Pattern matching with abstract data types[1]

F. WARREN BURTON AND ROBERT D. CAMERON
School of Computing Science, Simon Fraser University, Burnaby, British Columbia, Canada V5A 1S6
(*e-mail*: burton@cs.sfu.ca)

Abstract

Pattern matching in modern functional programming languages is tied to the representation of data. Unfortunately, this is incompatible with the philosophy of abstract data types.

Two proposals have been made to generalize pattern matching to a broader class of types. The *laws* mechanism of Miranda allows pattern matching with non-free algebraic data types. More recently, Wadler proposed the concept of *views* as a more general solution, making it possible to define arbitrary mappings between a physical implementation and a view supporting pattern matching. Originally, it was intended to include views in the new standard lazy functional programming language Haskell.

Laws and views each offer important advantages, particularly with respect to data abstraction. However, if not used with great care, they also introduce serious problems in equational reasoning. As a result, laws have been removed from Miranda and views were not included in the final version of Haskell.

We propose a third approach which unifies the laws and views mechanisms while avoiding their problems. Philosophically, we view pattern matching as a bundling of case recognition and component selection functions instead of a method for inverting data construction. This can be achieved by removing the implied equivalence between data constructors and pattern constructors. In practice, we allow automatic mapping into a view but not out of the view. We show that equational reasoning can still be used with the resulting system. In fact, equational reasoning is easier, since there are fewer hidden traps.

Capsule review

The tension between abstract datatypes and the syntactic sugar of pattern matching has long been recognized, and has led to two proposed solutions: laws and views. Unfortunately, these solutions cause problems for equational reasoning, so they have been rejected from current language designs. This paper shows that just a small modification to the earlier ideas allows us to have pattern matching and abstract data types along with safe equational reasoning. After explaining the idea, the authors give an extended example showing how to use it in practice.

1 Introduction

Pattern matching helps in writing simple and concise function definitions in modern functional programming languages. For example, with the Haskell algebraic data type

$$\textbf{data}\quad \textit{List alpha} \;=\; \textit{Nil} \mid \textit{Cons alpha}\ (\textit{List alpha})$$

[1] This work was supported in part by the Natural Science and Engineering Research Council of Canada.

we can write an append function as

$$\begin{aligned} \mathit{append}\ \mathit{Nil}\ \mathit{ys} &= \mathit{ys} \\ \mathit{append}\ (\mathit{Cons}\ x\ \mathit{xs})\ \mathit{ys} &= \mathit{Cons}\ x\ (\mathit{append}\ \mathit{xs}\ \mathit{ys}). \end{aligned}$$

There are two ways to think about pattern matching. We can think of pattern matching as inverting constructor functions. For example, when we evaluate *append* (*Cons 1 Nil*) *Nil*, then the *Cons* on the left hand side of the second equation defining *append* inverts the *Cons* in the expression *Cons 1 Nil*, extracting *1* into *x* and *Nil* into *xs*. This fits well with equational reasoning and allows us to argue

$$\begin{aligned} &\mathit{append}\ (\mathit{Cons}\ 1\ \mathit{Nil})\ \mathit{Nil} \\ =\ &\mathit{Cons}\ 1\ (\mathit{append}\ \mathit{Nil}\ \mathit{Nil}) \\ =\ &\mathit{Cons}\ 1\ \mathit{Nil}. \end{aligned}$$

The other way is to think of patterns as a bundling of case recognition and component selection functions. The *append* function can be rewritten to exhibit these operations explicitly

$$\begin{aligned} &\mathit{append}\ \mathit{xs}\ \mathit{ys}\ = \\ &\quad \mathbf{if}\ \mathit{is\_nil}\ \mathit{xs}\ \mathbf{then}\ \mathit{ys} \\ &\quad \mathbf{else}\ \mathit{Cons}\ (\mathit{hd}\ \mathit{xs})\ (\mathit{append}\ (\mathit{tl}\ \mathit{xs})\ \mathit{ys}). \end{aligned}$$

This is approximately what a compiler will produce as an intermediate form when compiling *append*. However, this form of *append* would be significantly more tedious to use in equational reasoning, even given suitable axioms such as

$$\begin{aligned} \mathit{is\_nil}\ \mathit{nil} &= \mathit{True} \\ \mathit{is\_nil}\ (\mathit{Cons}\ x\ \mathit{xs}) &= \mathit{False} \\ \mathit{hd}\ (\mathit{Cons}\ x\ \mathit{xs}) &= x \\ \mathit{tl}\ (\mathit{Cons}\ x\ \mathit{xs}) &= \mathit{xs}. \end{aligned}$$

With algebraic data types, either of these two views of pattern matching may be used.

It should be noted that pattern matching often forces some evaluation of arguments to a function, which has consequences for both semantics and efficiency. For example, given the function definition

$$\begin{aligned} f\ \mathit{True}\ \mathit{True} &= \mathit{True} \\ f\ x\ \mathit{False} &= \mathit{False} \\ f\ \mathit{False}\ y &= \mathit{False} \end{aligned}$$

exp will be evaluated when (*f exp False*) is evaluated, even though the second equation appears to be the appropriate equation and use of the second equation does not require the value of *exp*. In fact, if *exp* is $\perp$ then (*f exp False*) also will be $\perp$. In this case, the first equation must be held responsible for the result. We will not consider these issues in this paper, but will note that they remain essentially unchanged with the addition of the language features for pattern matching considered here.

Two proposals have been made to generalize pattern matching to a broader class of types. Miranda laws (Turner, 1985) allow pattern matching to be used with non-free algebraic data types. Simon Thompson (1988, 1990) has discussed ways of reasoning about algebraic data types with laws. More recently, the use of views (Wadler, 1987) has been proposed as a more general solution that makes it possible to define arbitrary mappings between a physical implementation and a view supporting pattern matching. Originally, it was intended to include views in the new standard lazy functional programming language Haskell (Hudak *et al.*, 1988). Both of these approaches are based on the correspondence between data constructors and patterns.

Laws and views each offer important advantages, particularly with respect to data abstraction. However they also introduce serious problems, particularly if they are not used with great care. As a result, laws have been removed from Miranda and views were not included in the revised version of Haskell (Hudak *et al.*, 1990). By thinking of patterns as a bundling of case recognition and component selection functions, we can keep the advantages of views while avoiding the problems, for the most part.

As a simple example, with either views or laws, it is possible to define a constructor *Ratio* that will eliminate common factors from its two arguments. For example, *Ratio 10 5* = *Ratio 2 1*. In Miranda with laws, this could be done as follows

$$\begin{aligned} &\mathit{rational} && ::= \quad \mathit{Ratio\ Num\ Num} \\ &\mathit{Ratio}\ m\ n && \Rightarrow \quad \mathit{Ratio}\ (m\ \text{`}\mathit{div}\text{`}\ q)\ (n\ \text{`}\mathit{div}\text{`}\ q), \quad \textbf{if}\ q \neq 1 \\ & && \quad \textbf{where} \\ & && \quad q \quad = \quad \mathit{normalize}\ m\ n. \end{aligned}$$

Here *normalize* is a suitable version of the *gcd* function that handles all the special cases.

One can easily define a function

$$\mathit{size}\ (\mathit{Ratio}\ a\ b) \quad = \quad a+b.$$

It is tempting to argue, using equational reasoning, that

$$\mathit{size}\ (\mathit{Ratio}\ 10\ 5) \quad = \quad 10+5 \quad = \quad 15$$

when *size* (*Ratio 10 5*) should be *3*.

With the approach we propose, *Ratio* may be used in patterns only. A function *ratio* may be used to construct data objects and the equation

$$\begin{aligned} &\mathit{ratio}\ m\ n \quad = \\ &\quad \mathit{Ratio}\ (m\ \text{`}\mathit{div}\text{`}\ q)\ (n\ \text{`}\mathit{div}\text{`}\ q) \\ &\quad \textbf{where} \\ &\quad q \quad = \quad \mathit{normalize}\ m\ n \end{aligned}$$

may be used in reasoning about a program. Thus, we do not require that *Ratio* invert the construction operation performed by *ratio*, but merely be useful for matching and selecting components of rational objects.

The remainder of this paper presents and analyses our specific proposals in the context of an early version of the Haskell programming language (Hudak *et al.*, 1990). However, the fundamental ideas do not depend on the choice of any particular language. In section 2 we propose a feature for the restricted export of data constructors and show how it can be used to provide pattern matching on non-free algebraic data types. This essentially provides a way of implementing Miranda laws in Haskell. Section 3 then considers how to reason about functions defined by pattern-matching on such non-free algebraic types. A restricted views mechanism is then described in section 4 to allow pattern matching using multiple views on arbitrary user-defined data abstractions. Several examples are considered in section 5. Section 6 concludes the paper.

2 Pattern matching for non-free abstract data types

Haskell's module facility allows abstract data types (ADTs) to be defined by exporting the name of an algebraic data type without its representation (i.e. its data constructors). For example, an ADT for rational numbers providing selector functions *numerator* and *denominator* and a constructor function *ratio* may be implemented as follows

```
module Ratio_Module (Rational, numerator, denominator, ratio)
where
data Rational   =   Ratio Int Int
numerator   ::   Rational -> Int
numerator (Ratio a b)   =   a
denominator   ::   Rational -> Int
denominator (Ratio a b)   =   b
ratio   ::   Int -> Int -> Rational
ratio m n   =
  Ratio (m `div` q) (n `div` q)
  where
  q   =   normalize m n
  normalize   ::   Int -> Int -> Int
  normalize m n
    | n == 0   =   error "Zero denominator in ratio"
    | m == 0   =   n
    | n < 0    =   -(gcd (abs m) (-n))
    | n > 0    =   gcd (abs m) n
  gcd   ::   Int -> Int -> Int
  gcd a b
    | a == b   =   b
    | a < b    =   gcd a (b-a)
    | a > b    =   gcd (a-b) b.
```

This ADT provides the desired semantics for our *Rational* data type, but requires that analysis be performed using the explicit selection operations *numerator* and *denominator* rather than pattern matching.

Suppose we allow the restricted export of data constructors for the purpose of pattern matching only. In this paper, we use the name of the data constructor prefixed by a ? mark to indicate such restricted export; some other syntax may be ultimately preferable. Using this extension, our *Rational* ADT may be reimplemented as follows

```
module Ratio_Module (Rational(?Ratio), ratio) where
data Rational  =  Ratio Int Int
ratio  ::  Int -> Int -> Rational
ratio m n  =
   Ratio (m `div` q) (n `div` q)
   where
   q  =  normalize m n
   as before.
```

Within the module, the constructor *Ratio* may be used as an ordinary algebraic type constructor, i.e. both for pattern matching and construction of objects. Outside of the module, *Ratio* may only be used for pattern matching, the actual construction of *Rational* values requires the use of *ratio* instead.

One may argue that exporting the data constructors of an ADT destroys its representation independence and information hiding properties, even if the constructors are made available only for pattern matching. This is a valid criticism. We will address it in section 4 with our proposal for resurrecting the views mechanism.

We can now write functions on the *Rational* data type using both *Ratio* and *ratio*, as appropriate. For example, addition of rational numbers requires conversion to a common denominator:

```
radd  ::  Rational -> Rational -> Rational
  radd (Ratio n1 d1) (Ratio n2 d2)
     =  ratio (n1 * d2 + n2 * d1) (d1 * d2).
```

Construction of *Rational* numbers using *ratio* ensures that they are always represented in canonical form; pattern matching such values with *Ratio* thus selects the canonical components.

Given the function definition

$$size\ (Ratio\ a\ b) \quad = \quad a + b$$

it is still valid to reason that *size* (*Ratio 10 5*) = 15. However, we cannot infer that *Ratio 10 5* = *Ratio 2 1*, so no contradiction arises. Indeed, *Ratio 10 5* is not even a valid object of type *Rational*. Such objects must be constructed using *ratio*. Thus we may specify an object *ratio 10 5* and equational reasoning gives us *ratio 10 5* = *Ratio 2 1*, so *size* (*ratio 10 5*) = *3* as desired.

Although we have described our extension to Haskell as a syntactic device to support pattern matching on ADTs, we can use proof techniques similar to those used by Thompson (1986, 1990). In essence, we think of the ADT as a Miranda 'lawful' type (Turner, 1985), i.e. a non-free algebraic data type. The corresponding free algebraic type is called the associated free type (AFT) of the ADT. We interpret the exported pattern constructors as the constructors of the AFT. For example, outside the *Rational* ADT, we interpret *Ratio* as the constructor of a free type, *Rational′*, associated with *Rational*, where

data *Rational′* = *Ratio Int Int*.

In fact, *Rational* is a subtype of *Rational′* consisting of all those *Ratio* values which represent ratios in canonical form. Thus *Ratio 2 1* is a value of type *Rational′* and of type *Rational*, while *Ratio 10 5* is only of type *Rational′*.

To specify objects of a non-free ADT such as *Rational*, we must provide constructors for the ADT corresponding to the constructors of the AFT. In the case of the *Rational* data type, the ADT constructor is *ratio*. The values constructed by *ratio* are of type *Rational* and just those values of type *Rational′* that are in canonical form.

Equations defining functions on type *Rational* using pattern matching with *Ratio* thus may be more properly thought of as defining functions on type *Rational′*. Note, however, that such functions cannot construct and return arbitrary *Rational′* objects, since their defining equations may only use *ratio* on their right-hand sides. When applied to objects of type *Rational*, then, they will always return objects of type *Rational*.

In describing the semantics and proof techniques for lawful functions, Thompson (1986, 1990) makes heavy use of the distinction between the constructor for a lawful type and the constructor of the AFT. In the case of Miranda laws, however, this distinction must be made at the metalanguage level; our proposal has the virtue of clarifying the distinction by introducing it into the source code. For example, *ratio* is a constructor for the ADT *Rational* and *Ratio* is a constructor for the AFT *Rational′*. We have also found this distinction to be useful in providing a proof technique not described by Thompson, namely induction over the constructors of the ADT. We present a detailed example of this technique in the next section.

3 Inductive proofs for non-free ADTs

To illustrate techniques for reasoning about non-free ADTs, we consider the example of ordered sets of numbers defined using the following (extended) Haskell module

```
module OrderedSetModule (OrderedSet(?Empty, ?Add), empty, add)
where
data  OrderedSet  =  Empty | Add Int OrderedSet
empty  ::  OrderedSet
empty  =  Empty
add  ::  Int -> OrderedSet -> OrderedSet
add m Empty  =  Add m Empty
add m (Add n x)
      | m < n    =  Add m (Add n x)
    | m == n   =  Add n x
    | m > n    =  Add n (add m x).
```

If a value *m* greater than the current first element of the ordered set is to be added, the last equation recursively ensures that it is placed after this first element and all other elements that are less than *m*. Thus values such as *Add 2* (*Add 3 Empty*) are proper values of type *OrderedSet* as well as the associated free type *OrderedSet′*, while *Add 3* (*Add 2 Empty*) is only a value of type *OrderedSet′*. Following our conventions, *empty* and *add* are the constructors of the ADT *OrderedSet* to be used outside the ADT for constructing ordered sets, while *Empty* and *Add* are the constructors of the AFT to be used outside the ADT only for pattern matching.

Thompson (1990) recommends that functions on a non-free ADT be implemented in a 'faithful' manner, i.e. so that they work independently of whether objects are in canonical form or not. This allows proofs about properties of such functions on the AFT to be immediately carried over to the corresponding functions of the ADT. However, one reason for using a canonical representation for a data type is so that you can take advantage of it in implementing functions on the data type. (Consider, for example, the implementation of an *intersect* function to perform set intersection. Using 'faithful' functions is likely to result in a $\Theta(n^2)$ implementation, whereas a $\Theta(n)$ implementation is possible making use of the orderedness of the representation.) We will show how induction can be used to establish properties of unfaithful functions.

Two useful functions on ordered sets test for set membership and remove an element, respectively. We provide implementations which make use of the canonical property for ordered sets

```
member :: num → OrderedSet → Bool
member a Empty = False
member a (Add b x)
  | a == b  =  True
    | a < b =  False
  | a > b   =  member a x
remove :: num → OrderedSet → OrderedSet
remove a Empty = Empty
remove a (Add b x)
  | a == b  =  x
    | a < b =  add b x
    | a > b =  add b (remove a x).
```

We design and write these functions to apply to objects of type *OrderedSet*. However, the equations also define functions on the AFT *OrderedSet′*. For example, *remove 2* (*Add 3* (*Add 2 Empty*)) = *Add 3* (*Add 2 Empty*). Of course, one should not expect the action of such functions on the AFT to necessarily obey properties that hold only for ordered sets.

The dichotomous view of these functions as being defined either over type *OrderedSet′* or type *OrderedSet* is useful because it provides two inductive bases for proving properties about them. We can show that a property *P*(*x*) holds for all finitely generated, well defined *x* in *OrderedSet′* by induction on its constructors *Empty* and

Add, i.e. by showing $P(Empty)$ and for any a, $P(Add\ a\ x)$, assuming that $P(x)$ is true. Since *OrderedSet* is a subtype of *OrderedSet′*, this also establishes $P(x)$ for all x in *OrderedSet*. Alternatively, we can directly show that $P(x)$ holds for finitely generated, well defined x in *OrderedSet* by induction over its constructors *empty* and *add*. Of course, such a property may not hold for arbitrary x in *OrderedSet′*.

Technically, we prove that a property $P(x)$ holds for all finitely generated, well defined x by using induction over the number of steps (i.e. uses of *empty* or *add*) required to generate an object. The approach can be extended to partial or infinite objects using the methods given in Bird and Wadler (1988).

In the case of an arbitrary ADT, it is not always clear which functions should be regarded as generators. For example, consider *OrderedSet* extended to include the operations *union* (merge two ordered sets), *min* (select the minimum value) and *deletemin* (remove the minimum value). Clearly, *min* is not a constructor because it does not return an *OrderedSet*. One solution is to treat all of the remaining functions as constructors in a proof. Alternatively, if it can be shown that any value generated by a function can be generated by other generators, then it can be removed from the set of generators. This is most easily done if the implementation of the function is in terms of other generators. For example, if *union* is defined in terms of *add* then *union* may be excluded from the set of generators that must be used in an induction proof. Removing *deletemin* from the list of constructors is not so easy. In fact, *deletemin empty* is a finitely generated object $\perp$ that cannot be generated using *empty* and *add* alone. Hence, *deletemin* normally cannot be excluded from the set of generators. However, if proofs include partial, and perhaps infinite, objects (see Bird and Wadler, 1988) then $\perp$ will be included as a generator along with *empty* and *add*, in which case *deletemin* can be excluded from the set of generators. Finally, a minimal set of constructors may not be unique. If we also add a function to generate a singleton *OrderedSet*, then that function and *union* could be used in the set of generators instead of *add*, which might simplify some proofs.

Induction over the ADT and over the AFT each are useful in verifying properties on type *OrderedSet*. Recall that functions on type *OrderedSet* are written using equations whose L.H.S. patterns are expressed in terms of the constructors of type *OrderedSet′*. Thus it is often easiest to use induction over *OrderedSet′* to establish basic properties of these functions. Once such properties have been established, further properties that hold only for type *OrderedSet*, but not in general for *OrderedSet′*, may be established by induction over *OrderedSet*.

Consider, for example, the proof of the following theorem:

Theorem 1
If x is a well defined OrderedSet and a is a well defined num, then member a (remove a x) = False.

This property holds for any well defined object in *OrderedSet*, but not for arbitrary values in *OrderedSet′* or if $x = \perp$ or $x = Add \perp Empty$ or $x = Add\ m\ y$, where y is not well defined. Note that any infinite set would be of type *OrderedSet′* but not *OrderedSet*.

We will verify Theorem 1 by induction over *OrderedSet*. To do so we establish a number of subsidiary properties as lemmas using induction over *OrderedSet′*. In these lemmas we will assume that all values are well defined.

Lemma 1
$add\ a\ (add\ a\ x) \quad = \quad add\ a\ x.$

Base step: show $add\ a\ (add\ a\ Empty) \quad = \quad add\ a\ Empty$. By equational reasoning:

$$\begin{aligned} & add\ a\ (add\ a\ Empty) \\ &\quad = \quad add\ a\ (Add\ a\ Empty) \\ &\quad = \quad Add\ a\ Empty \\ &\quad = \quad add\ a\ Empty. \end{aligned}$$

This establishes the base case.

Induction step: Assume $add\ a\ (add\ a\ y) = add\ a\ y$ and show $add\ a\ (add\ a\ (Add\ b\ y)) = add\ a\ (Add\ b\ y)$. The various cases can be broken down depending on the relative ordering of a and b. Table 1 shows the steps of equational reasoning that complete the proof. □

Lemma 2
$add\ a\ (add\ b\ x) \quad = \quad add\ b\ (add\ a\ x).$

There are three cases: (a) $a = b$, (2) $a < b$, and (3) $a > b$. Case 1 is established easily: $add\ a\ (add\ b\ x) = add\ a\ (add\ a\ x) = add\ b\ (add\ a\ x)$. Case 3 is the same as case 2 with the two sides of the equation interchanged. Therefore, we assume $a < b$ and prove the lemma by induction on *OrderedSet′*.

Base step: Show $add\ a\ (add\ b\ Empty) = add\ b\ (add\ a\ Empty)$. By equational reasoning

$$\begin{aligned} & add\ a\ (add\ a\ Empty) \\ &\quad = \quad add\ a\ (Add\ b\ Empty) \\ &\quad = \quad Add\ a\ (Add\ b\ Empty) \\ & add\ b\ (add\ a\ Empty) \quad = \quad add\ b\ (Add\ a\ Empty) \\ &\quad = \quad Add\ a\ (add\ b\ Empty) \\ &\quad = \quad Add\ a\ (Add\ b\ Empty). \end{aligned}$$

This establishes the base case.

Induction step: Assume $add\ a\ (add\ b\ y) = add\ b\ (add\ a\ y)$ and show $add\ a\ (add\ b\ (Add\ c\ y)) = add\ b\ (add\ a\ (Add\ c\ y))$. The various cases can be broken down depending on the relative ordering of a, b and c, (recalling also that $a < b$ by assumption). Table 2 shows the steps of equational reasoning that complete the proof. □

Lemma 3
$member\ a\ (add\ b\ x) \quad = \quad member\ a\ x, \quad$ **if** $a \neq b$.

Table 1. *Inductive case for Lemma 1*

Case	$add\ a\ (add\ a\ (Add\ b\ y)) =$	$add\ a\ (Add\ b\ y)) =$
$a = b$	$add\ a\ (Add\ b\ y)$	—
$a < b$	$add\ a\ (Add\ a\ (Add\ b\ y)) =$ $Add\ a\ (Add\ b\ y)$	$Add\ a\ (Add\ b\ y)$
$a > b$	$add\ a\ (Add\ b\ (add\ a\ y)) =$ $Add\ b\ (add\ a\ (add\ a\ y)) =$ $Add\ b\ (add\ a\ y)$ (by induction hypothesis)	$Add\ b\ (add\ a\ y)$

Table 2. *Inductive case for Lemma 2*

Case	$add\ a\ (add\ b\ (Add\ c\ y)) =$	$add\ b\ (add\ a\ (Add\ c\ y)) =$
$b = c$	$add\ a\ (Add\ c\ y) =$ $Add\ a\ (Add\ c\ y)$	$add\ b\ (Add\ a\ (Add\ c\ y)) =$ $Add\ a\ (add\ b\ (Add\ c\ y)) =$ $Add\ a\ (Add\ c\ y)$
$b < c$	$add\ a\ (Add\ b\ (Add\ c\ y)) =$ $Add\ a\ (Add\ b\ (Add\ c\ y))$	$add\ b\ (Add\ a\ (Add\ c\ y)) =$ $Add\ a\ (add\ b\ (Add\ c\ y)) =$ $Add\ a\ (Add\ b\ (Add\ c\ y))$
$b > c, a = c$	$add\ a\ (Add\ c\ (add\ b\ y)) =$ $Add\ c\ (add\ b\ y)$	$add\ b\ (Add\ c\ y) =$ $Add\ c\ (add\ b\ y)$
$b > c, a < c$	$add\ a\ (Add\ c\ (add\ b\ y)) =$ $Add\ a\ (Add\ c\ (add\ b\ y))$	$add\ b\ (Add\ a\ (Add\ c\ y)) =$ $Add\ a\ (add\ b\ (Add\ c\ y)) =$ $Add\ a\ (Add\ c\ (add\ b\ y))$
$b > c, a > c$	$add\ a\ (Add\ c\ (add\ b\ y)) =$ $Add\ c\ (add\ a\ (add\ b\ y)) =$ $Add\ c\ (add\ b\ (add\ a\ y))$ (by induction hypothesis)	$add\ b\ (Add\ c\ (add\ a\ y)) =$ $Add\ c\ (add\ b\ (add\ a\ y))$

This can be established by induction on *OrderedSet′* as follows.
Base step: $x = Empty$. By equational reasoning

$$\begin{aligned} & member\ a\ (add\ b\ Empty) \\ = \ & member\ a\ (Add\ b\ Empty) \\ = \ & False, \quad \textbf{if}\ a < b \\ = \ & member\ a\ Empty, \quad \textbf{if}\ a > b. \end{aligned}$$

Since $member\ a\ Empty = False$, the base case is established.

Induction step: Assume that $member\ a\ (add\ b\ y) = member\ a\ y$, if $a \neq b$ and show that $member\ a\ (add\ b\ (Add\ c\ y)) = member\ a\ (Add\ c\ y)$, if $a \neq b$. It is established in Table 3. □

Table 3. *Inductive case for Lemma 3*

Case	*member a (add b (Add c y))* =	*member a (Add c y))* =
$b = c$	*member a (Add c y)*	—
$b < c, a > b$	*member a (Add b (Add c y))* = *member a (Add c y)*	—
$b < c, a < b$	*member a (Add b (Add c y))* = *False*	*False*
$b > c, a = c$	*member a (Add c (add b y))* = *True*	*True*
$b > c, a < c$	*member a (Add c (add b y))* = *False*	*False*
$b > c, a > c$	*member a (Add c (add b y))* = *member a (add b y)* = *member a y* (by induction hypothesis)	*member a y*

Lemma 4
remove a (add b x) = add b (remove a x), **if** $a \neq b$.

We prove the lemma by induction on *OrderedSet′*.

Base step: Show *remove a (add b Empty) = add b (remove a Empty)*. By equational reasoning

$$\begin{aligned} & \textit{remove a (add b Empty)} \\ &= \textit{remove a (Add b Empty)} \\ &= \textit{Add b Empty}, \quad \textbf{if } a < b \\ &= \textit{Add b (remove a Empty)} \quad = \quad \textit{Add b Empty}, \quad \textbf{if } a > b \end{aligned}$$

$$\begin{aligned} & \textit{add b (remove a Empty)} \\ &= \textit{add b Empty} \\ &= \textit{Add b Empty}. \end{aligned}$$

This establishes the base case.

Induction step: Assume *remove a (add b y) = add b (remove a y)* and show *remove a (add b (Add c y)) = add b (remove a (Add c y))*. The various cases can be broken down depending on the relative ordering of *a*, *b* and *c*. Table 4 shows the steps of equational reasoning that complete the proof. □

Lemma 5
remove a (add a x) = remove a x.

We prove the lemma by induction on *OrderedSet′*.

Base step: show *remove a (add a Empty) = remove a Empty*. By equational reasoning

$$\textit{remove a (add a Empty)} \quad = \quad \textit{remove a (Add a Empty)} \quad = \quad \textit{Empty}$$

$$\textit{remove a Empty} \quad = \quad \textit{Empty}.$$

Table 4. *Inductive case for Lemma 4*

Case	*remove a (add b (Add c y))* =	*add b (remove a (Add c y))* =
$b < c, a < b$	*remove a (Add b (Add c y))* = *Add b (Add c y)*	*abb b (Add c y)* = *Add b (Add c y)*
$b < c, a > b$	*remove a (Add b (Add c y))* = *add b (remove a (Add c y))*	—
$b = c, a < b$	*remove a (Add c y)* = *Add c y*	*add b (Add c y)* = *Add c y*
$b = c, a > b$	*remove a (Add c y)* = *add c (remove a y)*	*add b (add c (remove a y))* = *add c (remove a y)* (by Lemma 1)
$b > c, a = c$	*remove a (Add c (add b y))* = *add b y*	*add b y*
$b > c, a < c$	*remove a (Add c (add b y))* = *Add c (add b y)*	*add b (Add c y)* = *Add c (add b y)*
$b > c, a > c$	*remove a (Add c (add b y))* = *add c (remove a (add b y))* = *add c (add b (remove a y))* (by induction hypothesis)	*add b (add c (remove a y))* = *add c (add b (remove a y))* (by Lemma 2)

Table 5. *Inductive case for Lemma 5*

Case	*remove a (add a (Add b y))* =	*remove a (Add b y)* =
$a < b$	*remove a (Add a (Add b y))* = *Add b y*	*Add b y*
$a = b$	*remove a (Add b y)*	—
$a > b$	*remove a (Add b (add a y))* *add b (remove a (add a y))* = *add b (remove a y)* (by induction hypothesis)	*add b (remove a y)*

This establishes the base case.

Induction step: Assume *remove a (add a y) = remove a y* and show *remove a (add a (Add b y)) = remove a (Add b y)*. The various cases can be broken down depending on the relative ordering of *a* and *b*. Table 5 shows the various steps of equational reasoning that complete the proof. □

Now we consider the proof of Theorem 1:

$$member\ a\ (remove\ a\ x) \quad = \quad False.$$

We use Lemmas 3–5 and induction on *OrderedSet*.

Base step: we verify the theorem for $x = empty$

$$\begin{aligned} & \mathit{member}\ a\ (\mathit{remove}\ a\ \mathit{empty}) \\ &= \mathit{member}\ a\ (\mathit{remove}\ a\ \mathit{Empty}) \\ &= \mathit{member}\ a\ \mathit{Empty} \\ &= \mathit{False}. \end{aligned}$$

Induction step: Assume that *member a* (*remove a y*) = *False* and verify that *member a* (*remove a* (*add b y*)) = *False* holds inductively. First suppose $a \neq b$

$$\begin{aligned} & \mathit{member}\ a\ (\mathit{remove}\ a\ (\mathit{add}\ b\ y)) & \\ &= \mathit{member}\ a\ (\mathit{add}\ b\ (\mathit{remove}\ a\ y)) & \text{Lemma 4} \\ &= \mathit{member}\ a\ (\mathit{remove}\ a\ y) & \text{Lemma 3} \\ &= \mathit{False} & \text{Hypothesis.} \end{aligned}$$

Now suppose $a = b$

$$\begin{aligned} & \mathit{member}\ a\ (\mathit{remove}\ a\ (\mathit{add}\ b\ y)) & \\ &= \mathit{member}\ a\ (\mathit{remove}\ a\ y) & \text{Lemma 5} \\ &= \mathit{False} & \mathit{Hypothesis.} \end{aligned}$$

This completes the proof of Theorem 1. □

Note that the final inductive proof over the ADT *OrderedSet* is supported by several lemmas established over the AFT *OrderedSet′*. These lemmas are of two kinds. Lemmas 1 and 2 establish basic properties of the ADT function *add*. In general, the primitive functions of a non-free ADT are always defined in terms of the AFT constructors, so their properties are naturally established over the AFT. Lemmas 3–5 establish properties of *member* and *remove*, which are client functions of the ADT. These properties hold, however, over the entire AFT, and their proofs take frequent advantage of the fact that the argument patterns for *remove* are defined in terms of the AFT constructors *Add* and *Empty*.

Induction over the AFT is not appropriate for Theorem 1, as it does not hold for arbitrary *x* in *OrderedSet′*. For example

$$\mathit{member}\ 3\ (\mathit{remove}\ 3\ (\mathit{Add}\ 3\ (\mathit{Add}\ 3\ \mathit{Empty}))) \quad = \quad \mathit{True}.$$

Theorem 1 states a property that holds only if we have an ordered set in canonical form, i.e. one that can be finitely constructed using *empty* and *add* only. Using the previously established lemmas, however, its proof is relatively straightforward by induction over the ADT.

4 Views revisited

As mentioned in section 2, the export of the data constructors for pattern matching compromises the representation independence of the ADT. However, this may be alleviated by merely exporting a *view* of the type which does not necessarily reflect its implementation. We thus propose to resurrect the views construct as described in an early draft of the proposed Haskell standard (Hudak *et al.*, 1988) and later removed (Hudak *et al.*, 1990), except we will require that the *toView* transformation be provided and the *fromView* transformation not be used. In terms of Wadler's (1987) original proposal for views, we propose that the *in* transformation be required and the *out* transformation be prohibited. The provided *toView* or *in* transformation maps the implementation type of the ADT to an algebraic view type, allowing the data constructors of the view type to be used for pattern matching. The *fromView* or *out* transformation would provide the inverse mapping and allow the view constructors to be used on the R.H.S. of equations for object construction. We prohibit this to avoid pitfalls in equational reasoning when the mappings are not complete inverses.

Following Hudak *et al.* (1988), the syntax of a view declaration is the same as that of an algebraic data type declaration except that

1. The keyword **view** is used in place of **data**.
2. The declaration contains a **where** part in which the *toView* transformation is defined.

In our proposal, the data constructors of a view declaration may only be exported for use in pattern matching, indicated using the prefix ? mark notation.

A good example is an ADT for complex numbers providing views for both cartesian and polar coordinates

```
module Complex_Module (Complex(?Cart), Complex(?Pole), cart, pole)
where
data  Complex  =  Cart Float Float
cart  =  Cart
view  Complex  =  Pole Float Float
   where
   toView (Cart x y)  =  Pole (sqrt(x^2+y^2)) (atan2 x y)
pole r t  =  cart (r * cos t) (r * sin t).
```

Functions on complex numbers may be defined using either view. For example

```
cadd (Cart x1 y1) (Cart x2 y2)  =  cart (x1+x2) (y1+y2)
 cmult (Pole r1 t1) (Pole r2 t2)  =  pole (r1 * r2) (t1+t2).
```

However, we do not allow mixing of views; for example, the following definition is not allowed

```
-- illegal
f (Pole 0 t)  =  0
f (Cart x y)  =  x+y.
```

Notice that we export *Cart* and *Pole* for use in patterns only, and export *cart* and *pole* for producing *Complex* values. From the client's point of view, it does not matter that we represent complex numbers using rectangular coordinates; we could easily change to a different representation without changing our client programs.

This example is similar to that presented in section 4 of Wadler's paper (1987).

However, in Wadler's example, the names *Pole* and *pole* were not distinguished. In section 10, Wadler states that for all angles *t1* and *t2*, the equation

$$Pole\ 0\ t1 \quad = \quad Pole\ 0\ t2$$

must be consistent with the rest of the program.[2] With our distinction between *Pole* and *pole* we can derive the equation:

$$\begin{aligned} toView(pole\ r\ t) \quad = \quad & Pole\ (sqrt\ ((r * cos\ t)\hat{\ }2 + (r * sin\ t)\hat{\ }2)) \\ & \qquad (atan2\ (r * cos\ t)\ (r * sin\ t)). \end{aligned}$$

We can simplify the mathematics and deal with the *toView* transformation as described below. However, we are never led to believe that *pole* = *Pole*, so avoid the problems experienced by Wadler.

Although one may think of a view as simply a different representation for the values of a given type, it is semantically modelled as a distinct type. The pattern constructors of a view are interpreted as the constructors of a free view type (FVT) that can be defined by replacing the keyword *view* by *data* and renaming the type. This is essentially the same as the approach taken by Wadler and used in the Haskell draft report.

A function f defined using the pattern constructors of a view has a direct interpretation as a function f' over the FVT. We also interpret f, in an overloaded sense, as a function f^* over the ADT, defined by the equation $f^* = f' . toView$. Since the FVT and the ADT are disjoint types, the value of $f\ x$ is uniquely defined for all x. This approach is somewhat different than the translational approach taken by Wadler.

We support equational reasoning on views with the following '*toView* elimination rule'. Given $\alpha = toView\ \beta$, one may infer that $f\ \alpha = f\ \beta$, if f is an overloaded function defined on both the ADT and the FVT. This follows from $f'\ (toView\ \beta) = f^*\ \beta$, giving $f'\ \alpha = f^*\ \beta$ when $\alpha = toView\ \beta$. Since this rule may be used in either direction, it may also be referred to as the '*toView* introduction rule'.

Consider the following view for type *Int*, which provides for the convenient definition of the *power* function using the standard divide-and-conquer algorithm

```
view Int = ZERO | DOUBLE Int | INC Int
  where
  toView n
        | n == 0              =  ZERO
        | n > 0 && even n     =  DOUBLE (n 'div' 2)
        | n > 0 && odd n      =  INC (n−1)

power x ZERO        =  1
power x (DOUBLE n)  =  y * y where y  =  power x n
power x (INC n)     =  x * power x n.
```

[2] That is, the program may contain no equations that would result in anomalies of equational reasoning. For example, the equation *angle* (*Pole r t*) = *t* would not be permitted because it would allow reasoning such as

$$1 = angle\ (Pole\ 0\ 1) = angle\ (Pole\ 0\ 2) = 2.$$

We model the view by introducing a FVT *DblView* and functions *power′* and *power** as follows

```
data  DblView  =  ZERO | DOUBLE Int | INC Int

power'  ::  Int -> DblView -> Int
power' x ZERO  =  1
power' x (DOUBLE n)  =  y * y where y = power* x n
power' x (INC n)  =  x * power*  n
power*  ::  Int -> Int -> Int
power* x n  =  power' x (toView n).
```

Note that the components stored in the *DblView* are objects of type *Int*, not of type *DblView*. Correspondingly, the *toView* transformation is not recursive.

It is interesting to compare our FVT model with the following alternative model specifying a recursive view type (RVT) *Dbl*

```
data  Dbl  =  ZERO | DOUBLE Dbl | INC Dbl
  where
  toView n
       n == 0               =  ZERO
       | n > 0 && even n    =  DOUBLE (toView (n `div` 2))
       | n > 0 && odd n     =  INC (toView (n - 1))
power'  ::  Int -> Dbl -> Int
power' x ZERO  =  1
power' x (DOUBLE n)  =  y * y   where   y = power' x n
power' x (INC n)  =  x * power' x n.
```

This model directly specifies the *ZERO-DOUBLE-INC* representation of integers as a stand-alone data type. Arguably, it more clearly illustrates the concept of defining exponentiation via divide-and-conquer on an appropriate data type.

However, the goal for our views mechanism is to provide pattern matching capabilities for different views of an ADT. From this viewpoint, application of the *DOUBLE* pattern to an *Int* should recognize if that *Int* can be characterized as the double of some other *Int*, and if so, determine that second *Int*. The *DblView* interpretation is more appropriate to this task. For example, it allows the selected object to be returned directly as a value of the ADT, as in the following definition of the function *halve*

```
halve  ::  Int -> Int
halve ZERO  =  0
halve (DOUBLE n)  =  n
halve (INC n)  =  halve n.
```

Modelled over the FVT *DblView*, these equations may be interpreted as follows

$$\begin{aligned}
&halve' \;::\; DblView \to Int\\
&halve'\ ZERO \;=\; 0\\
&halve'\ (DOUBLE\ n) \;=\; n\\
&halve'\ (INC\ n) \;=\; halve^*\ n\\
&halve^*\ n \;=\; halve'\ (toView\ n).
\end{aligned}$$

To model these equations using the recursive type *Dbl* would require that a *fromView* transformation be provided to convert the R.H.S. *n* in the second equation from a *Dbl* back to an *Int*.

Caution must be employed when using induction on views. One may be tempted to argue that *power* is well and correctly defined over the RVT constructors *ZERO*, *DOUBLE* and *INC*, and hence that it is correct for any *Int* which may be transformed to the view. However, such an argument only holds for values of the ADT which map to finite values of the RVT. In the example, finite mappings are defined for all non-negative integers, and *power* is correct over that domain.

If we changed the view slightly, as follows

```
view  Int  =  ZERO | DOUBLE Int | INC Int
  where
  toView n
    |n == 0   =  ZERO
    |even n   =  DOUBLE (n 'div' 2)
    |odd n    =  INC (n-1)
```

then the view can be used with any integer, positive, negative or zero. However, we cannot construct objects of the RVT for negative integers. An attempt to compute a negative power will result in non-termination.

Rather than using structural induction over the RVT, we can use mathematical induction on the FVT. We may demonstrate that a property holds for all elements that are visible with the view, provided it is possible to associate a non-negative integer size with each value and to insure that components of a value in the view always have smaller sizes than the value itself. For example, if we take *size* $i = i$, then we can use induction to prove that *power* is correct for non-negative integers. However, if we consider the domain of all integers, we cannot define a suitable *size* function, and hence induction cannot be used. If we take *size* $i =$ *abs* i then *size* $n >$ *size* (*INC* n) for negative n, which is not permitted. Similarly, if we take *size* $i = i$ we have problems with *DOUBLE*.

Problems like this are more obvious if we think in terms of generation rather than reduction. Clearly, we cannot generate negative numbers with zero, addition by one and doubling.

Our mechanism supports reasoning about more general views than either Wadler's original proposal or the combined *fromView* and *toView* functions of the draft

Haskell proposal. In Wadler's proposal, the *in* and *out* functions must be inverses defining an isomorphism between a subset of the ADT and a subset of the FVT. In the Haskell proposal, the constructors of the FVT were defined only for those cases in which the *toView* transformation inverts the *fromView* transformation. Our approach allows any functional relationship from the ADT to the view type. In particular, this includes non-invertible views such as projections.

5 Backward lists

We will reconsider an example examined by Wadler (1987). In his example in section 6, Wadler presents a backwards view of lists. In Wadler's notation, modified to resemble Haskell, backward lists are defined as follows

```
view  [list]  =  Nil | [alpha] `Snoc` alpha
  in (x:Nil)  =  Nil `Snoc` x
  in (x:(xs `Snoc` y))  =  (x:xs) `Snoc` y
  out (Nil `Snoc` x)  =  x:Nil
  out ((x:xs) `Snoc` y)  =  x:(xs `Snoc` y).
```

This can be restated in our proposed notation

```
view [alpha]  =  NIL | [alpha] `SNOC` alpha
  where
  toView []  =  NIL
  toView [x]  =  [] `SNOC` x
  toView (x:(xs `SNOC` y))  =  (x:xs) `SNOC` y
nil  =  []
[] `snoc` x  =  [x]
(x:xs) `snoc` y  =  x:(xs `snoc` y).
```

Wadler did not need the equations

```
toView []  =  NIL
nil  =  [],
```

since his *Nil* is used for both our *NIL* and []. Both of these equations reduce to *Nil* = *Nil* which does not need to be stated.

It appears that the recursive use of *SNOC* on the left hand side of the third equation defining *toView* was not allowed in the preliminary Haskell standard. We will leave the example as it stands, noting that this equation could be replaced with

```
toView xs  =  (init xs) `SNOC` (last xs).
```

Wadler uses the same name, *Snoc*, for both the object constructor, which we call *snoc*, and the pattern constructor, which we call *SNOC*. In Wadler's paper, the *in* and *out* equations are mirror images, so it appears that they are inverse transformations. Using Wadler's notation, it is tempting to define

$$last\ (xs\ `Snoc`\ x) \quad = \quad x$$

and reason that

$$last\ ([1,2,..]\ `Snoc`\ 0) \quad = \quad 0.$$

This is not the case, since evaluation of *last* ([*1*, *2*, ..] ‘*Snoc*‘ *0*) fails to terminate.

When we attempt to prove that

$$toView\ (xs\ `snoc`\ x) \quad = \quad xs\ `SNOC`\ x$$

we quickly find that the result holds only for finite lists. Having different names for *snoc* and *SNOC* prevents us from carelessly writing

$$last\ (xs\ `SNOC`\ x) \quad = \quad x$$

and reasoning that

$$last\ ([1,2,..]\ `snoc`\ 0) \quad = \quad 0.$$

We can easily show that

$$toView\ (xs\ `snoc`\ x) \quad = \quad xs\ `SNOC`\ x$$

for any well defined, finite list *xs*, by structural induction on the list constructors, as follows.

Base step: Show *toView* ([] ‘*snoc*‘ *x*) = [] ‘*SNOC*‘ *x*. By equational reasoning, *toView* ([] ‘*snoc*‘ *x*) = *toView* [*x*] = [] ‘*SNOC*‘ *x*.

Induction step: Assume *toView* (*xs* ‘*snoc*‘ *x*) = *xs* ‘*SNOC*‘ *x* and show that *toView* ((*y*:*xs*) ‘*snoc*‘ *x*) = (*y*:*xs*) ‘*SNOC*‘ *x*. By equational reasoning

$$\begin{array}{lll} \multicolumn{3}{l}{toView\ ((y{:}xs)\ `snoc`\ x)} \\ = & toView(y{:}(xs\ `snoc`\ x)) & \text{Def } snoc \\ = & toView\ (y{:}toView\ (xs\ `snoc`\ x)) & toView \text{ introduction} \\ = & toView(y{:}(xs\ `SNOC`\ x)) & \text{Hypothesis} \\ = & ((y{:}xs)\ `SNOC`\ x) & \text{Def } SNOC. \end{array}$$

We could use induction on the view constructors if preferred. In this case the base step would be *toView* (*nil* ‘*snoc*‘ *x*) = *NIL* ‘*SNOC*‘ *x* and the induction step would be to show that *toView* ((*xs* ‘*snoc*‘ *x*) ‘*snoc*‘ *y*) = (*xs* ‘*SNOC*‘ *x*) ‘*SNOC*‘ *y* assuming *toView* (*xs* ‘*snoc*‘ *x*) = *xs* ‘*SNOC*‘ *x*.

Of course, the corresponding rule for *NIL*

$$toView\ nil \quad = \quad NIL$$

follows directly from the defining equations.

6 Conclusion

Pattern matching is such a useful technique that it is worth supporting for other than algebraic data types.

Pattern matching is commonly viewed as a process of inverting data construction. However, if a data type does not form a free algebra, inverses do not exist. This leads to problems in these cases.

We have proposed an alternative view, in which patterns are considered to be bundled data recognition and component selection functions. We have shown that this avoids many of the pitfalls of the other approach while supporting data abstraction and equational reasoning.

Only straightforward changes to previous proposals are necessary to support this approach.

Acknowledgement

We would like to thank Simon Thompson for helpful comments on an earlier draft of this paper.

References

Bird, R. and Wadler, P. 1988. *Introduction to Functional Programming*. Prentice Hall.

Hudak, P., Wadler, P., Arvind, Boutel, B., Fairburn, J., Fasel, J., Hammond, K., Hughes, J., Johnsson, T., Kieburtz, D., Nikhil, R., Peyton Jones, S., Reeve, M., Wise, D. and Young, J. 1990. Report on the programming language Haskell: A non-strict purely functional language (Version 1.0). Technical Report YALEU/DCS/RR777, Yale University, Department of Computer Science.

Hudak, P., Peyton Jones, S. and Wadler, P., editors. 1988. Report on the Functional Programming Language Haskell: *SIGPLAN Notices*, **27** (5), May 1992.

Thompson, S. 1986. Laws in Miranda. *Proc. ACM Conference on LISP and Functional Programming*, 1–12.

Thompson, S. 1990. Lawful functions and program verification in Miranda. *Science of Computer Programming*, **13** (2–3): 181–218, May.

Turner, D. A. 1985. Miranda: A non-strict functional language with polymorphic types. In Jean-Pierre Jouannaud, editor, *Functional Programming Languages and Computer Architecture*, *Volume 201 of Lecture Notes in Computer Science*, Springer-Verlag, 1–16.

Wadler, P. 1987. Views: A way for pattern matching to cohabit with data abstraction. *Principles of Programming Languages 14*, 307–313.