair p q  = λv → p (fst v) # q (snd v)                  -- append bindings
```

We present two implementations of heterogeneous sequences below, one that results in a *direct-style program* that uses an accumulator and one that results in a style of programming resembling *continuation-passing style*.

3.1 Direct-style, uncurried

We first represent sequences as nested pairs terminated by an empty tuple. Such a flattened representation encodes traditional lists by replacing $[\,]$ by $()$ and $(x:xs)$ by (x,xs). It allows elements of different types in one sequence.

More precisely, a sequence is represented as a function that accepts a tail and that "conses" (using a pair) its elements onto that tail, as follows.

```
nil     = λac → ac          -- cons no element
one v   = λac → (v, ac)     -- cons v
m # n   = λac → m (n ac)    -- let n cons first, then m
```

It is straightforward to verify that *nil* is both a left and a right unit of # and that # is associative. In other words, # and *nil* form a *monoid* over heterogeneous sequences.

Example 3. The following expression constructs a sequence containing an integer, a boolean, and a string.

one 5 # *nil* # *one True* # *one* `"abc"` # *nil*

The type of this expression, $\alpha \to (Int, (Bool, (String, \alpha)))$, indeed reflects the number and the types of the elements of the corresponding sequence, as required. Passing this expression the empty tuple $()$ yields $(5, (True, (\texttt{"abc"}, ())))$.

When encoding heterogeneous sequences in direct-style like this, the well-typed Haskell expression

case $((1,2),3)$ **of** $((x,2),y) \to x+y$

can be represented by the well-typed

case $(match\ (pair\ (pair\ var\ (cst\ 2))\ var)\ ((1,2),3))$ **of**
$(x,(y,())) \to x+y$

3.2 Continuation-passing style, curried

Even though the structure of bindings is now independent of the structure of the corresponding pattern, we still rely on Haskell's built-in pattern matching to fetch the values of bound pattern variables. (Alternatively, we may use the functions *fst* and *snd* to fetch the values from the sequence, but such a solution would be cumbersome.)

To overcome this problem and to make bindings digestible, we *curry* the functions that process them. To enable such a change, we adopt a representation of sequences as functions that pass all the elements of a sequence to a given curried continuation. The operations on sequences are defined as follows.

```
nil     = λk → k                  -- pass no values to k
one v   = λk → k v                -- pass v to k
m # n   = λk → n (m k)            -- let m pass values first, then n
```

It is again straightforward to verify that # and *nil* form a monoid over heterogeneous sequences. Notice that according to this implementation, any non-functional value is a "continuation" of zero arguments: In general, the representation of a sequence of length n passes its elements to a function of n arguments. When $n = 0$, such a "function" can be a value of any type. As we shall see in the next section, this generalization allows us to represent a right-hand side of a pattern-matching clause as a value of base type when the corresponding pattern is ground (i.e., one that does not contain variables).

Example 4. The following expression constructs a sequence containing an integer, a boolean, and a string.

one 5 # *nil* # *one True* # *one* `"abc"` # *nil*

The type of this expression, $(Int \to Bool \to String \to \alpha) \to \alpha$, again reflects the number and the types of the elements of the corresponding sequence. Passing this expression the continuation $\lambda x\, y\, z \to (x,y,z)$ yields $(5, True, \texttt{"abc"})$.

When encoding heterogeneous sequences in continuation-passing style like this, the well-typed Haskell expression

case $((1,2),3)$ **of** $((x,2),y) \to x+y$

can be represented by the well-typed

$$\begin{array}{l} \mathit{match}\ (\mathit{pair}\ (\mathit{pair}\ \mathit{var}\ (\mathit{cst}\ 2))\ \mathit{var}) \\ \qquad ((1,2),3) \\ \qquad (\lambda x\ y \to x+y) \end{array}$$

Thus, we have replaced the need for Haskell's built-in pattern matching to fetch the values of pattern variables by the application of a curried function.

Danvy's statically typed implementation of *printf* in Standard ML (Danvy, 1998) use curried continuations in a similar fashion. But where Danvy's formatting combinators *consume* a heterogeneous sequence of values, our pattern-matching combinators *produce* such a heterogeneous sequence of values.

4 A library of typed patterns

With the basic mechanisms at hand, let us design a library that takes us beyond the capabilities of Haskell's built-in pattern matching. To this end, we extend the language of patterns presented so far with *disjunctions* and *conjunctions* (also known as *or*-patterns and *and*-patterns), *predicates*, and a *set-like view* on lists.

To support disjunctions, we add a failure continuation to the encoding of sequences. However, this addition rules out the implementation above where currying happens *on the fly.* (In a curried implementation, the failure continuation is expected to consume bindings. This contradicts the intuition that the failure continuation is applied only when a pattern does not match, and hence when no bindings are available.)

To address this issue, we employ an uncurried encoding of bindings as nested pairs during the construction of patterns, as in the direct-style implementation above. We then uncurry a given curried (success) continuation before passing it to a pattern. To this end, we use the following typed and generic implementation of uncurrying of functions.

$$\begin{array}{ll} \mathit{zero}\ f & = \lambda() \quad \to f \\ \mathit{succ}\ n\ f & = \lambda(x,xs) \to n\ (f\ x)\ xs \\ \mathit{uncurry}\ n\ f & = n\ f \end{array}$$

The functions *zero* and *succ* define a representation of numerals with the property that an n-ary curried function can be uncurried by the representation of n.[1] Fridlender and Indrika also use a higher-order representation of numerals in their generic and well-typed implementation of an n-ary *zipWith* (Fridlender & Indrika, 2000).

Example 5. To uncurry a function of two arguments, we apply the following representation of the numeral 2 to the function.

$$\mathit{succ}\ (\mathit{succ}\ \mathit{zero}) :: (\alpha \to \beta \to \gamma) \to (\alpha,(\beta,())) \to \gamma$$

[1] Our use of the term "uncurried" is unconventional: Rather than taking n-tuples as arguments, our uncurried functions take nested pairs of depth n as argument.

4.1 Basic patterns

We represent heterogeneous sequences in continuation-passing style using two continuations (a success continuation k_s and a failure continuation k_f) and an accumulator. We also add two constructions, *fail* and *catch*, that introduce an exception mechanism into the language of sequences:

$$\begin{aligned}
&\mathit{nil} &&= \lambda k_s\ k_f\ ac \to k_s\ ac\\
&\mathit{one}\ v &&= \lambda k_s\ k_f\ ac \to k_s\ (v, ac)\\
&m\ \#\ n &&= \lambda k_s\ k_f\ ac \to n\ (m\ k_s\ k_f)\ k_f\ ac\\
&\mathit{fail} &&= \lambda k_s\ k_f\ ac \to k_f\ ()\\
&m\ \text{`}\mathit{catch}\text{`}\ n &&= \lambda k_s\ k_f\ ac \to m\ k_s\ (\lambda() \to n\ k_s\ k_f\ ac)\ ac
\end{aligned}$$

Both (# , *nil*) and (*catch*, *fail*) form monoids over heterogeneous sequences.

A pattern consists of a pair of functions. The first increments a given numeral (in the representation presented above) by the number of variables of the pattern. Eventually, this component is used to curry success continuations. The second component implements the matching itself. The basic patterns are defined as follows.

$$\begin{aligned}
&\mathit{var} &&= (\mathit{succ}, &&\lambda v \to \mathit{one}\ v)\\
&\mathit{cst}\ v' &&= (\mathit{id}, &&\lambda v \to \textbf{if}\ v \equiv v'\ \textbf{then}\ \mathit{nil}\ \textbf{else}\ \mathit{fail})\\
&\mathit{pair}\ p\ q &&= (\mathit{curry}_{pq}, &&\lambda v \to \mathit{match}_p\ (\mathit{fst}\ v)\ \#\ \mathit{match}_q\ (\mathit{snd}\ v))\\
&&&\quad \textbf{where}\ (\mathit{curry}_p, \mathit{match}_p) = p\\
&&&\qquad\ \ (\mathit{curry}_q, \mathit{match}_q) = q\\
&&&\qquad\ \ \mathit{curry}_{pq} = \mathit{curry}_p\ \cdot\ \mathit{curry}_q
\end{aligned}$$

4.2 Matching

Before we continue, let us introduce a more realistic main function. To this end, we introduce *clauses* (pairs of patterns and right-hand sides) using the infix operator $\rightarrow\!\!\!\!\rightarrow$ and we separate two such clauses by ||. It is $\rightarrow\!\!\!\!\rightarrow$ that curries continuations properly. These operations are defined as follows.

$$\begin{aligned}
&p \rightarrow\!\!\!\!\rightarrow k = \lambda v\ k_f \to \mathit{match}_p\ v\ (\mathit{curry}_p\ \mathit{zero}\ k)\ k_f\ ()\\
&\qquad\qquad \textbf{where}\ (\mathit{curry}_p, \mathit{match}_p) = p\\
&c_1\ ||\ c_2 = \lambda v\ k_f \to c_1\ v\ (\lambda() \to c_2\ v\ k_f)
\end{aligned}$$

The following main operation then matches a given value against a group of clauses.[2]

$$\mathit{match}\ v\ cs = cs\ v\ (\lambda() \to \mathit{error}\ \texttt{"match"})$$

Example 6. With the correct precedences, these definitions allow us to write expressions such as the following. ($ is an infix apply operator of low precedence commonly used as an alternative to parenthesizing a function's argument. Here we can read it as "with.")

[2] To give it a stronger resemblance with Haskells **case** − **of**− construct, this final version of *match* takes the value to match against as its first argument.

```
match n $
      cst 0 ⟶⟶ "zero"
    || cst 1 ⟶⟶ "one"
    || var   ⟶⟶ λi → error ("not a binary digit: " ++ show i)
```

Since patterns are first-class values, *pattern abstraction* (Fähndrich & Boyland, 1997) is at our hands to extend the library of patterns. As a fist example, we can introduce anonymous pattern-matching functions as follows, using the infix operator $\mapsto$.

$$p \mapsto k = \lambda v \to match\ v\ \$\ p \longrightarrow\!\!\!\!\rightarrow k$$

Example 7. The following expression sums each pair of numbers in a list.

$$map\ (pair\ var\ var \mapsto \lambda x\ y \to x + y) :: [(Int, Int)] \to [Int]$$

4.3 Disjunctions

The disjunctive aspect is represented by the pattern combinators *none* and $\vee$, defined as follows. The pattern *none* does not match any value.

$$\begin{aligned} none &= (id, \quad \lambda v \to fail) \\ p \vee q &= (curry_{pq}, \lambda v \to match_p\ v\ \text{`}catch\text{`}\ match_q\ v) \\ &\quad \textbf{where}\ (curry_p, match_p) = p \\ &\qquad\quad (curry_q, match_q) = q \\ &\qquad curry_{pq} = \textbf{if}\ True\ \textbf{then}\ curry_p\ \textbf{else}\ curry_q \end{aligned}$$

The combinators $\vee$ and *none* form a monoid over patterns. The conditional in the implementation of disjunction serves to unify the types of the currying component associated with the two patterns. As a result, the two branches are required to produce bindings of equal type.

Example 8. The following pattern matches pairs whose first *or* second component is 1. For successful matches, it binds a variable to the "other" component.

$$pair\ (cst\ 1)\ var \vee pair\ var\ (cst\ 1)$$

4.4 Conjunctions

The conjunctive aspect is represented by the pattern combinators *any* and $\wedge$, defined as follows. The pattern *any* matches any value. It corresponds to the wildcard _ in Haskell.

Conjunctions are implemented as follows.

$$\begin{aligned} any &= (id, \quad \lambda v \to nil) \\ p \wedge q &= (curry_p \cdot curry_q, \lambda v \to match_p\ v\ \#\ match_q\ v) \\ &\quad \textbf{where}\ (curry_p, match_p) = p \\ &\qquad\quad (curry_q, match_q) = q \end{aligned}$$

These combinators also form a monoid over patterns. Conjunction generalizes layered patterns: A pattern like $var \wedge p$ matches any value that p matches and binds the value to a variable.

The infix operator ? matches a ground pattern against a value and yields a boolean value representing the status of the match.

$$p\ ?\ v = match\ v\ \$\ p \rightarrow\!\!\!\rightarrow True\ ||\ any \rightarrow\!\!\!\rightarrow False$$

Example 9. The following expression removes from a list all pairs whose first component is nonzero.

$$filter\ (pair\ (cst\ 0)\ any\ ?) :: [(Int, \alpha)] \rightarrow [(Int, \alpha)]$$

4.5 Predicates

It is sometimes useful to employ a predicate during matching. The following operation yields a pattern that matches only the values for which a given predicate p returns *True*.

$$is\ p = (id, \lambda v \rightarrow \textbf{if}\ p\ v\ \textbf{then}\ nil\ \textbf{else}\ fail)$$

Example 10. The following function implements integer exponentiation using the Russian-peasant algorithm.

```
power x n =
  match n $
        cst 0    -->> 1
    || is even -->> square (power x (n `div` 2))
    || is odd  -->> x * power x (n - 1)
  where square x = x * x
```

Example 11. The following expression removes from a list all those pairs whose first component is odd.

$$filter\ (pair\ (is\ even)\ any\ ?) :: [(Int, \alpha)] \rightarrow [(Int, \alpha)]$$

It is tempting to write

$$filter\ (is\ even\ ?) :: [Int] \rightarrow [Int]$$

but one can prove that the "left section"

$$(is\ p\ ?)$$

(which is a partial application of the infix operator ? to its first argument $is\ p$) is equivalent to p, for any function p of type $\alpha \rightarrow Bool$.

4.6 Sets

It may also be useful to view a list as if it was a set of values. The pattern *has p* scans a list for the first element matched by the pattern p. If no element matches, then the entire pattern fails:

$$\begin{array}{l} \mathit{has}\ p = (\mathit{curry}_p, \mathit{foldr}\ \mathit{catch}\ \mathit{fail}\ \cdot\ \mathit{map}\ \mathit{match}_p) \\ \qquad \textbf{where}\ (\mathit{curry}_p, \mathit{match}_p) = p \end{array}$$

Example 12. The function *fetch*, of type $\mathit{String} \to [(\mathit{String}, \alpha)] \to \alpha$, implements a lookup function:

$$\mathit{fetch}\ v = \mathit{has}\ (\mathit{pair}\ (\mathit{cst}\ v)\ \mathit{var}) \mapsto \lambda x \to x$$

Example 13. The function *first*, of type $(\alpha \to \mathit{Bool}) \to [\alpha] \to \alpha$, implements a function that takes a list and returns the first element of the list that satisfies the predicate f:

$$\mathit{first}\ f = \mathit{has}\ (\mathit{var} \wedge \mathit{is}\ f) \mapsto \lambda x \to x$$

We can take example 12 a step further by implementing lookup as a pattern rather than as a function. The resulting pattern binds a variable:

$$\mathit{get}\ v = \mathit{has}\ (\mathit{pair}\ (\mathit{cst}\ v)\ \mathit{var})$$

Example 14. We can not only implement *fetch* using the pattern combinator *get*, but also more complex matches. The code snippet below produces the value associated with either `"a"` or `"A"` in the list of pairs *env*. It complains if either both or none of the bindings exist (but not if two occurrences of `"a"` exists in the list):

$$\begin{array}{l} \mathit{match}\ \mathit{env}\ \$ \\ \quad \mathit{get}\ \texttt{"a"} \wedge \mathit{get}\ \texttt{"A"} \longrightarrow (\lambda \_\ \_ \to \mathit{error}\ \texttt{"ambiguity"}) \\ \quad \mid\mid \mathit{get}\ \texttt{"a"} \vee \mathit{get}\ \texttt{"A"} \longrightarrow (\lambda a \quad \to a) \\ \quad \mid\mid \mathit{any} \qquad\qquad\quad \longrightarrow \mathit{error}\ \texttt{"unbound"} \end{array}$$

4.7 Data abstraction

Since pattern combinators are first-class values, an abstract data type can be associated with a set of patterns without exposing its implementation. In the sense of Wadler (1987), we can define patterns that provide a concrete *view* on an abstract data type. For example, the function below uses pattern matching to define f^n, the n-times composition of a function f with itself. However, rather than fixing the representation of the integer n, it allows any representation that can be viewed as natural numbers built using a zero and a successor. This is accomplished by parameterizing the implementation over two patterns combinators, *zero* and *succ*:

$$\begin{array}{l} \mathit{makeCompose}\ \mathit{zero}\ \mathit{succ} = \mathit{iterate} \\ \quad \textbf{where}\ \mathit{iterate}\ n\ f\ x = \\ \qquad\qquad \mathit{match}\ n\ \$ \\ \qquad\qquad\quad \mathit{zero} \qquad \longrightarrow x \\ \qquad\qquad \mid\mid \mathit{succ}\ \mathit{var} \longrightarrow \lambda n' \to \mathit{iterate}\ n'\ f\ (f\ x) \end{array}$$

As witnessed by the following definitions of actual pattern combinators, we can view both built-in integers and lists as natural numbers.

$$
\begin{array}{lll}
\mathit{zero}_{\mathit{int}} & = (\mathit{id}, & \lambda v \to \textbf{if } v \equiv 0 \textbf{ then } \mathit{nil} \textbf{ else } \mathit{fail}) \\
\mathit{succ}_{\mathit{int}}\ p & = (\mathit{curry}_p, \lambda v \to \textbf{if } v \equiv 0 \textbf{ then } \mathit{fail} \textbf{ else } \mathit{match}_p\ (v-1)) & \\
 & \quad \textbf{where } (\mathit{curry}_p, \mathit{match}_p) = p & \\
\mathit{zero}_{\mathit{list}} & = (\mathit{id}, & \lambda v \to \textbf{if } \mathit{null}\ v \textbf{ then } \mathit{nil} \textbf{ else } \mathit{fail}) \\
\mathit{succ}_{\mathit{list}}\ p & = (\mathit{curry}_p, \lambda v \to \textbf{if } \mathit{null}\ v \textbf{ then } \mathit{fail} \textbf{ else } \mathit{match}_p\ (\mathit{tail}\ v)) & \\
 & \quad \textbf{where } (\mathit{curry}_p, \mathit{match}_p) = p &
\end{array}
$$

We instantiate the general function as follows.

$$
\begin{array}{l}
\mathit{compose\_int} = \mathit{makeCompose}\ \mathit{zero}_{\mathit{int}}\ \mathit{succ}_{\mathit{int}} \\
\mathit{compose\_list} = \mathit{makeCompose}\ \mathit{zero}_{\mathit{list}}\ \mathit{succ}_{\mathit{list}}
\end{array}
$$

Using these specialized functions, both of the following expressions yield 8.

$$
\begin{array}{l}
\mathit{compose\_int}\ 3\ (\times 2)\ 1 \\
\mathit{compose\_list}\ [10,20,30]\ (\times 2)\ 1
\end{array}
$$

4.8 On type safety

The type safety of the pattern combinators thus implemented follows by construction: There are no dynamic type checks in their implementation and they are well typed according to Haskell's type checker.

4.9 On efficiency

Matching is linear in the size of patterns that do not involve the combinator *has*. (Matching the pattern *has p* against a list amounts to matching p against each element in the list.) Experiments performed using Hugs suggest that matching of pattern combinators require 2–4 times as many reductions as matching using Haskell's built-in patterns. This overhead is primarily induced by managing continuations and by currying. Standard partial-evaluation techniques (β-reduction and η-expansion) can completely compile away both the manipulation of continuations and currying, hence eliminating the overhead.

5 Conclusion

New languages are often constructed by piling new features on top of an existing language's definition and by integrating these features in the existing language's implementation. However, it is a sign of expressiveness if new features can be implemented *within* an existing language without changing its definition.

Short of macros, functional languages such as Haskell and Standard ML require new features to be expressed in terms of typed higher-order functions. We have demonstrated how to extend — or, in Guy Steele's terminology, to "grow" (Steele Jr., 1999) — Haskell with our own statically typed implementation of pattern matching and we have shown how to extend this framework with patterns not currently supported by Haskell.

References

Danvy, Olivier. (1998). Functional unparsing. *Journal of functional programming*, **8**(6), 621–625.

Fähndrich, Manuel, & Boyland, John. (1997). Statically checkable pattern abstractions. *Pages 75–84 of:* Tofte, Mads (ed), *Proceedings of the 1997 ACM SIGPLAN international conference on functional programming (ICFP 1997)*. Amsterdam, The Netherlands: ACM Press.

Filinski, Andrzej. (1999). A semantic account of type-directed partial evaluation. *Pages 378–395 of:* Nadathur, Gopalan (ed), *International conference on principles and practice of declarative programming (PPDP 1999)*. Lecture Notes in Computer Science, no. 1702. Paris, France: Springer-Verlag.

Fridlender, Daniel, & Indrika, Mia. (2000). Do we need dependent types? *Journal of functional programming*, **10**(4), 409–415.

Rhiger, Morten. (1999). Deriving a statically typed type-directed partial evaluator. *Pages 25–29 of:* Danvy, Olivier (ed), *Proceedings of the ACM SIGPLAN workshop on partial evaluation and semantics-based program manipulation (PEPM 1999)*. BRICS Note Series, no. NS–99–1.

Steele Jr., Guy L. (1999). Growing a language. *Higher-order and symbolic computation*, **12**(3), 221–236.

Wadler, Philip. (1987). Views: a way for pattern matching to cohabit with data abstraction. *Pages 307–313 of:* O'Donnell, Michael J. (ed), *Proceedings of the fourteenth annual ACM symposium on principles of programming languages (POPL 1987)*. Munich, West Germany: ACM Press.

Yang, Zhe. (2004). Encoding types in ML-like languages. *Theoretical computer science*, **315**(1), 151–190. A preliminary version was presented at the 1998 ACM SIGPLAN International Conference on Functional Programming (ICFP 1998).