Another Look at Formal Parameters

Recall from long ago:

```
simple :: Int -> Int -> Int
simple a b = a + 3*b
```

- In the definition above, \(a\) and \(b\) are the formal parameters of `simple`.
- They act as names for input values that are passed into the function.
- Variables/names are simply one sort of pattern that can be used to represent the input that’s passed into a function.
- The simplest sorts of patterns are:
  1. Variables, such as \(a\), \(b\), `big`, `pic`
  2. Constants, such as `0`, `37`, `True`
  3. Wildcard (“don’t care”), represented as underscore (`_`)

Patterns and Tuples and Lists (Oh, My!)

Prof. Susan Older

9 February 2016

Patterns and Multiple-Equation Definitions (Take One)

We can define functions by using multiple equations:

```haskell
myFun :: Int -> Int -> Int
myFun 0 _ = 15
myFun x 0 = x+1
myFun x y = x*x + y
```

The Evaluation Rule:

Use the first equation whose patterns are matched by the input.

(But what does that mean?)

Simple Pattern Matching

A function’s formal parameters are patterns that are matched against its input (i.e., its actual parameters).

For now, very simple patterns (more powerful ones later):

<table>
<thead>
<tr>
<th>Pattern</th>
<th>Input that Matches Pattern</th>
<th>Effect (if any)</th>
</tr>
</thead>
<tbody>
<tr>
<td>Variable</td>
<td>anything</td>
<td>Binds input to variable</td>
</tr>
<tr>
<td>Constant</td>
<td>anything that evaluates to the given constant</td>
<td>None</td>
</tr>
<tr>
<td>Wildcard ((_))</td>
<td>anything</td>
<td>None</td>
</tr>
</tbody>
</table>

Examples

- Input `12*4` matches these patterns: `48`, `stuff`, `\_`
- Input `12*4` does not match these patterns: `47`, `False`
Patterns and Multiple-Equation Definitions (Take Two)

Consider the following:

myFun :: Int -> Int -> Int
myFun 0 _ = 15
myFun x 0 = x+1
myFun x y = x*x + y

Use the first equation whose patterns are matched by the input.

What are the values of the following?

1. myFun (6*2) (3-(4-1)) ⇝ 13
2. myFun (4-4) (2*3*4*5) ⇝ 15
3. myFun (7-2) (2*6) ⇝ 37
4. myFun (4-4) False
Type Error!

What’s the Point of Pattern Matching?

We will see that using patterns in equations lets us:

- Distinguish cases based on the structure of data
- Deconstruct “complex” data into simpler components and give names to those smaller pieces

Two specific ways to combine multiple values into a single package:

- Tuples
  - Contain a fixed number of items, possibly with different types
- Lists
  - Contain an arbitrary number of items, all having the same type

New Types from Old: Tuples

Tuple Types

Suppose \( t_1, t_2, \ldots, t_n \) are all types. Then there is a type

\[(t_1, t_2, \ldots, t_n),\]

whose values have form \((v_1, v_2, \ldots, v_n)\) (where \(v_i :: t_i\), for each \(i\)).

Examples

- \((\text{Char}, \text{Bool})\) has values \(\ ('A', True)\) and \(\ ('\#', False)\).
- \((\text{Int}, \text{Bool}, \text{Char}, \text{Float})\) has value \((7, False, 'M', 9.65)\).
- \((\text{Int}, \text{Char}, \text{String}, \text{Float}, \text{Bool})\) has value \((7, 'M', "abcd", 9.65, True)\).
- \((\text{Int}, \text{Char}), (\text{String}, \text{Float}, \text{Bool}))\) has value \((\(7, 'M', ("abcd", 9.65, True)\)).

Examples of Tuple Use

- \(\text{topRightQuad} :: (\text{Float}, \text{Float}) -> \text{Bool}\)
  \(\text{topRightQuad} (x,y) = x>0 \&\& y>0\)
- \(\text{topHalf} :: (\text{Float}, \text{Float}) -> \text{Bool}\)
  \(\text{topHalf} (_,y) = y>0\)
- \(\text{scale} :: \text{Int} -> (\text{Int}, \text{Int}) -> (\text{Int}, \text{Int})\)
  \(\text{scale} m (x,y) = (m*x, m*y)\)
- \(\text{minMax} :: \text{Int} -> (\text{Int}, \text{Int}) -> (\text{Int}, \text{Int})\)
  \(\text{minMax} a b = (\text{min} a b, \text{max} a b)\)
Pattern Matching Extended to Tuples

<table>
<thead>
<tr>
<th>Pattern</th>
<th>Input that Matches Pattern</th>
<th>Effect (if any)</th>
</tr>
</thead>
<tbody>
<tr>
<td>Variable</td>
<td>anything</td>
<td>Binds input to variable</td>
</tr>
<tr>
<td>Constant</td>
<td>any input that evaluates to that constant</td>
<td>None</td>
</tr>
<tr>
<td>Wildcard (_)</td>
<td>anything</td>
<td>None</td>
</tr>
<tr>
<td>((p_1, p_2, \ldots, p_n)) each (p_i) is pattern</td>
<td>((e_1, e_2, \ldots, e_n)) where each (e_i) matches (p_i) componentwise effects</td>
<td></td>
</tr>
</tbody>
</table>

\((y, (_{\ldots}, True), z)\) is matched by input \(('R', (45, not False), 12\cdot3)\):
- \(y\) is matched by `'R'` (**'R'** gets bound to name \(y\))
- \( (_{\ldots}, True)\) is matched by \((45, not False)\)
- \(z\) is matched by \(12\cdot3\) (**12\cdot3** gets bound to name \(z\))

New Types from Old: Lists

**List Types**

Suppose \(t\) is a type. Then \([t]\) is the type of lists over type \(t\).

\[
\begin{align*}
[True,False,False,True,False] & :: [Bool] \\
[5,10,15,24] & :: [Int] \\
[(5,True),(10,False),(13,True)] & :: [(Int, Bool)] \\
[[1,2,3],[10],[76,9],[3]] & :: [[Int]] \\
['q','w','e','r','t','y'] & :: [Char] (= String) \\
"qwert" & :: [Char] (= String)
\end{align*}
\]

- What’s the type of the empty list []? It’s polymorphic:
  \([] :: [a]\)

- Some handy notation:
  \([7..20] \quad [7,10..20] \quad [7..]\)

The List Constructor

\(\::\) is pronounced “cons”:

\[
(\::) :: a -> [a] -> [a]
\]

- \(30:[] \rightsquigarrow [30]\)
- \(5:(10,20,30) \rightsquigarrow [5, 10, 20, 30]\)
- \(True:[True, False] \rightsquigarrow [True, True, False]\)
- \('C':["u",'s','e"] \rightsquigarrow ["C", 'u','s','e"] (= "Cuse")
- \(1:2:3:[] \rightsquigarrow [1, 2, 3]\) (i.e., cons is right-associative)

A key idea:

Every list matches exactly one of the following two patterns:

- \([]\) (the empty list)
- \((x:xs)\) (a nonempty list)

Patterns, Tuples, and Lists

Polymorphism

**Two built-in functions:**

\(\text{fst } (x,y) = x\) -- returns first component of a pair

\(\text{snd } (x,y) = y\) -- returns second component of a pair

**What is the type of \(\text{fst}\)?**

\((\text{Int,Bool}) \rightarrow \text{Int}\)

\((\text{Char,Float}) \rightarrow \text{Char}\)

\(((\text{Bool,Float}),(\text{Int,Bool})) \rightarrow (\text{Bool,Float})\)

In fact, \(\text{fst\ and\ snd\ are\ polymorphic,\ and\ we\ write:}\)

\(\text{fst} :: (a,b) \rightarrow a\)

\(\text{snd} :: (a,b) \rightarrow b\)

(Here, \(a\) and \(b\) are type variables.)
Pattern Matching on Lists: Two Very Common Patterns

- [] is matched by empty list.
- (x:xs) is matched by nonempty list:
  first element is bound to x and rest of list is bound to xs

<table>
<thead>
<tr>
<th>Input</th>
<th>x</th>
<th>xs</th>
</tr>
</thead>
<tbody>
<tr>
<td>[1,2,3] = 1:[2,3]</td>
<td>1</td>
<td>[2,3]</td>
</tr>
<tr>
<td>[True] = True:[]</td>
<td>True</td>
<td>[]</td>
</tr>
<tr>
<td>[(5,’A’),(13,’g’)] = (5,’A’):[(13,’g’)]</td>
<td>(5,’A’)</td>
<td>[(13,’g’)]</td>
</tr>
<tr>
<td>[[1,2],[4],[8,9]] = [1,2]:[[4],[8,9]]</td>
<td>[1,2]</td>
<td>[[4],[8,9]]</td>
</tr>
</tbody>
</table>

What happens with the following patterns?

- (x:y:zs)  ((a,b):cs)

Recursion on Lists: Sample Functions

-- calculate length of a list
myLength :: [a] -> Integer
myLength [] = 0
myLength (x:xs) = 1 + myLength xs

-- calculate sum of a list of numbers
mySum :: [Integer] -> Integer
mySum [] = 0
mySum (x:xs) = x + mySum xs

-- count occurrences of particular char in string
count :: Char -> String -> Int
count ch [] = 0
count ch (c:cs)
  | ch == c = 1 + count ch cs
  | otherwise = count ch cs