Essential Constructs of Haskell

Basic Types and Functions

For those not familiar with Haskell, we briefly repeat its essential elements. Basic types are \texttt{Int} for the integers and \texttt{Bool} for the Booleans with elements \texttt{True} and \texttt{False}. Exponentiation is written in the form \(x^n\). The operations of conjunction and disjunction are denoted by \&\& and ||, resp. These are the semi-strict versions evaluating their arguments from left to right, i.e. satisfying

\[
\begin{align*}
x \&\& y &= \text{if } x \text{ then } y \text{ else False} \\
x || y &= \text{if } x \text{ then True else } y
\end{align*}
\]

The type of functions taking elements of type \(a\) as arguments and producing elements of type \(b\) as results is \(a \to b\). The fact that a function \(f\) has this type is expressed as \(f :: a \to b\).

Function application is denoted by juxtaposing function and argument, separated by at least one blank, in the form \(f \ x\). Functions of several arguments are mostly used in curried form \(f \ x1 \ x2 \ldots \ xn\). In this case \(f\) has the higher-order type \(f :: t1 \to (t2 \to \ldots (tn \to t) \ldots)\) or, abbreviated, \(f :: t1 \to t2 \to \ldots\ \tn \to t\) (the arrow \(\to\) associates to the right, whereas function application associates to the left).

Functions are defined by equations of the form \(f \ x = E\) or as (anonymous) lambda abstractions. Instead of \(\lambda x.E\) one uses the notation \(\lambda x \to E\).

A two-place function \(f :: a \to b \to c\) may also be used as an infix operator in the form \(x \ 'f' \ y\); this is equivalent to the usual application \(f \ x \ y\). Eg. instead of \texttt{div} \(x\ y\) one may also write \(x \ 'div' \ y\). To formulate a number of expressions and properties in a more readable way we use a small notational extension of this: we also use larger expressions (that do not contain the backquote) between backquotes as infix operators. Eg. for

\[
\text{zipWith} :: (a \to b \to c) \to [a] \to [b] \to [c]
\]

we may then write \(xs \ 'zipWith\ (+)' \ ys\ for the componentwise addition of lists \(xs\) and \(ys\).

For a binary operator \(?\), by supplying only one of its arguments one obtains a \textbf{residual function} or \textbf{operator section} of the form

\[
(x \ ?) = \ \lambda y \to x \ ? y \ \text{or} \ (\ ? y) = \ \lambda x \to x \ ? y
\]

Case Distinction and Assertions

Haskell offers several possibilities for doing case distinctions. One is the usual \texttt{if then else} construct. To avoid cascades of ifs, a function may also be defined in a style similar to the one used in mathematics. The notation is

\[
f x \\
| C1 = E1 \\
\ldots \\
| Cn = En
\]
The result is the value of the first expression $E_i$ for which the corresponding $C_i$ evaluates to $\textbf{True}$. If there is none, the result is undefined.

We also use this to make functions intentionally partial in order to enforce assertions about their parameters (see Section refsc:assert and [51]).

To avoid partiality one can use the predefined constant $\textbf{otherwise} = \textbf{True}$ and add a final clause

$$| \textbf{otherwise} = \textbf{En+1}$$

Yet another way of case distinction is provided by defining a function through argument patterns. Several equations indicate what a function does on inputs that have certain shapes. The equations are tried in textual order; if no pattern matches the current argument, the function is again undefined at that point.

**Example.** By the equations

$$f 0 = 5$$
$$f 1 = 7$$

the function $f :: \textbf{Int} \rightarrow \textbf{Int}$ is defined only for argument values 0 and 1.

**Lists**

The type of lists of elements of type $a$ is denoted by $[a]$. The list consisting of elements $x_1, \ldots, x_n$ is written as $[x_1, \ldots, x_n]$; in particular, $[]$ is the empty list. Concatenation is denoted by ++. Prefixing an element to a list is denoted by the colon operator:

$$x:xs = [x] ++ xs$$

The function $\textbf{length}$ returns the length of a list. The $i$th element of list $xs$ is selected by the expression $xs!!i$ (where numbering starts with 0).

A list may be split into two parts using the functions

$$\textbf{take}, \textbf{drop} :: [a] \rightarrow \textbf{Int} \rightarrow [a]$$

For non-negative integer $k$ the list $\textbf{take} k xs$ consists of the first $k$ elements of $xs$ if $k \leq \textbf{length} xs$ and of all of $xs$ if $k > \textbf{length} xs$. For negative $k$ the expression $\textbf{take} k xs$ is undefined. The list $\textbf{drop} k xs$ results by removing $\textbf{take} k xs$ from the front of $xs$. Hence one always has

$$\textbf{take} k xs ++ \textbf{drop} k xs = xs$$

A very useful specification feature is list comprehension in the form

$$[ f x \mid x \leftarrow L, px]$$

where $L$ is a list expression, $f$ some function on the list elements and $p$ a Boolean function. The symbol $\leftarrow$ may be viewed as a leftward arrow and pronounced as “drawn from” or as a form of element sign. In this latter view, the expression is the list analogue of the usual set comprehension $\{ f x \mid x \in S, p x \}$. The meaning of the list comprehension expression $[ f x \mid x \leftarrow L, p x]$ is again a list, constructed as follows:

- The elements of list $L$ are scanned in left-to-right order.
- On each such element $x$ the test $p$ is performed.
- If $p x = \textbf{True}$, $f x$ is put into the result list.
- Otherwise, $x$ is ignored.
The list \([m, m+1, \ldots, n]\) of integers may be denoted by the shorthand \([m..n]\). The right bound \(n\) may be omitted; then the expression denotes the infinite list \([m, m+1, \ldots]\).

A useful operation on non-empty lists is the folding of their elements using a binary operator \(f : :\):

\[
\text{foldr1 } f \ [x_1, \ldots, x_n] = f \ x_1 \ (f \ x_2 \ldots \ (f \ x_{n-1} \ x_n)\ldots)
\]

Eg. \(\text{foldr1 } (+) \ s\) computes the sum of all elements of \(s\). The function \(\text{foldr1}\) itself has the type \((a \to a \to a) \to [a] \to a\).

A variant of \(\text{foldr1}\) that also copes with empty lists is \(\text{foldr}\); it uses an additional argument \(e\) that specifies the value for empty lists. The defining equations read

\[
\text{foldr } f \ e \ [] = e
\]
\[
\text{foldr } f \ e \ [x] ++ x s = f \ x \ (\text{foldr } f \ e \ x s)
\]

Based on \(\text{foldr}\) one can define a universal quantifier over lists. For a predicate \(p : : a \to \text{Bool}\) one has

\[
\text{all } p \ x s = \text{foldr } (\&\&) \ True \ [p \ x | x \leftarrow x s]
\]

So \(\text{all } p \ x s\) yields \(True\) iff \(p \ x\) yields \(True\) for all \(x\) in \(x s\).