Software System Design and Implementation

Functional Programming

Gabriele Keller

The University of New South Wales
School of Computer Science and Engineering
Sydney, Australia
Course software

How to install Haskell (see 'Course Software' on COMP3141 website)

If you’re using a Mac and want the Haskell for Mac IDE, please reply to my email
What is functional programming?
Common properties of functional languages

• Functions are the main device to structure programs

• Based on the lambda calculus

• The use of higher-order functions is encouraged

• Side effects are used in a disciplined manner (pure functions)

• Sophisticated type systems (though, the Lisp family is dynamically typed)
Haskell

• Broad-spectrum programming language

• Widely used with over 2500 open-source libraries and tools

• Haskell is a principled language
  ‣ Purely functional
  ‣ Strictly isolating side effects
  ‣ Strongly typed with a surprisingly expressive type system

http://haskell.org/
Simple Functions in Haskell

harmonicMean x y = \(\frac{2 \times x \times y}{x + y}\)
Functions in Haskell

```
harmonicMean :: Double -> Double -> Double
harmonicMean x y = (2 * x * y)/(x + y)
```

Optional type annotation

```
c double harmonicMean(double x, double y)
{
    return (2 * x * y/(x + y));
}
```
harmonicMean x y = prod / sum

where
prod = 2 * x * y
sum  = x + y

Haskell

double harmonicMean(double x, double y)
{
    double prod = 2 * x * y;
    double sum  = x + y;
    return (prod / sum);
}

harmonicMean :: Double -> Double -> Double

Optional, but considered good style
prod :: Double
prod = 2 * x * y
sum :: Double
sum  = x + y

Can not be updated!
Just like in math.

Usually omitted

Demo
Simple functions

- Selection between multiple equations is by pattern matching or guards

Patterns

```plaintext
fact :: Int -> Int
fact 0 = 1
fact n = n * fact (n - 1)
```

Guards

```plaintext
fact :: Int -> Int
fact n | n == 0    = 1
      | otherwise = n * fact (n - 1)
```

Java

```java
int fact(int n) {
    int res = 1;
    while (n > 0) {
        res = res * n;
        n--;
    }
    return (res);
}
```
• Constants are simply functions with no arguments

```latex
theAnswer :: Int
theAnswer = 2 * 21
```
“->“ is right associative, that is

\[
\text{harmonicMean} :: \text{Double} \to \text{Double} \to \text{Double}
\]

is the same as

\[
\text{harmonicMean} :: \text{Double} \to (\text{Double} \to \text{Double})
\]

can be interpreted as a function that takes one Double value, and returns a new function as result

\[
\text{harmonicMean} :: \text{Double} \to \text{Double} \to \text{Double}
\]

\[
\text{harmonicMean} \ x \ y \ = \ (2 \times x \times y)/(x + y)
\]

\[
\text{janesFinal} :: \text{Double} \to \text{Double}
\]

\[
\text{janesFinal} \ \text{examMark} \ = \ \text{harmonicMean} \ 89 \ \text{examMark}
\]
"-> " is right associative, that is

\[ \text{harmonicMean} :: \text{Double} \rightarrow \text{Double} \rightarrow \text{Double} \]

is the same as

\[ \text{harmonicMean} :: \text{Double} \rightarrow (\text{Double} \rightarrow \text{Double}) \]

can be interpreted as a function that takes one Double value, and returns a new function as result

\[
\text{harmonicMean} :: \text{Double} \rightarrow \text{Double} \rightarrow \text{Double} \\
\text{harmonicMean} \ x \ y \ = \ (2 * x * y)/(x + y)
\]

\[
\text{janesFinal} :: \text{Double} \rightarrow \text{Double} \\
\text{janesFinal} \ \text{examMark} = \text{harmonicMean} \ 89 \ \text{examMark}
\]

\[
\text{jebsFinal} :: \text{Double} \rightarrow \text{Double} \\
\text{jebsFinal} = \text{harmonicMean} \ 75
\]
Type inference

- Compiler can work out (nearly) all types by itself
  - You can prototype, leaving out the types, and add them later

- Type signatures are documentation
  - The compiler makes sure the documentation is in sync with the code

- Type signatures catch bugs early — compiler complains if they are wrong!

This is the best of both worlds!
Higher-order functions

applyTwice :: (Double -> Double) -> Double -> Double
applyTwice fn x = fn (fn x)

> applyTwice sqrt 81
> applyTwice (\x -> x + 5) 81
Polymorphism has a different meaning in FP than in OO

Java's generics correspond to parametric polymorphism

```haskell
applyTwice :: (Double -> Double) -> Double -> Double
applyTwice fn x = fn (fn x)
```

Type variable
Types
Pre-defined types

• We already saw function types: \( a \rightarrow b \)

• We also saw elementary types: \( \text{Int, Float, Double, Char} \), and so on

• Tuples group multiple types: \((\),\((a, b), (a, b, c)\), and so on

\[
\text{harmonicMeanT} :: (\text{Double, Double}) \rightarrow \text{Double} \\
\text{harmonicMeanT} (x, y) = \frac{(2 \times x \times y)}{(x + y)}
\]

\[
\text{harmonicMeanT} :: (\text{Double, Double}) \rightarrow \text{Double} \\
\text{harmonicMeanT} \ pxy \\
= \frac{(2 \times (\text{fst pxy}) \times (\text{snd pxy}))}{(\text{fst pxy}) + (\text{snd pxy})}
\]

\[
\text{fst} :: (a, b) \rightarrow a - \text{functions defined in Prelude} \\
\text{snd} :: (a, b) \rightarrow b
\]
Lists

- Lists are either empty: \([\ ]\)

- ...or consist of a head and a tail: \(x : xs\)

- Lists are homogenous — all elements in one list have the same type

- List are parametric — different lists may contain elements of different type
Some operations on lists

• Length of a list

• Concatenating two lists

• Reversing the elements of a list

• Mapping a function over a list

In Haskell!
User-defined types

• Type synonyms (typedefs in C)

```c
type Point = (Float, Float)
```

```c
type Path = [Point]
```

• Algebraic data types

  ‣ Combination of structs and unions

  ‣ together with pointers in C
• Data types can be like structs in C (we call those data types **product types**)

```c
typedef struct {
    unsigned int x, y;
} Point2;
```

This is called a **data constructor**

```
data Point2 = MkPoint2 Float Float
```

Fields are not named, characterised by position in the definition.
data Point2 = MkPoint2 Float Float

point :: Point2
point = MkPoint2 1.3 2.45

typedef struct {
    unsigned int x, y;
} Point2;

Point2 point = {1.3, 2.45};
// or
Point2 point;
point.x = 1.3;
point.y = 2.45
data Point2 = MkPoint2
  { xPoint :: Float,
    yPoint :: Float
  }

point :: Point2
point = MkPoint2 1.3 2.45
— or
point = MkPoint2 {yPoint = 2.45, xPoint = 1.3}

typedef struct {
  unsigned int x, y;
} Point2;

Point2 point = {1.3, 2.45};
// or
Point2 point;
point.x = 1.3;
point.y = 2.45
data Point2 = MkPoint2
  { xPoint :: Float,
    yPoint :: Float
  }

distance :: Point2 -> Point2 -> Float
distance (MkPoint2 x1 y1) (MkPoint2 x2 y2) =
sqrt ((x2 - x1)^2 + (y2 - y1)^2)
• Problem: define a type to model hostnames, which can be either symbolic (string) or numeric address (4 integer values)

• Data types can be like unions in C (we call those data types sum types)

```c
enum tag {NUMERIC_IP, SYMBOLIC_IP};
struct mkNumericIP {
    enum tag theTag;
    unsigned short a, b, c, d;
}
struct mkSymbolicIP {
    enum tag theTag;
    char *hostname;
}
typedef union {
    struct mkNumericIP aNumericIP;
    struct mkSymbolicIP aSymbolicIP;
} Host;
```

```
data Host = MkNumericIP Int Int Int Int Int | MkSymbolicIP String
```
Product-Sum Types

• We call Haskell's data types also product-sum types

• They can be recursive as well

• In contrast to data types in C, but much like generics in Java and C#, Haskell data types can be parameterised

```haskell
data Maybe a = Nothing | Just a
```
Identifiers in Haskell

- Alphanumeric with underscores (_ _) and prime symbols (')

- Case matters

<table>
<thead>
<tr>
<th></th>
<th>Lower Case</th>
<th>Upper Case</th>
</tr>
</thead>
<tbody>
<tr>
<td>Functions &amp; variables</td>
<td>lower case</td>
<td>map, pi, (+), (++)</td>
</tr>
<tr>
<td>Data constructors</td>
<td>Upper case</td>
<td>True, Nothing, (:)</td>
</tr>
<tr>
<td>Type variables</td>
<td>lower case</td>
<td>a, b, c, eltType</td>
</tr>
<tr>
<td>Type constructors</td>
<td>Upper case</td>
<td>Int, Bool, IO</td>
</tr>
</tbody>
</table>
A larger example: Fractal trees