301AA - Advanced Programming [AP-2017]

Lecturer: Andrea Corradini andrea@di.unipi.it
Tutor: Lillo Galletta galletta@di.unipi.it

Department of Computer Science, Pisa
Academic Year 2017/18

AP-2017-15: Functional Programming
Functional Programming: Historical Origins

• The imperative and functional models grew out of work undertaken Alan Turing, Alonzo Church, Stephen Kleene, Emil Post, etc. ~1930s
  – different formalizations of the notion of an algorithm, or effective procedure, based on automata, symbolic manipulation, recursive function definitions, and combinatorics

• These results led Church to conjecture that any intuitively appealing model of computing would be equally powerful as well
  – this conjecture is known as Church’s thesis
Historical Origins

• Church’s model of computing is called the **lambda calculus**
  
  – based on the notion of parameterized expressions (parameter introduced letter \( \lambda \), hence the notation’s name)
  
  – allows one to define mathematical functions in a constructive/effective way
  
  – lambda calculus was the inspiration for functional programming
  
  – computation proceeds by substituting parameters into expressions, just as one computes in a high level functional program by passing arguments to functions
Basics of λ-calculus

λ-terms: \[ t ::= x \mid \lambda x.t \mid t \ t \mid (t) \]

• \( x \) \hspace{1cm} \textit{variable, name, symbol,…}
• \( \lambda x.t \) \hspace{1cm} \textit{abstraction}, defines an anonymous function
• \( t \ t' \) \hspace{1cm} \textit{application} of function \( t \) to argument \( t' \)
• \([\lambda\text{-abstraction}]\)
  \hspace{1cm} \textit{function definition} \hspace{1cm} x : A, \ t(x) : B \hspace{1cm} \lambda x.t : A \rightarrow B
• \([\beta\text{-conversion}]\)
  \hspace{1cm} \textit{function application} \hspace{1cm} (\lambda x.t) \ t' = t \ [t'/x]

A simple tutorial on lambda calculus:
http://www.inf.fu-berlin.de/lehre/WS03/alpi/lambda.pdf
Functional Programming Concepts

- Functional languages such as **Lisp, Scheme, FP, ML, Miranda, and Haskell** are an attempt to realize Church’s lambda calculus in practical form as a programming language.

- The key idea: do everything by composing functions
  - no mutable state
  - no side effects
Functional Programming Concepts

• Necessary features, many of which are missing in some imperative languages:
  – 1st class and high-order functions
  – recursion
    • Takes the place of iteration
  – powerful list facilities
    • Recursive function exploit recursive definition of lists
  – Polymorphism (typically universal parametric implicit)
    • Relevance of Container/Collections
  – fully general aggregates
    • Data structures cannot be modified, have to be re-created
  – structured function returns
  – garbage collection
    • Unlimited extent for locally allocated data structures
Other Related Concepts

• **Lisp** also has some features that are not necessary present in other functional languages:
  – programs are data
  – self-definition
  – read-evaluate-print interactive loop

• Variants of LISP
  – (Original) Lisp: purely functional, dynamically scoped
  – Common Lisp: current standard, statically scoped, very complex
  – Scheme: statically scoped, very elegant, used for teaching
Other functional languages: the ML family

• Robin Milner (Turing award in 1991, CCS, Pi-calculus, ...)
• Statically typed, general-purpose programming language
  – “Meta-Language” of the LCF theorem proving system
• Type safe, with type inference and formal semantics
• Compiled language, but intended for interactive use
• Combination of Lisp and Algol-like features
  – Expression-oriented
  – Higher-order functions
  – Garbage collection
  – Abstract data types
  – Module system
  – Exceptions
Other functional languages: Haskell

- Designed by committee in 80’s and 90’s to unify research efforts in lazy languages
  - Evolution of Miranda
  - Haskell 1.0 in 1990, Haskell ‘98, Haskell 2010

- Several features in common with ML, but **some differ**:

- Types and type checking
  - Type inference
  - Implicit parametric polymorphism
  - *Ad hoc polymorphism (overloading)*

- Control
  - *Lazy vs. eager evaluation*
  - Tail recursion and continuations

- Purely functional
  - *Precise management of effects*