## 4.2 User-Defined Datatypes

In this section, we describe algebraic datatypes (`data` declarations), renamed datatypes (`newtype` declarations), and type synonyms (`type` declarations). These declarations may only appear at the top level of a module.

### 4.2.1 Algebraic Datatype Declarations

| | | | |
|---|---|---|---|
| topdecl | -> | `data` [context `=>`] simpletype `=` constrs [deriving] | |
| simpletype | -> | tycon $tyvar_1$ ... $tyvar_k$ | (k>=0) |
| constrs | -> | $constr_1$ `\|` ... `\|` $constr_n$ | (n>=1) |
| constr | -> | con [`!`] $atype_1$ ... [`!`] $atype_k$ | (arity con = k, k>=0) |
| | `\|` | (btype \| `!` atype) conop (btype \| `!` atype) | (infix conop) |
| | `\|` | con `{` $fielddecl_1$ `,` ... `,` $fielddecl_n$ `}` | (n>=0) |
| fielddecl | -> | vars `::` (type \| `!` atype) | |
| deriving | -> | `deriving` (dclass \| `(`$dclass_1$`,` ... `,` $dclass_n$`)`) | (n>=0) |
| dclass | -> | qtycls | |

The precedence for *constr* is the same as that for expressions---normal constructor application has higher precedence than infix constructor application (thus `a : Foo a` parses as `a : (Foo a)`).

An algebraic datatype declaration has the form:

$$\texttt{data}\ cx \Rightarrow T\ u_1\ \ldots\ u_k = K_1\ t_{11}\ \ldots\ t_{1k_1}\ |\ \ldots |\ K_n\ t_{n1}\ \ldots\ t_{nk_n}$$

where *cx* is a context. This declaration introduces a new *type constructor T* with one or more constituent *data constructors* $K_1, ..., K_n$. In this Report, the unqualified term "constructor" always means "data constructor".

The types of the data constructors are given by:

$$K_i :: forall\ u_1\ \ldots\ u_k.\ cx_i \Rightarrow t_{i1} \rightarrow \ldots \rightarrow t_{ik_i} \rightarrow (T\ u_1\ \ldots\ u_k)$$

where $cx_i$ is the largest subset of *cx* that constrains only those type variables free in the types $t_{i1}, ..., t_{ik_i}$. The type variables $u_1$ through $u_k$ must be distinct and may appear in *cx* and the $t_{ij}$; it is a static error for any other type variable to appear in *cx* or on the right-hand-side. The new type constant *T* has a kind of the form $\kappa_1$->...->$\kappa_k$->* where the kinds $\kappa_i$ of the argument variables $u_i$ are determined by kind inference as described in Section 4.6. This means that *T* may be used in type expressions with anywhere between *0* and *k* arguments.

For example, the declaration

```
  data Eq a => Set a = NilSet | ConsSet a (Set a)
```

introduces a type constructor `Set` of kind *->*, and constructors `NilSet` and `ConsSet` with types

| | |
|---|---|
| `NilSet` | :: *forall a.* `Set` *a* |
| `ConsSet` | :: *forall a.* `Eq` *a* =>*a* ->`Set` *a* ->`Set` *a* |

In the example given, the overloaded type for `ConsSet` ensures that `ConsSet` can only be applied to values whose type is an instance of the class `Eq`. Pattern matching against `ConsSet` also gives rise to an `Eq a` constraint. For example:

```
  f (ConsSet a s) = a
```

the function `f` has inferred type `Eq a => Set a -> a`. The context in the `data` declaration has no other effect whatsoever.

The visibility of a datatype's constructors (i.e. the "abstractness" of the datatype) outside of the module in which the datatype is defined is controlled by the form of the datatype's name in the export list as described in Section 5.8.

The optional `deriving` part of a `data` declaration has to do with *derived instances*, and is described in Section

4.3.3.

### Labelled Fields

A data constructor of arity *k* creates an object with *k* components. These components are normally accessed positionally as arguments to the constructor in expressions or patterns. For large datatypes it is useful to assign *field labels* to the components of a data object. This allows a specific field to be referenced independently of its location within the constructor.

A constructor definition in a `data` declaration may assign labels to the fields of the constructor, using the record syntax (`C { ... }`). Constructors using field labels may be freely mixed with constructors without them. A constructor with associated field labels may still be used as an ordinary constructor; features using labels are simply a shorthand for operations using an underlying positional constructor. The arguments to the positional constructor occur in the same order as the labeled fields. For example, the declaration

```
  data C = F { f1,f2 :: Int, f3 :: Bool }
```

defines a type and constructor identical to the one produced by

```
  data C = F Int Int Bool
```

Operations using field labels are described in Section 3.15. A `data` declaration may use the same field label in multiple constructors as long as the typing of the field is the same in all cases after type synonym expansion. A label cannot be shared by more than one type in scope. Field names share the top level namespace with ordinary variables and class methods and must not conflict with other top level names in scope.

The pattern `F {}` matches any value built with constructor `F`, *whether or not* `F` *was declared with record syntax.*

### Strictness Flags

Whenever a data constructor is applied, each argument to the constructor is evaluated if and only if the corresponding type in the algebraic datatype declaration has a strictness flag, denoted by an exclamation point, "`!`". Lexically, "`!`" is an ordinary varsym not a *reservedop*; it has special significance only in the context of the argument types of a data declaration.

**Translation:**

A declaration of the form

`data` $cx$ `=>` $T\ u_1 \ldots u_k$ `=` $\ldots$ `|` $K\ s_1 \ldots s_n$ `|` $\ldots$

where each $s_i$ is either of the form `!` $t_i$ or $t_i$, replaces every occurrence of $K$ in an expression by

`(\` $x_1 \ldots x_n$ `->` $(\ ((K\ op_1\ x_1)\ op_2\ x_2)\ \ldots\ )\ op_n\ x_n)$

where $op_i$ is the non-strict apply function `$` if $s_i$ is of the form $t_i$, and $op_i$ is the strict apply function `$!` (see Section 6.2) if $s_i$ is of the form `!` $t_i$. Pattern matching on $K$ is not affected by strictness flags.

#### 4.2.2 Type Synonym Declarations

| | | | |
|---|---|---|---|
| topdecl | -> | `type` simpletype `=` type | |
| simpletype | -> | tycon $tyvar_1 \ldots tyvar_k$ | $(k>=0)$ |

A type synonym declaration introduces a new type that is equivalent to an old type. It has the form

`type` $T\ u_1 \ldots u_k$ `=` $t$

which introduces a new type constructor, $T$. The type $(T\ t_1 \ldots t_k)$ is equivalent to the type $t[t_1/u_1, \ldots, t_k/u_k]$. The type variables $u_1$ through $u_k$ must be distinct and are scoped only over $t$; it is a static error for any other type variable to appear in $t$. The kind of the new type constructor $T$ is of the form $\kappa_1$->...->$\kappa_k$->$\kappa$ where the kinds $\kappa_i$ of the arguments $u_i$ and $\kappa$ of the right hand side $t$ are determined by kind inference as described in Section 4.6. For example, the following definition can be used to provide an alternative way of writing the list type constructor:

```
  type List = []
```

Type constructor symbols $T$ introduced by type synonym declarations cannot be partially applied; it is a static error to use $T$ without the full number of arguments.

Although recursive and mutually recursive datatypes are allowed, this is not so for type synonyms, *unless an algebraic datatype intervenes*. For example,

```
type Rec a   =  [Circ a]
data Circ a  =  Tag [Rec a]
```

is allowed, whereas

```
type Rec a   =  [Circ a]        -- invalid
type Circ a  =  [Rec a]         -- invalid
```

is not. Similarly, `type Rec a = [Rec a]` is not allowed.

Type synonyms are a convenient, but strictly syntactic, mechanism to make type signatures more readable. A synonym and its definition are completely interchangeable, except in the instance type of an `instance` declaration (Section 4.3.2).

### 4.2.3 Datatype Renamings

| | | | |
|---|---|---|---|
| topdecl | -> | `newtype` [context `=>`] simpletype `=` newconstr [deriving] | |
| newconstr | -> | con atype | |
| | \| | con `{` var `::` type `}` | |
| simpletype | -> | tycon $\text{tyvar}_1$ ... $\text{tyvar}_k$ | $(k>=0)$ |

A declaration of the form

`newtype` $cx$ `=>` $T\ u_1\ \ldots\ u_k$ `=` $N\ t$

introduces a new type whose representation is the same as an existing type. The type `(`$T\ u_1\ \ldots\ u_k$`)` renames the datatype $t$. It differs from a type synonym in that it creates a distinct type that must be explicitly coerced to or from the original type. Also, unlike type synonyms, `newtype` may be used to define recursive types. The constructor $N$ in an expression coerces a value from type $t$ to type `(`$T\ u_1\ \ldots\ u_k$`)`. Using $N$ in a pattern coerces a value from type `(`$T\ u_1\ \ldots\ u_k$`)` to type $t$. These coercions may be implemented without execution time overhead; `newtype` does not change the underlying representation of an object.

New instances (see Section 4.3.2) can be defined for a type defined by `newtype` but may not be defined for a type synonym. A type created by `newtype` differs from an algebraic datatype in that the representation of an algebraic datatype has an extra level of indirection. This difference may make access to the representation less efficient. The difference is reflected in different rules for pattern matching (see Section 3.17). Unlike algebraic datatypes, the newtype constructor $N$ is *unlifted*, so that $N$ _|_ is the same as _|_.

The following examples clarify the differences between `data` (algebraic datatypes), `type` (type synonyms), and `newtype` (renaming types.) Given the declarations

```
data D1 = D1 Int
data D2 = D2 !Int
type S = Int
newtype N = N Int
d1 (D1 i) = 42
d2 (D2 i) = 42
s i = 42
n (N i) = 42
```

the expressions `( d1` _|_`)`, `( d2` _|_`)` and `(d2 (D2` _|_`) )` are all equivalent to _|_, whereas `( n` _|_`)`, `( n ( N ` _|_`) )`, `( d1 ( D1` _|_`) )` and `( s` _|_`)` are all equivalent to `42`. In particular, `( N` _|_`)` is equivalent to _|_ while `( D1` _|_`)` is not equivalent to _|_.

The optional deriving part of a `newtype` declaration is treated in the same way as the deriving component of a `data` declaration; see Section 4.3.3.

A `newtype` declaration may use field-naming syntax, though of course there may only be one field. Thus:

```
newtype Age = Age { unAge :: Int }
```

brings into scope both a constructor and a de-constructor:

```
Age   :: Int -> Age
unAge :: Age -> Int
```