Views for Standard ML

Chris Okasaki*
Columbia University
`cdo@cs.columbia.edu`

Abstract

In Standard ML, as in many other languages, programmers are often confronted with an unpleasant choice between pattern matching and abstraction. Because pattern matching can only be performed on concrete datatypes, programmers must often sacrifice either the convenience of pattern matching or the engineering benefits of abstraction. *Views* relieve this tension by allowing pattern matching on abstract datatypes. We propose a modest extension of Standard ML with views and define its semantics via a source-to-source translation back into Standard ML without views. We claim no particular technical innovation; rather, we have attempted to engineer a solution that blends as seamlessly as possible with the rest of the language, including the module system and the stateful features of the language.

1 Introduction

The convenience of pattern matching is one of the most seductive aspects of languages like Standard ML [6]. Not until the newcomer tries to write large programs does she encounter the Achilles' heel of this language feature: it can only be performed on concrete datatypes. Our intrepid programmer is faced with a dilemma worthy of Hobson. She can have the convenience of pattern matching, or the engineering benefits of abstraction, but not both. She can choose to hide the internal representations of her datatypes, usually resulting in code littered with predicates such as `null` and selection functions such as `head` and `tail`. Or she can expose these internal representations, allowing pattern matching on these datatypes but also allowing clients to construct possibly ill-formed values and making changes to her datatypes a nightmare.

Views offer a way out of this dilemma. They expose an abstract representation of a datatype to clients and allow pattern matching on that abstract representation, without making any commitment to a particular concrete representation. Furthermore, the exposed representation can be used for pattern matching only, not for the construction of values, so a client cannot create his own values using the exposed representation.

We begin by illustrating the features of our proposal through a series of examples, paying particular attention to the behavior of views in the presence of stateful computations. Next, we formally define our language extension via a source-to-source translation back into Standard ML without views. Finally, we discuss related work and conclude.

2 Examples

As an example of the use of views, consider a sequence ADT defined by the signature in Figure 1. The signature declares an abstract type

*Supported during the summer of 1998 by the Department of Computing Science at the University of Glasgow, with funds from the Scottish Higher Education Development Council.

```
signature SEQUENCE =
sig
   type 'a Seq
   viewtype 'a Seq = Empty | Cons of 'a * 'a Seq
   val empty  : 'a Seq
   val cons   : 'a * 'a Seq -> 'a Seq
   val append : 'a Seq * 'a Seq -> 'a Seq
end
```

Figure 1: A signature including a viewtype.

```
structure MyList :> SEQUENCE =
struct
   datatype 'a Seq = Empty | Cons of 'a * 'a Seq
   val empty = Empty
   val cons = Cons
   fun append (Empty, ys) = ys
     | append (Cons (x, xs), ys) = Cons (x, append (xs, ys))
end
```

Figure 2: An example where the viewtype in the signature is matched by a datatype.

```
type 'a Seq
```

and then declares a view over that type

```
viewtype 'a Seq = Empty | Cons of 'a * 'a Seq
```

This view states that values of the existing type `'a Seq` can be pattern matched as if they were built with the constructors `Empty` and `Cons`. A client of such a sequence module might thus write code such as

```
fun map f Empty = empty
  | map f (Cons (x,xs)) = cons (f x,map f xs)
```

Of course, a real sequence implementation would include its own `map` function, but this example shows that the convenient style of defining functions by pattern matching has been retained, even though the concrete representation is now hidden. The only difference between writing this map function over the usual concrete type versus over the abstract type is that the values `empty` and `cons` are used on the right-hand side to construct new sequences, rather than the constructors `Empty` and `Cons`.

The sequence implementation in Figure 2 shows that a datatype can be partially exported as a viewtype when the constructors of the datatype and the viewtype match. This has the very important benefit that clients of this module can use the constructors `Empty` and `Cons` only in patterns, not to construct new sequences. If the client wishes to construct a sequence, he must use the values that have been exported for that purpose: `empty`, `cons`, and `append`.

The next implementation, in Figure 3, shows that the concrete representation of a type can be completely different from its viewtype. The programmer must then include with the viewtype definition a function—called the *view transformation*—mapping the concrete representation to the abstract representation.

```
viewtype 'a Seq = Empty | Cons of 'a * 'a Seq
  where Nil => Empty
      | One x => Cons (x, Nil)
      | App (Cons (x, xs), ys) => Cons (x, append (xs, ys))
```

```
structure JoinList :> SEQUENCE =
struct
   datatype 'a Seq = Nil | One of 'a | App of 'a Seq * 'a Seq
     (* Invariant: Nil never child of App *)
   val empty = Nil
   fun cons (x, xs) = append (One x, xs)
   fun append (xs, Nil) = xs
     | append (Nil, ys) = ys
     | append (xs, ys)  = App (xs, ys)
   viewtype 'a Seq = Empty | Cons of 'a * 'a Seq
     where Nil => Empty
         | One x => Cons (x, Nil)
         | App (Cons (x, xs), ys) => Cons (x, append (xs, ys))
end
```

Figure 3: An implementation of sequences with a non-trivial viewtype transformation.

The view transformation is implicitly invoked whenever constructors of the viewtype appear in a pattern. Within the body of the transformation, the constructors of the viewtype are used to construct values of the abstract type. This is the only place where these constructors can appear in an expression, rather than a pattern. Note that, even though this function is anonymous, it can be invoked recursively by pattern matching against one of the view constructors, as in the `App` clause.

The `One` clause also illustrates an interesting point. It would be easy to mistakenly write the right-hand side of that clause as `Cons (x, Empty)` rather than `Cons (x, Nil)`. However, this mistake would be caught by the typechecker since the second argument of `Cons` must have type `'a Seq`, and `Empty` does not have this type! Although the pattern `Empty` can be treated as if it has type `'a Seq`, the expression `Empty` has some deliberately unnamed type that exists only within the view transformation.

Figure 4 illustrates most of the remaining features of views. The most interesting of these is the last: views are allowed to match less polymorphic signatures. In this regard, views are more like functions than datatypes.

3 Memoization of View Transformations

In Standard ML, functions can both depend on and affect the current state. Therefore, it is crucial to define exactly when each view transformation is called. We defer the formal definition to Section 4, but discuss some of the options here.

Consider the following views:

```
viewtype int = C of int
viewtype int = D of int
```

where the view transformations for `C` and `D` are functions that may manipulate state.

We begin with an easy decision. In the expression

```
case e of 0 => 0
        | C 1 => 1
        | D x => x
```

when should the view transformations for `C` and `D` be called? Although other choices are conceivable, such as calling both at the beginning of the match, the only reasonable choice is to call the view transformation for `C` if and when the match against `0` fails, and to call the view transformation for `D` if and when the match against `C 1` fails.

- A single type may have multiple views.

```
viewtype 'a Seq = EmptyC | Cons of 'a * 'a Seq
viewtype 'a Seq = EmptyS | Snoc of 'a Seq * 'a
```

- Views may be written for arbitrary types.

```
viewtype 'a = And of 'a * 'a
  where x => And (x,x)

viewtype ''a * ''a = Same of ''a | Diff of ''a * ''a
  where (x,y) => if x = y then Same x else Diff (x,y)

viewtype int * int = MinMax of int * int
  where (x,y) => if x < y then MinMax (x,y) else MinMax (y,x)
```

- View transformations may be mutually recursive with ordinary functions.

```
viewtype 'a list = Last of 'a | NoLast
  where [] => NoLast
      | xs => Last (last xs)

withfun last [] = raise LastExn
      | last (x :: NoLast) = x
      | last (x :: Last y) = y
```

- View transformations may be mutually recursive with other view transformations.

```
viewtype 'a Seq = LastEven of 'a | NoLastEven
  where Empty => NoLastEven
      | Cons (x, NoLastOdd) => LastEven x
      | Cons (x, LastOdd y) => LastEven y

and 'a Seq = LastOdd of 'a | NoLastOdd
  where Cons (x, LastEven y) => LastOdd y
      | _ => NoLastOdd
```

- Views can match less polymorphic signatures.

```
structure S : sig viewtype int list = Head of int | NoHead end =
struct
  viewtype 'a list = Head of 'a | NoHead
    where (x::xs) => Head x
        | [] => NoHead
end
```

Figure 4: More examples of views.

Now, in the expression

```
case e of C 0 => 0
        | C 1 => 1
        | D 2 => 2
        | D 3 => 3
        | C 4 => 4
        | C 5 => 5
        | D 6 => 6
        | D x => x
```

how many times should the view transformations for `C` and `D` each be called? There are three reasonable choices: at most four times, at most twice, and at most once. These correspond respectively to calling a view transformation whenever any constructor from the view is matched, whenever any contiguous block of constructors from the view is matched, or only when the first constructor from the view is matched. We choose the last interpretation, which essentially memoizes applications of a view transformation within a single match, not only on the grounds of efficiency, but because it corresponds most closely with our intuitions about exhaustiveness and redundancy in matches. For example, given the views

```
viewtype int = F of bool
viewtype int = G of bool
```

the match

```
case e of F true => 0
        | G true => 1
        | F false => 2
```

is not necessarily exhaustive under either of the other interpretations, even though the `F` clauses appear to cover all possible cases, because the view transformation for `F` might initially return false, but later return true (possibly because of an assignment made by the view transformation for `G`). Similarly, in the match

```
case e of F true => 0
        | G true => 1
        | F true => 2
```

the third clause is not necessarily redundant with the first. Under the memoizing interpretation, both examples obey our intuitions—the first match is exhaustive and the second match is redundant.

Note that memoization only applies to constructors that appear in the same position. For example, both of the following matches call the view transformation for `C` twice:

```
case e of (C x,C y) => x+y
case e of C (C x) => x
```

4 Syntax and Semantics

Next, we formally define the syntax and semantics of views. The new syntax required to support views is minimal. We add two new keywords (`viewtype` and `withfun`) and three new productions to the syntax described in *The Definition of Standard ML (Revised)* [6].

spec	::=	`viewtype` *ty* `=` *condesc*
dec	::=	`viewtype` *viewbind* ⟨`withfun` *fvalbind*⟩
viewbind	::=	*ty* `=` *conbind* `where` *match* ⟨`and` *viewbind*⟩

We define the static and dynamic semantics of the new language constructs implicitly via a set of rewrite rules back into Standard ML without views. These rules fire in any order until no more rules apply. Some of the rules are context-sensitive in that they need to know whether a given identifier is a constructor defined by a view (and if so, which view). We could implement this explicitly by parameterizing the rewrite rules with an environment, but we choose to leave the environment implicit.

Many of the rules generate new identifiers. These are not fresh identifiers *per se.* They are guaranteed not to clash with any user identifiers, but are chosen to be consistent with each other. For example, each view constructor is renamed from C to C'. The exact form of C' does not matter, as long as every occurrence of the same C is renamed to the same C'. We assume that the rules can distinguish between user identifiers and identifiers generated by other rules.

For brevity, we show the rules for only a subset of the language. For example, we ignore nullary constructors, records, and tuples larger than two. Extending these rules to cover the full language is straightforward.

Viewtypes Inside signatures, each view specification is expanded into a datatype specification and a function specification.

(VIEW-SPEC)

`viewtype` ty = C_1 `of` ty_1 | $\cdots$ | C_k `of` ty_k

$\Longrightarrow$

`datatype` $(\alpha_1, \ldots, \alpha_k)\ V$ = C'_1 `of` α_1 | $\cdots$ | C'_k `of` α_k
`val` $viewV\ :\ ty \rightarrow (ty_1, \ldots, ty_k)\ V$

The new identifiers V and $viewV$ are generated from $C_1 \ldots C_k$; the new identifier C'_i is generated from C_i.

Each view declaration expands into a datatype declaration and a function declaration. This is quite similar to the rule for view specifications, except for the need to deal with recursion between view transformations and ordinary functions.

(VIEW-DEC)

`viewtype` ty_1 = $C_{1,1}$ `of` $ty_{1,1}$ | $\cdots$ | C_{1,k_1} `of` ty_{1,k_1} `where` $match_1$
  `and ...`
  `and` ty_n = $C_{n,1}$ `of` $ty_{n,1}$ | $\cdots$ | C_{n,k_n} `of` ty_{n,k_n} `where` $match_k$
`withfun` $fvalbind$

$\Longrightarrow$

`datatype` $(\alpha_{1,1}, \ldots, \alpha_{1,k_1})\ V_1$ = $C'_{1,1}$ `of` $\alpha_{1,1}$ | $\cdots$ | C'_{1,k_1} `of` α_{1,k_1}

$\vdots$

`datatype` $(\alpha_{n,1}, \ldots, \alpha_{n,k_n})\ V_n$ = $C'_{n,1}$ `of` $\alpha_{n,1}$ | $\cdots$ | C'_{n,k_n} `of` α_{n,k_n}
`val rec` $viewV_1\ :\ ty_1 \rightarrow (ty_{1,1}, \ldots, ty_{1,k_1})\ V_1$ = `fn` $match_1$

$\vdots$

  `and` $viewV_n\ :\ ty_n \rightarrow (ty_{n,1}, \ldots, ty_{n,k_n})\ V_n$ = `fn` $match_n$
  `and` $fvalbind'$

where $fvalbind'$ is $fvalbind$ desugared from `fun`-style into `val rec`-style.

The seemingly bizarre form of the datatypes generated by these rules is so that a view declaration can match a less polymorphic view specification. Functions currently have this property, whereas datatypes are required to match exactly. Hence, we generalize the datatype as much as possible and rely on the type assigned to the view transformation to determine whether a view declaration and specification are compatible.

Also, note that both of these rules assume that the constructors of a view are first sorted into some canonical order. Otherwise, viewtype specifications and declarations would fail to match whenever they list constructors in different orders. Datatype definitions in Standard ML are currently insensitive to the order in which constructors are written, and we want that property for viewtype definitions as well.

Datatypes Each datatype expands into two datatypes and a function. The first datatype is a verbatim copy of the original. The other datatype and the function are for when a datatype is used as a view of itself.

(DATA-SPEC)

$\texttt{datatype}\ \textit{tyvarseq}\ T = C_1\ \texttt{of}\ \textit{ty}_1 \mid \cdots \mid C_k\ \texttt{of}\ \textit{ty}_k$

$\Longrightarrow$

$\texttt{datatype}\ \textit{tyvarseq}\ T = C_1\ \texttt{of}\ \textit{ty}_1 \mid \cdots \mid C_k\ \texttt{of}\ \textit{ty}_k$
$\texttt{datatype}\ (\alpha_1, \ldots, \alpha_k)\ V = C'_1\ \texttt{of}\ \alpha_1 \mid \cdots \mid C'_k\ \texttt{of}\ \alpha_k$
$\texttt{val}\ \textit{viewV} : \textit{tyvarseq}\ T \rightarrow (\textit{ty}_1, \ldots, \textit{ty}_k)\ V$

(DATA-DEC)

$\texttt{datatype}\ \textit{tyvarseq}\ T = C_1\ \texttt{of}\ \textit{ty}_1 \mid \cdots \mid C_k\ \texttt{of}\ \textit{ty}_k$

$\Longrightarrow$

$\texttt{datatype}\ \textit{tyvarseq}\ T = C_1\ \texttt{of}\ \textit{ty}_1 \mid \cdots \mid C_k\ \texttt{of}\ \textit{ty}_k$
$\texttt{datatype}\ (\alpha_1, \ldots, \alpha_k)\ V = C'_1\ \texttt{of}\ \alpha_1 \mid \cdots \mid C'_k\ \texttt{of}\ \alpha_k$
$\texttt{val}\ \textit{viewV} : \textit{tyvarseq}\ T \rightarrow (\textit{ty}_1, \ldots, \textit{ty}_k)\ V = \texttt{fn}\ C_1\ x_1 \Rightarrow C'_1\ x_1 \mid \cdots \mid C_k\ x_k \Rightarrow C'_k\ x_k$

Of course, the full rules must allow for mutually recursive datatypes.

View constructors in expressions Constructors introduced by views can be used as expressions only within the view transformation for that view. Any use outside the view transformation produces a static error.

(VIEWCON-EXP)

$$C \Longrightarrow \begin{cases} C' & \text{if occurs within own view transformation} \\ \textit{error} & \text{otherwise} \end{cases}$$

Case expressions Not surprisingly, the majority of the rewrite rules deal with pattern matching. Pattern matching can occur in several places in Standard ML, but we give the rules only for explicit case expressions.

Figure 5 describes the rules for rewriting case expressions. The basic strategy is first to expand each case expression into a series of nested case expressions, each of which tests at most a single constructor; then to insert calls to the appropriate view transformation for each occurrence of a view constructor; and finally to merge identical calls to view transformations.[1] For this merging phase to work properly, it is important that new identifiers naming intermediate values be generated consistently. The dot operator $v.id$ deterministically combines two identifiers into a new identifier. (There will never be any confusion with dots indicating selection from a structure.)

Many of the rules introduce new bindings using $\textsc{let}\ x = e\ \textsc{in}\ e'$ as shorthand for $(\texttt{fn}\ x \Rightarrow e')\ e$. This prevents any confusion over unwarranted generalization of type variables, as might occur if we used `let` expressions instead.

Note that these rewrite rules are intended to define the semantics of pattern matching with views, not to suggest an implementation. They are horribly inefficient, in at least two respects.

[1]These rules are similar in spirit to the formal semantics of pattern matching presented in the Haskell report [5].

(CASE-BEGIN)

$\texttt{case}\ e\ \texttt{of}\ match \quad \Longrightarrow \text{LET}\ v = e\ \text{IN}\ \texttt{case}\ v\ \texttt{of}\ match \mid \_ \Rightarrow \texttt{raise Match}$

$\{v \text{ fresh, } e \text{ not some generated identifier } v'\}$

(CASE-SEQ)

$\texttt{case}\ v\ \texttt{of}\ match \mid match' \mid \_ \Rightarrow e \Longrightarrow \texttt{case}\ v\ \texttt{of}\ match \mid \_ \Rightarrow (\texttt{case}\ v\ \texttt{of}\ match' \mid \_ \Rightarrow e)$

(CASE-AS)

$\texttt{case}\ v\ \texttt{of}\ x\ \texttt{as}\ p \Rightarrow e \mid \_ \Rightarrow e' \quad \Longrightarrow \texttt{case}\ v\ \texttt{of}\ p \Rightarrow (\text{LET}\ x = v\ \text{IN}\ e) \mid \_ \Rightarrow e'$

(CASE-CON)

$\texttt{case}\ v\ \texttt{of}\ C\ p \Rightarrow e \mid \_ \Rightarrow e' \quad \Longrightarrow \texttt{case}\ v\ \texttt{of}\ C\ v.C \Rightarrow (\texttt{case}\ v.C\ \texttt{of}\ p \Rightarrow e \mid \_ \Rightarrow e') \mid \_ \Rightarrow e'$

$\{p \text{ not generated identifier } v'\}$

(CASE-TUP)

$\texttt{case}\ v\ \texttt{of}\ (p_1, p_2) \Rightarrow e \mid \_ \Rightarrow e' \quad \Longrightarrow \texttt{case}\ v\ \texttt{of}\ (v.1, v.2) \Rightarrow$

$(\texttt{case}\ v.1\ \texttt{of}\ p_1 \Rightarrow (\texttt{case}\ v.2\ \texttt{of}\ p_2 \Rightarrow e \mid \_ \Rightarrow e') \mid \_ \Rightarrow e')$

(CASE-VIEW)

$\texttt{case}\ v\ \texttt{of}\ C\ v.C \Rightarrow e \mid \_ \Rightarrow e' \quad \Longrightarrow \text{LET}\ v' = viewV\ v\ \text{IN}\ \texttt{case}\ v'\ \texttt{of}\ C'\ v.C \Rightarrow e \mid \_ \Rightarrow e'$

$\{C \text{ a view constructor with transformation } viewV,\ v' = v.viewV\}$

(VIEW-MEMO)

$\text{LET}\ v' = viewV\ v\ \text{IN}\ \ldots (\text{LET}\ v' = viewV\ v\ \text{IN}\ e) \ldots$

$\Longrightarrow \text{LET}\ v' = viewV\ v\ \text{IN}\ \ldots e \ldots$

Figure 5: Rules for rewriting case expressions.

First, rule CASE-SEQ sequentializes each match, preventing many of the optimizations possible for parallel matches. Second, rules CASE-CON and CASE-TUP duplicate code, resulting in potentially exponential increases in code size.

Both problems can be reduced by running backward variants of rules CASE-SEQ, CASE-AS, CASE-CON, and CASE-TUP in a postpass recombining sequentialized matches into parallel matches whenever possible. This will usually be successful, failing only when calls to view transformations intervene, when switching from matching constructors of one viewtype (or datatype) to another, or when a particular view transformation is memoized in one occurrence of duplicated code, but not in another. The explosion in code size can be further alleviated by ensuring that any duplicated code is small to be begin with, for example by replacing large expressions on the right-hand sides of clauses with function calls.

Ref patterns In the rules for rewriting case expressions, we have ignored patterns involving the `ref` constructor. This is not an oversight; rather, we propose eliminating this controversial language feature in favor of the following view on reference cells:

```
viewtype 'a ref = Ref of 'a
  where r => Ref (!r)
```

Then `ref` can be demoted from a constructor to a value.

If `ref` is left as a constructor, then its behavior in patterns should be identical to the above view. In particular, the dereference should occur the first time the `ref` pattern is encountered, and it should occur only once. This can be accomplished by the rules in Figure 6.

(CASE-REF)

$\texttt{case}\ v\ \texttt{of ref}\ v.\mathit{ref} \Rightarrow e \mid \_ \Rightarrow e' \quad \Longrightarrow \text{LET}\ v.\mathit{ref}\ \texttt{=}\ \texttt{!}v\ \text{IN}\ e$

(REF-MEMO)

$\text{LET}\ v'\ \texttt{=}\ \texttt{!}v\ \text{IN}\ \ldots(\text{LET}\ v'\ \texttt{=}\ \texttt{!}v\ \text{IN}\ e)\ldots \Longrightarrow \text{LET}\ v'\ \texttt{=}\ \texttt{!}v\ \text{IN}\ \ldots e\ldots$

Figure 6: Rules for rewriting `ref` patterns.

5 Efficiency

One argument that is sometimes used against views is that pattern matching involving views is less efficient than pattern matching against concrete datatypes. However, this should not be surprising since abstraction usually carries a performance penalty.[2] As long as code not using views is not adversely affected, and as long as code using views is at least comparable in efficiency to code accessing abstract datatypes in other ways (such as through `null`, `head`, and `tail` functions), questions of efficiency should be much less of a concern than questions such as ease of reading, writing, and maintaining programs.

A related argument against views is that, because pattern matching is usually very fast, programmers might be misled by views into writing very inefficient programs, whereas if they were required to make the function calls explicit, they would be more careful. If this is a major concern, then it would be easy to adopt a naming convention that distinguishes view constructors from datatype constructors as a reminder that view constructors in patterns indicate function calls.

6 Related Work

Wadler introduced views in [9]. He allowed view constructors to appear in both patterns and expressions, and therefore required two view transformations—one mapping from the concrete type to the abstract type (to be used in patterns) and one mapping from the abstract type to the concrete type (to be used in expressions). Unfortunately, this dual role played by view constructors proved problematic for equational reasoning.

Burton and Cameron [2] solve the problems with equational reasoning by forbidding the use of view constructors in expressions. This also eliminates the need for the inverse view transformation. However, for the sake of efficient compilation, they forbid mixing constructors from different views within a single match.

Palao *et al.* [7] and Erwig [4] allow the mixing of arbitrary view constructors, but effectively limit each view type to a single constructor. This is only feasible because their view transformations are all *partial*—a match failure in the view transformation does not cause an error, but rather causes the pattern that invoked the view transformation to fail. A major drawback of this single-constructor approach is that related constructors cannot share work during the view transformation.

The proposal most similar to ours is that of Burton *et al.* [3]. However, since their proposal was for Haskell, they do not consider issues such as interactions with state or with Standard ML's module system.

Aitken and Reppy [1] considered a different solution to the problem of pattern matching on abstract datatypes. Their proposal for *abstract value constructors* allows patterns to be named. Then an occurrence of an abstract constructor expands into the named pattern. This proposal

[2]When the concrete representation and view transformation are known to the compiler, pattern matching involving views can often be optimized into pattern matching against the concrete type [7, 4]. However, in the common case that the view is received as a functor argument, these optimizations will usually not apply.

supports very efficient compilation, but has much more limited applicability. For example, because the named patterns have a fixed shape, and do not allow any computation to occur, it is impossible to write an abstract value constructor that matches the last element of an arbitrary list.

Peyton Jones's *pattern guards* [8] offer another alternative to views. Pattern guards allow bindings within the guard of a pattern to be visible within the right-hand side of the clause. If a pattern within the guard fails to match, then the guard fails. Views and pattern guards solve many of the same problems, but there are problems for which views are much more convenient than pattern guards, and vice versa. Pattern guards are extremely attractive in Haskell, which already supports the components out of which pattern guards are built, namely guards and list comprehension syntax. However, they are less attractive in Standard ML, which currently supports neither.

7 Conclusions

A large part of Standard ML's appeal is the way the module system encourages the kinds of abstractions that are necessary for building large systems. Unfortunately, the restriction of pattern matching to concrete datatypes *discourages* such abstractions. By allowing pattern matching on abstract datatypes, views eliminate this disincentive. We sincerely hope that some solution to this problem is adopted soon, even if not the one described here.

References

[1] Aitken, W. E., and Reppy, J. H. Abstract value constructors. In *ACM SIGPLAN Workshop on ML and its Applications* (June 1992), pp. 1–11.

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

[3] Burton, F. W., Meijer, E., Sansom, P., Thompson, S., and Wadler, P. A (sic) extension to haskell 1.3 for views. Distributed on the Haskell mailing list, Oct. 1996.

[4] Erwig, M. Active patterns. In *International Workshop on the Implementation of Functional Languages* (Sept. 1996), pp. 21–40.

[5] Hudak, P., et al. Report on the functional programming language Haskell, Version 1.2. *SIGPLAN Notices 27*, 5 (May 1992).

[6] Milner, R., Tofte, M., Harper, R., and MacQueen, D. *The Definition of Standard ML (Revised).* The MIT Press, Cambridge, Massachusetts, 1997.

[7] Palao Gostanza, P., Peña, R., and Núñez, M. A new look at pattern matching in abstract data types. In *ACM SIGPLAN International Conference on Functional Programming* (May 1996), pp. 110–121.

[8] Peyton Jones, S. A new view of guards. Distributed on the Haskell mailing list, Apr. 1997.

[9] Wadler, P. Views: A way for pattern matching to cohabit with data abstraction. In *ACM Symposium on Principles of Programming Languages* (Jan. 1987), pp. 307–313.