A Type with a View

January 31, 1999

Abstract

All modern approaches used for the definition of abstract types in programming languages do not allow values of these types to be used in pattern-matching. This limits the use of abstract types in programs. In this paper we propose a new form of abstract type definition, based on Wadler's notion of views, which allows pattern-matching for values of abstract types.

With our approach, a type definition provides, in a simple and uniform way, either a type synonym, a datatype, an abstract type, an abstract type with views, a subtype of an existing type, or a module (collection of declarations).

The paper presents a very simple semantics of a kernel language with the proposed type definition mechanism. The mechanism can be efficiently implemented, and can use pattern-matching on constructors of the representation type whenever there is an isomorphism between (the relevant view of) the abstract type and its representation type; the decision of when such an isomorphism exists is simple and based directly on the abstract type definition.

With datatype definitions, if a type (implementation) is changed then all program parts that perform pattern-matching on constructors of values of that type need to be modified. With abstract type definitions, a representation may be changed without the need for these modifications, but no pattern-matching can be performed on values of this type. A type with a view changes this situation. Its representation (implementation) may be changed without having to change user's code, if the view (interface) is not changed.

Keywords: abstract type, pattern-matching, type definitions

1 Introduction

The following approaches exist for the definition of abstract types in programming languages.

A first approach, used for example with the abstract data type construct of SML, uses a *datatype* as the representation type. For example, a definition of an abstract data type `set`, for sets of values which can be compared for equality, defining a union operation between sets, is sketched below:

```
abstype ''a set = set of ''a list with
  ...
 fun union (set x) (set x') = set (union' x x')
```

Function `union' (x, x')` is assumed here to be defined so as to include all elements of `x` not already in `x'` (note that `x'`, representing a set, is assumed to consist of distinct elements). To define the union of sets recursively, we might write:

```
abstype ''a set = set of ''a list with
  ...
 fun union (set []) s = s
   | union (set (a::x)) (s@set y) =
       if a mem y then union (set x) s
       else let val set x' = union (set x) s
            in set (a::x')
```

Another approach, that avoids the explicit "wrapping" and "unwrapping" of constructors (required by the use of a datatype as the representation type), is adopted for example in Gofer and Hugs[Jon98]. Abstract type values are considered as values of the representation type in definitions of "functions of the abstract type", and as values of the abstract type itself elsewhere (unless stated otherwise in type signatures of the abstract type functions). The example above can be written in Gofer/Hugs as follows:

```
type Set a = [a] in ... , union

union [] s = s
union (a:x) s
  | a `elem` s = union x s
  | otherwise  = a:(union x s)
```

In this approach, the abstract type is *transparent* (i.e. synonymous with the representation type) in the definition of the abstract type functions (specified after **in**). Otherwise, the abstract type is *opaque* (the implementation is hidden, since values of the abstract type are not compatible with values of other types, and can only be used with the abstract type functions).

In Gofer and Hugs, such transparency, applied to operations overloaded for both the abstract and representation types, has an unfortunate consequence. In Gofer and Hugs, these overloaded operations never use the operation defined for the abstract type (despite the use of any type signature). Furthermore, the use of any other function which should

use itself an instance of overloaded operation defined for the abstract type, applies instead the overloaded function for the representation type.

For example, in Gofer/Hugs, the overloaded equality in `s::Set a == s'::Set a`, used in the definition of functions of abstract type `Set`, refers to list equality, instead of set equality.

In Haskell [Pe97, Tho96] and SML [MTH89, Pau96], abstract types can be defined by means of the module system. In Haskell, the approach requires the use of datatypes as representation types; an abstract type is defined by not exporting the constructors of a datatype. In this way, only the exported functions can be used to operate on values of that type. As with the abstract type construct in SML, this approach uses explicit constructors in the definition of the abstract type functions.

In another approach based on the module system, a signature (i.e. a description of the interface of a module) can be used to abstract from the representation type of a module (called a structure in SML). In such a signature, only the (so-called) arity of a type can be specified, in order to hide the representation type. It should be noted that the module system in this approach is not simply used as a means of controlling the name space.

All these approaches, used for the definition of abstract types, do not allow values of these types to be used in pattern-matching. This limits the use of abstract types in programs. In this paper we propose a new form of abstract type definition, based on that used in Gofer and Hugs, which allows pattern-matching for values of abstract types.

Abstract type definitions can use Wadler's notion of *views*[Wad87b]. A type definition can now give, in a simple and uniform way, either a type synonym, a datatype, an abstract type, an abstract type with views, a subtype of an existing type, or a module (collection of declarations). As with Wadler's views, the main motivation is to provide the possibility for values of abstract types to be used in pattern-matching, giving these values a first-class status.

It is possible to specify more than one view associated with a given abstract type. The representation can be changed without changing the abstract type view, in which case users won't have to modify their code. This enables an easy and incremetal extension of types, with propagation of type information.

With regards to the relation between abstract and representation types, it holds that:

- the representation type may have values with no abstract counterpart. For example, for an abstract type `Rational` implemented with representation type `Integer` $\times$ `Integer`, all values `(`n`,0)`, where n is an integer, have no abstract counterpart; `DateOfYear` with representation type `Day` $\times$ `Month` $\times$ `Year` may have several values with no abstract counterpart (for example, with `Year = Integer`).

- the representation type may have more than one value that have the same abstract counterpart. For example, for the `Rational` type above, (m, n) and (m', n') represent the same rational number if $m \times n' = m' \times n$; for a queue $a_1 \cdots a_n b_m \cdots b_1$ represented as a pair of lists $([a_1, \ldots, a_n], [b_1, \ldots, b_m])$, values $([], [b_1, \ldots, b_m])$ and $([b_m, \ldots, b_1], [])$ represent the same queue.

When writing an abstract type definition, an abstract value should always be guaranteed to have a valid representation. When more than one representation exists for the same abstract value, an abstraction function specifies one of them as canonical.

The paper describes the semantics of a kernel language with the proposed type definition mechanism, which can lead to a simple and efficient implementation, based on transformations to pattern-matching on values of the representation type whenever there is an isomorphism between (the relevant view of) the abstract type and its representation type; the decision of when such an isomorphism exists is simple: it exists exactly when no abstraction function is needed (for that view of the abstract type).

The type of values of the representation type is synonymous with that of the abstract type in definitions occurring inside the abstract type definition. These values are interpreted (unless explicitly indicated otherwise, by means of a type annotation), as values of the abstract type. This allows overloaded operations defined for the abstract type to be used inside the abstract type definition.

With datatype definitions, if a type (implementation) is changed then all program parts that perform pattern-matching on constructors of values of that type need to be modified. With abstract type definitions, a representation may be changed without the need for these modifications, but no pattern-matching can be performed on values of this type. A type with a view changes this situation. Its representation (implementation) may be changed without having to change user's code, if the view (interface) is not changed.

The rest of the paper is organized as follows. Section 2 presents the type definition construct by means of several Haskell-like examples, section 3 gives the semantics and section 4 concludes.

2 Proposal

This section introduces, by means of several Haskell-like examples, the ideas related to a unifying programming language construct to support the definition of synonym, data, abstract and sub-types.

2.1 Stack and Rational

We start with a very simple and widely used example, of type `Stack`:

```
type Stack a = [a]
  in emptyStack = []
     push = (:)
     pop = tail
     top = head
```

The definition patently specifies the identity isomorphism between stack and list types. In spite of this, values of type `Stack` cannot (yet) be used in pattern-matching.

The bindings for `emptyStack`, `push`, `pop` and `top` can be used to create, and operate on, values of type `Stack`.

In this example, `push` has type `a -> Stack a -> Stack a`. This means that, in the abstract type definition, values of the representation type are typed as values of the abstract type. This can be overriden by means of explicit type signatures. In the abstract type definition, the abstract and representation types are synonymous.

To enable pattern-matching, a view must be given:

```
type Stack a = [a]
  in emptyStack = []
     push = (:)
     pop = tail
     top = head
  is EmptyStack | Push a (Stack a)
```

This is all that is required for values of type `Stack` to be used in pattern-matching.

The constructors of values of type `Stack`, namely `EmptyStack` and `Push`, have corresponding bindings that give values of the representation type, respectively `emptyStack` and `push`.

Functions `pop` and `top` above could as well be defined, respectively, by `pop (Push a x) = x` and `top (Push a x) = a`. A more complete and careful definition would also give proper error indications for the cases of popping from and accessing the top element of an empty stack.

The following example defines abstract type `Rational`, illustrating the definition of an abstraction function, that occurs due to the possibility of having more than one representation for the same abstract value. Type `Rational` is defined as follows:

```
type Rational = (Integer, Integer)
  in rat (x, y) = reduce (x * signum y) (abs y)
     where reduce _ 0 = ... invalid rational number:denominator=0
           reduce x y = (x `quot` d, y `quot` d)
           d = gcd x y
  is Rat (x, y)
```

It should be noted that, in an expression like **let** `Rat (x,y) = rat (4, 2)` **in** `...` , for example, `(x,y)` is equal to `(2, 1)`, and not to `(4,2)`.

The absence of constructors and of bindings for abstract type functions and values indicates a transparent (synonymous) type definition, as in **type** `Stack a = [a]`. In the absence of the type equality and bindings, we have a data type definition, as in **type** `Stack` **is** `EmptyStack | Push a (Stack a)`. In the abscence of constructors and type equality, we have bindings, forming a module.

2.2 Type Queue

It has already been mentioned that abstract values should always be constructed so that it is guaranteed that they have a valid representation. This ensures that pattern-matching of abstract values will never fail because a value has been created with no correct abstract view. If there exists more than one valid representation for the same abstract value, an abstraction function must be defined in order to choose a given (so-called canonical) representation.

Type `Queue` defined below illustrates a case with more than one representation for the same abstract value:

```
type Queue a = ([a],[a])
  in emptyQueue = ([],[])
     ins a (f,r) = (f, a:r)
     rem ([],[]) = error ...
     rem (a:f, r) = (a, (f,r))
     rem ([], r) = rem (reverse r, [])
```

For example, values `([], [2,1])` and `([1,2], [])` represent the same queue.

In such case, a canonical representation is required for pattern-matching:

```
type Queue a = ([a],[a])
  in emptyQueue = ([],[])
     ins a (f,r) = (f, a:r)
     rem ([],[]) = error ...
     rem (a:f, r) = (a, (f,r))
     rem ([], r) = rem (reverse r, [])
  is EmptyQueue | RView a (Queue a)
     abs ([], []) = EmptyQueue
         (f, a:r) = RView a (f,r)
         (f, []) = abs ([], reverse f)
```

Thus, for example, `([], [2,1])` is the canonical representation for a queue with `2` in the rear and `1` as the only other element.

2.3 Complex with Cartesian and Polar views

This section illustrates an example with two views (cartesian and polar) for the same type (of complex numbers), which can be mixed together. Cartesian coordinates is the chosen representation:

```
type Complex = (Real, Real)
  in cart x y = (x,y)
     pole r t = (r*cos t, r*sin t)
  is Cart x y
  is Pole x y
       abs(x,y) = Pole (sqrt(x^2 + y^2), atan2 y x)
```

We can now write, for example:

```
(Cart x y) + (Cart x' y') = cart (x+x') (y+y')
(Pole r t) * (Pole r' t') = pole (r*r') (t+t')
magnitude (Pole r t) = r
abs z = cart (magnitude z) 0
```

2.4 Extending Types

Normally, one cannot change the representation of a type without having to modify every piece of code that performs pattern-matching on constructors of values of that type. No longer: with pattern-matching on abstract types, we can change the representation, modify and provide new, more efficient abstract functions that operate on the changed representation, and simply provide an abstraction function corresponding to the new representation. If the abstract view can remain the same, users will not have to modify their code.

Consider for example a binary search tree implemented by means of a binary tree:

```
type BTree a is Leaf | Node a (BTree a) (BTree a)
type BSTree a = BTree a
  in leaf = Leaf
     insert ... = ...
     delete ... = ...
     size Leaf = 0
     size (Node a t t') = 1 + size t + size t'
  is BTree a
```

We can define, for example, a function to return the so-called n-th element of the search tree, as follows:

```
gElem::  Int -> BSTree a -> a
gElem n Leaf = error ...
gElem n (Node v t t')
    | n < st = gElem (n-1) t
    | n > st = gElem (n-1-st) t'
    | n == st= v
  where st = size t
```

Suppose then that we want to change the representation of the binary search tree, by adding an extra field to hold the size of the tree. Then, we can maintain the same abstract view, and change only abstract type functions, as follows:

```
type BSTree a = L | N a (BTree a) (BTree a) Int
  in leaf = L
     insert ... = ...
     delete ... = ...
     size L = 0
     size (N a t t' n) = n
  is BTree a
     abs N a t t' n = Node a t t'
```

Function `size`, using the new modified representation, must be defined in the abstract type definition.

2.5 SubTypes

A Wadler's view[Wad87b] defines a subtype of a given (viewed) type, not a new abstract type. For example, type `Peano`[Wad87b, Section 2] is a subtype of `Int` that avoids the inefficient datatype representation **data** `Peano = Zero | Succ Peano .` We write it as follows:

```
type Peano <= Int
  is zero = 0
     succ::  Int -> Peano
     succ n | n >= 0 = n+1
  is Zero | Succ n
    abs 0 = Zero
        n = Succ(n-1)
```

The explicit signature for `succ` is used to allow, for example, `succ 4` instead of `succ(succ (succ (succ zero)))`. The latter form is still allowed, as `Peano` is a subtype of `Int` (cf. `Peano <= Int`).

The condition `n>=0` guarantees that no invalid representation is created for a value of type `Peano`.

A user of type `Peano` can write a recursive addition function on peano values as follows:

```
Zero+n = n
(Succ n')+n = succ(n+n')
```

Considering that the binding for `(+)` above involves a recursive definition, we obtain `(+)::Peano -> Peano -> Peano`.

3 Semantics of Pattern Matching

This section describes the semantics of bindings that use constructors of values of abstract types, and illustrates it with some examples.

Following the Haskell report [Pe97], the semantics is defined by translation to a simpler case expression. We need to be concerned only with the translation of case expressions, since all patterns (occurring in function bindings, lambda abstractions, pattern bindings etc.) ultimately translate into case expressions.

The semantics is based on using the abstraction function to convert values of the representation type into values of the abstract type, and on performing pattern-matching on values of the abstract type.

We consider the semantics of a case expression e_0 in the following form, to which more general case expressions may be semantically translated:

```
case x of
  p -> e
  _ -> e'
```

where x is a variable, p is a pattern of a given (view of) abstract type T, and e, e' are expressions.

We let $p \equiv C\, x_1 \cdots x_m$, for some variables $x_1, \ldots, x_m$, and define the semantics of e_0 as follows:

$$
\begin{array}{l}
[\![e_0]\!] = \textbf{if } \textbf{abs}(x) = C\, e_1 \cdots e_n, \text{ for some } e_1, \ldots, e_n, \\
\quad \textbf{then } \texttt{case } e_1 \texttt{ of} \\
\qquad\qquad x_1' \texttt{ -> } \ldots \texttt{ case } e_n \texttt{ of} \\
\qquad\qquad\qquad\qquad x_n' \texttt{ -> } [\![e[x_1 :\to x_1', \ldots, x_n :\to x_n']]\!] \\
\quad \textbf{else } [\![e']\!]
\end{array}
$$

The notation $e[x_i :\rightarrow e_i]$ denotes the textual substitution of e_i for x_i in e.

Letting $p \equiv C\, x_1 \cdots x_n$ does not impose any restrictions since, for any other patterns $p_1, \ldots, p_n$, we have:

$$
\begin{array}{l}
[\![\ \texttt{case}\ x\ \texttt{of}\ \{C\, p_1 \cdots p_n\ \texttt{->}\ e;\ \_\ \texttt{->}\ e'\}\ ]\!] = \\
\qquad \texttt{case}\ x\ \texttt{of}\ \{ \\
\qquad\quad C\, x_1 \cdots x_n\ \texttt{->}\ \texttt{case}\ x_1\ \texttt{of}\ \{ \\
\qquad\qquad\qquad\qquad p_1\ \texttt{->}\ \ldots\ \texttt{case}\ x_n\ \texttt{of}\ \{p_n\ \texttt{->}\ [\![e]\!];\ \_\ \texttt{->}\ [\![e']\!]\} \\
\qquad\qquad\qquad\qquad \_\ \texttt{->}\ [\![e']\!]\ \} \\
\qquad\quad \_\ \texttt{->}\ [\![e']\!]\ \}
\end{array}
$$

Values abtsract type T are represented as values of type R; view types (i.e. abstract types) are constructed only for pattern-matching, using the abstraction function (**abs**) defined for that view.

In some cases it is not necessary that the abstraction function be explicitly defined. These are cases where there exists a "simple" isomorphism between the abstract and representation types, in the sense that, for each constructor of the abstract type of a given arity, there exists a unique corresponding constructor of its representation type with the same arity. In such a case, whose occurrence is easy to be checked statically, the abstraction function is implicitly defined by $\mathbf{abs}(C'_i\, x_{i1} \cdots x_{ik_i}) = C_i\, x_{i1} \cdots x_{i_{k_i}}$, for each $i = 1, \ldots, n$, where $\{C_i\}$ and $\{C'_i\}$ (for $i = 1, \ldots, n$) are respectively the sets of constructors of type T and its representation type, and $C \equiv C_k$, for some $1 \leq k \leq n$.

The following example illustrates the transformation of a case expression to the form of e_0 above:

$$
\begin{array}{l}
[\![\ \texttt{case}\ e\ \texttt{of}\ \{\ \texttt{alts}\ \}\ ]\!] = \\
\qquad (\backslash x\ \texttt{->}\ \texttt{case x of}\ \{\ \texttt{alts}\ \})\ e
\end{array}
$$

It should also be noted that an implementation is not expected to use this semantics directly, since that would generate inefficient code. A more efficient implementation can use the same general ideas explored in e.g. [Wad87a].

In particular, the implementation can be based on values of the representation type, whenever no abstraction function is defined, or when the abstraction function is defined and its right-hand side does not use functions, but only variables and constructors.

For example, the abstract view of `Rational` is isomorphic to its representation type `(Integer, Integer)` (no abstraction function is defined), and the translation of:

$$\textbf{let}\ \texttt{Rat(x,y)} = \texttt{rat(x,y)}\ \textbf{in}\ \ldots$$

can be given as follows:

```
case rat(x,y) of
  (x,y) -> ...
```

The canonical representation can be used in order to keep performing pattern-matching on values of the representation type, if the abstraction function is defined by using only variables and constructors, as illustrated by the following example. The translation of:

```
size q = case q of
  EmptyQueue -> 0
  RView a q  -> 1 + size q
```

can be given as follows:

```
size q = case canonical q of
  ([], []) -> 0
  (f, a:r) = 1 + size (f,r)
```

where function **canonical** is obtained directly from the abstraction function:

```
canonical ([], []) = ([], [])
          (f, a:r) = (f, a:r)
          (f, [])  = canonical ([], reverse f)
```

If function calls are used in the definition of the abstraction function, as for example in the case of complex numbers (section 2.3), pattern-matching is done on values of the abstract type, using the abstraction function. For example, the translation of:

```
(Pole r t) * (Pole r' t') = pole (r*r') (t+t')
```

can be given as follows:

```
x * y = case abs (x) of
          Pole r t -> case abs (y) of
                        Pole r' t' -> pole (r*r') (t+t')
```

4 Conclusion

A simple language construct has been presented. It constitutes a unifying mechanism that supports the definition of type synonyms, datatypes, subtypes, modules (collections of declarations), abstract types, and abstract types with views. Abstract types with views allow pattern-matching for values of abstract types.

A very simple semantics of pattern-matching for values of abstract types has been presented. The proposed construct for the support of abstract types has a simple and efficient implementation. In particular, it can use pattern-matching on constructors of the representation type whenever there is an isomorphism between (the relevant view of) the abstract type and its representation type; the decision of when such an isomorphism exists is simple and based directly on the abstract type definition.

Further work is planned in order to implement a prototype supporting the proposed type definition construct. This will provide scope to obtain more experience on the expected advantages of this construct, in particular with respect to the ability of using pattern-matching on values of abstract types and changing the implementation of an abstract type without changing its views.

References

[Jon98] Mark Jones. Hugs – The Haskell User's Gofer System. http://www.haskell.org/hugs/, 1998.

[MTH89] Robin Milner, Mads Tofte, and Robert Harper. *The Definition of Standard ML.* MIT Press, 1989.

[Pau96] Lawrence Paulson. *ML for the Working Programmer.* Cambridge University Press, 1996. 2nd edition.

[Pe97] John Peterson and Kevin Hammond (eds.). Report on the Programming Language Haskell, A Non-strict, Purely Functional Language (Version 1.4). Technical report, Haskell committee, April 1997.

[Tho96] Simon Thompson. *Haskell: The Craft of Functional Programming.* Addison-Wesley, 1996.

[Wad87a] Philip Wadler. *Efficient Compilation of Pattern-Matching*, chapter 5 of The Implementation of Functional Programming Languages, Simon Peyton Jones, pages 78–103. Prentice-Hall, 1987.

[Wad87b] Philip Wadler. Views: a way for pattern matching to cohabit with data abstraction. *POPL'87*, 14:307–313, 1987.