Chapter 11

User-defined datatypes and type synonyms

11.1 Datatype definitions

In addition to the wide range of built-in datatypes described in section 7, Gofer also allows the definition of new datatypes using declarations of the form:

$$\texttt{data}\ \textit{Datatype}\ a_1 \ldots a_n \ \texttt{=}\ \textit{constr}_1\ \texttt{|}\ \ldots\ \texttt{|}\ \textit{constr}_m$$

where *Datatype* is the name of a new type constructor of arity $n \geq 0$, $a_1, \ldots, a_n$ are distinct type variables representing the arguments of *DatatypeName* and $\textit{constr}_1$, $\ldots$, $\textit{constr}_m$ ($m \geq 1$) describe the way in which elements of the new datatype are constructed. Each *constr* can take one of two forms:

- $\textit{Name}\ \textit{type}_1 \ldots \textit{type}_r$ where *Name* is a previously unused constructor function name (i.e. an identifier beginning with a capital letter). This declaration introduces *Name* as a new constructor function of type:

  $$\textit{type}_1 \rightarrow \ldots \rightarrow \textit{type}_r \rightarrow \textit{Datatype}\ a_1 \ldots a_n$$

- $\textit{type}_1 \oplus \textit{type}_2$ where $\oplus$ is a previously unused constructor function operator (i.e. an operator symbol beginning with a colon). This declaration introduces $(\oplus)$ as a new constructor function of type:

  $$\textit{type}_1 \rightarrow \textit{type}_2 \rightarrow \textit{Datatype}\ a_1 \ldots a_n$$

Only the type variables a_1, $\ldots$, a_n may appear in the type expressions in each *constr* in the definition of *Datatype*.

As a simple example, the following definition introduces a new type `Day` with elements `Sun`, `Mon`, `Tue`, `Wed`, `Thu`, `Fri` and `Sat`:

```
data Day = Sun | Mon | Tue | Wed | Thu | Fri | Sat
```

Simple functions manipulating elements of type `Day` can be defined using pattern matching:

```
what_shall_I_do Sun = "relax"
what_shall_I_do Sat = "go shopping"
what_shall_I_do _   = "looks like I'll have to go to work"
```

Another example uses a pair of constructors to provide a representation for temperatures which may be given using either of the centigrade or fahrenheit scales:

```
data Temp = Centigrade Float | Fahrenheit Float

freezing                  :: Temp -> Bool
freezing (Centigrade temp) = temp <= 0.0
freezing (Fahrenheit temp) = temp <= 32.0
```

The following example uses a type variable on the left hand side of the datatype definition to implement a `Set` type constructor for representing sets using a list of values:

```
data Set a = Set [a]
```

For example, `Set [1,2,3]` is an element of type `Set Int`, representing the set of integers $\{1, 2, 3\}$ whilst `Set ['a']` represents a singleton set of type `Set Char`. As this example shows, it is possible to use the same name simultaneously as both a type constructor and as a constructor function.

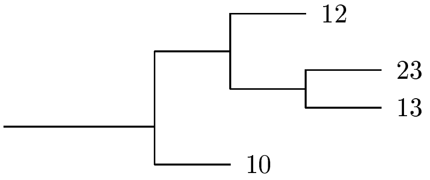
Datatype definitions may also be recursive, using the name of the datatype being defined on the right hand side of the datatype definition (mutually recursive datatype definitions are also permitted). The following example is taken from the Haskell report [5] and defines a type representing binary trees with values of a particular type at their leaves:

```
data Tree a = Lf a | Tree a :^: Tree a
```

For example,

```
(Lf 12 :^: (Lf 23 :^: Lf 13)) :^: Lf 10
```

has type `Tree Int` and represents the binary tree:



As an example of a function defined on trees, here are two definitions using recursion and pattern matching on tree valued expressions which calculate the list of elements at the leaves of a tree traversing the branches of the tree from left to right. The first definition uses a simple definition, whilst the second uses an 'accumulating parameter' giving a more efficient algorithm:

```
leaves, leaves'  :: Tree a -> [a]

leaves (Lf l)  = [l]
leaves (l:^:r) = leaves l ++ leaves r

leaves' t = leavesAcc t []
            where leavesAcc (Lf l)  = (l:)
                  leavesAcc (l:^:r) = leavesAcc l . leavesAcc r
```

Using the binary tree above as an example:

```
? leaves ((Lf 12 :^: (Lf 23 :^: Lf 13)) :^: Lf 10)
[12, 23, 13, 10]
(24 reductions, 73 cells)
? leaves' ((Lf 12 :^: (Lf 23 :^: Lf 13)) :^: Lf 10)
[12, 23, 13, 10]
(20 reductions, 58 cells)
?
```

11.2 Type synonyms

Type synonyms are used to provide convenient abbreviations for type expressions. A type synonym is introduced by a declaration of the form:

`type` *Name* $a_1 \ldots a_n$ = *expansion*

where *Name* is the name of a new type constructor of arity $n \geq 0$, $a_1, \ldots, a_n$ are distinct type variables representing the arguments of *Name* and *expansion* is a type expression. Note that the only type variables permitted in the expansion type are those on the left hand side of the synonym definition. Using this declaration any type expression of the form:

Name $type_1 \ldots type_n$

is treated as an abbreviation of the type expression obtained from expansion by replacing each of the type variables $a_1, \ldots, a_n$ with the corresponding type $type_1$, $\ldots$, $type_n$.

The most frequently used type synonym is almost certainly the `String` type which is a synonym for `[Char]`:

```
type String = [Char]
```

(This definition is actually built in to the Gofer system, but the effect is the same as if this declaration were included in the standard prelude.)

Note that the types of expressions inferred by Gofer will not usually contain any type synonyms unless an explicit type signature is given, either using an explicitly typed expression (section 10.6) or a type declaration (section 9.12):

```
? :t ['c']
['c'] :: [Char]
? :t ['c'] :: String
['c'] :: String
?
```

Unlike the datatype declarations described in the previous section, recursive (and mutually recursive) synonym declarations are not permitted. This rules out examples such as:

```
type BadSynonym = [BadSynonym]
```

and ensures that the process of expanding all of the type synonyms used in any particular type expression will always terminate. The same property does not hold for the illegal definition above, in which any attempt to expand the type BadSynonym would lead to the non-terminating sequence:

```
BadSynonym ==> [BadSynonym] ==> [[BadSynonym]] ==> ....
```