PROGRAMMING IT IN HASKELL

This document tells you some ways that you can write programs in Haskell. In the left-hand column are general programming advice, and suggestions about the specifics of writing in Haskell. In the right-hand column you can find examples to illustrate these ideas.

GETTING STARTED

First we need to get a clear idea of the problem. What are the inputs; what are the outputs? What are their types?

We have to think about how to use Haskell to represent the types of the problem; we first look at cases where choices are clear, and revisit types later.

We can then start our definition in Haskell. We have to name the function(s) we are going to write, and give their type(s).

We need to know this information before we start writing our definitions.

Usually we are not working from scratch; now is the time to review what we know already that might be relevant to the problem:

• What functions does the language provide already which might be useful? You can look in the standard preludes to find out about these.

• Have we solved a simpler problem before? If so, we can perhaps take the definition as a guide, or modify it to work in the new problem.

• Have we solved a related problem before?

• Can we use a function we have defined already in solving this problem?

DEFINING A FUNCTION

Function definitions in Haskell consist of a number of conditional equations. At the start of each, after the function name, there are patterns, which show to which data each equation applies. After this there may be multiple clauses, representing different cases, and a where clause to hold local definitions.

For example, we may want to find the greatest of three integers. The inputs are three integers (Int in Haskell), the output is an integer.

In the ‘greatest of three’ example we have the function maxThree of type

\[ \text{Int} \rightarrow \text{Int} \rightarrow \text{Int} \rightarrow \text{Int} \]

The type after the last arrow is the type of the result; the others are the types of the arguments (or inputs).

In the context of the example, max is useful as it gives the maximum of two arguments.

We can define maxi to take the maximum of two arguments:

\[
\text{maxi} \ a \ b \ |
\begin{array}{l}
\text{a} \geq \text{b} \\
\text{otherwise}
\end{array}
\begin{array}{l}
= \text{a} \\
= \text{b}
\end{array}
\]

We might already have found the minimum of three numbers; the two problems are close, and we can modify the minimum to maximum.

In our running example, we can in fact use maxi or max in defining maxThree:

\[
\text{maxThree} \ a \ b \ c = \text{maxi} \ a \ (\text{maxi} \ b \ c)
\]

In this section we take the running example of finding the maximum of a list of positive integers. We can begin by naming it and giving it a type, thus:

\[
\text{maxList} :: [\text{Int}] \rightarrow \text{Int}
\]

An obvious question raised by the specification is what to do about an empty list? Since we have lists of positive numbers, we can signal that a list is empty by returning the result 0 in that case.
We can start by designing the patterns to which each equation will apply. Each type has characteristic patterns which are often (but not always) used. In the case of lists we have patterns for an empty and a non-empty list; for instance.

Given the patterns we look next at how to work out their corresponding results. What will help?

- It usually helps to think of examples. These clarify the typical cases, and how the definitions might work.

- Often definitions are recursive: the value at \((a:x)\) is calculated using the value at \(x\), or the value at \(n\) is calculated from the value at \((n-1)\).

- In working out values, we maybe need to divide into cases. These give guards, which follow the vertical bars; the corresponding result is separated from the guard by an equals sign.

Can we break down how the value is calculated into a number of smaller calculations?

- We can use the where clause to make these smaller calculations, for instance.

\[
\begin{align*}
\text{maxList } (a:x) &= \begin{cases} 
\text{maxList } x & \text{maxList } x > a \\
\text{a} & \text{otherwise}
\end{cases} \\
\text{where } \text{maxL} &= \text{maxList } x
\end{align*}
\]

MORE COMPLEX DEFINITIONS: BREAKING THE PROBLEM DOWN

A problem is often solved by breaking it into parts. These parts might be functions which are to be called by other functions, or to be composed together.

- Function composition is useful in many examples. A task is broken into parts, the inputs being transformed to an intermediate value, then the result is calculated from this value.

How many characters in a list of strings?

\[
\begin{align*}
\text{charCount} : : [\text{String}] & \to \text{Int} \\
\text{countEach} : : [\text{String}] & \to [\text{Int}] \\
\text{sum} : : [\text{Int}] & \to \text{Int}
\end{align*}
\]

First find the length of each string, then sum the results giving the definition

\[
\text{charCount } \text{stList} = \text{sum } (\text{countEach } \text{stList})
\]

or directly,

\[
\text{charCount} = \text{sum } . \text{countEach}
\]
• Built in functions are helpful in suggesting ways of breaking a problem down.

We want to count the number of characters in each string in a list, that is, apply a function to every member of a list, so

\[
\begin{align*}
\text{countEach stList} & = \text{map countString stList} \\
\text{Of course, countString is built in too} & = \text{map length stList}
\end{align*}
\]

If we want to calculate the maximum of three numbers and the number of times that maximum occurs we can write

\[
\begin{align*}
\text{maxThreeCount a b c} & = (\max, \text{count}) \\
\text{where} & \\
\max & = \text{maxThree a b c} \\
\text{count} & = \text{if } a == b \&\& b == c \text{ then } 3 \\
& \quad \text{else } ...
\end{align*}
\]

Suppose we are asked to build an index for a document. We will need functions to split the document into lines, words and so on; to order words and entries etc. These can be built and tested separately.

An example occurs if we are trying to find out whether one string is a substring of another.

In deciding whether the string \(st\) is a substring of \((a:x)\), it will either be a substring of \(x\), or a substring of \((a:x)\) starting at the front: we need a function to decide the latter: \(\text{frontSubStr}\).

\[
\begin{align*}
\text{subStr st (a:x)} & = \text{subStr st x ||} \\
\text{frontSubStr st (a:x)} & 
\end{align*}
\]

A good example would be to define \([1..n]\) if it was not already built in. We start by saying

\[
[1..n] = 1:[2..n]
\]

but where do we go now? We have to define \([m..n]\) instead:

\[
\begin{align*}
[m..n] \mid m<n & = m:[m+1..n] \\
& \quad \text{otherwise } = []
\end{align*}
\]

• Another way of breaking a problem down is to write the solution using things which then have to be defined themselves in a \texttt{where} clause.

• The methods suggested here are top down: we work down from the original problem (the top). It can also be useful to work bottom up, writing functions we know we will need in our overall solution.

• Sometimes we have to solve a related problem, in addition to the original one..

• Sometimes we have to generalise a problem, seemingly making it more complicated, in order to get the solution. This happens when trying to write a recursion fails…

DESIGNING DATA TYPES

We need to know the built in types of the language.

Types can be given names.

The types can be combined to give representations of many more complex objects.

The base types are \texttt{Int} and other numerical types; \texttt{Bool} and \texttt{Char}. Compound types are tuples \((t1, t2, \ldots)\), lists \([t]\) and function types \(t1 -> t2\).

Type synonyms are given in Haskell thus:

\[
\begin{align*}
\text{type Name} & = [\text{Char}] \\
\text{type Age} & = \text{Int}
\end{align*}
\]

A person might be represented by their name and age

\[
\text{type Person} = (\text{Name, Age})
\]
In a functional programming language, functions can be thought of as arguments and results of other functions. map takes a function which is to be applied to every element of a list.

filter takes a property, which is represented as a function taking an element to a Bool, as well as the list to be filtered.

If a type contains different kinds of object, then we might well use an algebraic type to represent it. A simple example is of geometrical shapes on a two-dimensional grid.

The type will be Shape, and will contain circles, lines and triangles.

data Shape = Circle ... | Line ... | Triangle ...

Points on the grid will be represented by Point (to be defined). A circle is given by its radius and centre, a line by its end points and a triangle by its three corners:

data Shape
   = Circle Float Point | Line Point Point | Triangle Point Point Point

• First we name the type and think of the different kinds of object which it contains.

• Next we have to think of the components of the different kinds of object. This completes the definition. This process works equally well for recursive types like trees.

© Simon Thompson, May 1996