5. Functional Programming

Slides:

- *17 Functional Programming*
- *18 Lambda and Laziness*

The key idea: do everything by composing functions

- no mutable state
- no side effects
- (no fun, IMHO)

Main Concepts

- 1st class and high-order functions: functions can be denoted, passed as argument to other functions and returned as result
- Recursion instead of iteration
- Powerful list facilities
- Polymorphism: universal parametric implicit
- Data structures cannot be modified, them must be recreated

ML family

Meta-language. Includes: Standard ML, Caml, OCaml, F#.

Features:

- Type safe, with type inference and formal semantics
- Both compiled and interactive use
- Expression-oriented
- Higher order functions
- Anonymous functions: lambda
- Abstract data types
- Garbage collector
- Module system
- Exceptions
- Impure: allows side-effects

Haskell

Features:

- Type checking and type inference (cast not allowed)
- Polymorphism: implicit parametric and ad hoc (overloading)
- Lazy evaluation
- Tail recursion and continuations
- Purely functional
- Variables are bound to expression, without evaluating them (lazy evaluation, functions don't evaluate its arguments until them are needed)

Core Haskell

- Basic types
  - Unit
  - Boolean
  - Integer
  - Real
  - Character
  - String
  - Tuple
  - List
  - Record
- Patterns
- Declarations
- Functions
- Polymorphism
- Type declarations
- Type Classes
- Monads
- Exceptions

Laziness

Functions and data constructors don’t evaluate their arguments until they need them.

In several languages there are forms of lazy evaluation (if-then-else, shortcutting `&&` and `||`)

Lambda calculus

`λx.t`

Binding

An occurrence of x is free in a term t if it is not in the body of an abstraction λx. t, otherwise it is bound. λx is a binder.
Example: in `λx.λy.λz.(x+z)` x and z are bound, y is free.

β-reduction

`(λx.t) t' = t[t'/x]`

Encode functions in λ

`f(x,y) = <exp>` ≡ `f = λx.λy.<exp>`

Parameter passing mechanism

- Applicative order evaluation: parameter are evaluated before applying the function (eager evaluation, parameter passed by value).
- Normal order evaluation: functions evaluated first, arguments if and when needed (parameter passed by name)

Parameter passing modes

- in | in/out | out

Parameter passing mechanisms

- value (in)
  - need (in): copy as an expression, evaluated the first time is needed.
  - name (in + out): same of call by need, but the parameter is evaluated every time (substitution in the body)
- reference (in + out)
  - sharing (in/out): the value is copied, but the value is a reference. Is the same of the call by value, but in the reference model.
- result (out)
- value/result (in + out)

Value vs reference

- Value copy the value into the variable (copy of data)
- Reference copy the reference to the value into the variable (shared data)

Reference vs pointer

- Reference to x: address of the cell in which is stored x
- Pointer to x: location containing the address (reference) of x