REVISTA COLOMBIANA DE COMPUTACIÓN
Volumen 3, número 1
Págs. 21-40

A View on Abstract and Extensible Types*

Lucília Figueiredo† Carlos Camarão‡

Abstract

This paper presents a type declaration construct which provides 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), in the style of modern functional programming languages which provide support for parametric polymorphism. A view on an abstract type allows pattern-matching on values of this type. Subtyping can be defined by: i) restriction of the set of values of the parent type through the use of constructor functions; ii) extension of the functionality of the parent type by modifying or providing new abstract functions; iii) concrete subtyping of views. These concepts and the proposed type declaration construct support the idea that programs are composed of (extensible) type declarations, the latter being formed by function and value definitions.

1 Introduction

This paper proposes a new type declaration construct that provides 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), in the style of modern functional programming languages which provide support for parametric polymorphism. The aim of the proposed construct is to provide more expressive (modular, extensible) type declarations, by allowing the definition of:

1. Pattern-matching on values of abstract types, giving these values a first-class status.
2. Overloading of symbols between abstract and representation types.
3. Modules as windows, that is, used simply to control the visibility of names.
4. Subtyping of concrete and abstract types.

The paper presents the motivations behind the proposal and includes examples of type declarations illustrating the points above. The main focus of this paper is on pattern-matching on values of abstract types, which is described both formally and informally. A formalization of the type system for the proposed type declaration construct is not included, being a subject of ongoing work. The subset of the type system which deals with overloading is described in [5, 6].

Before describing the proposed type declaration construct, we briefly review existing approaches for the definition of abstract types in modern functional programming languages;

*This work has been partially supported by FAPEMIG

†Universidade Federal de Ouro Preto, Departamento de Computação, Instituto de Ciências Exatas e Biológicas, Ouro Preto 35400-000, Brasil, *email*: lucilia@dcc.ufmg.br

‡Universidade Federal de Minas Gerais, Departamento de Ciência da Computação, Instituto de Ciências Exatas, Belo Horizonte 31270-010, Brasil, *email*: camarao@dcc.ufmg.br

none of these allow values of abstract types to be used in pattern matching. Proposals for eliminating this restriction [33, 37, 34, 4, 12, 24] usually require the representation type of an abstract type to be a datatype (which introduces new constructors). As a consequence, a lot of "wrapping" and "unwrapping" of constructors is necessary, in the definition of the abstract type functions.

1.1 Abstract types in modern functional programming languages

The first approach we consider is the `abstype` construct of SML [19, 25], which uses a *datatype* as the representation type of an abstract type, and where the definitions of the abstract type functions are usually implemented by means of operations already defined for the representation type (this approach has now been superseded by one based on SML module system; see below). For example, a definition of an abstract data type of (polymorphic) sets, with a corresponding union operation, is sketched below (where we assume that `union'` is defined on lists):

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

For readers not familiar with SML,`''a set` and `''a list` represent polymorphic types (of sets and lists, respectively) in SML; the use of two quotes is SML's way of indicating that type variable `''a` can only be instantiated to types whose values can be compared for equality. The first occurrence of `set` in this example is the name of the (polymorphic) abstract type being defined, and the second occurrence introduces a value constructor — that, given a list which is an instance of the polymorphic type `''a list`, constructs a value of the datatype used to represent sets.

In this approach, the representation type must be a new datatype, so as to distinguish values of the representation type from values of the abstract type. Due to this requirement, there is a lot of "wrapping" and "unwrapping" of constructors in the definition of the abstract type functions (wrapping means that a constructor must be explicitly used in order to construct a value of the abstract type, and unwrapping means that pattern-matching must be used to obtain a value of the underlying type used in the representation of values of the abstract type). This can be seen, for example, if we want to define the union of sets recursively (in SML, `[]` denotes an empty list, and `(a::x)` a list with head `a` and tail `x`, so that `(::)` has then type `'a -> 'a list -> 'a list`):

```
abstype ''a set = set of ''a list with
  ...
  fun  union (set [])    s       = s
    |  union (set(a::x)) (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, which avoids this problem, is adopted for example in Gofer and Hugs [16], by means of *restricted type synonyms*. The example above can be written in Gofer/Hugs as follows:

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

union ::  Eq a => Set a -> Set a -> Set a
union []    s = s
union (a:x) s
  | a `elem` s = union x s
  | otherwise  = a:(union x s)
```

For readers not familiar with Gofer (or Haskell[17, 35]), `[a]` represents a polymorphic type of lists; the restriction that `a` can be instantiated only for types whose values can be compared for equality is indicated in the type of function `union`, by the "constraint" `Eq a`, which indicates that type variable `a` can be instantiated only for types which are instances of type class `Eq`. Type classes specify signatures for overloaded symbols (and may also specify *default* definitions); instances of type classes then give a definition for each of these symbols (except that a default definition, given for some symbol in the type class, is "inherited" if not explicitly redefined in the instance declaration). In Gofer and Haskell, `[]` denotes an empty list (as in SML), and `(a:x)` is analogous to `(a::x)` in SML. `` `elem` `` is Gofer/Haskell's way of transforming prefix to infix notation (i.e. `a `elem` s` is the same as `elem a s`); the reverse transformation is done by enclosing an operator between parentheses (e.g. `(:) a x` is the same as `a:x`). In the sequel, we will use a Haskell-like notation (see e.g. [3, 35, 14]).

The purpose of a restricted type synonym is, as its name indicates, to restrict the use of values of this type to a particular set of functions. Explicit type annotations must be used to indicate that a parameter value is (or must be) an argument of the restricted type. Also, in the definitions of the abstract type functions, values of the restricted type are considered as values of the representation type. Outside, values of these types can only be used by the functions named after **in** (in the declaration of the restricted type).

A limitation of this approach is that, in Gofer and Hugs, overloaded operations cannot be defined for both the abstract and the representation types. For example, the equality operator `(==)` cannot be defined to compare values of the abstract type `Set a`, because it is already defined for lists.

Other approaches for the definition of abstract types use the module system, as for example in Haskell and SML. In Haskell, representation types are required to be datatypes (otherwise representation and abstract types would be synonymous). An abstract type is defined by not exporting the constructors of a datatype, so that 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, so wrapping and unwrapping is a problem.

In SML, wrapping and unwrapping is avoided in the approach based on the module system. A signature (description of the interface of a module) can be used to abstract from the representation type of a module (called structure). 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, in this approach, the module system is not simply used to control the visibility of names (that is, to specify which names defined in a module are visible outside, and which names visible outside can be used inside). It is used also to change the types of symbols.

The definition of abstract types using the module system, however, also does not allow pattern-matching on values of abstract types. Chris Okasaki's proposal [24] for extending SML solves this problem, but for the definition of an abstract type it requires that a new datatype be defined as the representation type, and wrapping and unwrapping becomes again a problem.

While pattern-matching for values of abstract type and the notion of views have already been widely discussed, a construct supporting this notion has, somewhat surprisingly, never before been combined with a construct for the definition of abstract types (as done in this paper). The construct proposed in this paper also allows more than one view associated to 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 that uses the abstract type. This enables an easy and incremental extension of types, with propagation of type information.

We proceed by presenting the proposed abstract syntax of the type declaration construct (Section 2.1) and several examples that illustrate its use (Section 2.2). Section 3 gives the semantics of pattern-matching on values of abstract types. Section 4 presents more details about related work and Section 5 concludes.

2 Proposal

We firstly present the abstract syntax of a mini-language with the proposed type declaration construct (Section 2.1), and then describe it by means of several illustrative examples (Section 2.2).

2.1 Abstract Syntax

A program in our mini-language consists of a set of type declarations, one of which should contain the program's main function. Dependencies between these declarations are specified by import (use) clauses, which are omitted here for simplicity.

The abstract syntax of the language is presented in Figure 1, where meta-symbols are written in a gray box (as in ::= or |), to avoid confusion.

Meta-variable α is used for type or constructor variables and C for type constructors. Each type constructor C in $C\ \tau_1 \ldots \tau_n$ and each type variable α in $\alpha\ \tau_1 \ldots \tau_n$ is assumed to have an arty, which is the value of n. It is also assumed that there is a function type constructor ($\rightarrow$), which has arity 2 and is used in infix notation. A type variable with an arity greater than zero is called a constructor variable. Meta-variable ξ denotes a constructor variable; meta-variable x denotes the name of a term variable or function, and meta-variable c denotes a value constructor.

2.2 Pattern-matching on abstract values

For pattern-matching on values of abstract types, an *abstraction function* is used, to convert values of the representation type to values of the abstract type. As an example, an abstract type of complex numbers can use cartesian coordinates to represent complex numbers and permit pattern-matching on constructors that use polar coordinates. In this case, the abstraction function converts cartesian to polar coordinates.

The relation from abstract to representation values is, in general, a one-to-many relation (see Figure 2, taken from [15], where the abstraction function is called a *retrieve function*):

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

Program	P	::=	$D \mid D\ P$	
Type declaration	D	::=	**type** $C\ a_1 \ldots a_n$ [<= δ] [= δ] [**in** B] [V] [W]	$(n \geq 0)$
View	V	::=	**is** δ [**abs** B]	
Local Definition	W	::=	**where** L [W]	
Definition list	L	::=	$B \mid D \mid L\ L$	
Binding	B	::=	$p = e$ [W]	
Simple type	τ	::=	$a\ \tau_1 \ldots \tau_n \mid C\ \tau_1 \ldots \tau_n$	$(n \geq 0)$
Data type	δ	::=	$c_1\ \tau_{11} \ldots \tau_{1n_1} \mid \ldots \mid c_k\ \tau_{k1} \ldots \tau_{kn_k}$	$(k \geq 1, n_1, \ldots, n_k \geq 0)$
Pattern	p	::=	$x \mid c\ p_1 \ldots p_n$	$(n \geq 0)$
Expression	e	::=	$x \mid \lambda p.e \mid e\ e' \mid$ **let** b **in** e	
			$\mid$ **case** e **of** $a \mid C.x \mid \xi.x \mid$ **Self**$.x$	
Case list	a	::=	p -> e [W] $\mid a\ a'$	

Figure 1: Abstract Syntax of Type Declarations

- the representation type may have more than one value with the same abstract counterpart. For example, for type `Rational`, (m, n) and (m', n') represent the same 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 (cf. section 2.3.1).

The abstract type definition proposed in this paper is based on that used in Gofer and Hugs, but also allows for the possibility of defining views for the abstract type, thus allowing pattern-matching on values of abstract types.

The type of values of the representation type is synonymous with that of the abstract

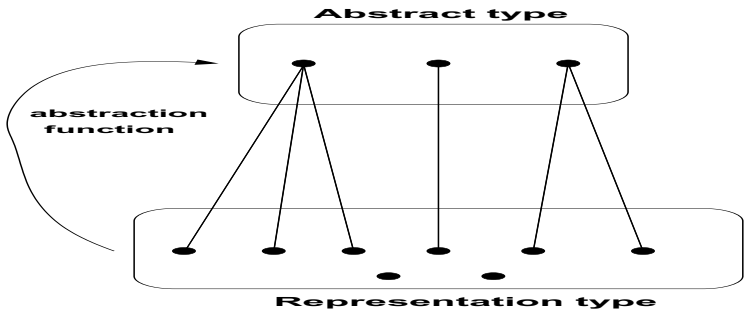

Figure 2: The abstraction function, from a representation to an abstract type

type in definitions occurring inside the abstract type definition (cf. Section 2.3). Such a value is interpreted as a value of the abstract type, unless i) there is an explicit type annotation that indicates that it is a value of the representation type, or ii) the value is an argument of an abstract type constructor function (cf. section 2.3).

2.3 Abstract versus Concrete

We present first a very simple and widely used example, of a polymorphic type of stacks, using our proposed type declaration construct:

```
type Stack a = [a]
  cons empty   = []
  in   push    = (:)
       pop     = tail
       top     = head
       isEmpty = null
```

In an abstract type declaration, *constructor* functions are defined after keyword **cons** and *transformer* and *reader* functions are defined after **in**.

Constructor functions create a value of the abstract type from values of other types given as arguments. They have type $t_1 \rightarrow t_2 \rightarrow \ldots \rightarrow t_n \rightarrow t$, where t is the abstract type and $t_i \neq t$, for all $i = 1, \ldots, n$. Transformer functions are such that at least one of the t_i's (in $t_1 \rightarrow t_2 \rightarrow \ldots \rightarrow t_n \rightarrow t$) is equal to t. Reader functions have type $t_1 \rightarrow t_2 \rightarrow \ldots \rightarrow t_n \rightarrow t_{n+1}$, where $t_{n+1} \neq t$.

Let t' be the representation type of t. In the abstract type definition, values of type t' are typed as values of the abstract type t. The substitution of t for t' occurs textually, after type inference, and is called the *the rule of textual substitution of the abstract for the representation type*. For example, in **type** `T a = [[a]]` **in** `f xss = concat (map concat xss)`, type `[[[a]]] → [a]` of `f` is transformed to `[T a]` (and not to e.g. `T [a]`). This can be overridden by means of an explicit type annotation for `f`. An abstract function type can include t' in its signature only if a type annotation is explicitly specified or t' is an argument of a constructor function of the abstract type.

The only *constructor function* of type `Stack a` is `empty`, which creates an empty stack. *Transformer* functions `push` and `pop` respectively insert and remove a value from the top of a stack. `push` has (is inferred to have) type `a -> Stack a -> Stack a` and `pop` has type `Stack a -> Stack a`. `top` gives the value at the top. It is a (so-called) *reader* function.

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

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

The constructors of values of (the view of) type `Stack a`, namely `EmptyStack` and `Push`, can be used *only* for pattern-matching. For example, functions `pop` and `top` could as well be defined by `pop (Push a x) = x` and by `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).

A note on terminology: `EmptyStack` and `Push` are *constructors* of values of (a view of) type `Stack a`, whereas `empty` is the only *constructor function* of this type (in contrast to constructors of abstract values, constructor functions may be used in expressions that are not patterns).

Constructors `EmptyStack` and `Push` have corresponding constructors of the representation type, respectively `[]` and `(:)`. This correspondence is inferred automatically from the types of the constructors of the abstract and representation types (cf. section 3). Informally, the correspondence is automatically inferred whenever, for each constructor of the abstract type of a given type, there exists a unique corresponding constructor of its representation type with "the same" type (here "the same" considers that the abstract type and its representation type are identical). When this correspondence exists, an abstraction function (which would be trivial in the example above, with bindings `[] = EmptyStack` and `(:) = Push`) does not need to be defined.

Abstract values should always be constructed so that it is guaranteed that they have a valid representation. This ensures that pattern-matching on abstract values will never fail because a value has been created with no correct abstract view. Consider, for example, the following definition of type `Rational`:

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

Function `rat`, which has (is inferred to have) type `Integer -> Integer -> Rational`, avoids having more than one representation for the same abstract value. In an expression like **let** `Rat x y = rat 4 2` **in**..., for example, `x` and `y` are equal to `2` and `1`, and not to `4` and `2` (and **let** `Rat 4 y = rat 4 2` **in**... will yield a pattern-matching failure).

Using our type declaration construct, a type declaration can 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 (discussed in Section 2.4), or a module (collection of declarations). The absence of constructors and of bindings for abstract type functions 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 a` **is** `EmptyStack | Push a (Stack a)`. In this case, constructors can be used both for pattern-matching and for construction of values of the data type.

In the absence of constructors and type equality, we have bindings, forming a module (a collection of declarations). A **where** clause may be used in a type definition, for local definitions (of internal use only). Declarations that do not occur in a **where** clause are, on the other hand, visible outside (a declared name, with its type, is said to be *exported*).

2.3.1 Abstraction Function

The following simple example illustrates the use of an abstraction function to convert cartesian to polar coordinates of complex numbers. The definition of the abstract type `Complex` uses cartesian coordinates for the representation but a polar view (for pattern-matching):

```
type Complex = (Float, Float)
  cons cart x y = (x,y)
  is Pole Float Float
     abs (x,y) = Pole (sqrt(x^2 + y^2)) (atan2 y x)
```

Consider now an example where there exist more than one valid representation for the same abstract value. In this case, an abstraction function must be defined in order to choose a given (so-called canonical) representation. A polymorphic type of queues, defined below, provides an illustrative example:

```
type Queue a = ([a],[a])
  cons emptyQ       = ([],[])
  in   enq a (f,r)  = (f, a:r)
       deq ([],[])  = error ...
       deq (a:f, r) = (a, (f,r))
       deq ([], r)  = deq (reverse r, [])
```

For example, values `([],[2,1])` and `([1,2],[])` represent the same queue. In such case, a canonical representation should be defined for pattern-matching. The significance of the abstraction function specifying one value in the representation as canonical can be seen in the following example:

```
type Queue a = ([a],[a])
  cons emptyQ        = ([],[])
  in   enq a (f,r)   = (f, a:r)
       deq EmptyQ    = error ...
       deq (Deq a q) = q
  is EmptyQ | Deq a (Queue a)
     abs ([], []) = EmptyQ
         ([], r)  = abs (reverse r, [])
         (a:f, r) = Deq a (f, r)
```

For example, `([1,2],[])` is the canonical representation for a queue with `1` in the front and `2` (as the only other element) in the rear. This is necessary so that the same bindings are produced in pattern-matching on, for example, the queues created by expressions (`enq 2 . enq 1`) `emptyQ` and (`deq . enq 2 . enq 1 . enq 0`) `emptyQ`.

We can now define, for example, a breadth first traversal on trees as follows:

```
type Tree a is EmptyT | Node (Tree a) a (Tree a)
breadthFirst t = traverse (enq t emptyQ)
   where traverse EmptyQ                = []
         traverse (Deq EmptyT q)        = traverse q
         traverse (Deq (Node t a t') q) = a: traverse((enq t'.enq t) q)
```

2.3.2 More than one view

This section illustrates the definition of more than one view for the same abstract type, which can be mixed together. Consider the following example of complex numbers, with two views (cartesian and polar), each with its own abstraction function:

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

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
```

Note that operations on complex numbers can now choose between a cartesian or polar view of complex numbers (for example, cartesian coordinates have been chosen for addition and polar coordinates for multiplication of complex numbers).

2.4 Subtyping

Subtyping of concrete types can be based on an explicit rule for each type constructor, as usual, or on overloading of value constructors. In this last case, each constructor of a subtype is also a constructor of its parent type. Subtyping of concrete types by overloading of value constructors (called *constructor overloading*) is the subject of Section 2.4.2.

Subtyping of abstract types can be based on: i) restriction of the set of values of the parent type (through the use of the functions defined in the abstract type, as illustrated by type `Nat` below), ii) extension of the functionality of the parent type by modifying or providing new transformer or reader functions or iii) subtyping of views. The subsumption property holds in all cases: a value of the subtype may be used whenever a value of its parent (super) type may be used.

Our first example of subtyping is type `Nat`[37, cf. section 2], which is a subtype of `Int` that avoids the inefficient datatype representation **type** `Nat` **is** `Zero | Succ Nat`.

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

In contrast with the case in which a new abstract type is defined (with the use of = instead of <=), values of type Nat can be used whenever values of type Int can. The explicit signature for succ is used to allow, for example, succ 3 instead of succ(succ (succ (succ zero))). The latter form is still allowed, as Nat is a subtype of Int. The condition n>=0 guarantees that no invalid representation is created for a value of type Nat.

A programmer can now write a recursive addition on values of type Nat as follows. Considering that the binding for (+) below involves a recursive definition, we obtain (+):: Nat -> Nat -> Nat.

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

A more illustrative example is given next. Simple monads are defined and used to write extendable modular interpreters, without the inconvenience of wrapping and unwrapping values of datatypes (cf. [38, 3]). We start with a trivial monad:

```
type TM a = T a
  cons return x = x
  in   T x ▷ f  = f x
is T a
```

return has type a -> TM a (note that return is a constructor function). The type inferred for (▷) is TM a -> (a -> b) -> b, which is more general than intended type TM a -> (a -> TM b) -> TM b (this intended type could be explicitly specified, if desired).

An exception monad ExcM a (where a computation may either raise an exception or return a value) may be defined as an abstract subtype of TM a (ExcM a is an extension of TM a, providing a new constructor function):

```
type ExcM a = Exc a
  cons return x       = Return x
  in   (Raise e)  ▷ f = Raise e
       (Return x) ▷ f = f x
       raise          = Raise
  is Exc a
  where type Exc a is Return a | Raise Exception
        type Exception = String
```

In this example, the definition of type Exc a is local to the definition of ExcM a; it is used both as the representation type and as a view of type ExcM a.

Note that Return and Raise — used in the body of functions return and (▷) in the example above — are used as constructors of the representation type (Exc a), which is later converted to the abstract type, to give the types of return and (▷), by the rule of textual substitution of the abstract for the representation type (Section 2.3). The types of above definitions of return and (▷) are, respectively, a -> ExcM a and ExcM a -> (a -> ExcM a) -> ExcM a.

A state monad (where a computation accepts an initial state and returns a pair with a value and a final state) may be defined as follows:

```
type StateM a State = State -> (a, State)
  cons (return x) s = (x, s)
  in   (f ▷ g)    s = (g x) s'
       where (x, s') = f s
```

When the representation type is a function type, as in the above example, the types of the abstract type functions are inferred with the assumption that function definitions are pointwise. Thus, for example, in the above definition, the type inferred for `return` is `a -> StateM a State`. The type inferred for `(▷)` is `(State -> (b,c)) -> (b -> c -> (a, State)) -> StateM a State`, a more general type than intended type `StateM a State -> (a -> StateM a State) -> StateM a State`, which could be explicitly specified, if desired.

2.4.1 Overloading

An important characteristic of our type declaration construct is the possibility of considering the use of names relative to a certain type (constructor) — i.e. names defined in a certain type declaration. Calls to symbols defined in specific types use the notation $t.f$, where t is a type constructor and the definition of f occurs inside t's definition. This is called a *qualified* function call. A chosen symbol (**Self**) may be used instead of the name of the type being defined.

The following example provides a way to view a counter as a state monad, where `State` is an integer, initialized to zero at the start of a computation and incremented with `tick`:

```
type CounterM <= StateM Int ()
  cons tick s = Self.return () (s+1)
```

The notation **Self**`.return` refers to the definition of `return` given in the definition of `StateM`, which is inherited by `CounterM`.

We now define simple modular monadic interpreters for arithmetic expressions using abstract types, with types of arithmetic expressions defined stepwisely, by extension of simpler ones. The interpreter uses the possibility of extending previous definitions, does not involve wrapping and unwrapping of constructors of newtypes, and uses overloading. Overloading is as supported by system `CT`[5], which does not require class or instance declarations, allowing the introduction of overloaded definitions in usual let-bindings. We define:

```
type TExpr a is Con a
type TEval m a = TExpr a -> m a
  cons eval (Con x) = m.return x

type TIEval m = TEval m Int
```

Considering that the type of `m.return` is `a -> m a`, the type of `eval` defined above can be inferred to be `TEval m a`.[1]

[1] If, instead of `m.return`, the definition of `eval` used just `return`, then this could cause a type error, if the type of `return` was not `a -> m a`. This would be the case if `return` is overloaded for different monad type constructors `m`. In this case, `m` would have a constrained polymorphic type: given standard definitions of `return` for different monad type constructors, the constrained polymorphic type of `return` would be written in system CT as: { `return: a -> m a`}. `a -> m a`.

An abstract subtype of `TIEval` can be defined as follows, where contravariance of the function type constructor is used, as `Expr` is a (concrete) supertype of `TExpr Int`:

```
type Expr is Con Int | Div Expr Expr
type Eval m <= TIEval m = Expr -> m Int
  cons eval (Div e e') =
            (eval e) m. ▷λa.
            (eval e')m. ▷λb.
            m.return (a `div` b)
```

Type `Eval m` inherits all abstract function equations defined for `TIEval m`, and extends it with one more. Its representation type (`Expr -> m Int`) is a subtype of the representation type of `TIEval` (namely, `TExpr Int -> m Int`).

The trivial, exception and state monads can be extended to provide a "way out" of monads, which means in these cases simply showing returned values. These extensions can be given as follows, using `show`:

```
type TM' a <= TM a
  in show (T x) = "value:  " ++ show x
```

```
type ExcM' a <= ExcM a
  in show (Raise e)  = "exception:  " ++ e
     show (Return x) = "value:  " ++ show x
```

```
type CounterM' <= CounterM
  in show f = "value:  " ++ show x ++
              "count:  " ++ show s
     where (x, s) = f 0
```

Supposing that `show` has type `a -> Int`, the types of each definition of `show` above would be, respectively, `TM' a -> String`, `ExcM' a -> String` and `(Int -> (a,b)) -> String`[2]

Given an expression `e` of type `Expr`, the results of the evaluator `eval` for arithmetic expressions, defined in `Eval m`, can be shown by annotating `Eval TM'` as below:

```
show(eval e:: Eval(TM' Int))
```

The type annotation defines parameter `m` of `Eval`, making the use of the monadic operations unambiguous in the definition of `eval`. Function `eval` can be easily defined so as to call `raise` in case of division by zero, and call `tick` to count the number of divisions performed.

Monadic operations can be combined. For example, the exception and state monads can be combined by the following distinct monad transformers:

[2]In fact, `show` would typically have a constrained polymorphic type, written in system CT as `{ show: a -> String}. a -> String` (where type variable `a` is implicitly quantified). This would cause the types of each definition of `show` above to be, respectively (with type variables implicitly quantified):

```
{ show: a -> String }. TM' a          -> String
{ show: a -> String }. ExcM' a        -> String
{ show: a -> String }. (Int -> (a,b)) -> String
```

```
type ExcMT m a = m (ExcM a)
  cons return = (m.return) . (ExcM.return)
  in   p ▷ f = p m.▷ f'
         where f' (Raise e)  = m.return (raise e)
               f' (Return x) = m.return (ExcM.return (f x))
       promote g = g m.▷ (λx. Self. return x)
```

```
type StMT m a State = State -> m (a, State)
  cons return x = (m.return) . (StateM.return x)
  in   (p ▷ f) s = m.return ((p StateM.▷ f) s)
       promote g = g StateM.▷ (λx. Self. return x)
```

2.4.2 Constructor Overloading

Subtyping originated from overloading of value constructors is illustrated in this section by considering 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
  cons leaf       = Leaf
  in   insert ... = ...
       delete ... = ...
       size Leaf = 0
       size (Node a t t') = 1 + size t + size t'
  is BTree a
```

Instead of introducing new constructors, the view may specify existing constructors of another view or datatype. This overloading of constructors indicates that `BSTree a` is a subtype of `BTree a`, based on interface subtyping. Note that there is no wrapping and unwrapping here, i.e. there is no need to use explicit constructors to convert between values of type `BTree` and `BStree`. Type `BTree` might also be defined locally, i.e. in a where clause of type `BSTree`.

A function to return the so-called n-th element of the search tree can be defined as follows:

```
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
```

Assuming that `size` is defined only for `BSTree`, the type of `gElem` is `Int->BSTree a-> a`.

2.5 Changing the Representation Type

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.

Suppose 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 below; for example, function `gElem`, defined previously, need not be modified.

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

3 Semantics of Pattern Matching

This section describes the semantics of pattern-matching on (view) values of abstract types. Following the Haskell report [17], the semantics of pattern-matching is defined by translation to a simpler case expression; we thus need to be concerned only with the translation of case expressions. We consider the semantics of a case expression e_0 in the following form, to which more general case expressions may be semantically translated (c.f. [17]):

$$\begin{array}{l}\texttt{case } x \texttt{ of}\\ \quad p \texttt{ -> } e\\ \quad \texttt{\_ -> } e'\end{array}$$

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

The semantics is based on representing values of the abstract type as values of the representation type, and on constructing values of the abstract (view) type only for pattern-matching, using the abstraction function defined for that view. Letting C be a meta-variable for type constructors and $p \equiv C\ x_1 \cdots x_m$, for some variables $x_1, \ldots, x_m$, the semantics of e_0 is given as follows (notation $e[x_i :\rightarrow e_i]$ denotes the textual substitution of e_i for x_i in e):

$$\begin{array}{l}[\![e_0]\!] = \texttt{case abs}(x) \texttt{ of } \{\\ \quad C\ e_1 \cdots e_n \texttt{ -> case } e_1 \texttt{ of } \{\\ \qquad\qquad x'_1 \texttt{ -> } \ldots \texttt{ case } e_n \texttt{ of}\\ \qquad\qquad\qquad\qquad x'_n \texttt{ -> } e[x_i :\rightarrow x'_i]^{i=1..n}[\![\}\\ \quad \texttt{\_ -> } [\![e']\!] \}\end{array}$$

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:

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

As already noted, 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 type, there exists a unique corresponding constructor of its representation type with the same type. In such a case, whose occurrence is easy to be checked statically, the abstraction function is implicitly defined by $\mathbf{abs}(C'_i\,x_{i1} \cdots x_{ik_i}) = C_i\,x_{i1} \cdots x_{i_{l_i}}$, 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 $1 \leq k \leq n$. Of course, if no such unique correspondence exists, the abstraction function must be defined explicitly. For example, in **type** `X is A | B | C Integer`, the fact that `A` and `B` have the same type requires the definition of an abstraction function in the definition of an abstract type.

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. [37]. In particular, the implementation of pattern-matching 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 (ultimately[3]) be given as follows:

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

It is easy to use the representation type and a canonical representation to perform 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
  EmptyQ -> 0
  Deq a q-> 1 + size q
```

[3]We say ultimately because let-bindings are (at least in Haskell) lazily evaluated and case expressions are strict.

can be given as follows:

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

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

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

More formally (cf. page 15), if the definition of **abs** specifies a sequence of equations $C'_i\, x_{i1} \cdots x_{ik_i} = C_i\, e_{i1} \cdots e_{il_i}$, for each $i = 1, \ldots, n$, where $\{C_i\}$ and $\{C'_i\}$ are respectively the sets of constructors of type T and its representation type, and $e_{i1} \cdots e_{il_i}$ are variables (for $i = 1, \ldots, n$), then function **canonical** can be defined, in order to perform pattern-matching on values of the representation type, from the definition of **abs**, simply by substituting **canonical** for **abs** and $C_i\ x_{i1} \cdots x_{il_i}$ for $C'_i\ e_{i1} \cdots e_{ik_i}$. Otherwise (i.e. if in $C'_i\ e_{i1} \cdots e_{ik_i}$ some expression is not a variable), as for example in the case of complex numbers (section 2.3.2), a simple implementation of pattern-matching may be based on pattern-matching on values of the abstract type and on 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 Related Work

The first effort in order to reconcile data abstraction and the definition of functions by pattern matching was made with Miranda laws [33]. Laws are equations relating the data constructors (of a data type), which are interpreted as rewriting rules defining a canonical representation for values of the defined type. Laws were not included in the final version of Miranda, because of well-known problems in equational reasoning, as discussed in [34, 4]. Besides that, it is not very convenient for pattern matching (cf. Gostanza, Peña and Núñez[12]).

Wadler's views [37] allow for the definition of arbitrary mappings between the implementation of an abstract type and views supporting pattern matching. For each view, two functions are defined: from the view to the representation (`in`) and from the representation to the view (`out`). These functions must be inverses of each other (they must be injective), which is very restrictive, but necessary in order to make equational reasoning feasible [BurtonCameron93]. There is also some loss of abstraction, since one of the views always correspond to the actual representation type and thus the representation cannot be hidden.

To solve the problems with equational reasoning in programs using laws, Thompson [34] distinguishes between two different uses of a data constructor: as a pattern and as a function that constructs a value of the viewed type.

Burton and Cameron [4] implemented this idea, making this distinction explicit into the source code of programs: constructors and the normalizing function receive different names.

Their work unifies the mechanisms of laws and views, while avoiding their problems. Pattern matching is viewed as a bundling of case recognition and component selection functions, instead of a method for inverting data construction. Only a mapping out of a view needs to be specified, eliminating the mentioned restriction of requiring an isomorphism between the representation type and the view. They show that equational reasoning can then be used.

The approach adopted for the definition of views in this paper is very similar to Burton and Cameron's [4]: view constructors are explicitly distinguished from corresponding functions and only an abstraction function from the representation to the view is required (or, in our work, is not required if there exists a simple isomorphism between the view and the representation type). However, in all these works, views are defined for datatypes; in other words, the representation type of an abstract type is required to be a new datatype. To hide the representation type, Burton and Cameron use the module system (as in Haskell), with a special notation for exporting view constructors so as to restrict their use to patterns only.

Gostanza, Peña and Nuñez's work [12] proposes a more radical extension to the notion of views. In their work, pattern matching can involve arbitrary computations (and thus guards can be used to control pattern matching). "Partial" views can be defined, and a definition of a function over an abstract type can use constructors belonging to different views. As with other approaches, a new datatype is required for the definition of an abstract type.

The combination of subtyping with parametric polymorphism and type inference has been the subject of a great number of studies [9, 11, 10, 31, 20, 8, 1, 32, 13, 36, 18, 27, 28, 30, 26, 21]. However, there exist few practical language implementations nowadays that use type inference algorithms and support parametric polymorphism and subtyping (see e.g. [29, 22], which are the basis of the languages OCAML[7] and O'Haskell[23, 2], respectively). The main reasons seem to be related to the inefficiency of these algorithms, the fact that types, even for relatively simple expressions, can become excessively long and cumbersome to read, and the use of restrictions in order to tackle the problem of type inference. This paper has focused on pattern-matching of abstract values. A formalization of the type system and the study of type subtype inference in the context of our type declaration construct are subjects of ongoing work.

5 Conclusion

This paper presents a type declaration construct which provides, in a simple 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). It supports the definition of views for an abstract type, thus allowing pattern matching on values of abstract types. Several examples illustrate the use of the proposed construct.

It has been exposed clearly in this paper the relation between representation and (views of) abstract types, clarifying when and why it is necessary to define an abstraction function.

Problems with existing constructs for the support of abstract types in modern programming languages, which were summarized in this paper, are avoided by the proposed solution.

The proposed construct has a simple and efficient implementation, according to the simple semantics of pattern-matching for values of abstract types that has been presented. 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 based directly on the abstract type definition.

The proposed type declaration construct facilitates the construction of modular programs, as well as their extension and modification. When a datatype (implementation) is

changed, 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. An abstract type with a view changes this situation. Its representation (implementation) may be changed without imposing changes on the user's code that use the abstract type, if the view is not changed.

The definition of a subtype relation between concrete types can be based on i) an explicit rule for each type constructor, as usual, or ii) on overloading of value constructors. For abstract types, on the other hand, the subtyping relation is based either on i) restricting the set of values of the parent type (through the use of the abstract type functions), or ii) the extension of the functionality of the parent type by modifying or providing new transformer or reader functions, or, finally, v) on concrete subtyping of views (interfaces). The subsumption property holds in all cases: a value of the subtype may be used whenever a value of its parent (super) type may.

Further work involves extending system CT [5, 6] in order to provide full support for the type declaration construct presented in this paper, as well as implementing a language with this 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, and of modular extension of programs through abstract subtyping.

References

[1] Alexander Aiken and Edward L. Wimmers. Type inclusion constraints and type inference. In *Proceedings of ACM Functional Programming Languages and Computer Architecture*, 1993.

[2] R. Bailey. *Functional Programming with Hope*. Ellis Horwood, 1990.

[3] Richard Bird. *Introduction to Functional Programming using Haskell*. Prentice-Hall, 1998. 2nd ed.

[4] F. Warren Burton and Robert D. Cameron. Pattern matching with abstract data types. *Journal of Functional Programming*, 3(2):171–190, April 1993.

[5] Carlos Camarão and Lucília Figueiredo. Type Inference for Overloading without Restrictions, Declarations or Annotations. *Proc. of FLOPS'99, LNCS 1722*, pages 37–52, 1999.

[6] Carlos Camarão and Lucília Figueiredo. Type Inference for Overloading. Technical report, UFMG, 2001. Submitted for publication. Available at http://www.dcc.ufmg.br/~camarao/ct-tech-rep.ps.

[7] Guy Cousineau and Michel Mauny. *The Functional Approach to Programming*. Cambridge University Press, 1998.

[8] Pierre-Lous Curien and Giorgio Ghelli. Coherence of subsumption: Minimum typing and type-checking in $F_{\leq}$. *Mathematical Structures in Computer Science*, 2:55–91, 1992. Also in Theoretical Aspects of Object-Oriented Programming: Types, Semantics, and Language Design, The MIT Press, 1993.

[9] You-Chin Fuh and Prateek Mishra. Type Inference with Subtypes. In *2nd European Symposium on Programming (ESOP'88)*, pages 94–114, 1988. Springer-Verlag LNCS 300.

[10] You-Chin Fuh and Prateek Mishra. Type inference with subtypes. *Theoretical Computer Science*, 73(2):155–175, 1990.

[11] You-Chin Fuh and Prateek Mishra. Polymorphic Subtype Inference: Closing the Theory-Practice Gap. In *Proceedings of TAPSOFT'89*, volume 2, pages 167–183, 1997.

[12] Pedro P. Gostanza, Ricardo Peña, and Manuel Nuñez. A New Look at Patternn Matching in Abstract Data Types. *ACM SIGPLAN International Conference on Functional Programming*, pages 110–121, May 1996.

[13] My Hoang and John Mitchell. Lower bounds on type inference with subtypes. *Conference Record of POPL'95: the 25th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, pages 176–185, 1995.

[14] Paul Hudak. *The Haskell School of Expression: Learning Functional Programming through Multimedia*. Cambridge University Press, 2000.

[15] Cliff B. Jones. *Systematic Software Development using VDM*. Prentice-Hall, second edition, 1990.

[16] Mark Jones et al. Hugs98. http://www.haskell.org/hugs/, 1998.

[17] Simon Peyton Jones et al. The Haskell 98 Report, 1998. http://haskell.org/definition.

[18] Simon Marlow and Philip Wadler. A practical subtyping system for Erlang. In *Proceedings of ICFP'97: the ACM SIGPLAN International Conference on Functional Programming*, pages 136–149, 1997.

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

[20] John Mitchell. Type inference with simple subtypes. *Journal of Functional Programming*, 1(3):245–285, July 1991.

[21] Johan Nordlander. Pragmatic Subtyping in Polymorphic Languages. In *Proceedings of ICFP'98: the ACM SIGPLAN International Conference on Functional Programming*, 1998. volume 34(1) of ACM SIGPLAN Notices, pages 216-227, June 1999.

[22] Johan Nordlander. Polymorphic Subtyping in O'Haskell. In *Proceedings of the APPSEm Workshop on Subtyping and Dependent Types in Programming*, 2000.

[23] Johan Nordlander. Polymorphic subtyping in o'haskell. In *Proceedings of the APPSEm Workshop on Subtyping and Dependent Types in Programming*, 2000.

[24] Chris Okasaki. Views for Standard ML. In *Proc. 1998 ACM SIGPLAN Workshop on ML*, 1998.

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

[26] François Pottier. A Framework for Type Inference with Subtyping. In *Proceedings of ICFP'98: the ACM SIGPLAN International Conference on Functional Programming*, 1998. volume 34(1) of ACM SIGPLAN Notices, pages 228-238, June 1999.

[27] Jacob Rehof. Minimal Typings in Atomic Subtyping. *Proceedings of POPL'97: the 24th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, 1997.

[28] Jakob Rehof. *The Complexity of Simple Subtyping Systems*. PhD thesis, University of Copenhagen, 1998.

[29] Didier Rémy and Jérôme Vouillon. Objective ML: An effective object-oriented extension to ML. *Theory And Practice of Object Systems*, 4(1):27–50, 1998.

[30] Dilip Sequeira. *Type Inference with Bounded Quantification*. PhD thesis, University of Edinburgh, 1998.

[31] Geoffrey Smith. *Polymorphic Type Inference for Languages with Overloading and Subtyping*. PhD thesis, Cornell University, 1991.

[32] Geoffrey Smith. Principal type schemes for functional programs with overloading and subtyping. *Science of Computer Programming*, 23:197–226, 1994.

[33] Simon Thompson. Laws in Miranda. In *Proceedings of the ACM Conference on Lisp and Functional Programming*, pages 1–12, 1986.

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

[35] Simon Thompson. *Haskell: The Craft of Functional Programming*. Addison-Wesley, 1999. second edition.

[36] Valery Trifonov and Scott Smith. Subtyping Constrained Types. *Proc. SAS'96. LNCS 1145*, pages 349–365, 1996.

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

[38] Philip Wadler. The essence of functional programming. In *Conference Record of the 19th ACM Symposium on Principles of Programming Languages*, pages 1–14, 1992.