Fachhochschule Nordwestschweiz

Functional Programming

http://www.fotolia.com/id/15792

Introduction

Daniel Kröni
University of Applied Sciences Northwestern Switzerland

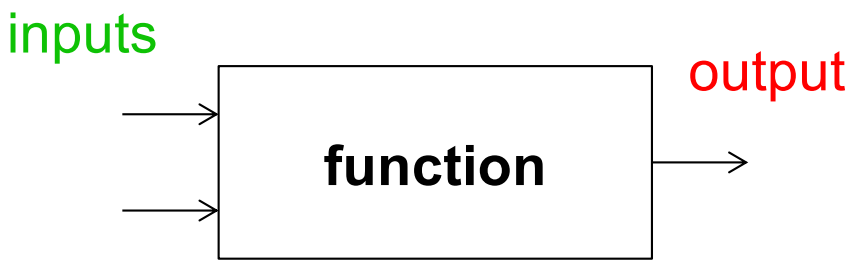
Learning Targets

You

- understand the concept of functions
- know what it means to apply a function to an argument
- have a working Haskell installation on your computer

What is a Function?

- **A Function**
  - can be pictured as a box with some inputs and an output

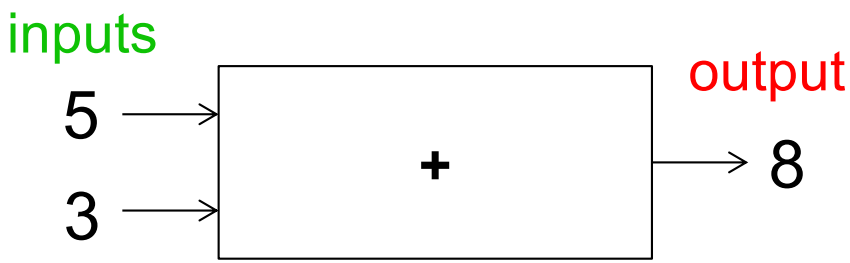


  - gives an output value which depends upon the input value(s)

- **We will often use the term result for the output and the term arguments and parameters for the input**

Function Application

- **Giving inputs to a function is called function application**



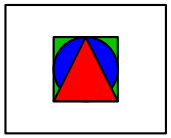
- **The function takes the inputs and computes the output**
- **Rules:**
  - **No side effects are allowed!**
  - **The output depends on the inputs only!**
- **Haskell functions are pure!**
  - For every specific input, a function always computes exactly the same output!

Models of Computation

- **Imperative: Step by step instructions**
  - Changing memory cells

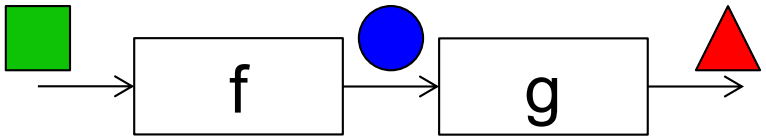
```
⇨ f();
⇨ g();
```

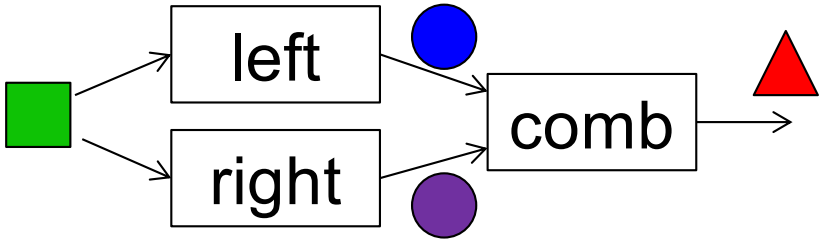
var



Spooky action at a distance

- **Functional: Applying functions to arguments**
  - Transforming data through pipelines of functions

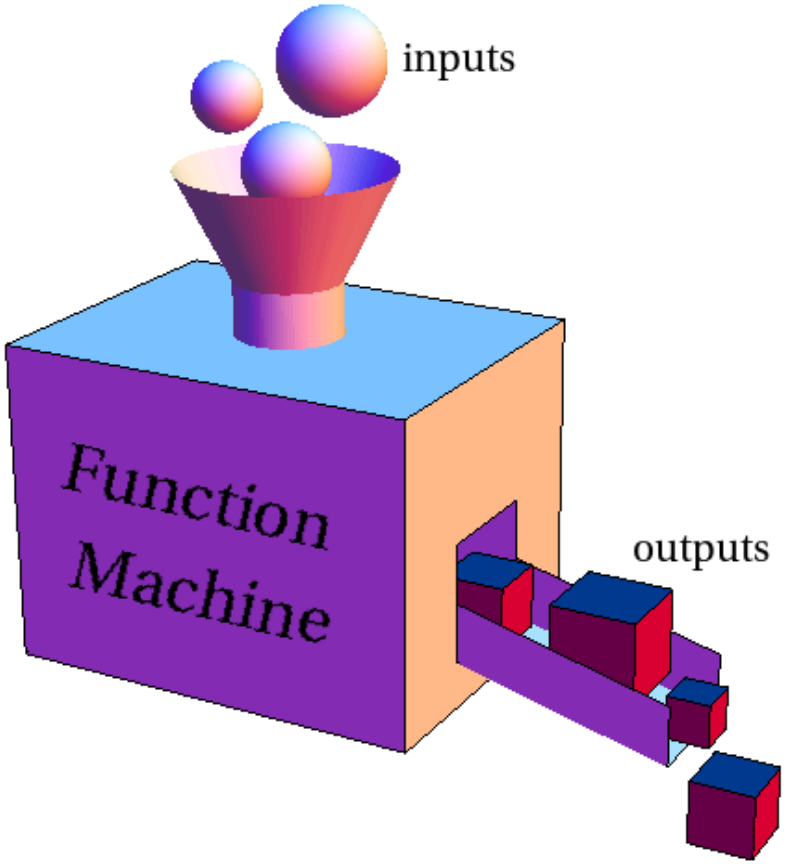




result = g (f ■)

result = comb (left ■) (right ■)

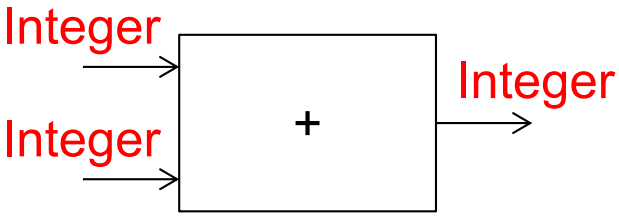
The Function Machine

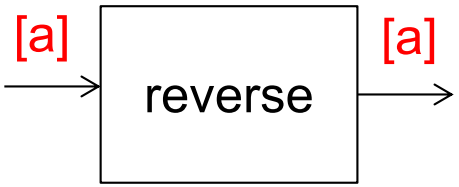


http://mathinsight.org/image/function_machine

Types

- **The input data which a function can accept as well as the output data has to be of a specific type.**
- **Examples:**





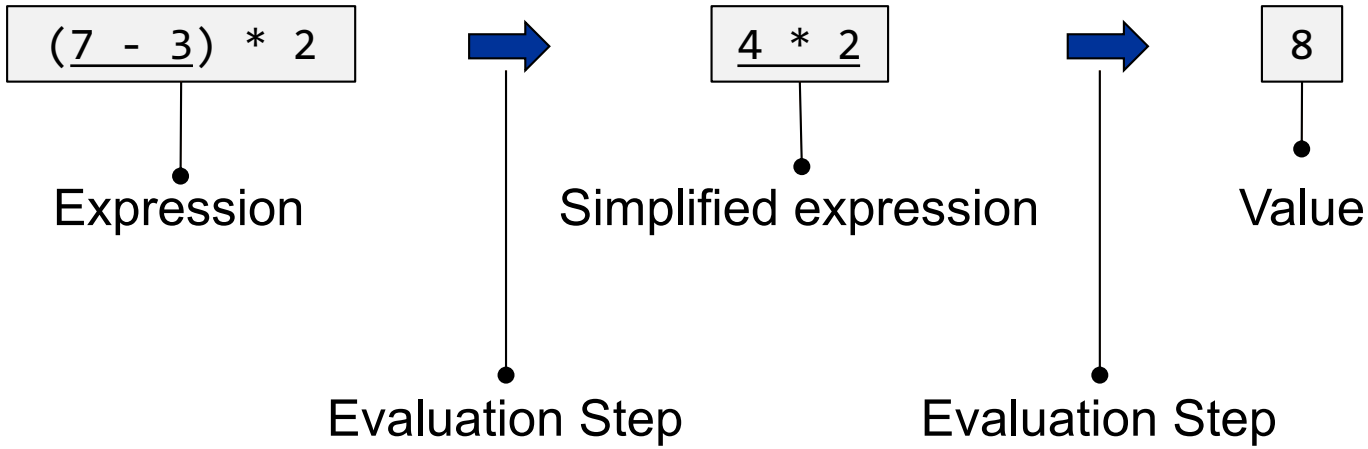
- **Type errors**
  - Applying reverse to a Bool (rather than to a list with elements of any type)

```
dk — ghci — ghci — ghc -B/Users/dk/.ghcup/ghc/8.8.4/lib/ghc-8.8.4 --int...
→  ~ ghci
GHCi, version 8.8.4: https://www.haskell.org/ghc/  :? for help
Loaded GHCi configuration from /Users/dk/.ghci
Prelude> reverse True

<interactive>:1:9: error:
    • Couldn't match expected type ‘[a]’ with actual type ‘Bool’
    • In the first argument of ‘reverse’, namely ‘True’
      In the expression: reverse True
      In an equation for ‘it’: it = reverse True
    • Relevant bindings include it :: [a] (bound at <interactive>:1:1)
```

Expressions and Evaluation in Haskell

- **Evaluation is the process of calculating the resulting value of an expression.**



Worksheet: Haskell Setup

Key learnings

- **You know how to start GHCi and evaluate simple expressions**

dk — ghci — ghci — ghc -B/Users/dk/.ghcup/ghc/8.8.4/lib/ghc-8.8.4 --interact...

```
➜  ~ ghci
GHCi, version 8.8.4: https://www.haskell.org/ghc/  :? for help
Loaded GHCi configuration from /Users/dk/.ghci
Prelude> reverse "puteS lleksaH :teehskroW"
"Worksheet: Haskell Setup"
Prelude> 2 ^ 10
1024
Prelude> print "Hallo fprog!"
"Hallo fprog!"
Prelude>
```

Definitions

- **A functional program in Haskell consists of definitions**
- **A definition associates a name with a value of a particular type**

```
name :: type
name = expression
```

- **Example:**

```
size :: Integer
size = (7 - 3) * 2
```

Associates the name size with the value of the expression, 8, whose type is Integer.

In Java:

```
int size = (7 - 3) * 2;
```

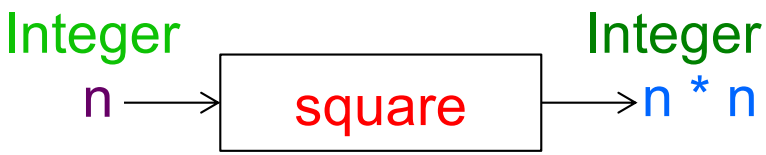
Function Definition Example I

- **Defining the square function in Haskell**

```haskell
square :: Integer -> Integer
square n =  n * n
```

```java
int square(int n) {
  return n * n;
}
```

- **Diagrammatically**



- **The first line declares the type `Integer -> Integer`**
  - The arrow '->' signifies that it is a function type
    - Taking an input/argument of type Integer
    - Returns a result/value of type Integer
- **Read as: "square is a function taking an Integer to an Integer."**

Haskell vs. Java Syntax Comparison

```haskell
-- Definitions
size :: Integer
size =  12

square :: Integer -> Integer
square n =  n * n


mul:: Integer -> Integer -> Integer
mul x y =  x * y

-- Application
square 2
mul 1 size
mul (square 2) 3
```

```java
// Definitions
int size = 12;


int square(int n) {
  return n * n;
}

int mul(int x, int y) {
  return x * y;
}
// Application
square(2);
mul(1, size);
mul(square(2), 3);
```

Function Definition in General

Types of the parameters

Type of result

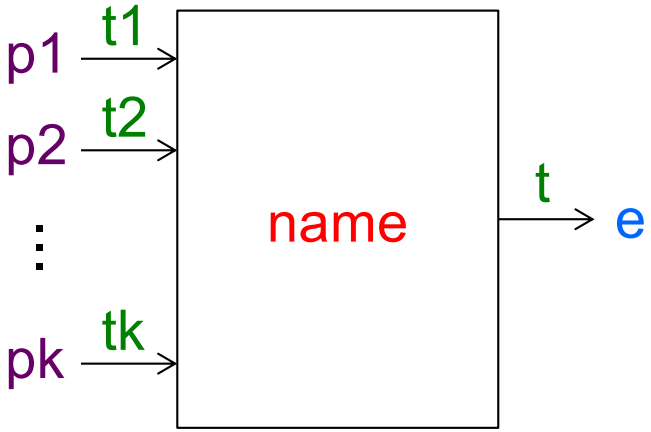
Name of the function being defined

```
name :: t1  ->  t2  ->  ...  ->  tk  ->  t
name    p1      p2      ...     pk  =   e
```

parameters

Result, defined in terms of the parameters

- **Diagrammatically**



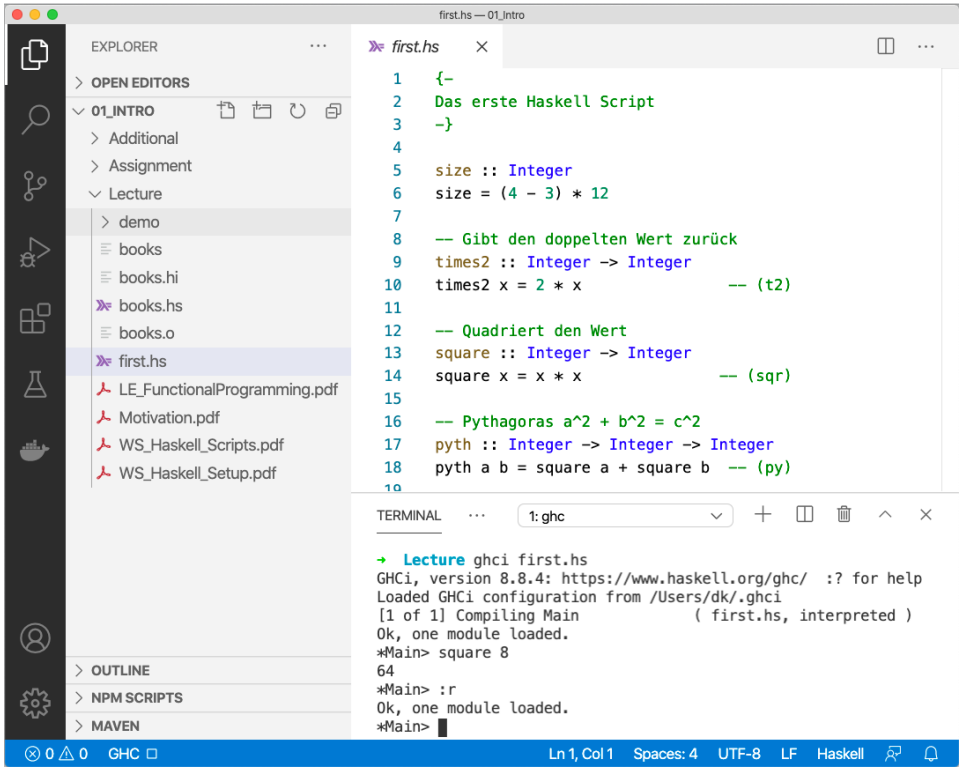

```
t name(t1 p1, t2 p2, …, tk pk) {
  return e;
}
```

Worksheet: Haskell Script

Key learnings

- **A Haskell program is stored in a file whose name ends with .hs**

- **The development cycle is**
  1. Edit and save source code
  2. Load into GHCi
     :l filename
     :r
  3. Experiment with expressions
  4. Goto 1.



Function Application / Evaluation

- **Function application is evaluated by replacing every occurrence of a parameter with the given argument**
- **Example:**
  - To evaluate:

```
23 - (times2 (3+1))
```

    - We need to use the definition of the function:

```
times2 :: Integer -> Integer
times2 n = 2*n
```

    - We replace the parameter `n` with the argument `(3+1)` giving

```
times2 (3+1) = 2*(3+1)
```

  - By replacing equals by equals we arrive at

```
23 - (2*(3+1))
```

Function Application / Evaluation

- **Example 2:**

```
times2 :: Integer -> Integer
times2 n = 2*n                                  -- (t2)
```

Comment with label

Call by name

```
23 – (times2 (3+1))
~> 23 – (2*(3+1))                               using (t2)
~> 23 – (2*4)                                   arithmetic
~> 23 – 8                                       arithmetic
~> 15                                           arithmetic
```

Explanations (may refer to label)

Call by value

```
23 – (times2 (3+1))
~> 23 – (times2 4)                              arithmetic
~> 23 – (2*4)                                   using (t2)
~> 23 – 8                                       arithmetic
~> 15                                           arithmetic
```

Function Application / Evaluation

- **Example 3:**

```
times2 :: Integer -> Integer
times2 n = 2*n                          -- (t2)

negSum :: Integer -> Integer -> Integer
negSum a b = - (a + b)                  -- (ns)
```

negSum (times2 3) (times2 (-4))	
~> negSum (2 * 3) (times2 (-4))	using (t2)
~> negSum 6 (times2 (-4))	arithmetic
~> negSum 6 (2 * (-4))	using (t2)
~> negSum 6 (-8)	arithmetic
~> - (6 + (-8))	using (ns)
~> - (-2)	arithmetic
~> 2	arithmetic

Why Pure Functions are Great

- **Given an unknown function named f**

```
f :: Integer -> Integer
```

- **What is the result / value of the following expression?**

```
(f 42) – (f 42)
```

Always 0! Because pure functions always return the same result when given the same arguments!

Why Objects are Dangerous

- **In comparison take the following unknown Java method**

```
class X { ...
  public int m(int i) { ... }
}
```

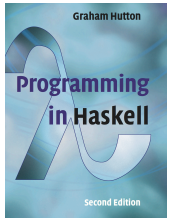
- **What can be said about the result of the following expression?**

```
X x = ...;
x.m(42) - x.m(42)
```

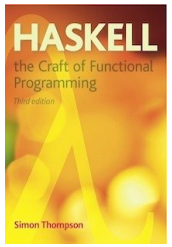
Nothing! Because methods can behave differently on every invocation!

```
class X {
  int cnt = 3;
  public int m(int i) {
    if(--cnt == 0) {
      killBambi(); deleteHD();
      return i * cnt;
    }
    return i * 3;
  }
}
```

Further Reading



Chapter 1 and Chapter 2



Chapter 1 and Chapter 2

Introduction

http://learnyouahaskell.com/introduction