

Outline of Lecture 2

- Lambda calculus and lambda expressions
- Evaluation in Haskell and the notion of closure
- Proof and testing in Haskell
- Booleans and Integers in Haskell
- Guards and conditional expressions
- Primitive recursion (if time allows it)

The lambda calculus

- The lambda calculus (or λ -calculus) is a mathematical theory for expressing computations based on function abstraction and application using variable binding and substitution
- Introduced by mathematician Alonzo Church in 1930s as part of his research of the foundations of mathematics
- The semantical basis of the Haskell language
- All the Haskell features are translatable into the corresponding lambda expressions. That makes Haskell *pure* functional language

Lambda calculus (cont.)

- Lambda calculus has four kinds of basic components or expressions: constants, variables, abstractions, and combinations of those
- Variables are used as names for potential inputs for functions
- An abstraction is a function, consisting of its *head* and the *body*, separated by a dot (.)
- Example: $\lambda x. x+1$
- The head is λ (lambda) followed by a variable (parameter) name
- The body is another expression, describing the function output

Lambda calculus (cont.)

- The variable name in the head is the parameter and binds all the instances of that variable in the body
- That means that, when we apply this function to any argument, each variable x in that function body will have the value of that argument
- Lambda abstraction by itself has no name \Rightarrow it is an *anonymous function*
- An abstraction (function) is a generalisation of some typical calculation for any parameter value, abstracting the calculation via introduction of names
- When we apply the abstraction to arguments, we replace the names with values, thus making it concrete

Alpha equivalence

- When you write lambda expressions like $(\lambda x. x)$, the only meaning of the variable x is to bind parameter values in the body
- The actual parameter name does not matter
- That means that lambda expressions like $(\lambda x. x)$, $(\lambda d. d)$, $(\lambda z. z)$ are equivalent. They all are the same function
- Mathematically, such equivalence between lambda terms is called *alpha equivalence*

Beta reduction

- When we apply a function to an argument (a combination of an abstraction followed by an expression), we substitute the input expression for all instances of the bound variable within the abstraction body
- We also eliminate the head of abstraction, since its only purpose was to bind a variable

$$\begin{aligned} & (\lambda x. x + 1) 2 \\ = & \text{\{beta reduction, } [x := \dots] \text{ denotes substitution\}} \\ & (x + 1) [x := 2] \\ = & \text{\{substituting } x \text{ with } 2 \text{ in the body\}} \\ & 2 + 1 \\ = & \text{\{simplifying\}} \\ & 3 \end{aligned}$$

Beta reduction (cont.)

Function application is left-associative. Example:

$$\begin{aligned} & (\lambda x. x) (\lambda y. y) (z * 5) \\ = & \{ \text{beta reduction, } [x := \dots] \text{ denotes substitution} \} \\ & (x [x := (\lambda y. y)]) (z * 5) \\ = & \{ \text{simplifying} \} \\ & (\lambda y. y) (z * 5) \\ = & \{ \text{beta reduction, } [y := \dots] \text{ denotes substitution} \} \\ & y [y := z * 5] \\ = & \{ \text{simplifying} \} \\ & z * 5 \end{aligned}$$

The process of beta reduction stops when there are either no more abstractions left to apply or no more arguments to apply to

Free variables

- The function variables defined in the head are called *bound* variables for that function
- If there are variables not named in the abstraction head, they are called *free*
- Free variables are NOT affected by beta reduction process:

$$(\lambda x. x y) f = f y$$

- Alpha equivalence or other transformation cannot be applied if it makes a free variable into bound:

$$(\lambda x. x y) \neq (\lambda y. y y) \quad \text{forbidden!}$$

Multiple arguments

- Each lambda expression can only bind one parameter and thus accept only one argument
- Functions that require multiple arguments have multiple nested heads
- The standard notation $\lambda x y. \dots$ is just a shorthand for

$$\lambda x. (\lambda y. \dots)$$

- When you apply it once and eliminate the first (leftmost) head, the next one applied and so on
- This principle is called currying (named after mathematician Haskell Curry)

Multiple arguments (cont.)

Example:

$$\begin{aligned} & (\lambda x y. x y) (\lambda z. a) 1 \\ = & \text{\{unfolding the shorthand notation\}} \\ & (\lambda x. (\lambda y. x y)) (\lambda z. a) 1 \\ = & \text{\{beta reduction, [x := ...] denotes substitution\}} \\ & ((\lambda y. x y) [x := (\lambda z. a)]) 1 \\ = & \text{\{simplifying\}} \\ & (\lambda z. a) 1 \\ = & \text{\{beta reduction, [z := ...] denotes substitution\}} \\ & a [z := 1] \\ = & \text{\{simplifying, no effect on a free variable\}} \\ & a \end{aligned}$$

Calculation and evaluation in Haskell

- Expression calculation and evaluation in Haskell is based on the described beta-reduction process on lambda expressions
- Each function in Haskell, e.g.,

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

is actually declared as (is a syntactic sugar for) a named lambda expression

$$\text{double} = \lambda n. 2 * n$$

- If needed, functions can be directly constructed as anonymous lambda expressions (with ASCII `\` used instead of λ)

Calculation and evaluation in Haskell (cont.)

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

Evaluation and reduction in a step-by-step manner, following beta reduction rules described earlier:

$$\begin{aligned} & 23 - (\text{double } (3 + 1)) \\ \rightsquigarrow & 23 - (2 * (3 + 1)) \\ \rightsquigarrow & 23 - (2 * 4) \\ \rightsquigarrow & 23 - 8 \\ \rightsquigarrow & 15 \end{aligned}$$

Substitution, rewriting, and simplification (in a general case, after checking some extra conditions/pattern matching)

The notion of closure

Function definitions in Haskell must be fully defined as *closed expressions* (i.e., without undefined free variables or identifiers).

Function formal parameters – not considered as free.

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

The definition `double` is closed. The only variable `n` is a function parameter.

```
square_1 :: Integer -> Integer
square_1 n = square (n+1)
```

The definition `square_1` is closed, provided `square` is defined.

The notion of closure (continued)

- The "free" variables/identifiers should be resolved (during compilation) from the local context (scope)
- The local context: previously entered definitions (in ghci) or the whole file (in a module description), also the value and function definitions from the imported modules
- The process of resolving "free" variables (and the resulting full definition) is called *closure*
- Once function closure successfully calculated, the function definition stays fixed (always the same results for the same inputs)

Proof and testing

- The value and function definitions are very close to mathematical descriptions (equations) \Rightarrow the correctness can be easily verified
- The definitions and their properties can manually checked (by step-by-step rewriting and evaluation)
- Or, may be be verified by automated tools (model checkers, theorem provers)
- Or, may be tested by automatically generating test cases for various parameter values

Proof and testing (cont.)

- Haskell provides a simple test generator, called `quickCheck`
- Can be simply imported (in the Haskell version 8) with the command

```
import Test.QuickCheck
```

- Example: checking the property `square_1 x == x*x + 2*x + 1`

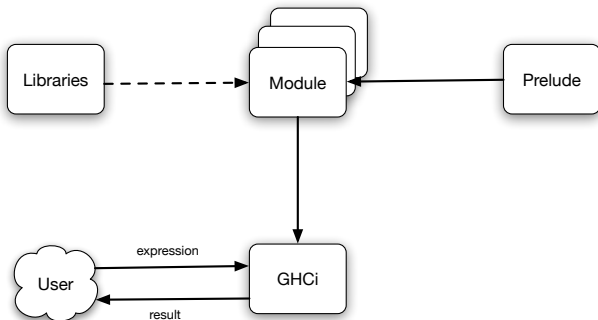
```
quickCheck prop_square_1
```

where

```
prop_square_1 x =  
    square_1 x == x*x + 2*x + 1
```

- `quickCheck` takes a function of the type `a -> Bool` (the condition/property to be checked) as a parameter

The Haskell module system



Libraries – precompiled (collections of) imported modules (like Test)

Prelude – the default (always loaded) module containing a bunch of standard, commonly used definitions

Syntax conventions

- In ghci, the interpreter typically evaluates one line a time. A several declarations can be combined using ";", e.g.,

```
> z :: Float; z = 3.5
```

- There is also the multi-line mode `:{ ... :}` in ghci
- In slightly older ghci versions (before 8.0), `let` keyword was required for such temporary local declarations

```
> let z = 3.5
```

Syntax conventions (cont.)

- The above declaration (using ";") is accepted in a module file as well
- Long declarations in a module file can be split into several lines, for instance

```
ff x y z =  
    (x+y >= z) &&  
    (x+y <= z+10) &&  
    ...  
gg x = ...
```

- A general rule (in a module file): as long as the next line is more indented (i.e., has more preceding spaces) than the first declaration line, the compiler considers it as a part of the same declaration

Syntax conventions (cont.)

- Identifiers – names starting with a letter
- Named values, variables and type variables – with a small letter
- Types, type constructors, module names, type classes – (typically) with a capital letter
- Reserved identifiers: `case`, `class`, `data`, `default`, `deriving`, `do`, `else`, `if`, `import`, `in`, `infix`, `infixl`, `infixr`, `instance`, `let`, `module`, `newtype`, `of`, `then`, `type`, `where`
- Operator names – starting with a special symbol, e.g., `!`, `%`, `#`, `*`, `+`, `.`, `...`. Cannot start with :
- Operators are by default infix. Can be written before arguments (e.g., for checking its type, or defining a new operator) by enclosing them into parentheses, like `(+)`, `(&&)`

The Booleans

- The Boolean type in Haskell is called `Bool`
- Literals/constants: `True` and `False`. The results of tests, e.g., checking for equality
- Standard operations: `&&` (logical and), `||` (logical or), `not`
- Defining Boolean functions:

```
exOr :: Bool -> Bool -> Bool
exOr x y = (x || y) && not( x && y)
```

The Booleans (cont.)

- We can also use a combination of literals and variables on the left hand side of the equations defining `exOr`

```
exOr_1 :: Bool -> Bool -> Bool
exOr_1 True x = not x
exOr_1 False x = x
```

- A simple example of [pattern matching](#); Also – more than one equation (definition case) in a function declaration
- Checking with `quickCheck`:

```
quickCheck prop_exOrs
```

where

```
prop_exOrs x y =
    exOr x y == exOr_1 x y
```

The Integers: Integer and Int

- The Haskell type `Integer` contains positive and negative whole numbers
- Literals (can be used for pattern matching): `0`, `45`, `-232`, `34782099`, ...

```
ff :: Integer -> Integer
ff 0 = 23
ff 23 = 24
ff x = 0
```

Evaluated in the order of declarations \Rightarrow the last equation applied only if $x \neq 0$ and $x \neq 23$

- Standard operations: `+`, `-`, `*`, `^`, `div`, `mod`, ...
- Relational operators: `>`, `>=`, `==`, `/=`, `,`, `<=`, `<`
(defined for any ordered type)

The Integers: Integer and Int (cont.)

- The Haskell type `Int` contains the bounded range of integers between `minBound` ... `maxBound`. The actual values of `minBound` and `maxBound` depend on the Haskell implementation and the computer architecture ($2^{63} - 1$ in the example below)

```
> maxBound :: Integer  
9223372036854775807
```

- In contrast, `Integer` is theoretically infinite (practically, bounded only by the available computer memory)
- All standard operations and relational operators on `Integer` and `Int` are overloaded

Overloading

- Integers, Ints and Booleans can be all compared for equality using the same operator `==`. This means that in different cases `==` has the types `Integer -> Integer -> Bool`, `Int -> Int -> Bool`, `Bool -> Bool -> Bool` respectively
- An example of function (operator) *overloading*. `==` can be used for any type for which we can define equality comparison. Its generic (polymorphic) type indicates this

```
> :type (==)  
Eq t => t -> t -> Bool
```

- More about polymorphic Haskell types and type classes – later
- Most (if not all) of the above operators are overloaded

Definitions with guards

- A *guard* is a boolean expression, which is used to express different cases in the definition of a function
- An example (the Prelude definition of `max`):

```
max :: Integer -> Integer -> Integer
max x y
  | x >= y = x
  | otherwise = y
```

- If the first guard is evaluated to `True`, the function returns `x` as its result. Otherwise, `y` is returned.

Definitions with guards (cont.)

- In general case:

```
name x1 x2 ... xk
  | g1 = res1
  | g2 = res2
  ...
  | otherwise = res
```

- The guards are evaluated in the order they presented. The next guard is checked only after the previous one has failed
- The otherwise branch is not compulsory (if you did not cover all the cases, you can get "the cases are non exhaustive" exception in the case a function cannot be evaluated for the given arguments)

Definitions with guards (cont.)

- Another example – a function returning the maximum of three integers:

```
max3 :: Integer -> Integer -> Integer -> Integer
max3 x y z
  | x >= y && x >= z = x
  | y >= z = y
  | otherwise = z
```

Conditional expressions

- Guards allows us to distinguish different cases on the function definition level
- We can also write general conditional expression using the `if ... then ... else ...` construct of Haskell. Depending on the evaluation of the `if` condition, the result value of the respective branch is returned
- Yet another version of `max`:

```
max' :: Integer -> Integer -> Integer
max' x y =
    if x>=y then x else y
```

- Can be convenient in some cases, however guards in general are more expressive and readable

Primitive recursion

- Recursion: a definition of a function or another object which refers to itself
- Very important (and very heavily relied on in Haskell) programming mechanism
- We start by considering a simple yet quite powerful form of recursion – *primitive recursion*

Primitive recursion (cont.)

- Based on primitive induction principle on natural numbers
- General template:

```
fun n
  | n==0 = ...
  | n > 0 = ... fun(n-1)
  ...
```

- The first branch defines the **base case**, stopping the recursion and returning the end result
- The second branch defines the **recursive case** (inductive case), describing the result in terms of a more simple case $n-1$
- Such template guarantees termination of recursion (for positive integer numbers)

Primitive recursion (cont.)

- Example (factorial):

```
fac :: Integer -> Integer
fac n
  | n==0 = 1
  | n > 0 = fac(n-1)*n
```

- What happens for `fac (-1)`? The "non-exhaustive pattern" error.

```
fac n
  | n==0 = 1
  | n > 0 = fac(n-1)*n
  | otherwise = 0
```

- Alternatively (by defining own exceptions),
| otherwise = error "Only defined for natural numbers"

- The slides can be downloaded from the Moodle course page
- Exercises (Set 1) will be added to Moodle later today
- The solutions should be uploaded to Moodle or sent by e-mail to me (linas.laibinis@mif.vu.lt). The first way is preferable
- The deadline for uploading: September 29th (Friday)