In: *8th Int. WS on Implementation of Functional Languages*, 1996, LNCS 1268, 21-40.

Active Patterns

Martin Erwig

FernUniversität Hagen, Praktische Informatik IV
58084 Hagen, Germany
erwig@fernuni-hagen.de

Abstract. Active patterns apply preprocessing functions to data type values before they are matched. This is of use for unfree data types where more than one representation exists for an abstract value: in many cases there is a specific representation for which function definitions become very simple, and active patterns just allow to assume this specific representation in function definitions. We define the semantics of active patterns and describe their implementation.

1 Introduction

Pattern matching is a well-appreciated concept of (functional) programming. It contributes to concise function definitions by implicitly decomposing data type values. In many cases, a large part of a function's definition is only needed to prepare for recursive application and to finally lead to a base case for which the definition itself is rather simple. Such function definitions could be simplified considerably if these preparatory computations could be factorized and given in a separate place. Recognizing that the same normalization is often used in more than one function definition this seems even more reasonable since avoiding code means reducing the risk of introducing programming errors.

We propose to hide these computations in the definition of a special kind of patterns that allow the programmer to assume the "simple base case" in a function definition. Consider, for example, the implementation of sets based on lists. Function definitions for membership test, insertion and deletion typically check whether the first list element equals the argument element to be searched, inserted, or deleted, and, depending on the result, a value is returned or the function being defined is recursively called. Given the following data type definition:[1]

```
datatype 'a set = Empty | Add of 'a * 'a set
```

the function definition, for example, for membership test is:

```
fun member x (Add (y,s)) = if x=y then true else member x s
  | member x Empty       = false
```

[1] Throughout this paper we will use ML syntax.

If we can use a pattern that, when it matches, guarantees that the required element is in the first position of the list, the function definition becomes remarkably simple. Assume for a moment we have already defined such a pattern, say `Add' (x,s)`, which transforms any set value `v` matched against into a `set`-term `Add (x,t)` (if possible, that is, if `x` is contained in `v`). Then the definition of `member` is:

```
fun member x (Add' (x,s)) = true
 |  member x s            = false
```

Informally, `member` now works as follows: when called with an element `x` and a set value `v`, the computational part of `Add'` is applied to `v` to move the item `x` to the first position. If this can be performed successfully, we know that `x` is contained in `v`, and thus we can simply return `true`. Moreover, the rest of `v` is bound to the variable `s`. Otherwise, that is, if `v` cannot be rearranged as demanded, the match fails and the second line of `member` applies. Under the assumption that lists representing sets do not contain duplicates (this has to be ensured by the insertion function) a function definition for deletion is simply given by:

```
fun delete x (Add' (x,s)) = s
 |  delete x s            = s
```

Finally, insertion leaves a term already containing the element to be inserted unchanged or otherwise simply performs insertion by applying the `Add`-constructor.

```
fun insert x (Add' (x,s)) = Add (x,s)
 |  insert x s            = Add (x,s)
```

The given function definitions come up to equational specifications of abstract data types, yet they are efficiently executable function definitions. Of course, the correctness of the above definitions critically depends on the proper definition for `Add'`: on the one hand, the process of rearranging must preserve the represented set value, and on the other hand, rearrangement has to fail exactly if `x` is not contained in `v`.

Due to their computational content we call patterns like `Add'` *active patterns*. A definition of an active pattern actually consists of two parts: a *function* that performs the described rearrangement and an *interface* to the function that serves two purposes:

1. The interface defines the *syntax* of the pattern, that is, the actual appearance when used in the left hand side of a function definition. The appearance is derived from the constructor term produced by its function and differs only in the constructor name to distinguish it from an ordinary constructor term.[2]

[2] In this paper we use the convention that the pattern name is the constructor name followed by an apostrophe. This helps identifying the constructor which is actually to be expected at the position of an active pattern.

2. The interface introduces the *free variables* of the pattern's function, if there are any. Recall that the function transforms a term into a specific form which is often guided by an external value. (In the example above this was the element `x`.) This value might be a constant or a variable, which then appears free in the pattern function. We call these variable positions in an active pattern *free positions.*

The definition of an active pattern is introduced by the keyword `pat`. For example, the definition of `Add'` is:

```
pat Add' (x,_) =
    Add (y,s) => if x=y then Add (y,s)
                        else let val Add' (x,t)=s
                              in Add (x,Add (y,t)) end
```

Informally, the above code reads as follows: `Add'` is an active pattern, and the associated pattern function might use the free variable `x`. The function itself expects a term `Add (y,s)`, and if `x` equals `y`, that term is already the result and the unbound variables of the pattern (the positions of which are indicated in the interface by an underscore) are bound to the corresponding parts of the term. If $\mathtt{x} \neq \mathtt{y}$, the pattern function searches for `x` in `s`, that is, `s` is rearranged by recursively matching the active pattern `Add'`. If this succeeds, `x` is swapped with `y`, yielding the desired result. Thus if `x` is contained in the argument term, it will "bubble up" by recursively being exchanged with preceding elements in the term. Note that the function expects a term `Add (y,s)`, that is, it fails for `Empty` which means that if `x` is not found in the term, the function would usually (that is, according to the semantics of Standard ML) raise a `Match` exception. But this is handled by the semantics to move to the next case (following the current active pattern) in the function definition.

It might seem the described effects could also be achieved by a language extension allowing just repeated variables in patterns, but this is not the case, since plain non-linear patterns miss the computational part of active patterns, that is, they match only if the argument is already in the desired shape. For example, `member 2 (Add (1,Add (2,Empty)))` correctly yields `true` for the definition with active patterns, but it gives `false` with just repeated variables in patterns which are merely checked for equality.

As in the above example, the work of active patterns is often guided by an external value which results in repeated variables in patterns, but active patterns do not always require non-linear patterns. Consider, for example, the implementation of queues by two lists [Bur82]. The first list contains the front part of the queue, and the second list contains the rear part of the queue in reverse order. This makes it possible to append new elements by simply consing to the second list while dequeuing can still be performed by simply removing the first element of the first list. Now when the first list is empty, it is exchanged with the reverse of the second list. We use the following data type.

```
datatype 'a queue = Q of 'a list * 'a list
```

Appending to the queue needs no active pattern at all, the definition is:

```
fun append x (Q (f,r)) = Q (f,x::r)
```

Using a pattern that asserts the first list is not empty:

```
pat Q' (_,_) = (Q ([],r)) => Q (rev r,[])
             | q          => q
```

inspection and deletion of the front element is now very simple:

```
fun front   (Q' (x::_,_)) = x
fun dequeue (Q' (x::f,r)) = Q (f,r)
```

Here the rearrangement of the data type value just depends on the value itself, no external value is needed, and patterns stay linear. Another example is the implementation of priority queues based on lists. There an active pattern has to move the list's minimum to the front which does not require an external value either.

In the next section we consider more applications of active patterns, and in Section 3 we define static and dynamic semantics of a core language containing active patterns. In Section 4 we show a straightforward implementation of active patterns using a rather simple source code transformation. We also sketch a fusion algorithm that is able to optimize some function definitions very well. We comment on related work in Section 5, and some conclusions are drawn in Section 6.

2 More Examples

Active patterns can be used to write highly concise function definitions. We illustrate this with an implementation of binary search trees.

Trees are represented by the following data type.

```
datatype tree = Nil | Node of int * tree * tree
```

Now we can define an active pattern `Node'` that realizes a generalized form of rotation for moving a specific node to the root:

```
pat Node' (x,_,_) = (t as Node (y,l,r)) =>
    if x=y then t else
    if x<y then let val Node' (x,l',r') = l
                in Node (x,l',Node (y,r',r)) end
           else let val Node' (x,l',r') = r
                in Node (x,Node (y,l,l'),r') end
```

Using this active pattern the code for testing membership in a tree looks very similar to the definition given for `member`:

```
fun isin x (Node' (x,_,_)) = true
 |  isin x _               = false
```

More interesting is the function for deletion. The definition employs a function `delmin` that deletes the minimum from a binary tree and returns the minimum itself and the remaining tree:

```
fun delmin (Node (x,Nil,r)) = (x,r)
 |  delmin (Node (x,l,r))   = let val (m,s) = delmin l
                               in (m,Node (x,s,r)) end
```

Now the function `remove` can be defined by:

```
fun remove x (Node' (x,Nil,r)) = r
 |  remove x (Node' (x,l,Nil)) = l
 |  remove x (Node' (x,l,r))   = let val (m,s) = delmin r
                                  in Node (m,l,s) end
 |  remove x t                 = t
```

This is already a quite compact definition. However, we can go even further by extending the data type by a new variant serving as an interface between `remove` and `delmin`.

```
datatype tree =
   Nil
 | Node   of int * tree * tree
 | DelMin of int * tree
```

This extension is not very problematic since it is only used locally within the definition of `remove`; `isin` (and also a possible function for insertion) are not affected. Now we can define an active pattern `DelMin'` for extracting the minimum of a tree by returning a term `DelMin (i,t)`.

```
pat DelMin' (x,_) =
      Node (x,Nil,r) => DelMin (x,r)
    | Node (x,l,r)   => let val DelMin' (m,s) = l
                         in DelMin (m,Node (x,s,r)) end
```

This could also be specified by using the active pattern recursively in the LHS:

```
pat DelMin' (_,_) =
      Node (x,Nil,r)           => DelMin (x,r)
    | Node (x,DelMin' (m,s),r) => DelMin (m,Node (x,s,r))
```

Now, `remove` can be defined even more succinctly:

```
fun remove x (Node' (x,Nil,r)) = r
 |  remove x (Node' (x,l,Nil)) = l
 |  remove x (Node' (x,l,DelMin' (m,s))) = Node (m,l,s)
 |  remove x t = t
```

Another interesting application area for active patterns are graphs: in order to realize graph algorithms in functional languages it is convenient to view a graph inductively as a term `Graph (v,p,s,g)` adding a node `v` with predecessors `p` and successors `s` to another graph `g`. Now many graph algorithms traverse graphs in a particular node order. This means to access a graph repeatedly at specific nodes, which is possible by an active pattern `Graph'` having `v` as a free variable. This is explained in more detail in [Erw97].

3 Syntax and Semantics

In the following we consider the language defined in Figure 1.[3] As usual we have constants, variables, abstraction, and application. Tuples are needed to build data type terms. In addition to value declarations, local definitions for active patterns are also possible. Both kinds of `let`-expressions allow for recursive definitions.

exp	::=	*con*	*pat*	::=	*atpat*
	|	*var*		|	*cpat*
	|	(exp_1, ..., exp_n)	*cpat*	::=	*con atpat*
	|	*exp exp*	*atpat*	::=	_
	|	`fn` *match*		|	*con*
	|	`let val` *pat*=*exp* `in` *exp* `end`		|	*var*
	|	`let pat` *cpat*=*match* `in` *exp* `end`		|	(pat_1, ..., pat_n)
match	::=	*rule* ⟨ `|` *match*⟩			
rule	::=	*pat* `=>` *exp*			

Fig. 1. Syntax of Expressions and Patterns.

Note that a function does not simply abstract over a variable, but is given by a collection of rules where a rule associates an expression with a pattern. This is the place where active patterns come into play. A pattern is either atomic, or it is a constructor pattern which is an application of a constructor to an atomic pattern. An atomic pattern is a wildcard, a variable, or a tuple of patterns.

The use of repeated variables in patterns is restricted to the following case: in the preorder traversal of a pattern, the first occurence of each variable must be in non-free position (that is, not matching a parameter of a pattern function), and each further occurrence must be in free position. This guarantees that repeated

[3] Phrases enclosed in angle bracket are optional.

variables are just used to supply parameters for active patterns.

For example, the function definition for `member` using this austere syntax is:[4]

```
let pat Add' (x,_) =
    Add (y,s) => if x=y then Add (y,s)
                        else let val Add' (x,t)=s
                              in Add (x,Add (y,t)) end
 in
let val member = fn x =>
        fn (Add' (x,s)) => true
         | s            => false
 in
    exp
end end
```

3.1 Static semantics

The static semantics include, of course, type inference rules, but especially for active patterns there is a more fine-grained analysis possible which we call "constructor checking". We begin with the description of the type system. The language of types is given in Figure 2.

$$
\begin{array}{rcll}
\tau & ::= & \alpha & \text{type variable} \\
 & | & \tau \rightarrow \tau & \text{function type} \\
 & | & (\tau_1, \ldots, \tau_n) & \text{tuple type } (n > 1) \\
 & | & \langle\, \tau \,\rangle \ \mathit{con} & \text{data type} \\
\sigma & ::= & \forall \alpha_1, \ldots, \alpha_n . \tau & \text{type scheme}
\end{array}
$$

Fig. 2. Types.

The typing rules for constants, variables, application and local value definition are standard. Type checking for abstractions over patterns might not be so well-known, and this is explained best by dealing with two kinds of sentences in the type inference system: (1) $A \vdash \mathit{exp} : \tau$ says that expression *exp* has type τ under the set of assumptions A. Usually, A is a finite map from variables and constructors to types, but for a constructor of an active pattern we store its type together with its interface in A. (2) For a pattern we derive, in addition to its type, also a set of assumptions (for the variables occurring within the pattern). We write such assertions as $A \vdash \mathit{pat} \hookrightarrow (A', \tau)$.

Since we have to treat repeated variables in patterns we must be careful about the definition of the union of two assumptions. Defining $A + A'$ as usual by "overwriting" the definitions in A by those in A' leaves some type errors

[4] Note that we still keep conditionals. To be precise, `if` c `then` e_1 `else` e_2 translates to `(fn true =>` e_1 `| false =>` e_2`)` c.

undetected. We thus require identity on common subdomains, that is:

$$(A + A')(x) = \begin{cases} A(x) & \text{if } x \in dom(A) \wedge (x \in dom(A') \Rightarrow A(x) = A'(x)) \\ A'(x) & \text{if } x \in (dom(A') - dom(A)) \\ \bot & \text{otherwise} \end{cases}$$

The type system is given in Figure 3, and the derivation of assertions from patterns is shown in Figure 4.

$$\text{CON}_\vdash \ \frac{A(con) \succ \tau}{A \vdash con \ : \ \tau} \qquad\qquad \text{VAR}_\vdash \ \frac{A(var) \succ \tau}{A \vdash var \ : \ \tau}$$

$$\text{TUP}_\vdash \ \frac{A \vdash exp_1 \ : \ \tau_1 \qquad \ldots \qquad A \vdash exp_n \ : \ \tau_n}{A \vdash (exp_1, \ldots, exp_n) \ : \ (\tau_1, \ldots, \tau_n)}$$

$$\text{APP}_\vdash \ \frac{A \vdash exp \ : \ \tau' \rightarrow \tau \qquad A \vdash exp' \ : \ \tau'}{A \vdash exp \ exp' \ : \ \tau}$$

$$\text{ABS}_\vdash \ \frac{A \vdash match \ : \ \tau}{A \vdash \texttt{fn} \ match \ : \ \tau} \qquad\qquad \text{MATCH}_\vdash \ \frac{A \vdash rule \ : \ \tau \qquad A \vdash match \ : \ \tau}{A \vdash rule \ \texttt{|} \ match \ : \ \tau}$$

$$\text{RULE}_\vdash \ \frac{A \vdash pat \ \hookrightarrow \ (A', \tau') \qquad A + A' \vdash exp \ : \ \tau}{A \vdash pat \ \texttt{=>} \ exp \ : \ \tau' \rightarrow \tau}$$

$$\text{LET}_\vdash \ \frac{A \vdash pat \ \hookrightarrow \ (A', \tau') \qquad A + A' \vdash exp' \ : \ \tau' \qquad A + gen(A, A') \vdash exp \ : \ \tau}{A \vdash \texttt{let val} \ pat\texttt{=}exp' \ \texttt{in} \ exp \ \texttt{end} \ : \ \tau}$$

$$\text{LETP}_\vdash \ \frac{\begin{array}{c} A(con') = \tau'' \rightarrow \tau' \qquad A \vdash atpat \hookrightarrow (A', \tau'') \qquad A + A' \vdash match \ : \ \tau' \rightarrow \tau' \\ A + \{con' \mapsto (gen(A, \tau'' \rightarrow \tau'), atpat)\} \vdash exp \ : \ \tau \end{array}}{A \vdash \texttt{let pat} \ con' \ atpat\texttt{=}match \ \texttt{in} \ exp \ \texttt{end} \ : \ \tau}$$

Fig. 3. Type System.

The operation *gen* in rules LET$_\vdash$ and LETP$_\vdash$ is for the generalization of types to type schemes so that different occurrences of variables and patterns can be instantiated to different types. Let $\alpha_1, \ldots, \alpha_k$ be the type variables that are contained in type τ, but not in the assumption A. Then $gen(A, \tau) = \forall \alpha_1, \ldots, \alpha_k.\tau$. Now *gen* is extended in a natural way to assumptions. For

$$A' = \{var_1 \mapsto \tau_1, \ldots, var_n \mapsto \tau_n\}$$

we have

$$gen(A, A') = \{var_1 \mapsto gen(A, \tau_1), \ldots, var_n \mapsto gen(A, \tau_n)\}$$

Note also in rules LET$_\vdash$ and LETP$_\vdash$ that using assumptions $A + A'$ in the inference of *exp'*, respectively *match*, accounts for recursive value and pattern definitions. Apart from this, A' is needed in the inference of *match* to provide types for its free variables. However, A' is not used in the inference of *exp* in rule LETP$_\vdash$, to ensure that free variables of *match* are defined within *exp*; only the generalized type of the active pattern's constructor might be used. In addition to this type the interface (*atpat*) is also put into the assumption to enable the identification of variables in free positions within the rule $\text{ACTIVE}'_{\vdash}$. This is done as follows: variables in the actual pattern *atpat* that are in free position with respect to the interface *atpat′* are determined by a function F defined by:

$$\begin{array}{lcl} F(var', var) & = & \{var\} \\ F(con\ atpat', con\ atpat) & = & F(atpat', atpat) \\ F((pat'_1, \ldots, pat'_n), (pat_1, \ldots, pat_n)) & = & \cup_{i=1}^{n} F(pat_i, pat'_i) \end{array}$$

In all other cases, $F(pat, pat') = \emptyset$. The assumption A' is then restricted to those variables not contained in this set. This is needed to reject definitions, such as `fun foo (Add' (x,_)) = x`, which would lead to a runtime error because `x` is undefined.

$$\text{WILD}'_{\vdash}\ \frac{}{A \vdash \_ \hookrightarrow (\{\}, \tau)} \qquad \text{VAR}'_{\vdash}\ \frac{}{A \vdash var \hookrightarrow (\{var \mapsto \tau\}, \tau)}$$

$$\text{CON}'_{\vdash}\ \frac{A(con) \succ \tau}{A \vdash con \hookrightarrow (\{\}, \tau)}$$

$$\text{TUP}'_{\vdash}\ \frac{A \vdash pat_1 \hookrightarrow (A_1, \tau_1) \quad \ldots \quad A \vdash pat_n \hookrightarrow (A_n, \tau_n)}{A \vdash (pat_1, \ldots, pat_n) \hookrightarrow (A_1 + \ldots + A_n, (\tau_1, \ldots, \tau_n))}$$

$$\text{ACTIVE}'_{\vdash}\ \frac{A(con) = (\tau', atpat') \qquad \tau' \succ \tau'' \to \tau \qquad A \vdash atpat \hookrightarrow (A', \tau'')}{A \vdash con\ atpat \hookrightarrow (A'|_{dom(A') - F(atpat', atpat)}, \tau)}$$

$$\text{CPAT}'_{\vdash}\ \frac{A(con) \succ \tau' \to \tau \qquad A \vdash atpat \hookrightarrow (A', \tau')}{A \vdash con\ atpat \hookrightarrow (A', \tau)}$$

Fig. 4. Derivation of Assumptions.

As already mentioned, within static elaboration we can actually do a bit more for active patterns than just type checking: we know that all rules of a pattern's match have to build a term of exactly the same shape (which additionally has to match the pattern's interface). For example, each case in the function of `Add'` – actually, there is just one case – has to return a term with an outermost `Add`-constructor; an `Empty`-constructor, which would be legal according to the type system, makes no sense here and should be prevented.

We can formalize this *constructor checking* by re-using the above type system in a slightly modified way: we simply assume for each constructor *con* the "type" *con* instead of the associated data type (resolving name clashes between constructor and type names appropriately), that is, rule $\text{CON}_{\vdash}$ changes to

$$\frac{A(con) = con}{A \vdash con :: con}$$

For example, whereas in the type system `true` and `false` would be both mapped to the type `bool`, the modified "constructor system" infers for `true` the constructor `true` and for `false` the constructor `false`. Now, within this modified rule system the sentence *exp* :: *con* states that the outermost constructor of expression *exp* is *con*.

Constructor checking actually has two applications: first, we can strengthen the type checking of active patterns by enforcing equal constructors in all rules of the pattern function. This is reflected by the adding the following precondition to rule LETP$_{\vdash}$:

$$\{\,\} \vdash \mathit{match} :: \tau' \to \mathit{con}$$

This means that in addition to proving a function type for *match* we must also be able to infer a corresponding constructor. (The name *con* is chosen to distinguish the constructor from con', which is the name of the active pattern.)

The second application of constructor checking is found in the transformation of active patterns: there we have to refer to a pattern's associated constructor (for example, `Add` is the associated constructor of `Add'`). This can be expressed as follows. If con' is defined by the expression `let pat` con' *atpat*=*match* `in` *exp* `end`, the associated constructor of con' is *con* if $\{\,\} \vdash \mathit{match} :: \tau' \to \mathit{con}$.

Though described here as a separate phase, in practice constructor checking should be integrated with type checking.

3.2 Operational semantics

An expression is evaluated call-by-value relative to an environment E and reduces to a value which is either a constant, a tuple of values, or a closure. A closure is a triple $\mathit{clos} = (\mathit{match}, E, E')$ where E captures the environment at the time of the definition of *match* and E' contains definitions for variables that are bound to recursive functions. Terms (values of data types) are represented by pairs (constructor, argument).

Moreover, we need a value representation for evaluated active patterns since in addition to the pattern function (which reduces to a closure) we need the interface in the evaluation, too. This is used when in an application of an active pattern an expression *exp* appears at the position of a free variable *var*. Then we must use the additional binding $\{\mathit{var} \mapsto v\}$ (v being the value of *exp*) in the evaluation of the active pattern. In order to identify the free variable positions we thus need the interface when applying active patterns.

A pattern is evaluated against a value and yields an environment giving bindings for its variables. However, this is only true if the pattern really matches the value, otherwise the special object FAIL is returned. FAIL is not a value, it is a semantic object that controls the order of pattern matching. FAIL propagates to rule MATCH$_{\Rightarrow}$ expressing that when a rule successfully matches, then the obtained result is the result of the whole match and if a rule does not match, that is, when it yields FAIL, the remaining match is evaluated. If FAIL is not caught eventually by rule MATCH$_{\Rightarrow}$, this is left an undefined situation by our semantics and can be considered a runtime error. (In the semantics of Standard ML [MTH90], a match exception is generated.) In the semantic rules we make use of what we will call the "FAIL-assumption": if any precondition of a rule yields FAIL, then the conclusion also yields FAIL. Places where FAIL might occur are indicated by appending "/FAIL" to the usually expected object. The semantics is given in Figure 5.

$$\text{CON}_{\Rightarrow} \quad \frac{}{E \vdash con \Rightarrow con} \qquad \text{TUP}_{\Rightarrow} \quad \frac{E \vdash exp_1 \Rightarrow v_1 \quad \ldots \quad E \vdash exp_n \Rightarrow v_n}{E \vdash (exp_1, \ldots, exp_n) \Rightarrow (v_1, \ldots, v_n)}$$

$$\text{VAR}_{\Rightarrow} \quad \frac{E(var) = v}{E \vdash var \Rightarrow v} \qquad \text{ABS}_{\Rightarrow} \quad \frac{}{E \vdash \texttt{fn}\ match \Rightarrow (match, E, \{\})}$$

$$\text{APP}_{\Rightarrow} \quad \frac{E \vdash exp \Rightarrow (match, E', E'') \qquad E \vdash exp' \Rightarrow v' \qquad E' + \operatorname{rec} E'', v' \vdash match \Rightarrow v}{E \vdash exp\ exp' \Rightarrow v}$$

$$\frac{E \vdash exp \Rightarrow con \qquad E \vdash exp' \Rightarrow v}{E \vdash exp\ exp' \Rightarrow (con, v)}$$

$$\text{LET}_{\Rightarrow} \quad \frac{E \vdash exp' \Rightarrow v' \qquad E, v' \vdash pat \Rightarrow E' \qquad E + \operatorname{rec} E' \vdash exp \Rightarrow v}{E \vdash \texttt{let val}\ pat\texttt{=}exp'\ \texttt{in}\ exp\ \texttt{end} \Rightarrow v}$$

$$\text{LETP}_{\Rightarrow} \quad \frac{E + \operatorname{rec}\{con' \mapsto (atpat, (match, E, \{\}))\} \vdash exp \Rightarrow v}{E \vdash \texttt{let pat}\ con'\ atpat\texttt{=}match\ \texttt{in}\ exp\ \texttt{end} \Rightarrow v}$$

$$\text{MATCH}_{\Rightarrow} \quad \frac{E, v \vdash rule \Rightarrow v'}{E, v \vdash rule\ \texttt{|}\ match \Rightarrow v'} \qquad \frac{E, v \vdash rule \Rightarrow \text{FAIL} \qquad E, v \vdash match \Rightarrow v'}{E, v \vdash rule\ \texttt{|}\ match \Rightarrow v'}$$

$$\text{RULE}_{\Rightarrow} \quad \frac{E, v \vdash pat \Rightarrow E'/\text{FAIL} \qquad E + E' \vdash exp \Rightarrow v'}{E, v \vdash pat\ \texttt{=>}\ exp \Rightarrow v'/\text{FAIL}}$$

Fig. 5. Operational Semantics.

The operator rec employed in rules $\text{APP}_{\Rightarrow}$ and $\text{LET}_{\Rightarrow}$ performs a one-step unfolding of recursive function definitions, that is, with

$$\operatorname{rec}(E, v) \;=\; \begin{cases} (match, E', E) & \text{if } v = (match, E', E'') \\ (pat, (match, E', E)) & \text{if } v = (pat, (match, E', E'')) \\ v & \text{otherwise} \end{cases}$$

we obtain for an environment $E = \{var_1 \mapsto v_1, \ldots, var_n \mapsto v_n\}$ the collection of unrolled definitions $\operatorname{rec} E = \{var_1 \mapsto \operatorname{rec}(E, v_1), \ldots, var_n \mapsto \operatorname{rec}(E, v_n)\}$.

In rule $\text{LETP}_{\Rightarrow}$ the interface and function definition of the active pattern are pushed into the environment in which expression *exp* is to be evaluated. This definition is used in rule $\text{ACTIVE}'_{\Rightarrow}$ (see below).

The pattern matching semantics are given separately in Figure 6. The last three rules describe the matching of patterns against constructor terms (data type values). Rule $\text{ACTIVE}'_{\Rightarrow}$ describes the essence of active pattern matching: to match an active pattern $con'\ atpat$ against a value v the pattern function *match* obtained from the environment E (1st precondition) is used to transform v into a term (con, v') (3rd precondition). However, before *match* can be applied, we must ensure that bindings exist for all free variables of *match*. This is done by matching the actual "pattern call" *atpat* against the interface of con' (2nd precondition). To describe this we need another rule system defining the matching of patterns against patterns, see Figure 7. Note that matching the constructor of v, which happens to be *con*, is done during the matching of *match*. Thus v' denotes just the argument of the rearranged term. Finally, the variables of *atpat* are bound by matching *atpat* against v' (last precondition). Rule $\mathrm{CPAT}'_{\Rightarrow}$ applies whenever *con* is not defined in the environment E. In that case the constructor of the term to be matched must be the same as in the pattern, otherwise matching fails.

$$\mathrm{WILD}'_{\Rightarrow}\quad \frac{}{E, v \vdash \_ \Rightarrow \{\,\}} \qquad\qquad \mathrm{VAR}'_{\Rightarrow}\quad \frac{}{E, v \vdash var \Rightarrow \{var \mapsto v\}}$$

$$\mathrm{CON}'_{\Rightarrow}\quad \frac{}{E, con \vdash con \Rightarrow \{\,\}} \qquad\qquad \frac{con' \neq con}{E, con' \vdash con \Rightarrow \mathrm{FAIL}}$$

$$\mathrm{TUP}'_{\Rightarrow}\quad \frac{E, v_1 \vdash pat_1 \Rightarrow E_1/\mathrm{FAIL} \qquad \ldots \qquad E, v_n \vdash pat_n \Rightarrow E_n/\mathrm{FAIL}}{E, (v_1, \ldots, v_n) \vdash (pat_1, \ldots, pat_n) \Rightarrow E_1 + \ldots + E_n/\mathrm{FAIL}}$$

$$\mathrm{ACTIVE}'_{\Rightarrow}\quad \frac{\begin{array}{c} E(con') = (atpat', (match, E_1, E_2)) \qquad E, atpat \vdash atpat' \Rightarrow E_P \\ E + E_1 + \mathrm{rec}\, E_2 + E_P, v \vdash match \Rightarrow (con, v')/\mathrm{FAIL} \\ E, v' \vdash atpat \Rightarrow E'/\mathrm{FAIL} \end{array}}{E, v \vdash con'\ atpat \Rightarrow E'/\mathrm{FAIL}}$$

$$\mathrm{CPAT}'_{\Rightarrow}\quad \frac{con \notin dom(E) \qquad E, v \vdash atpat \Rightarrow E'/\mathrm{FAIL}}{E, (con, v) \vdash con\ atpat \Rightarrow E'/\mathrm{FAIL}}$$

$$\frac{con \notin dom(E) \qquad con \neq con'}{E, (con', v) \vdash con\ atpat \Rightarrow \mathrm{FAIL}}$$

Fig. 6. Pattern Matching Semantics.

When matching patterns against patterns the main case is matching a variable *var* (of the interface) against another pattern *pat*: rule $\mathrm{VAR}''_{\Rightarrow}$ expresses that this results in a binding $\{var \mapsto v\}$ whenever *pat* reduces to value v. This is well-defined since, disregarding the wildcard, patterns make up a subset of expressions. (Formally, we need yet another rule system defining evaluation of such patterns, but actually this is perfectly done by the semantics of Figure 5.) The meaning of the remaining rules are: matching a wildcard does not give any binding ($\mathrm{WILD}''_{\Rightarrow}$), and recursive application for tuples ($\mathrm{TUP}''_{\Rightarrow}$) and terms ($\mathrm{CPAT}''_{\Rightarrow}$).

4 Implementation

In the following we define a function $\mathcal{T}$ that transforms an expression possibly containing active patterns into a semantically equivalent expression without

$$\text{WILD}''_{\Rightarrow}\ \frac{}{E, pat \vdash \_ \Rightarrow \{\}} \qquad \text{VAR}''_{\Rightarrow}\ \frac{pat \neq \_ \qquad E \vdash pat \Rightarrow v}{E, pat \vdash var \Rightarrow \{var \mapsto v\}}$$

$$\text{TUP}''_{\Rightarrow}\ \frac{E, pat_1 \vdash pat'_1 \Rightarrow E_1 \quad \ldots \quad E, pat_n \vdash pat'_n \Rightarrow E_n}{E, (pat_1, \ldots, pat_n) \vdash (pat'_1, \ldots, pat'_n) \Rightarrow E_1 + \ldots + E_n}$$

$$\text{CPAT}''_{\Rightarrow}\ \frac{E, atpat \vdash atpat' \Rightarrow E'}{E, con\ atpat \vdash con\ atpat' \Rightarrow E'}$$

Fig. 7. Matching Pattern Calls Against Interfaces.

active patterns.

The idea is to replace active patterns by functions that perform the desired rearrangement for matching values and leave other values unchanged. Uses of active patterns are then replaced by applications of these functions. Consider the following pattern definition

`let pat` *con′ atpat*=*match*

where $var_1, \ldots, var_n$ is the sequence of free variables of *atpat* (obtained by a preorder traversal). Then

$$\mathcal{P}(atpat) = \texttt{fn}\ var_1\ \texttt{=>}\ \ldots\ \texttt{=>}\ \texttt{fn}\ var_n\ \texttt{=>}$$

denotes the additional parameters of the pattern function that are needed in the definition of the corresponding function. The definition is:

`let val` *con′*=$\mathcal{P}(atpat)$ `fn` *match* `| x => x`

Since *match* is only defined for one constructor of a data type we append a rule realizing an identity mapping to allow non-matching values to pass *con′* unchanged. The need for this can be seen as follows: whereas a non-matching active pattern simply moves evaluation to the next case of the function definition, an application of the corresponding function would be undefined.

Active patterns can be used in two ways: (i) on the LHS of a rule, and (ii) in value definitions. When used in a rule

con′ atpat `=>` *exp*

the LHS is replaced by a fresh variable $\overline{var}$ that does not yet occur anywhere in the currently transformed expression. This variable matches any value, say v. The transformation of the value and the pattern matching both happens on the RHS: first, the newly defined function *con′* is applied to $\overline{var}$ (which is bound to v) and possibly additional arguments. If v matches the data type case of *con′*, it is transformed into the representation v', otherwise v stays unchanged. After that the actual pattern matching takes place. In the first case, v' must be matched against the (non-active) pattern *con atpat*, and *exp* is returned.

Otherwise, no rearrangement did happen, and the match following the current rule must be applied to v. We can achieve this behavior by a function consisting of a rule for pattern *con atpat* and the rest of the current match. Thus, as a first approximation, the transformation of the rule gives something like:

$\overline{var}$ `=> (fn` *con atpat* `=>` $\langle 1 \rangle$ $\langle 2 \rangle$`)` `(`*con'* $\langle 3 \rangle$ *var*`)`

$con = \mathcal{C}(con')$ denotes the constructor associated with the active pattern. We assume that the function $\mathcal{C}$ has been determined by the type/constructor checking phase as described in Section 3.1.

In the transformed expression, $\langle 1 \rangle$ denotes the transformation of *exp*, and $\langle 2 \rangle$ stands for the transformation of the remaining match, which itself is an abstraction applied to *var*. Finally, $\langle 3 \rangle$ denotes the additional arguments for the free variables of *con'*'s pattern interface. These values are given by subexpressions of the currently transformed rule's LHS. More precisely, each parameter var_i gets its argument from the expression exp_i that occurs (in the rule's LHS) in the same position as var_i in the pattern interface. We can compute all arguments in the correct order by performing a parallel preorder traversal of the pattern interface and *atpat*: whenever we encounter a free variable in the interface, we have found in *atpat* the corresponding expression. Let $\mathcal{I}(con')$ denote the interface of the active pattern constructor *con'* (like $\mathcal{C}$, $\mathcal{I}$ is also given by static elaboration). Then the function $\mathcal{A}$ computes the sequence of argument expressions with respect to a pattern interface as follows (ε denotes the empty word).

$$\begin{array}{lcl}
\mathcal{A}(\_, exp) & = & \varepsilon \\
\mathcal{A}(var, exp) & = & exp \\
\mathcal{A}((pat_1, \ldots, pat_n), (exp_1, \ldots, exp_n)) & = & \mathcal{A}(pat_1, exp_1) \;\ldots\; \mathcal{A}(pat_n, exp_n) \\
\mathcal{A}(con\ atpat, con\ exp) & = & \mathcal{A}(atpat, exp)
\end{array}$$

Thus, the missing arguments are given by the expression $\mathcal{A}(\mathcal{I}(con'), atpat)$.

Finally, consider the transformation of a value declaration containing an active pattern, such as

`let val` *con' atpat*=*exp'* `in` *exp* `end`

The rearrangement of the active pattern happens by applying *con'* to $\mathcal{A}(\mathcal{I}(con'), atpat)$ (giving the arguments for the additional parameters) and to the translation of *exp'*. Again, *exp'* might not match *con*, in which case it should pass unchanged. So we again transform the value declaration into a function application with two cases: one for the pattern *con atpat* and one for other values.

The complete translation algorithm is shown in Figure 8. The angle brackets in the translation of rules say that the enclosed code is only produced if *match* is not empty, that is, consists of one or more rules.

A possible optimization is to group a sequence of function equations with the same active pattern *con'* together to avoid possibly repeated calls to the function *con'*. To illustrate the algorithm, we transform the definition of `member`. We apply

$$
\begin{aligned}
\mathcal{T}\llbracket con \rrbracket &= con\\
\mathcal{T}\llbracket var \rrbracket &= var\\
\mathcal{T}\llbracket (exp_1,\ldots,exp_n) \rrbracket &= (\mathcal{T}\llbracket exp_1\rrbracket,\ldots,\mathcal{T}\llbracket exp_n\rrbracket)\\
\mathcal{T}\llbracket exp\ exp' \rrbracket &= \mathcal{T}\llbracket exp\rrbracket\ \mathcal{T}\llbracket exp'\rrbracket\\
\mathcal{T}\llbracket \texttt{fn}\ match \rrbracket &= \texttt{fn}\ \mathcal{T}\llbracket match\rrbracket
\end{aligned}
$$

$$
\mathcal{T}\llbracket pat\ \texttt{=>}\ exp\ \langle\ \texttt{|}\ \ match\rangle \rrbracket\ =
$$

$$
\begin{cases}
\begin{array}{l}
\overline{var}\ \texttt{=>}\ \mathcal{T}\llbracket \texttt{(fn}\ con\ atpat\ \texttt{=>}\ exp \\
\qquad\qquad \langle\ \texttt{|}\ \texttt{\_}\ \texttt{=>}\ \texttt{(fn}\ match\texttt{)}\ \ var\,\rangle\texttt{)}\rrbracket \\
\qquad\qquad\quad \texttt{(}con'\ \mathcal{A}(\mathcal{I}(con'), atpat)\ var\texttt{)}
\end{array} & \begin{array}{l}\text{if } pat = con'\ atpat\\ \quad \wedge\ \mathcal{C}(con') = con\end{array}\\[2ex]
pat\ \texttt{=>}\ \mathcal{T}\llbracket exp\rrbracket\ \langle\ \texttt{|}\ \ \mathcal{T}\llbracket match\rrbracket\,\rangle & \text{otherwise}
\end{cases}
$$

$$
\mathcal{T}\llbracket \texttt{let val}\ pat\texttt{=}exp'\ \texttt{in}\ exp\ \texttt{end}\rrbracket\ =
$$

$$
\begin{cases}
\begin{array}{l}
\texttt{let val}\ con\ atpat\texttt{=}con'\ \mathcal{A}(\mathcal{I}(con'), atpat)\ \mathcal{T}\llbracket exp'\rrbracket \\
\ \ \texttt{in}\ \mathcal{T}\llbracket exp\rrbracket\ \texttt{end}
\end{array} & \begin{array}{l}\text{if } pat = con'\ atpat\\ \quad \wedge\ \mathcal{C}(con') = con\end{array}\\[2ex]
\texttt{let val}\ pat\texttt{=}\mathcal{T}\llbracket exp'\rrbracket\ \texttt{in}\ \mathcal{T}\llbracket exp\rrbracket\ \texttt{end} & \text{otherwise}
\end{cases}
$$

$$
\begin{array}{l}
\mathcal{T}\llbracket \texttt{let pat}\ con'\ \ atpat\texttt{=}match\ \texttt{in}\ exp\ \texttt{end}\rrbracket\ \ =\\
\quad \texttt{let val}\ con'\texttt{=}\mathcal{P}(atpat)\ \texttt{fn}\ \mathcal{T}\llbracket match\rrbracket\ \texttt{|}\ \texttt{x => x in}\ \mathcal{T}\llbracket exp\rrbracket\ \texttt{end}
\end{array}
$$

Fig. 8. Transformation of Active Patterns.

$\mathcal{T}$ to the following expression.

```
let pat Add' (x,_)=add in
let val member=memb
 in exp end end
```

where *add* and *memb* represent the parts of the corresponding expression from the beginning of Section 3.

In the first step we obtain the expression:

$$
\begin{array}{l}
\texttt{let val Add'=}\mathcal{P}\texttt{((x,\_))}\ \texttt{fn}\ \mathcal{T}\llbracket add\rrbracket\ \texttt{| x => x}\\
\ \texttt{in}\ \mathcal{T}\llbracket \texttt{let val member=}memb\ \texttt{in}\ exp\ \texttt{end}\rrbracket\ \texttt{end}
\end{array}
$$

which immediately transforms to:

$$
\begin{array}{l}
\texttt{let val Add'=fn x => fn}\ \mathcal{T}\llbracket add\rrbracket\ \texttt{| x => x}\\
\ \texttt{in let val member=}\mathcal{T}\llbracket memb\rrbracket\\
\ \texttt{in}\ \mathcal{T}\llbracket exp\rrbracket\ \texttt{end end}
\end{array}
$$

Next we consider the translation of *add* and *memb* separately. Since the LHS of match *add* is not an active pattern $\mathcal{T}\llbracket add\rrbracket$ gives:

```
Add (y,s) => 𝒯⟦ if x=y then Add (y,s)
                      else let val Add' (x,t)=s
                            in Add (x,Add (y,t)) end ⟧
```

which is

```
Add (y,s) => if x=y then Add (y,s)
                    else 𝒯⟦ let val Add' (x,t)=s
                            in Add (x,Add (y,t)) end ⟧
```

With $\mathcal{C}($`Add'`$) =$ `Add`, $\mathcal{I}($`Add'`$) =$ `(x,_)`, and $\mathcal{A}($`(x,_)`$,$ `(x,t)`$) =$ `x` we obtain:

```
let val Add (x,t) = Add' x s in Add (x,Add (y,t)) end
```

In the translation of *memb* the first abstraction over `x` is passed unchanged by $\mathcal{T}$. We get the expression:

```
fn x => fn 𝒯⟦(Add' (x,s)) => true | s => false⟧
```

Now we let `z` be a fresh variable, and with $\mathcal{A}($`(x,_)`$,$ `(x,s)`$) =$ `x` we obtain:

```
fn x => fn z =>
   (fn Add (x,s) => true | _ => (fn s => false) z) (Add' x z)
```

Summarizing, the complete translation is:

```
let val Add'=fn x =>
        fn Add (y,s) => if x=y then Add (y,s) else
                          let val Add (x,t) = Add' x s
                           in Add (x,Add (y,t)) end
         | x         => x
 in
let val member=fn x => fn z =>
        (fn Add (x,s) => true
          | _         => (fn s=>false) z) (Add' x z)
 in 𝒯⟦exp⟧ end end
```

We recognize that the resulting function for `member` is not very efficient because of the terms that are unnecessarily built by `Add'`: in the worst case (when `x` is the last element), the set representing list is duplicated. This is not surprising since we use a rather general rearrangement function (`Add'`) which has to build a reorganized value in the computation of a specific task where that value itself is never needed. Is there a way to improve the function definitions so to avoid unnecessary computations? Immediately, the *deforestation* fusion technique of Wadler [Wad90] comes to mind. However, deforestation cannot be applied since the definition of `Add'` is not "treeless" (due to the term-producing `Add'`-call being an argument of another function call). The method of Chin [Chi92] is applicable, but it seems to ignore `Add'` during optimization since the definition of `Add'` produces function terms.

Still in search of the appropriate technique to apply we shortly sketch another fusion scheme that is tailored for the optimization of active patterns. (What follows is still under development, and some restrictions are expected to be dropped in future.) Consider the definition of a function f containing one rule with an active pattern con' *atpat* `=>` *exp*. The translation resulting from $\mathcal{T}$ gives something like

```
(fn con atpat => exp | ...) (con' ...)
```

The goal is to fuse the definition of f for this case with the definition of the function con' to eliminate any computations for variables in the definition of con' that are not used in *exp*. This can be achieved by the following three fusion rules (in all other cases fusion is recursively applied to sub-expressions). A general precondition for the method is that there is at most one variable shared by *atpat* and *exp*, that is, only one part of the term computed by con' is actually used within *exp*. This prevents the introduction of repeated computations.

(F1) Replace "global" terms[5] *con* exp' in con' by a properly instantiated version of *exp*[6]
(F2) Replace recursive applications of con' by f.
(F3) Replace patterns *con pat* in anonymous functions (resulting from the translation of `let val`-expressions) by patterns matching the results of *exp*.

The last rule deserves some comments: The need for replacing function patterns can be seen as follows: a *con*-pattern catches the result of a con'-call. Since con'-applications are replaced by f the cases in function patterns have to be adapted. In particular, since the pattern occurs in that case of f's definition where the active pattern was used, the corresponding result expression *exp* is relevant here. Now, if *exp* is a constant, say c, this is obviously the pattern to be substituted. More general, if we can determine that *exp* will return any constant of $c_1, \ldots, c_n$, we can replace the rule by a set of rules, each for one constant c_i. If *exp* is a variable *var*, the pattern is replaced by *var*. All other cases are not completely clear at the moment (except a straightforward extension to tuples), and we abort the fusion process in those cases.

Let us illustrate the fusion rules in the optimization of `member`. $\mathcal{F}[\![exp]\!]$ denotes the application of the fusion algorithm to expression *exp*. An implicit argument is the function definition being actually fused; this should be clear from the context. Recall the $\mathcal{T}$-translation of `Add'` from above. Fusion moves over abstraction and conditional, so we get the expression:

[5] These are terms that can be returned as a result of con'.
[6] Each (local) variable in *exp* is substituted by the matching sub-expression of exp', in the sense of matching *con* exp' against *con atpat* as defined in Figure 6.

```
fn x => fn Add (y,s) =>
           if x=y then 𝓕⟦Add (y,s)⟧ else
              (fn 𝓕⟦Add (x,t)⟧ => 𝓕⟦Add (x,Add (y,t))⟧
                | x              => x) 𝓕⟦Add' x s⟧
         | x         => x
```

For `member` we obtain the following fused sub-expressions/patterns:

$$\begin{array}{lcll}
\mathcal{F}[\![\texttt{Add (y,s)}]\!] & = & \texttt{true} & \text{(F1)} \\
\mathcal{F}[\![\texttt{Add (x,t)}]\!] & = & \texttt{true} & \text{(F3)} \\
\mathcal{F}[\![\texttt{Add (x,Add (y,t))}]\!] & = & \texttt{true} & \text{(F1)} \\
\mathcal{F}[\![\texttt{Add' x s}]\!] & = & \texttt{member x s} & \text{(F2)}
\end{array}$$

This results in the function:

```
fn x => fn Add (y,s) => if x=y then true else
                           (fn true => true
                             | x    => x) member x s
         | x         => x
```

We see that no intermediate `Add`-terms are constructed. With some algebraic postprocessing we can arrive at the original `member`-definition without active patterns.

Currently, the presented fusion method is rather crude and somewhat limited. Algebraic transformations are needed to get concise definitions, and functions like `insert` cannot be optimized at all.

5 Related Work

Pattern matching with unfree data types has been addressed by Miranda *laws* [Tur85, Tho90], Wadler's *views* [Wad87], or the recent work of Burton and Cameron [BC93]. Common to all approaches is the mapping of equal terms to a canonical representation.

The demand for a canonical representation is a rather strong requirement which often entails a certain overspecification of the data type. For example, implementing sets by lists requires keeping lists in sorted order. This approach has the following two drawbacks:

1. *Overspecification restricts applicability.* Keeping elements sorted requires a comparison function on set elements. In contrast, working with unsorted lists only needs equality on list elements.
2. *Overspecification might need more computation than necessary.* All three set operations need linear time when implemented on base of sorted lists. With unsorted lists, `member` and `delete` are again linear, but `insert` takes just constant time if we allow duplicates.[7]

[7] However, if the number of duplicates is large compared to the size of the represented sets, then `member` and `delete` are no longer linear in the size of the represented set.

In contrast, active patterns do not require a canonical representation and thus offer more freedom in the implementation.

Views and Miranda laws both cause some trouble with equational reasoning because constructors of non-free data types can be used to construct values. This is overcome by the proposal of [BC93] where these constructors may be only used within patterns. We expect the same for active patterns since they too can only be used in patterns.

Context Patterns [Moh96] are intended to give direct access to arbitrary deeply nested sub-parts of terms, but they only work for free data types.

The *abstract value constructors* of [AR92] provide a kind of macro facility to denote terms in a more convenient (and abstract) way. However, no computations in the sense of changing the representation of the matched values are performed. We could cover abstract value constructors by active patterns if we dropped the restriction that the argument type must be the same as the result type. (We would have to adapt the semantics and the implementation.) The intent of active patterns is, however, different: they are meant as a device for abstracting data type laws and not as a macro language.

Finally, the *active destructors* introduced in [PGPN96] are essentially functions that can be used within patterns to produce bindings. Active destructors can perform computations much like active patterns, and they are even more general since they (like abstract value constructors) have no type restriction imposed. In [PGPN96] it is also sketched how active destructors can use external values, but active destructors only work with linear patterns. However, in many cases it is just the combination of external values and non-linear patterns that is needed, namely pushing one function parameter into an active pattern defining another parameter. Thus the set, tree, and graph examples cannot be expressed by active destructors.

6 Conclusions

Active patterns might be considered problematic since some applications require non-linear patterns which are not part of most functional languages. One implication is that the evaluation order of arguments is restricted: before an active pattern can be evaluated, its free variables must be bound. (In ML this is not a problem since the evaluation order is fixed left-to-right.) A related aspect is that parallel pattern matching is generally not possible in languages with imperative features (such as references and exceptions in ML) since the computations of active patterns make the order of pattern matching significant.

On the other hand, active patterns offer a *powerful abstraction* concept: reorganizations of data type values are removed from function definitions and are given in separate pattern definitions. There are two benefits gained from this: first, function definitions become remarkably simple, and second, by "factorizing" the data type laws, the risk of introducing errors in function definitions is reduced since reorganizational expressions do not have to be repeated for each new function definition. The effort needed to integrate active patterns into a functional language depends on the destination language. As far as ML is concerned, this presents no great difficulties: only the parser (and maybe the type checker) need to be extended; thanks to the presented source code transformation, the rest of a language implementation can be left unchanged. Finally, the sketched fusion technique is a first step to obtaining *efficient implementations* of functions defined with active patterns.

Acknowledgements

Thanks to Pedro Palao Gonstanza and John Boyland for their helpful comments on a previous version of this paper.

References

[AR92] W. E. Aitken and J. H. Reppy. Abstract Value Constructors. In *ACM Workshop on ML and its Applications*, pages 1–11, 1992.

[BC93] F. W. Burton and R. D. Cameron. Pattern Matching with Abstract Data Types. *Journal of Functional Programming*, 3(2):171–190, 1993.

[Bur82] F. W. Burton. An Efficient Functional Implementation of FIFO queues. *Information Processing Letters*, 14:205–206, 1982.

[Chi92] W. N. Chin. Safe Fusion of Functional Expressions. In *ACM Conf. on Lisp and Functional Programming*, pages 11–20, 1992.

[Erw97] M. Erwig. Functional Programming with Graphs. In *2nd ACM SIGPLAN Int. Conf. on Functional Programming*, pages 52–65, 1997.

[Moh96] M. Mohnen. Context Patterns in Haskell. In *8th Int. Workshop on Implementation of Functional Languages*, LNCS (this volume), 1996.

[MTH90] R. Milner, M. Tofte, and R. Harper. *The Definition of Standard ML*. MIT Press, Cambridge, MA, 1990.

[PGPN96] P. Palao Gonstanza, R. Peña, and M. Núñez. A New Look at Pattern Matching in Abstract Data Types. In *1st ACM SIGPLAN Int. Conf. on Functional Programming*, pages 110–121, 1996.

[Tho90] S. Thompson. Lawful Functions and Program Verification in Miranda. *Science of Computer Programming*, 13:181–218, 1990.

[Tur85] D. A. Turner. Miranda: A Non-strict Functional Language with Polymorphic Types. In *Conf. on Functional Programming and Computer Architecture*, LNCS 201, pages 1–16, 1985.

[Wad87] P. Wadler. Views: A Way for Pattern Matching to Cohabit with Data Abstraction. In *ACM Symp. on Principles of Programming Languages*, pages 307–313, 1987.

[Wad90] P. Wadler. Deforestation: Transforming Programs to Eliminate Trees. *Theoretical Computer Science*, 73:231–284, 1990.

This article was processed using the LaTeX macro package with LLNCS style