EXTENDED PATTERN MATCHING FOR EMBEDDED LANGUAGES

Gergely Dévai (Budapest, Hungary)

Communicated by Zoltán Horváth

(Received December 21, 2011; revised January 25, 2012; accepted February 14, 2012)

Abstract. Users of embedded languages might want to pattern match on embedded programs. Making this possible requires a considerable effort from the developer of the language, because the underlying data types are usually hidden.
This paper first analyses the available solutions for this problem. As *pattern synonyms* [13] and *function patterns* [6] seem promising, a compromise between these two is proposed: *restricted function patterns*. These are more general than pattern synonyms, but it is still possible to process them at compilation time. It is interesting that this proposal makes Haskell's rules about matching numeric literals more regular. It also provides Erlang's list prefix patterns in a consistent way instead of ad hoc implementations.
Finally, a lightweight prototype implementation is presented, which implements the functionality of the proposal, but cannot give the static guarantees that proper compiler support could achieve.

1. Problem definition

It is usually desirable to hide the implementation details of libraries and provide abstract interfaces for the users. On the other hand, this prevents

Key words and phrases: Restricted function patterns, pattern matching, EDSLs.
1998 CR Categories and Descriptors: D.3.3.
A preliminary, draft version of this paper was included in the Draft Proceedings of Trends in Functional Programming 2011.
The Research is supported by the European Union and co-financed by the European Social Fund (grant agreement no. TÁMOP 4.2.1./B-09/1/KMR-2010-0003).

the user from performing pattern matching, a convenient feature of functional languages. There are numerous proposals to tackle this old problem. This paper will definitely not try to solve this issue in general, but will inspect it from the point of view of embedded languages and explore the possibility of using a special class of expressions in patterns.

Let us start with a toy embedding which in turn is a suitable model to study problems to be solved when creating embedded languages.

```
data Expr = Symbol String | Expr :$ Expr
```

Function symbols and values are represented by their names via the `Symbol` constructor, while application (`:$`) can be used to build compound expressions by applying a function expression to an argument. Using this simple type, one can already define basic arithmetic operations and start manipulating arithmetic expressions. The most convenient way to do this in Haskell is the instantiation of the `Num` class.

```
instance Num Expr where
    fromInteger n = Symbol $ show n
    a + b = Symbol "+" :$ a :$ b
    a * b = Symbol "*" :$ a :$ b
    ...
```

Using the terminology of language embedding, the `Expr` type is used to build the abstract syntax tree of embedded programs, while the `Num` instance serves as the user interface and defines a piece of the syntax. It seems like a good idea to hide the internal representation, i.e. the constructors of the `Expr` type behind a module boundary and only expose the functions in the `Num` class to the user.

Let us suppose that one would like to use this language to optimize expressions based on the arithmetic laws $0 + a = a$, $1 * a = a$ and $0 * a = 0$. The desirable implementation of this would use pattern matching of the following form:

```
optimize :: Expr -> Expr
optimize (0 + a) = a
optimize (1 * a) = a
optimize (0 * a) = 0
optimize a       = a
```

However, this is invalid as the left-hand sides of these equations are not patterns. In fact, without further support from the library, it is impossible to implement this transformation, because the interface provides functions only for constructing, but not for deconstructing expressions.

The next section briefly summarizes the currently available techniques. It concludes that pattern synonyms and function patterns are quite close to what we want to achieve here. Based on these, section 3 defines restricted function patterns and section 4 highlights some of their advantages. Section 5 addresses the difficulties related to the proposal. Finally, the last two sections present a library to test the functionality of the extended pattern matching and conclude the paper.

2. Available solutions

2.1. Selectors

One way to make deconstruction of expressions possible while hiding its constructors is to provide the user with a set of functions to examine expressions and ask for their parts. These functions fall into two categories: some of them provide information on the form of the expression and thus can be used to separate cases, other functions return components of an expression of a given form.

```
is_add :: Expr -> Bool
is_add (Symbol "+" :$ _ :$ _) = True
is_add _ = False

is_mul :: Expr -> Bool
is_mul (Symbol "*" :$ _ :$ _) = True
is_mul _ = False

arg1 :: Expr -> Expr
arg1 (_ :$ a :$ _) = a

arg2 :: Expr -> Expr
arg2 (_ :$ _ :$ a) = a
```

The functions `is_add` and `is_mul` yield true if the expression in question is a compound one with topmost operation addition or multiplication respectively. Unpacking the arguments of such expressions is done by selectors like `arg1` and `arg2`. Using these functions one can implement the desired optimization transformation:

```
optimize :: Expr -> Expr
optimize e
    | is_add e && arg1 e == 0  = arg2 e
    | is_mul e && arg1 e == 1  = arg2 e
    | is_mul e && arg1 e == 0  = 0
    | otherwise                = e
```

This approach is problematic for the following reasons:

- A considerable amount of utility functions must be provided to make deconstruction of expressions convenient.
- Partial functions like `arg1` and `arg2` in our example makes the library dangerous.
- Implementation of the optimization transformation is far from the clarity of the desirable solution envisioned at the end of section 1.

2.2. Data type for matching

This solution enables pattern matching by adding a data type to the interface of the library. A function to convert expressions to this additional type is also provided.

```
data Arith
    = Expr :+: Expr
    | Expr :-: Expr
    | Expr :*: Expr
    | Other

arith :: Expr -> Arith
arith (Symbol "+" :$ a :$ b) = a :+: b
arith (Symbol "-" :$ a :$ b) = a :-: b
arith (Symbol "*" :$ a :$ b) = a :*: b
arith _ = Other
```

One can first use the `arith` function to convert an expression to an arithmetic expression. This transformation may fail, for example if the topmost operation of the expression is not one of the selected arithmetic functions. In this case the result is `Other`. Pattern matching can distinguish between failure and success as well as examine the topmost operation and access its arguments in the latter case.

There are two possibilities to construct the `Arith` type:

- If it is a recursive data type (i.e. the constructor parameters are of type `Arith`), pattern matching can analyse the structure in depth, but the extracted arithmetic expressions need to be converted back to `Expr`. This means that an inverse conversion is needed in addition.

- In the setting above (i.e. the constructor parameters are of type `Expr`), pattern matching is limited to one level at a time[1], but the arguments extracted are already of type `Expr`. This saves us from writing the inverse conversion.

Using this solution, one gets the following implementation of our model transformation:

```
optimize :: Expr -> Expr
optimize e = case arith e of
    0 :+: a -> a
    1 :*: a -> a
    0 :*: a -> 0
    _       -> e
```

This solution is much more satisfactory compared to that of section 2.1, but still not perfect:

- Arithmetic operations are represented in a different way in patterns (`:+:`) and expressions (`+`).

- There is still a considerable overhead when writing the library: extra data type(s) and conversion function(s) are needed to make pattern matching possible.

2.3. View patterns

This is a light-weight Haskell extension [3] that goes well with the solution of the previous section. The conversion from `Expr` to `Arith` can happen inside the pattern:

```
optimize :: Expr -> Expr
optimize (arith -> 0 :+: a) = a
optimize (arith -> 1 :*: a) = a
optimize (arith -> 0 :*: a) = 0
optimize e = e
```

[1] Note that the *view patterns* extension described in section 2.3 elegantly removes this limitation.

To match an expression with a view pattern, the value is first transformed using the function before the arrow and the result is matched with the pattern on its right hand side. The real power of this extension is shown when view patterns are nested. One can write patterns like the one in this function:

```
distr (arith -> a :*: (arith -> b :+: c)) = a * b + a * c
```

A similar solution can be achieved with the transformational patterns described in [12]. These extensions sometimes make the solution of section 2.2 more elegant, but the problems listed there (different operators in patterns and expressions, implementation overhead) still apply.

2.4. Active patterns

Instead of view patterns, there is a feature called *active patterns* [17] in F#. This language element merges the data type to be used for matching and the conversion function into one artifact. The example is provided in *haskellish pseudocode* instead of F# to ease the comparison with other solutions presented.

```
(|(:+:)|(:*:)|(:-:)|Other|) e = case e of
  Symbol "+" :$ a :$ b -> a :+: b
  Symbol "-" :$ a :$ b -> a :-: b
  Symbol "*" :$ a :$ b -> a :*: b
  _                    -> Other
```

The definition of an active pattern is like a function definition where the name of the function is replaced by the enumeration of the resulting data type's constructors. Now we can formulate the optimization function using pattern matching with active patterns:

```
optimize (0 :+: a) = a
optimize (1 :*: a) = a
optimize (0 :*: _) = 0
optimize e = e
```

This is even more elegant than the pattern matching solutions presented so far, as there is no need for explicit application of the conversion function. Yet, this is not entirely satisfactory: extra work (definition of the active pattern) is needed to make pattern matching possible, and we still have to use different operators in patterns and expressions.

2.5. Pattern synonyms

The *Strathclyde Haskell Enhancement* [13] is a preprocessor for Haskell that implements a selection of proposed language extensions. One of them is called *pattern synonyms*. One can define pattern synonyms using the following syntax:

```
pattern Add x y = Symbol "+" :$ x :$ y
pattern Mul x y = Symbol "*" :$ x :$ y
```

`Add` and `Mul` are allowed to appear both in expressions and in patterns. They are simply replaced with the right-hand sides of their definitions much like macros. The idea goes back to *abstract value constructors* introduced in [5]. To make semantics clear and the replacement possible at compile time, strict rules constrain the definition of pattern synonyms. The names are capitalized identifiers and the right-hand sides have to be valid patterns except that pattern synonyms are also allowed in place of constructors. Consequently, they are linear: all variables are used exactly once in the right-hand side of the definition. However, the currently available implementation does not check this property.

Using the pattern synonyms defined above, our optimization function takes this form:

```
optimize :: Expr -> Expr
optimize (Add 0 e) = e
optimize (Mul 1 e) = e
optimize (Mul 0 e) = 0
optimize e = e
```

The result is similar to that of section 2.4 and would yield the same code if infix pattern synonyms were supported. So far this solution requires the least additional effort from the programmer of the library. It may sometimes even be possible to provide pattern synonyms instead of functions in the public interface, but the limitations on the form of identifiers may make this undesirable and it is sometimes (like in our case with the functions of the `Num` class) impossible.

2.6. Function patterns

Curry [11] is a *functional logic programming language* based on Haskell and Prolog. Its syntax is similar to that of Haskell; therefore, the definition of the `Expr` data type seen in section 1 is completely valid Curry code. Unfortunately, Curry does not support type classes and numeric literals are not polymorphic. This implies the following changes in the front-end of our toy language embedding:

```
num :: Int -> Expr
num n = Symbol $ show n

(+.) :: Expr -> Expr -> Expr
x +. y = Symbol "+" :$ x :$ y

(*.) :: Expr -> Expr -> Expr
x *. y = Symbol "*" :$ x :$ y
```

The reason for mentioning Curry in this paper is that it provides the best solution for the problem at hand: *function patterns* [6]. Patterns in Curry are allowed to contain not only constructors and variables, but also functions, including built-in and user-defined ones.

```
optimize :: Expr -> Expr
optimize (num 0 +. x) = x
optimize (num 1 *. x) = x
optimize (num 0 *. _) = num 0
optimize e = e
```

If it were possible to overload arithmetic operations and numeric literals in Curry, this definition would be exactly our wish. Note that this is achieved without any further support from the library, because the functions used to construct expressions can also be used in patterns to deconstruct them.

However, the semantics of this extended pattern matching in Curry is far from that of Haskell. For example, the function

```
f :: [Int] -> ([Int], [Int])
f (xs ++ ys) = (xs,ys)
```

has multiple results for non-empty lists. The function application f [1,2] yields the following set of results: `([],[1,2])`, `([1],[2])`, `([1,2],[])`. This is acceptable in a functional logic language, but unsuitable for languages like Haskell. Furthermore, processing function patterns of Curry is a complex task that causes runtime overhead.

3. Restricted function patterns

The solutions presented in the previous section can be divided into the following categories:

- Section 2.1 avoids pattern matching. It is inconvenient and unsafe.
- Sections 2.2-2.4 are all approximations of *views* [18]. The basic idea behind them is that the value to be matched is first transformed to something that can be matched. The problem here is that, in addition to implementing functions to *construct* entities of the embedded language, a considerable effort is needed to provide ways of *deconstructing* them via pattern matching.
- An orthogonal possibility is shown in sections 2.5 and 2.6. Both pattern synonyms and function patterns provide a way to *build patterns*.

The goal here is to define a restricted class of function patterns that are more powerful than pattern synonyms, but much more restricted than Curry's function patterns in order to provide static guarantees and no performance overhead. The idea is to allow arbitrary expressions in patterns, if there is an equivalent valid (traditional) pattern.

Definition 3.1. If functions $f\, x_1\, x_2\, \ldots\, x_n = e$ and $f'\, x_1\, x_2\, \ldots\, x_n = e'$ are both of type $A_1 \to A_2 \to \cdots \to A_n \to B$, such that they are (extensionally) equal and e' is a valid pattern, then e is a **restricted function pattern** (RFP), and e' is its **canonical form**.

For example, all patterns in the "desired implementation" of the optimize function presented at the end of section 1 are restricted function patterns. The following table shows their canonical forms.

RFP	Canonical form
`0+a`	`(Symbol "+" :$ Symbol "0") :$ Symbol "a"`
`1*a`	`(Symbol "*" :$ Symbol "1") :$ Symbol "a"`
`0*a`	`(Symbol "*" :$ Symbol "0") :$ Symbol "a"`
`a`	`a`

Is RFP a given expression or not? This is not a decidable problem in general as results about the *halting problem* show. Nevertheless, it is possible to construct a decision algorithm that accepts only RFPs. It rejects all non-RFPs, but also *some* of the RFPs. For example, symbolic execution of the expression with a fixed limit on the number of reduction steps is such an algorithm. An expression is accepted if symbolic execution reaches the canonical form. If it is impossible to continue symbolic execution (for example, non-trivial pattern matching is performed on a parameter) or the number of reduction steps reach the limit, the function is rejected. Note however that the result of this decision algorithm is hard to predict, which is a major concern from the user's point of view. Finding a decision algorithm that is powerful enough and easily predictable is an important future task.

Even if symbolic execution is not the ideal decision algorithm, it has a useful feature: it provides the canonical form of all accepted RFPs. This way the compiler can simply replace RFPs by the corresponding canonical form. As those are "normal" patterns, it is possible to process them in the usual way. Note that all this happens in compile time: an RFP is transformed to an ordinary pattern making no runtime overhead and no change to the pattern matching algorithm. This also means that RFPs are deterministic: if the match succeeds, each variable in the pattern is bound to a single value. In contrast, Curry's function patterns are processed at runtime and are indeterministic.

In fact, a smart compiler already does something similar for optimization purposes. It may perform symbolic computation to reduce the expressions as much as possible to increase runtime performance. RFPs are good candidates for such an optimization. The difference is that the optimization-related transformations happen in expressions instead of patterns.

4. Applications

4.1. The problem at hand

The problem defined in section 1 can be solved in an elegant way using RFPs.

```
optimize :: Expr -> Expr
optimize (0 + a) = a
optimize (1 * a) = a
optimize (0 * a) = 0
optimize a       = a
```

- This solution is as clear and readable as that of Curry.
- There is no extra work needed by the creator of the library to enable pattern matching.
- The RFPs can be replaced with their canonical forms at compilation time causing no runtime overhead.

4.2. List prefix patterns

In the *Erlang* programming language, `"prefix" ++ Str` is a valid pattern [4]. In fact, it is syntactic sugar for the list pattern `[$p,$r,$e,$f,$i,$x | Str]`. These correspond to `‘‘prefix’’ ++ str` and `’p’ : ’r’ : ’e’ : ’f’ : ’i’ : ’x’ : str` in Haskell, which are an RFP and its canonical form. This means that one can use RFPs to implement a function that chops off the `‘‘prefix’’` prefix of input strings:

```
unprefix :: String -> String
unprefix ("prefix" ++ str) = str
unprefix str = str
```

While in Erlang this feature is an ad hoc extension to pattern matching, RFPs provide it in a consistent way.

This example shows the weekness of the naive decision algorithm mentioned in section 3. In case a very long string literal were used in the example above, symbolic execution would reach the limit on the number of steps and the pattern would be rejected.

This example also demonstrates that RFPs are more powerful than pattern synonyms, because it is impossible to define the append function as a pattern synonym. On the other hand, RFPs are more restricted than Curry’s function patterns, because neither `str1 ++ str2` nor `str ++ ‘‘postfix’’` are RFPs.

4.3. Matching numeric literals

One of the ingenious features of Haskell that makes this language particularly suitable for language embedding is that numeric literals are polymorphic. The literal `0` for example can be of any type that instantiates the `Num` class. In the case of our toy embedding, the literal `0` of type `Expr` means `Symbol ‘‘0’’` because of the implementation of the `fromInteger` function for the `Expr` type. In fact, an integer literal n is automatically transformed to `fromInteger` n by the compiler.

The unfortunate bit is that the same has to happen also in patterns. In the `Num` instance below, the `fromInteger` function is `undefined`, and this makes pattern matching unexpectedly unsafe.

```
data Foo = Foo
    deriving (Eq, Show)

instance Num Foo where
```

```
    fromInteger _ = undefined

f :: Foo -> Bool
f 0 = False
f _ = True
```

Evaluating the expression `f Foo` in the Haskell interpreter yields an exception, because the `fromInteger` function is called to transform the literal `0` in the pattern to type `Foo`. If we change the implementation of `fromInteger` so that it does infinite recursion, then so does the matching.

If `fromInteger` terminates normally, its result is compared to the value to be matched using the `(==)` function. In the example above the compiler generates a default implementation of `(==)` for the type `Foo` because of the *deriving clause* at the end of the data type definition. However, the programmer is allowed to implement it in such a way that it raises an exception or hangs, and in that case pattern matching is fooled again. This design contradicts the goal that pattern matching should be safe and fast.

Is it possible to correct this? As discussed above, the compiler calls the function `fromInteger` to convert integer literals to the desired type. Let us write this conversion explicitly in the pattern even if this is incorrect Haskell code:

```
f (fromInteger 0) = False
```

Now it is clear that the Haskell compiler in fact uses function patterns! The problem is that there is no guarantee that these are RFPs. What the compiler should do is the following:

- If `fromInteger 0` is an RFP (like in the case of `Expr` or the built-in numeric types), then using the literal `0` is all right: it should be replaced by the canonical form of `fromInteger 0`, and the usual pattern matching algorithm can be used instead of equality. This way there is no need to make `Eq` a superclass of `Num`.

- If `fromInteger 0` is not an RFP (for example, if it is undefined), then the compiler should complain about the pattern. (More on this in section 5.2.)

This means that the RFP extension discussed in this paper would make Haskell's current pattern matching rules related to numeric literals more consistent and safer.

5. Problems and solutions

5.1. Lexical ambiguity

Haskell patterns contain constructors and variables. Constructors are capitalized identifiers or operators starting with a colon, while variables are identifiers starting with a lower-case letter. In contrast, RFPs can contain ordinary functions that are syntactically indistinguishable from variables.

Let us suppose that `x` is defined to be a zero-arity function of type `Int` as `x = 5`. Now it is not clear if `x` in the pattern `(x:xs)` is the zero-arity function with value 5 or a free variable of the pattern. Note that this ambiguity is only present in the case of symbols without arguments: in the RFP `(a b)` the symbol `a` cannot be a variable unless we allow higher-order patterns. Let us list the possible solutions for this problem:

- The same issue is present in the Agda programming language [1] that allows lower-case constructors. Agda's rule says that `x` can only be a variable if it is not defined as a constructor.
- In contrast, Curry considers these symbols as variables and gives a warning if they shadow zero-arity functions.
- Another possible solution is to invent some syntax to distinguish variables from functions in dubious cases.

5.2. Type system issues

A much more important problem is that type systems used in practice cannot express that an expression is an RFP. This is problematic, because changing the implementation of a function may make the compiler complain about patterns even if the type of the function is untouched. The following example illustrates the problem.

```
double :: Expr -> Expr
double x = 2 * x
```

The expression `double x` is an RFP with canonical form `Symbol ‘‘*’’ :$ Symbol ‘‘2’’ :$ x`. Let us use it in the following function definition:

```
halve :: Expr -> Expr
halve (double x) = x
halve x = Symbol "div" :$ x :$ 2
```

What happens if we change the implementation of `double` slightly?

```
double :: Expr -> Expr
double x = x + x
```

The expression `double x` now reduces to `Symbol ‘‘+’’ :$ x :$ x`, which is not a valid pattern any more, at least not in a language with linear patterns. The compiler in this case will reject the first equation of the `double` function. This may be annoying since the type of `double` did not change.

There are two possible solutions: either the type system is extended, or the compiler has to handle non-restricted function patterns in a different way. The rest of this section explores these possibilities.

5.2.1. Extending the type system

There are certain properties that make a function suitable to be included in RFPs. Suppose we could express these properties in type signatures and had a type system that, based on that type information, is able to infer if an expression is an RFP or not. These changes like in the example above would be reflected in the functions' types. To have such a type system, there are three aspects to consider: well-definedness, parametricity and linearity.

Well-definedness. If the function $\backslash x_1\, x_2\, \ldots\, x_n \rightarrow e$ is not well defined for a set of (well-defined) arguments $v_1\, v_2\, \ldots\, v_n$, then e cannot be an RFP. Raising an exception, returning `undefined` or going into infinite recursion are not allowed. In type theory, many different type systems were designed to ensure the termination of expressions.

Parametricity. Parametric functions use their parameters as black boxes: no pattern matching is allowed on them. For example, the function `singleton` below is parametric, but `length` is not.

```
singleton :: a -> [a]
singleton x = [x]

length :: [a] -> Int
length []     = 0
length (x:xs) = 1 + length xs
```

Parametric functions are important from our perspective, because this property ensures that symbolic execution is possible: it is possible to reduce the function pattern without binding its free variables. Together with well-definedness, this ensures that symbolic execution terminates and leads to an expression containing constructors and variables only.

Note that non-parametric functions are still allowed in RFPs provided that their parameters are values known at compilation time. An example is the append function used in the list prefix pattern in section 4.2. It is parametric in its second argument but not in the first one. That is why `‘‘prefix’’ ++ str` is an RFP, but `str ++ ‘‘postfix’’` is not.

Parametric functions form a well-studied class, which is important for language embedding also because of the *higher order abstract syntax* (HOAS) technique [15]. For the sake of an example let us extend the `Expr` type from section 1 with a new constructor for λ-abstraction. An elegant solution is this one: `Lam (Expr -> Expr)`. The new constructor `Lam` has become part of the abstract syntax of the language and it is a higher order function: hence the name of the technique (HOAS). This solution makes nontrivial functions (like β-reduction, for example) quite easy to implement. On the other hand, passing a non-parametric function to `Lam` results in an expression which has no corresponding λ-term, like in the case of the following example.

```
Lam $ \x -> case x of
  Symbol "a" -> Symbol "a"
  otherwise  -> Symbol "b"
```

This is why parametricity is important for the HOAS technique. A great amount of work has gone into solving these kind of problems. An early one is an extension to the ML language [14] that makes pattern matching possible on parametric functions. In [7] a type system is developed that can distinguish the parametric and non-parametric function spaces. Such a type system would also be useful for checking RFPs.

Linearity. In many functional languages patterns must be linear: using the same variable more than once in a pattern is invalid. In such a language we have to ensure that the canonical forms of RFPs are also linear. Linear type systems [19] are well studied and are important from the perspective of destructive updates and interaction with the real word while still keeping referential transparency. A notable example is the Clean language, which implements uniqueness typing [16] for this purpose.

5.2.2. Warnings instead of errors

If the type system is not strong enough to express that an expression is RFP, the compiler should not reject non-RFP patterns for the reason discussed at the beginning of this section. So what should the compiler do if it finds a non-RFP pattern (or it is not able to prove that it is an RFP)? A possibility is to give a warning to the user and replace the pattern with one that fails to match any value. This way the decision of the compiler to accept or reject a pattern will only depend on the types of the functions involved. Changing their implementation (without touching their types) may affect only the runtime behaviour of the pattern match and may result in warnings.

6. Implementation

Before creating a proper language extension, more research is needed about the issues discussed in section 5. Nevertheless, it was already possible to create a lightweight Haskell library that provides the functionality of restricted function patterns, even though it is not able to check the restrictions at compile time and cannot guarantee the same runtime performance that built-in compiler support could. This library is currently available for testing [8] in the *Hackage library database* [2].

6.1. Public interface

Using this library, one can implement the optimization function discussed in section 2 as follows.

```
opt :: Expr -> Expr
opt e = match e $ do
    with $ \a -> 0 + a ~> a
    with $ \a -> 1 * a ~> a
    with $ \a -> 0 * a ~> 0
    with $ \a -> a     ~> a
```

The `match` function gets two arguments: the value to be matched and a sequence of cases. In order to mimic the syntax of Haskell's *case expressions*, the cases are listed in a monadic environment (hence the `do` keyword). Cases are created using the `with` function of arity one. Its argument is a function with arbitrary number of arguments (including zero) and it produces a pattern and a corresponding result combined by the `(~>)` operator.

If only RFPs are used as patterns, the library guarantees that a match is equivalent to a case expression with the corresponding canonical forms. However, the library is not able to check at compile time if the patterns are really RFPs; this is the responsibility of the user.

- If the pattern is not well-defined (due to `undefined` or non-termination), the match may fail, terminate with `undefined` or hang.

- If the pattern is not parametric, the match may fail, succeed or (in most of the cases) the *nonparametric pattern* error is raised.
- If the pattern is not linear, the match may fail or the *nonlinear pattern* error is raised.

It might be disappointing that non-parametric patterns makes matching so unpredictable, but it is important to note that a successful match is always correct. Whenever a value v matches a case[2] $\backslash x_1 x_2 \ldots x_n \rightarrow p \rightsquigarrow e$ and the variables get bound to the values $v_1, v_2, \ldots, v_n$ respectively, the value $(\backslash x_1 x_2 \ldots x_n \rightarrow p)\, v_1\, v_2\, \ldots\, v_n$ is well-defined and really matches v. This also holds for erroneous patterns.

In order to make this kind of pattern matching possible for a new data type, one has to make it an instance of the `Matchable` class. This instance defines the components of compound values. For example, the `Machable` instance of the `Expr` type is the following.

```
instance Matchable Expr where
    Symbol s .=. Symbol z    = Just [s :=: z]
    (e :$ f) .=. (g :$ h)    = Just [e :=: g, f :=: h]
    _ .=. _                  = Nothing
```

This means that

- symbols match, if their names match,
- function applications match, if both the functions and the arguments match,
- there is no other way for expressions to match.

Matchable instances are trivial and could be automatically created by the compiler.

6.2. Pattern matching algorithm

Matching a value v with a case $c = \backslash x_1 x_2 \ldots x_n \rightarrow p \rightsquigarrow e$ is performed as follows. First, the system creates parameters of two kinds that we denote by T and F. The function c is applied to these parameters such that x_1 becomes T and all others from x_2 to x_n become F. Let us denote the result by $p^* \rightsquigarrow e^*$. Now p^* is matched with v using an algorithm described below and this leads to one of the following results:

[2]In mathematical notation $\rightsquigarrow$ will denote the `(~>)` operator of the library.

- If the match fails, the next case is tried.
- If the match succeeds, it provides a value v_1 that x_1 will be bound to. The binding is done by applying c to v_1. This yields the function $c_1 = \backslash x_2 \ldots x_n \to p_1 \rightsquigarrow e_1$. Then the whole algorithm is called again recursively with this reduced case that has one argument less than the original.

After eliminating all arguments without failure, one gets $p_n \rightsquigarrow e_n$. If p_n matches v, then the case fires and the result is e_n.

Now let us describe how a pattern *pat* is matched with a value *val*:

- If *pat* is the parameter T then *val* is returned as a candidate for the next binding.
- If *pat* is the parameter F then the match succeeds, but no value is returned.
- If *pat* is not a parameter then $val \,.\!=\!.\, m$ is called (see the `Matchable` class above). It provides a list of match conditions, and matching is done recursively for all pairs in this list:
  - If any of them fail, the whole match fails.
  - If all of them succeed and either none of them or at least two of them return a value, then the pattern is not linear and an error is raised.
  - If all of them succeed and exactly one returns a value, then the match succeeds and returns the value.

Matching p_n with v is done similarly, but in this case no value is returned (as there are no more variables to bind).

What are the parameters T and F technically? Besides the function `(.=.)` seen before, the `Matchable` class contains two more functions :

```
makeParam   :: Bool -> a
isParam     :: a -> Maybe Bool
```

Parameter T and F are `makeParam True` and `makeParam False` respectively. The `isParam` function decides if its argument is a parameter or not. The following rules must hold:

- $isParam\ (makeParam\ True)\ ==\ Just\ True$
- $isParam\ (makeParam\ False)\ ==\ Just\ False$

- If a is not a parameter, then $isParam\ a\ ==\ Nothing$.

Users of the library do not have to implement these functions (unless they want to). The default implementation of `makeParam` throws an exception that wraps its argument. `isParam` tries to evaluate its argument to head normal form. If this succeeds then it was not a parameter, otherwise the exception is caught and the wrapped logical value is returned. This is only possible within the IO monad, but it is hidden using the `unsafePerformIO` function. One reason for writing custom implementations of `makeParam` and `isParam` may be that one works with exceptions interfering with the ones used by the library.

7. Conclusion

Embedded languages and similar functional libraries often hides the constructors of their data types. Instead they provide functions in their public interfaces that can be used for data construction. This design makes pattern matching in user code impossible. To make it possible after all, the library has to provide additional tools to deconstruct data. This paper searches ways to avoid (or considerably reduce) this additional work.

Pattern synonyms and *function patterns* are promising techniques. While the former one is too restrictive, the latter one is too general for a functional programming language. Therefore, this paper proposes a compromise between the two: *restricted function patterns*. A class of expressions is defined that is safe to be allowed in patterns, because they can be replaced with equivalent traditional patterns at compilation time.

Problems related to this language extension are analysed and solutions are proposed. Finally, a lightweight implementation is presented that provides the functionality of restricted function patterns. The author of this paper has already used this library in two embedded language projects [10, 9].

The most important future work in this topic is the design of a decision algorithm for a well defined subset of RFPs. The algorith must be smart enough to allow all RFPs that are useful in practice and, at the same time, must be intuitive enough to give results easily predictable by the users. The basic concepts to consider are already discussed in section 5.2 of this paper, and preliminary research shows that the combination of symbolic execution with the size change principle [20] will yield a suitable algorithm.

References

[1] *The Agda programming language*,
`http://wiki.portal.chalmers.se/agda/`

[2] *HackageDB*,
`http://hackage.haskell.org/`

[3] *View patterns: lightweight views for Haskell*,
`http://hackage.haskell.org/trac/ghc/wiki/ViewPatterns`

[4] *Erlang/OTP System Documentation*, 2011.
`http://www.erlang.org/doc/pdf/otp-system-documentation.pdf`

[5] **Aitken, W.E. and J.H. Reppy,** Abstract value constructors, in: *ACM SIGPLAN Workshop on ML and its Applications*, 1992, 1–11.

[6] **Antoy, S. and M. Hanus,** Declarative programming with function patterns, in: *Logic Based Program Synthesis and Transformation*, Lecture Notes in Computer Science, **3901**, Springer, Berlin/Heidelberg, 2006, 6–22.

[7] **Despeyroux, J., F. Pfenning and C. Schürmann,** Primitive recursion for higher-order abstract syntax, in: P. de Groote and J. Roger Hindley (eds.) *Typed Lambda Calculi and Applications*, Lecture Notes in Computer Science, **1210**, Springer, Berlin/Heidelberg, 1997, 147–163.

[8] **Dévai, G.**, *Restricted Function Patterns (Haskell library)*,
`http://hackage.haskell.org/package/funpat`

[9] **Dévai, G.**, Embedding a Proof System in Haskell, in: Z. Horváth, R. Plasmeijer and V. Zsók (eds.) *Central European Functional Programming School*, Lecture Notes in Computer Science, **6299**, Springer Berlin/Heidelberg, 2010, 354–371.

[10] **Dévai, G., M. Tejfel, Z. Gera, G. Páli, Gy. Nagy, Z. Horváth, E. Axelsson, M. Sheeran, A. Vajda, B. Lyckegård and A. Persson,** Efficient Code Generation from the High-level Domain-specific Language Feldspar for DSPs, in: *Proceedings of the 8th Workshop on Optimizations for DSP and Embedded Systems*, 2010, 12–20.

[11] **Hanus, M.** (ed.), *Curry: An Integrated Functional Logic Language (version 0.8.2)*, 2006,
`http://www.informatik.uni-kiel.de/˜curry/report.html`

[12] **Erwig, M. and S.P. Jones,** Pattern Guards and Transformational Patterns, in: *Haskell Workshop*, 2000.

[13] **McBride, C.,** *Strathclyde Haskell Enhancement*,
`http://personal.cis.strath.ac.uk/˜conor/pub/she/`

[14] **Miller, D.,** An extension to ML to handle bound variables in data structures: Preliminary Report, in: *Proceedings of the First Workshop on Logical Frameworks*, 1990, 323–336.

[15] **Pfenning, F. and C. Elliot,** Higher-order abstract syntax, *SIGPLAN Not.*, **23** (1988), 199–208.

[16] **Plasmeijer, R. and M. van Eekelen,** Keep it Clean: a Unique Approach to Functional Programming, *SIGPLAN Not.*, **34** (1999), 23–31.

[17] **Syme, D., G. Neverov and J. Margetson,** Extensible pattern matching via a lightweight language extension, *SIGPLAN Not.*, **42** (2007), 29–40.

[18] **Wadler, P.,** Views: a way for pattern matching to cohabit with data abstraction, in: *Proceedings of the 14th ACM SIGACT-SIGPLAN symposium on Principles of programming languages (POPL)*, ACM, New York, NY, USA, 1987, 307–313.

[19] **Wadler, P.,** Linear types can change the world!, in: *IFIP TC 2 Working Conference on Programming Concepts and Methods*, 1990, 347–359.

[20] **Lee, C.S., N.D. Jones and A.M. Ben-Amram,** The size-change principle for program termination, *SIGPLAN Not.*, **36:3**, ACM, New York, NY, USA, 2001, 81–92.

G. Dévai
Department of Programming Languages and Compilers
Faculty of Informatics
Eötvös Loránd University
H-1117 Budapest, Pázmány P. sétány 1/C
Hungary
`deva@elte.hu`