MATHFUN Lecture FP2
Introduction to Functional Programming, Part 2

Matthew Poole

moodle.port.ac.uk

School of Computing
University of Portsmouth

2014/15

Introduction to Lecture

- In the first lecture we introduced the fundamental building block of functional programs: the function definition.
- We begin this lecture by showing how to **trace** the evaluation of any expression that includes a call to a function definition.

Introduction to Lecture

- In the first lecture we introduced the fundamental building block of functional programs: the function definition.
- We begin this lecture by showing how to **trace** the evaluation of any expression that includes a call to a function definition.
- We then introduce how slightly more complex definitions can be written using **guards** (generalisations of conditionals).
- We show how functions involving guards are evaluated.

Introduction to Lecture

- In the first lecture we introduced the fundamental building block of functional programs: the function definition.
- We begin this lecture by showing how to **trace** the evaluation of any expression that includes a call to a function definition.
- We then introduce how slightly more complex definitions can be written using **guards** (generalisations of conditionals).
- We show how functions involving guards are evaluated.
- Finally, we consider how functional code can be simplified by allowing functions to be defined locally within one another.

Evaluation and calculation

- We know that we can understand imperative programs by tracing the effect of executing their statements on the program state.
- We can understand the operation of functional programs by **evaluating** expressions step-by-step (known as **calculation**).

Evaluation and calculation

- We know that we can understand imperative programs by tracing the effect of executing their statements on the program state.
- We can understand the operation of functional programs by **evaluating** expressions step-by-step (known as **calculation**).
- For example, consider the function definition:

```
twiceSum x y = 2 * (x + y)
```

- We can evaluate an expression such as

```
twiceSum 4 (2 + 6)
```

by replacing the formal parameters `x` and `y` by the expressions `4` and `(2 + 6)` in the right hand side of the definition, to give:

```
2 * (4 + (2 + 6))
```

Evaluation and calculation

- We can show a complete calculation of an expression as follows:

```
twiceSum 4 (2 + 6)
```

Evaluation and calculation

- We can show a complete calculation of an expression as follows:

$$\underline{\texttt{twiceSum 4 (2 + 6)}}$$
$$\leadsto \texttt{2 * (4 + } \underline{\texttt{(2 + 6)}}\texttt{)} \qquad\qquad \texttt{def of twiceSum}$$

Evaluation and calculation

- We can show a complete calculation of an expression as follows:

$\underline{\texttt{twiceSum 4 (2 + 6)}}$	
$\leadsto$ `2 * (4 +` $\underline{\texttt{(2 + 6)}}$`)`	`def of twiceSum`
$\leadsto$ `2 *` $\underline{\texttt{(4 + 8)}}$	`arithmetic`

Evaluation and calculation

- We can show a complete calculation of an expression as follows:

$$
\begin{array}{ll}
\underline{\texttt{twiceSum 4 (2 + 6)}} & \\
\leadsto \texttt{2 * (4 + }\underline{\texttt{(2 + 6)}}\texttt{)} & \texttt{def of twiceSum} \\
\leadsto \texttt{2 * }\underline{\texttt{(4 + 8)}} & \texttt{arithmetic} \\
\leadsto \underline{\texttt{2 * 12}} & \texttt{arithmetic}
\end{array}
$$

Evaluation and calculation

- We can show a complete calculation of an expression as follows:

$$
\begin{array}{lll}
& \underline{\texttt{twiceSum 4 (2 + 6)}} & \\
\rightsquigarrow & \texttt{2 * (4 + }\underline{\texttt{(2 + 6)}}\texttt{)} & \texttt{def of twiceSum} \\
\rightsquigarrow & \texttt{2 * }\underline{\texttt{(4 + 8)}} & \texttt{arithmetic} \\
\rightsquigarrow & \underline{\texttt{2 * 12}} & \texttt{arithmetic} \\
\rightsquigarrow & \texttt{24} & \texttt{arithmetic}
\end{array}
$$

Evaluation and calculation

- We can show a complete calculation of an expression as follows:

$\underline{\texttt{twiceSum 4 (2 + 6)}}$	
$\leadsto$ `2 * (4 + `$\underline{\texttt{(2 + 6)}}$`)`	`def of twiceSum`
$\leadsto$ `2 * `$\underline{\texttt{(4 + 8)}}$	`arithmetic`
$\leadsto$ $\underline{\texttt{2 * 12}}$	`arithmetic`
$\leadsto$ `24`	`arithmetic`

- Note that there are usually alternative ways to perform any calculation (and it doesn't matter which one we choose);

Evaluation and calculation

- We can show a complete calculation of an expression as follows:

$$
\begin{array}{lll}
 & \underline{\texttt{twiceSum 4 (2 + 6)}} & \\
\rightsquigarrow & \texttt{2 * (4 + }\underline{\texttt{(2 + 6)}}\texttt{)} & \texttt{def of twiceSum} \\
\rightsquigarrow & \texttt{2 * }\underline{\texttt{(4 + 8)}} & \texttt{arithmetic} \\
\rightsquigarrow & \underline{\texttt{2 * 12}} & \texttt{arithmetic} \\
\rightsquigarrow & \texttt{24} & \texttt{arithmetic}
\end{array}
$$

- Note that there are usually alternative ways to perform any calculation (and it doesn't matter which one we choose); e.g.:

$$
\begin{array}{lll}
 & \texttt{twiceSum 4 }\underline{\texttt{(2 + 6)}} & \\
\rightsquigarrow & \underline{\texttt{twiceSum 4 8}} & \texttt{arithmetic} \\
\rightsquigarrow & \texttt{2 * }\underline{\texttt{(4 + 8)}} & \texttt{def of twiceSum} \\
\rightsquigarrow & \underline{\texttt{2 * 12}} & \texttt{arithmetic} \\
\rightsquigarrow & \texttt{24} & \texttt{arithmetic}
\end{array}
$$

Guards

- Guards are Boolean expressions used in function definitions to give alternative results dependent on the parameter values.

Guards

- Guards are Boolean expressions used in function definitions to give alternative results dependent on the parameter values.
- The following function gives the maximum of two `Int` values:

```
maxVal :: Int -> Int -> Int
maxVal x y
    | x >= y        = x
    | otherwise     = y
```

guards

results corresponding to guards

Guards

- Guards are Boolean expressions used in function definitions to give alternative results dependent on the parameter values.
- The following function gives the maximum of two `Int` values:

```
maxVal :: Int -> Int -> Int
maxVal x y
   | x >= y          = x
   | otherwise       = y
```

guards

results corresponding to guards

- If the first guard (`x >= y`) evaluates to True, then the result is the corresponding value (`x`).
- Otherwise, if the first guard is false, the second guard is evaluated, and so on (an `otherwise` guard always holds).

Guards

- We know from last week that there is an `if...then...else...` construct, which can sometimes be used in place of guards; e.g.,

```
maxVal :: Int -> Int -> Int
maxVal x y
  = if x >= y then x else y
```

Guards

- We know from last week that there is an `if...then...else...` construct, which can sometimes be used in place of guards; e.g.,

```
maxVal :: Int -> Int -> Int
maxVal x y
  = if x >= y then x else y
```

- However, we'll tend to use guards since they more easily allow for multiple cases; for example:

```
maxThree :: Int -> Int -> Int -> Int
maxThree x y z
    | x >= y && x >= z      = x
    | y >= z                = y
    | otherwise             = z
```

Calculation involving guards

- Below is a calculation of an expression involving `maxThree`; the steps in which guards are being evaluated begin with `??`:

  <u>`maxThree 3 2 5`</u>

Calculation involving guards

- Below is a calculation of an expression involving `maxThree`; the steps in which guards are being evaluated begin with `??`:

```
maxThree 3 2 5
??  3 >= 2 && 3 >= 5                first guard
```

Calculation involving guards

- Below is a calculation of an expression involving `maxThree`; the steps in which guards are being evaluated begin with `??`:

	$\underline{\texttt{maxThree 3 2 5}}$	
`??`	$\underline{\texttt{3 >= 2}}$ `&& 3 >= 5`	`first guard`
`??`	$\leadsto$ `True &&` $\underline{\texttt{3 >= 5}}$	`def of >=`

Calculation involving guards

- Below is a calculation of an expression involving `maxThree`; the steps in which guards are being evaluated begin with `??`:

```
maxThree 3 2 5
??  3 >= 2 && 3 >= 5              first guard
??  ⇝ True && 3 >= 5              def of >=
??  ⇝ True && False               def of >=
```

Calculation involving guards

- Below is a calculation of an expression involving `maxThree`; the steps in which guards are being evaluated begin with `??`:

```
    maxThree 3 2 5
??  3 >= 2 && 3 >= 5              first guard
??  ⇝ True && 3 >= 5              def of >=
??  ⇝ True && False               def of >=
??  ⇝ False                       def of &&
```

Calculation involving guards

- Below is a calculation of an expression involving `maxThree`; the steps in which guards are being evaluated begin with `??`:

```
maxThree 3 2 5
??  3 >= 2 && 3 >= 5           first guard
??  ⇝ True && 3 >= 5           def of >=
??  ⇝ True && False            def of >=
??  ⇝ False                    def of &&
??  2 >= 5                     second guard
```

Calculation involving guards

- Below is a calculation of an expression involving `maxThree`; the steps in which guards are being evaluated begin with `??`:

```
maxThree 3 2 5
??   3 >= 2 && 3 >= 5            first guard
??   ⇝ True && 3 >= 5            def of >=
??   ⇝ True && False             def of >=
??   ⇝ False                     def of &&
??   2 >= 5                      second guard
??   ⇝ False                     def of >=
```

Calculation involving guards

- Below is a calculation of an expression involving `maxThree`; the steps in which guards are being evaluated begin with `??`:

	<u>`maxThree 3 2 5`</u>	
`??`	<u>`3 >= 2`</u> `&& 3 >= 5`	`first guard`
`??`	$\leadsto$ `True &&` <u>`3 >= 5`</u>	`def of >=`
`??`	$\leadsto$ <u>`True && False`</u>	`def of >=`
`??`	$\leadsto$ `False`	`def of &&`
`??`	<u>`2 >= 5`</u>	`second guard`
`??`	$\leadsto$ `False`	`def of >=`
`??`	<u>`otherwise`</u>	`third guard`

Calculation involving guards

- Below is a calculation of an expression involving `maxThree`; the steps in which guards are being evaluated begin with `??`:

<u>`maxThree 3 2 5`</u>		
`??`	<u>`3 >= 2`</u> `&& 3 >= 5`	`first guard`
`??`	$\rightsquigarrow$ `True &&` <u>`3 >= 5`</u>	`def of >=`
`??`	$\rightsquigarrow$ <u>`True && False`</u>	`def of >=`
`??`	$\rightsquigarrow$ `False`	`def of &&`
`??`	<u>`2 >= 5`</u>	`second guard`
`??`	$\rightsquigarrow$ `False`	`def of >=`
`??`	<u>`otherwise`</u>	`third guard`
`??`	$\rightsquigarrow$ `True`	`def of otherwise`

Calculation involving guards

- Below is a calculation of an expression involving `maxThree`; the steps in which guards are being evaluated begin with `??`:

```
maxThree 3 2 5
??   3 >= 2 && 3 >= 5              first guard
??   ⇝ True && 3 >= 5              def of >=
??   ⇝ True && False               def of >=
??   ⇝ False                       def of &&
??   2 >= 5                        second guard
??   ⇝ False                       def of >=
??   otherwise                     third guard
??   ⇝ True                        def of otherwise
⇝ 5
```

Local definitions

- Let's consider a "slightly complicated" function definition from Worksheet 1:

```
distance :: Float -> Float -> Float -> Float -> Float
distance x1 y1 x2 y2 = sqrt ((x1-x2)^2 + (y1-y2)^2)
```

- We might want to "break down" the definition's expression to make it easier to read.

Local definitions

- Let's consider a "slightly complicated" function definition from Worksheet 1:

```
distance :: Float -> Float -> Float -> Float -> Float
distance x1 y1 x2 y2 = sqrt ((x1-x2)^2 + (y1-y2)^2)
```

- We might want to "break down" the definition's expression to make it easier to read.
- We can do this using local definitions. These are introduced using the `where` keyword after the expression.
- For example:

```
distance x1 y1 x2 y2 = sqrt (dxSq + dySq)
                       where
                       dxSq = (x1 - x2)^2
                       dySq = (y1 - y2)^2
```

Local definitions

- We see that:
  - the main expression uses the local definitions; and
  - the local definitions use the function's parameters.
- Local definitions can only be used within the functions that they are defined; they are "hidden" from the rest of the program.

Local definitions

- We see that:
  - the main expression uses the local definitions; and
  - the local definitions use the function's parameters.
- Local definitions can only be used within the functions that they are defined; they are "hidden" from the rest of the program.
- We could go a step further with this example:

```
distance x1 y1 x2 y2 = sqrt (dxSq + dySq)
                       where
                       dx = x1 - x2
                       dy = y1 - y2
                       dxSq = dx^2
                       dySq = dy^2
```

- Note that the local definitions can appear in any order.

Local definitions

- As another example, consider the function definition:

```
sumPosCubes :: Float -> Float -> Float -> Float
sumPosCubes x y z = abs (x^3) + abs (y^3) + abs (z^3)
```

  which takes the cubes of its parameters, uses `abs` to make them non-negative, and then sums them.

Local definitions

- As another example, consider the function definition:

```
sumPosCubes :: Float -> Float -> Float -> Float
sumPosCubes x y z = abs (x^3) + abs (y^3) + abs (z^3)
```

  which takes the cubes of its parameters, uses `abs` to make them non-negative, and then sums them.

- This is clearly repetitive code, and is maybe better written as:

```
sumPosCubes x y z = posCube x + posCube y + posCube z
                    where
                    posCube a = abs (a ^ 3)
```

- We see that the local definition `posCube` has its own parameter.

Local definitions

- As another example, consider the function definition:

```
sumPosCubes :: Float -> Float -> Float -> Float
sumPosCubes x y z = abs (x^3) + abs (y^3) + abs (z^3)
```

  which takes the cubes of its parameters, uses `abs` to make them non-negative, and then sums them.
- This is clearly repetitive code, and is maybe better written as:

```
sumPosCubes x y z = posCube x + posCube y + posCube z
                    where
                    posCube a = abs (a ^ 3)
```

- We see that the local definition `posCube` has its own parameter.
- We could have defined `posCube` as a normal (non-local) definition, but it is unlikely to be useful outside of `sumPosCubes`.

Local definitions

- As a final (contrived) example, consider a function that counts how many of its (two) parameters that are multiples of 10.
- This can be written using guards:

```
numMultsOf10 :: Int -> Int -> Int
numMultsOf10 a b
    | multOf10 a && multOf10 b  = 2
    | multOf10 a || multOf10 b  = 1
    | otherwise                 = 0
    where
    multOf10 x = mod x 10 == 0
```

- We see that, even with multiple guards, we use a single `where` at the bottom to introduce local definition(s).

Local definitions

- As a final (contrived) example, consider a function that counts how many of its (two) parameters that are multiples of 10.
- This can be written using guards:

```
numMultsOf10 :: Int -> Int -> Int
numMultsOf10 a b
    | multOf10 a && multOf10 b  = 2
    | multOf10 a || multOf10 b  = 1
    | otherwise                 = 0
    where
    multOf10 x = mod x 10 == 0
```

- We see that, even with multiple guards, we use a single `where` at the bottom to introduce local definition(s).
- Q: Is there a better way of doing this (without guards)?