Chapter 2 - Types and Functions

September 10, 2017

1 Haskell's type system

Types provide *abstraction* - we can know something is a string without having to understand the underlying implementation of strings.

Haskell's type system is:

- **Strong** - here meaning that you can only write expressions that obey Haskell's typing rules, and that values are not automatically cast to their correct type if they are incorrect.
- **Static** - The compiler knows the types in the program at compile time, so if there are any errors, the code won't run.
- **using Type inference** - declaring the types of expressions is optional, theu can also be deduced by the compiler.

2 Other things to note about the type system

Function application has higher precedence than using operators, so compare 2 3 == LT is the same as (compare 2 3) == LT

Function application is left associative, so a b c d is the same as ((a b) c) d. Type signatures are right associative, so $Int \rightarrow [a] \rightarrow [a]$ is the same as $Int \rightarrow ([a] \rightarrow [a])$.

To specify polymorphic types, we use a type variable, which must begin with a lowercase letter. For example, we can write functions on all lists as $[a] \rightarrow a$ or $[a] \rightarrow a$, etc. (becuse list is a polymorphic type).

Parametric polymorphism is where the type variable is substituted with the actual type on evaluation. A parameterised type in Haskell is similar to a type variable in Java generics. C++ templates also bear a resemblance to parametric polymorphism.

The empty tuple is the unit type, (). It has one value, () and is a bit like void in other languages.

The record that we use to track an unevaluated expression is referred to as a *thunk*.

3 Examples of types

$$
\begin{aligned}
&\mathit{ex1\_1} :: \mathit{Bool} \\
&\mathit{ex1\_1} = \mathit{False}
\end{aligned}
$$

$$
\begin{aligned}
&\mathit{ex1\_2} :: ([\,\mathit{String}\,], \mathit{Char}) \\
&\mathit{ex1\_2} = ([\,\texttt{"foo"}, \texttt{"bar"}\,], \texttt{'a'})
\end{aligned}
$$

$$
\begin{aligned}
&\mathit{ex1\_3} :: [\,(\mathit{Bool}, [\,[\,\mathit{Char}\,]\,])\,] \\
&\mathit{ex1\_3} = [\,(\mathit{True}, [\,]), (\mathit{False}, [\,[\,\texttt{'a'}\,]\,])\,]
\end{aligned}
$$

4 Functions used in this chapter

4.1 take

take is a function in the Prelude. take n, applied to a list xs, returns the prefix of xs of length n, or xs itself if n ¿ length xs:

$$
\begin{aligned}
&\mathit{take'} :: \mathit{Int} \to [\,a\,] \to [\,a\,] \\
&\mathit{take'}\ 0\ \_ = [\,] \\
&\mathit{take'}\ n\ [\,] = [\,] \\
&\mathit{take'}\ n\ (x : \mathit{xs}) \\
&\quad \mid n < 0 = [\,] \\
&\quad \mid n \leqslant \mathit{length}\ (x : \mathit{xs}) = x : \mathit{take'}\ (n - 1)\ \mathit{xs} \\
&\quad \mid \mathit{otherwise} = [\,]
\end{aligned}
$$

4.2 drop

drop is a function in the Prelude. drop n xs returns the suffix of xs after the first n elements, or [] if n ¿ length xs:

$$
\begin{aligned}
&\mathit{drop'} :: \mathit{Int} \to [\,a\,] \to [\,a\,] \\
&\mathit{drop'}\ 0\ \mathit{xs} = \mathit{xs}
\end{aligned}
$$

$$
\begin{aligned}
&\mathit{drop}'\ n\ [\,] = [\,]\\
&\mathit{drop}'\ n\ \mathit{xs}\\
&\quad \mid n < 0 = \mathit{xs}\\
&\quad \mid n \leqslant \mathit{length}\ \mathit{xs} = \mathit{drop}'\ (n-1)\ (\mathit{tail}\ \mathit{xs})\\
&\quad \mid \mathit{otherwise} = [\,]
\end{aligned}
$$

4.3 (

fst and snd) fst takes the first element of a pair

$$
\begin{aligned}
&\mathit{fst}' :: (a, b) \rightarrow a\\
&\mathit{fst}'\ (a, \_) = a
\end{aligned}
$$

snd takes the second element of the pair

$$
\begin{aligned}
&\mathit{snd}' :: (a, b) \rightarrow b\\
&\mathit{snd}'\ (\_, b) = b
\end{aligned}
$$

4.4 null

null indictes if a list is empty:

$$
\begin{aligned}
&\mathit{null}' :: [\,a\,] \rightarrow \mathit{Bool}\\
&\mathit{null}'\ [\,] = \mathit{True}\\
&\mathit{null}'\ \_ = \mathit{False}
\end{aligned}
$$

4.5 last

last will extract the last element of a list, which must be finite and non-empty:

$$
\begin{aligned}
&\mathit{last}' :: [\,a\,] \rightarrow a\\
&\mathit{last}'\ [\,] = \mathit{error}\ \texttt{"empty list"}\\
&\mathit{last}'\ [\,x\,] = x\\
&\mathit{last}'\ (x : \mathit{xs}) = \mathit{last}'\ \mathit{xs}
\end{aligned}
$$

$$
\begin{aligned}
&\mathit{lastButOne} :: [\,a\,] \rightarrow a\\
&\mathit{lastButOne}\ (x : y : [\,]) = x\\
&\mathit{lastButOne}\ (x : \mathit{xs}) = \mathit{lastButOne}\ \mathit{xs}\\
&\mathit{lastButOne}\ \_ = \mathit{error}\ \texttt{"Not enough elements"}
\end{aligned}
$$

if the list is empty, it throws an exception.