Types

Well-typed programs cannot “go wrong.” — Robin Milner

- Evaluation preserves well-typedness.
- A well-typed program never gets stuck (in an undefined state).

Well-typedness is a safety property.

Safety ≡ some particular bad thing never happens.
Liveness ≡ some particular good thing eventually happens.

Haskell is a strongly typed language.

- strongly typed ⇒ good safety properties
- strongly typed ⇒ very fussy
- weakly typed ⇒ you can get away with murder
  but often you are the victim

Arguing with the type inference engine (it always wins)

```haskell
*Main> ['a',('q','z')]
<interactive>:1:6:
  Couldn't match expected type 'Char' with actual type '(t0, t1)'
  In the expression: ('q', 'z')
  In the expression: ['a', ('q', 'z')]
  In an equation for 'it': it = ['a', ('q', 'z')]

*Main> :t 'a'
'a' :: Char

*Main> :t ('q','z')
('q','z') :: (Char, Char)

*Main> it 4==5
4==5 :: Bool
```

Explicitly declaring types

```haskell
someFuns.hs

zapUpper :: [Char] -> [Char]
zapUpper cs = [ c | c <- cs, c 'notElem' ['A'..'Z'] ]

addThree :: Int -> Int -> Int -> Int
addThree x y z = x+y+z
```
Standard types

- Int
- Integer
- Float
- Double
- Bool
- Char
- Type variables: a, b, c, x..., t, t1, t2, ...
- Tuple types: (), (t1,t2), (t1,t2,t3), ...
- List types: [t]

Type classes, a first look

Type classes are “clubs” types can join.

There are:
- membership requirements,
- membership benefits, and
- membership cards you can show to get into places

Some standard type classes:

http://haskell.org/onlinereport/basic.html#sect6.3

The Eq type class

*Main> :i Eq

class Eq a where
  (==) :: a -> a -> Bool
  (=/=) :: a -> a -> Bool
-- Defined in GHC.Classes
instance Eq Integer -- Defined in ‘integer-gmp-1.0.0.0:GHC.Integer.Type’ :
  instance Eq a => Eq (Maybe a) -- Defined in ‘GHC.Base’

*Main> :t (==)
(==) :: Eq a => a -> a -> Bool

sample.hs

twoEqChar :: Char -> Char -> Char -> Bool
twoEqChar c1 c2 c3 = (c1==c2) || (c1==c3) || (c2==c3)

twoEq :: (Eq a) => a -> a -> a -> Bool
  twoEq x1 x2 x3 = (x1==x2) || (x1==x3) || (x2==x3)

Some other type classes

Ord — for types that can be put in an order

*Main> :t (<)
(<) :: Ord a => a -> a -> Bool

Show — for types that can be printed
  Read — for types that can be read
  Enum — for sequentially ordered types
Bounded — for types with lower and upper bounds
  Num — for numeric types
Floating — for floating point types
Integral — for whole number types

*Main> fromIntegral (length [1,2,3,4]) + 3.2
7.2
### Defining functions

Haskell program \(\approx\) series of definitions and comments

Haskell definition \(\approx\) type declarations + equations

#### General format

\[
\text{name} \;::\; t_1 \to t_2 \to \ldots \to t_k \to t
\]

- **argument types**
- **result type**

#### Examples

\[
\text{isPositive} \;::\; \text{Int} \to \text{Bool}
\]

\[
isPositive \;\text{num} = (\text{num}>0)
\]

\[
\text{foo} \;::\; \text{Int} \to \text{Int} \to \text{Int}
\]

\[
\text{foo} \;\text{x} \;\text{y} = \text{x} + (\text{twice} \;\text{y}) - 6
\]

### Patterns: Constants and Variables

- A function definition can be a sequence of equations.
- When a function is applied to some values, the equations are tried from top to bottom to find one that “succeeds” for these values.
- The form of the left-hand-side of a defining equation is \(\text{funName} \;\text{pat}_1 \ldots \text{pat}_n\)
- A pattern that is a constant value matches only that value.
- A pattern that is a variable matches any value.

\[
\text{lucky7} \;::\; \text{Int} \to \text{String}
\]

\[
lucky7 \;7 = "\text{You win}"
lucky7 \;x = "\text{You lose}"
\]

\[
\text{myFun} \;::\; \text{Int} \to \text{Int} \to \text{Int}
\]

\[
\text{myFun} \;\text{0} \;\text{y} = 15
\]

\[
\text{myFun} \;\text{x} \;\text{0} = \text{x} + 11
\]

\[
\text{myFun} \;\text{x} \;\text{y} = \text{x} + \text{y} * \text{y} + 3
\]

#### What happens if none of the equations succeed?

### Patterns: Tuples

Correct...

\[
\text{addVectors} \;::\; (\text{Double}, \text{Double}) \to (\text{Double}, \text{Double}) \to (\text{Double}, \text{Double})
\]

\[
\text{addVectors} \;\text{a} \;\text{b} = (\text{fst} \;\text{a} + \text{fst} \;\text{b}, \text{snd} \;\text{a} + \text{snd} \;\text{b})
\]

but this is preferred

\[
\text{addVectors'} \;::\; (\text{Double}, \text{Double}) \to (\text{Double}, \text{Double}) \to (\text{Double}, \text{Double})
\]

\[
\text{addVectors'} (\text{x}, \text{y}) (\text{x}, \text{y}) = (\text{x} + \text{x}, \text{y} + \text{y})
\]

\[
\text{first} \;::\; (\text{a}, \text{b}, \text{c}) \to \text{a}
\]

\[
\text{first} (\text{x}, \ldots, \ldots) = \text{x}
\]

\[
\text{second} \;::\; (\text{a}, \text{b}, \text{c}) \to \text{b}
\]

\[
\text{second} (\ldots, \text{y}, \ldots) = \text{y}
\]

\[
\text{third} \;::\; (\text{a}, \text{b}, \text{c}) \to \text{c}
\]

\[
\text{third} (\ldots, \ldots, \text{z}) = \text{z}
\]

\_ is the wildcard pattern—it matches anything.

### Patterns: List comprehensions

\[
\ast\text{Main}\text{>} \;[\text{a} \;\text{b} \mid (\text{a}, \text{b}) \leftarrow [(2,3),(9,4),(0,5)]]
\]

\[
[5,13,5]
\]

\[
\ast\text{Main}\text{>} \;[\text{a} \mid (\text{a}, 5) \leftarrow [(10,5),(2,3),(9,4),(0,5)]]
\]

\[
[10,0]
\]
Patterns: Lists

The \((x:xs)\) pattern

\[
\text{head'} :: [a] \rightarrow a \\
\text{head'} (x:_,:) = x \\
\text{head'} [] = \text{error } "\text{Can't call head on an empty list, silly!}"
\]

tell :: (Show a) => [a] \rightarrow String

tell [] = "The list is empty"

tell (x:[]) = "The list has one element: " ++ show x

tell (x:y:[]) = "The list has two elements: " ++ show x ++ " and " ++ show y

tell (x:y:_) = "This list is long. The first two elements are: " ++ show x ++ " and " ++ show y

\text{okAdd, betterAdd} :: (Num a) => [a] \rightarrow a

\text{okAdd} (x:y:z:) = x + y + z

\text{betterAdd} \ x s = \text{sum} (\text{take} 3 \ x s)

(Why better?)

Aside: Guards, more generally

If you have a function definition that includes the line

\[
\text{fun p1 ... pk } \mid \text{test} = \text{e} \\
\]

It means that:

- if the patterns match and \text{test} succeeds
- then return the value of \text{e}
- else don’t use this line but try the next.

E.g.:  

\[
\text{fact n } \mid n<0 = \text{error } "\text{fact given a negative argument}"
\]

\[
\text{fact 0} = 1 \\
\text{fact n} = n \times \text{fact} (n-1)
\]

Guards

Often patterns are not enough to distinguish cases

\[
\begin{array}{c|c|c|c}
\text{name p1 ... pk} & \mid \text{test1} = \text{e1} & \mid \text{test2} = \text{e2} & \mid\text{otherwise} = \text{ek} \\
\end{array}
\]

Examples

- This is a redefinition of the \text{max} function.
  \[
  \text{max} :: \text{Int} \rightarrow \text{Int} \rightarrow \text{Int}
  \]
  \[
  \text{max} \ x \ y \\
  \mid (x\leq y) = y \\
  \mid \text{otherwise} = x
  \]

- \text{maxThree} \ x \ y \ z = \text{the max of the three numbers}
  \[
  \text{maxThree} :: \text{Int} \rightarrow \text{Int} \rightarrow \text{Int}
  \]
  \[
  \text{maxThree} \ x \ y \ z \\
  \mid (x\geq y) && (x\geq z) = x \\
  \mid (y\geq z) = y \\
  \mid \text{otherwise} = z
  \]

where and let, 1

You can introduce \textit{local variables} that are visible only inside a definition. E.g.

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

\[
\text{maxSq} \ x \ y = \text{max} \ x2 \ y2
\]

\[
\text{where}
\]

\[
x2 = x*x -- x2 is a local variable to \text{maxSq}
\]

\[
y2 = y*y -- y2 is a local variable to \text{maxSq}
\]

\[
\text{maxSq'} :: \text{Int} \rightarrow \text{Int} \rightarrow \text{Int}
\]

\[
\text{maxSq'} \ x \ y = \text{max} \ (\text{sq} x) \ (\text{sq} y)
\]

\[
\text{where}
\]

\[
\text{sq} x = x \times x -- \text{sq is a function def local to maxSq'}
\]

Alternatively,

\[
\text{maxSq''} :: \text{Int} \rightarrow \text{Int} \rightarrow \text{Int}
\]

\[
\text{maxSq''} \ x \ y = \\
\text{let} \ sq x = x \times x -- sq is a function def local to maxSq'
\]

\[
in \ \text{max} \ (\text{sq} x) \ (\text{sq} y)
\]
where and let, 2

How are these two things different?

- let’s are expressions
- where is part of the syntax for function definitions

- let y = (let x = 3 in x+2) in y+11
  - x

- let y = (x+2 where x=3) in y+11
  - y

f x y

where z = x*x

let z = x*x

in f x y

describeList, describeList’ :: [a] -> String

describeList ls

= "The list is " ++ case ls of
  [] -> "empty."
  [x] -> "a singleton list."
  xs -> "a longer list."

-- alternatively

describeList’ ls = "The list is " ++ what ls

where what [] = "empty."
  what [x] = "a singleton list."
  what xs = "a longer list."

Layout: Indentation matters!!!

- Haskell has a 2D syntax*.
- Basic idea: layout determines where a definition start & stops
- The Rule: A definition ends at the first piece of text that lies at the same indentation (or to the left of) the start of that definition.

-- OK, if ugly.

fun1 :: Int -> Int
fun1 x = x
  +1

-- This is misformated

fun2 :: Int -> Int
fun2 x = x
  +1

-- This is also bad

fun3, fun4 :: Int -> Int
fun3 x = x + 10
  fun4 x = x *20

* But there are {`,}s, and ;'s around if you really, really need them.

Case statements: Pattern matching in expressions

Syntax

case expression of
  pattern1 -> result1
  pattern2 -> result2
  pattern3 -> result3
  ...

describeList', describeList'' :: [a] -> String

describeList ls

= "The list is " ++ case ls of
  [] -> "empty."
  [x] -> "a singleton list."
  xs -> "a longer list."

-- alternatively

describeList’’ ls = "The list is " ++ what ls

where what [] = "empty."
  what [x] = "a singleton list."
  what xs = "a longer list."