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 \textit{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

• We shall see later the basic of lambda-calculus
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**
Downloading Haskell

https://www.haskell.org/platform/

A multi-OS distribution
designed to get you up and running quickly, making it easy to focus on using Haskell. You get:

- the Glasgow Haskell Compiler
- the Cabal build system
- the Stack tool for developing projects
- support for profiling and code coverage analysis
- 35 core & widely-used packages

Prior releases of the Platform are also available.
Core Haskell

- Basic Types
  - Unit
  - Booleans
  - Integers
  - Strings
  - Reals
  - Tuples
  - Lists
  - Records

- Patterns
- Declarations
- Functions
- Polymorphism
- Type declarations
- Type Classes
- Monads
- Exceptions
Overview of Haskell

• Interactive Interpreter (ghci): read-eval-print
  – ghci infers type before compiling or executing
  – Type system does not allow casts or similar things!

• Examples

```haskell
Prelude> 5==4
False
Prelude> :set +t  -- enables printing of types
Prelude> 'x'
'x'
it :: Char
Prelude> (5+3)-2
6
it :: Num a => a  -- ?? generic constrained type
Prelude> :t map  -- type of a function
map :: (a -> b) -> [a] -> [b]
```
Overview by Type

• Booleans

True, False :: Bool

not :: Bool -> Bool

and, or :: Foldable t => t Bool -> Bool

if ... then ... else ... --types must match

• Characters & Strings

'a','b',';','\t', '2', 'X' :: Char

"Ron Weasley" :: [Char] --strings are lists of chars
Overview by Type

• Numbers

0, 1, 2, ..., Num p => p --type classes to disambiguate

1.0, 3.1415 :: Fractional a => a -> a -> a

(45 :: Integer) :: Integer -- explicit typing

+, *, -, ... :: Num a => a -> a -> a
-- infix + becomes prefix (+)
-- prefix binary op becomes infix `op`
/
:: Fractional a => a -> a -> a

div, mod :: Integral a => a -> a -> a

^ :: (Num a, Integral b) => a -> b -> a
Simple Compound Types

• Tuples

("AP", 2017) :: Num b => ([Char], b) -- pair
fst :: (a, b) -> a
snd :: (a, b) -> b

('4', True, "PLP") :: (Char, Bool, [Char])

• Lists

[] :: [a] -- NIL, polymorphic type
1 : [2, 3, 4] :: [Integer] -- infix cons notation
[1,2]++[3,4] :: [Integer] -- concatenation

data Person = Person {firstName :: String,
                      lastName :: String}
hg = Person { firstName = "Hermione",
                lastName = "Granger"}
More on list constructors

ghci> [1..20]    -- ranges
[1,2,3,4,5,6,7,8,9,10,11,12,13,14,15,16,17,18,19,20]
ghci> ['a'..'z']
"abcdefghijklmnopqrstuvwxyz"
ghci> [3,6..20]    -- ranges with step
[3,6,9,12,15,18]
ghci> [7,6..1]
[7,6,5,4,3,2,1]

ghci> [1..]        -- an infinite list: runs forever
ghci> take 10 [1..] -- prefix of an infinite lists
[1,2,3,4,5,6,7,8,9,10] -- returns!
ghci> take 10 (cycle [1,2])
[1,2,1,2,1,2,1,2,1,2]
ghci> take 10 (repeat 5)
[5,5,5,5,5,5,5,5,5,5]

How does it work??? Later...
Patterns and Declarations

- Patterns can be used in place of variables
  \(<\text{pat}> ::= \langle\text{var}\rangle \mid \langle\text{tuple}\rangle \mid \langle\text{cons}\rangle \mid \langle\text{record}\rangle \ldots\>

- Value declarations
  - General form: \(<\text{pat}> = <\text{exp}>\)
  - Examples

    ```
    myTuple = ("Foo", "Bar")
    (x,y) = myTuple   -- x = "Foo", y = "Bar"
    myList = [1, 2, 3, 4]
    z:zs = myList     -- z = 1, zs = [2,3,4]
    ```

- Local declarations

    ```
    let (x,y) = (2, "FooBar") in x * 4
    ```
Anonymous Functions (lambda abstraction)

• Anonymous functions

\( \lambda x \rightarrow x+1 \)  --like Lisp lambda, function (...) in JS

\((\lambda x \rightarrow x+1)5\)  \Rightarrow  6

\( f = \lambda x \rightarrow x+1 \)  \( f\ 7 \)  \Rightarrow  8

• Anonymous functions using patterns

Prelude> h = \((x,y)\) \rightarrow x+y

h :: Num a => (a, a) \rightarrow a

Prelude> h (3, 4) \Rightarrow  7

Prelude> h 3 4  \Rightarrow  error

Prelude> k = \((z:zs)\) \rightarrow \text{length}\ zs

k :: [a] \rightarrow \text{Int}

Prelude> k "hello"  \Rightarrow  4
Function declarations

• Function declaration form

  <name> <pat₁>  = <exp₁>
  <name> <pat₂>  = <exp₂> ...

• Examples

  \[
  f (x,y) = x+y \quad \text{–--argument must match pattern } (x,y)
  \]

  length [] = 0
  length (x:s) = 1 + length(s)
More Functions on Lists

- Apply function to every element of list

\[
\begin{align*}
\text{map } f \; [] &= [] \\
\text{map } f \; (x:xs) &= f \; x : \text{map } f \; xs
\end{align*}
\]

\[
\text{map } (\lambda x \rightarrow x+1) \; [1,2,3] = [2,3,4]
\]

- Reverse a list

\[
\begin{align*}
\text{reverse } [] &= [] \quad \text{-- quadratic} \\
\text{reverse } (x:xs) &= (\text{reverse } xs) \; ++ \; [x]
\end{align*}
\]

\[
\begin{align*}
\text{reverse } xs &= \quad \text{-- linear, tail recursive} \\
&\quad \text{let } \text{rev } ( [ ], \text{accum } ) = \text{accum} \\
&\quad \quad \text{rev } ( y:ys, \text{accum } ) = \text{rev } ( ys, y:\text{accum } ) \\
&\quad \text{in } \text{rev } ( xs, [] )
\end{align*}
\]
On laziness

- Haskell is a lazy language
- Functions and data constructors don’t evaluate their arguments until they need them
- In several languages there are forms of lazy evaluations (`if-then-else`, shortcutting `&&` and `||`)

```c
if (x != 0) return y/x; else return 0; // ok
if (x != 0 && y/x > 5) return 0; else return 1; // ok
if (x != 0 & y/x > 5) return 0; else return 1; // no
```

```c
int choose(boolean e1, boolean e2){
    if (e1 && e2) return 0; else return 1;
}
choose(x!=0, y/x>5) // ???
```

- Ok in Haskell, thanks to **Normal Order evaluation** and **Call by Need** parameter passing...
A digression on $\lambda$-calculus
\[\lambda\text{-calculus: syntax}\]

\[\lambda\text{-terms:} \quad t ::= x \mid \lambda x.t \mid t t \mid (t)\]

- \(x\) \textit{variable, name, symbol,}...
- \(\lambda x.t\) \textit{abstraction}, defines an anonymous function
- \(t t'\) \textit{application} of function \(t\) to argument \(t'\)

Terms can be represented as abstract syntax trees

\textbf{Syntactic Conventions}

- Applications associates to left
  \[t_1 t_2 t_3 \equiv (t_1 t_2) t_3\]
- The body of abstraction extends as far as possible
  - \(\lambda x. \lambda y. x y x \equiv \lambda x. (\lambda y. (x y) x)\)

A simple tutorial on lambda calculus:
http://www.inf.fu-berlin.de/lehre/WS03/alpi/lambda.pdf
Free vs. Bound Variables

• An occurrence of $x$ is **free** in a term $t$ if it is not in the body of an abstraction $\lambda x. t$
  – otherwise it is **bound**
  – $\lambda x$ is a **binder**

• Examples
  – $\lambda z. \lambda x. \lambda y. x \ (y \ z)$
  – $(\lambda x. x) \ x$

• Terms without free variables are **combinators**
  – Identity function: $id = \lambda x. x$
  – First projection: $fst = \lambda x. \lambda y. x$
Operational Semantics

\[ \text{function application} \]

\[ (\lambda x. t) t' = t [t'/x] \]

\[ (\lambda x. x) y \rightarrow y \]

\[ (\lambda x. x (\lambda x. x)) (u r) \rightarrow u r (\lambda x. x) \]

\[ (\lambda x (\lambda w. x w)) (y z) \rightarrow \lambda w. y z w \]
λ-calculus as a functional language

Despite the simplicity, we can encode in λ-calculus most concepts of functional languages:

• Functions with several arguments
• Booleans and logical connectives
• Integers and operations on them
• Pairs and tuples
• ...
Functions with several arguments

• A definition of a function with a single argument associates a name with a $\lambda$-abstraction

\[
\begin{align*}
  f \ x &= \langle \text{exp} \rangle \quad \text{-- is equivalent to} \\
  f &= \lambda x.\langle \text{exp} \rangle
\end{align*}
\]

• A function with several argument is equivalent to a sequence of $\lambda$-abstractions

\[
\begin{align*}
  f(x, y) &= \langle \text{exp} \rangle \quad \text{-- is equivalent to} \\
  f &= \lambda x.\lambda y.\langle \text{exp} \rangle
\end{align*}
\]

• “Currying” and “Uncurrying”

\[
\begin{align*}
  \text{curry} &: (\ (a,\ b) \to c) \to a \to b \to c \\
  \text{uncurry} &: (a \to b \to c) \to (a,\ b) \to c
\end{align*}
\]
Church Booleans

• $T = \lambda t.\lambda f.t$ -- first
• $F = \lambda t.\lambda f.f$ -- second
• $\text{and} = \lambda b.\lambda c.bcF$
• $\text{or} = \lambda b.\lambda c.bTc$
• $\text{not} = \lambda x.xFT$
• $\text{test} = \lambda l.\lambda m.\lambda n.lmn$

\[
\begin{align*}
\text{test } F & \quad u \quad w \\
& \rightarrow (\lambda l.\lambda m.\lambda n.lmn) \; F \; u \; w \\
& \rightarrow (\lambda m.\lambda n.Fmn) \; u \; w \\
& \rightarrow (\lambda n.Fun) \; w \\
& \rightarrow Fuw \\
& \rightarrow w
\end{align*}
\]

\[
\begin{align*}
\text{and } T & \quad F \\
& \rightarrow (\lambda b.\lambda c.bcF) \; T \; F \\
& \rightarrow (\lambda c.TcF) \; F \\
& \rightarrow TFF \\
& \rightarrow F
\end{align*}
\]

\[
\begin{align*}
\text{not } F \\
& \rightarrow (\lambda x.xFT) \; F \\
& \rightarrow FFT \\
& \rightarrow T
\end{align*}
\]
Pairs

- pair = λf.λs.λb.b f s
- fst = λp.p T
- snd = λp.p F

\[
\begin{align*}
{\text{fst}} & (\text{pair } u \ w) \\
\to & (\lambda p. p \ T) \ (\text{pair } u \ w) \\
\to & (\text{pair } u \ w) \ T \\
\to & (\lambda f. \lambda s. \lambda b. b \ f \ s) \ u \ w \ T \\
\to & (\lambda s. \lambda b. b \ u \ s) \ w \ T \\
\to & (\lambda b. b \ u \ w) \ T \\
\to & T \ u \ w \\
\to & u
\end{align*}
\]
### Numerals

- \(c_0 = \lambda s. \lambda z. z\)
- \(c_1 = \lambda s. \lambda z. s\ z\)
- \(c_2 = \lambda s. \lambda z. s\ (s\ z)\)
- \(c_3 = \lambda s. \lambda z. s\ (s\ (s\ z))\)
- \(\text{succ} = \lambda n. \lambda s. \lambda z. s\ (n\ s\ z)\)
- \(\text{plus} = \lambda m. \lambda n. \lambda s. \lambda z. m\ s\ (n\ s\ z)\)
- \(\text{times} = \lambda m. \lambda n. m\ (\text{plus}\ n)\ c_0\)
- Turing Complete

**succ c2**

\[
\begin{align*}
\text{succ c2} & \rightarrow (\lambda n. \lambda s. \lambda z. s\ (n\ s\ z))\ c2 \\
& \rightarrow (\lambda s. \lambda z. s\ (c2\ s\ z)) \\
& \rightarrow (\lambda s. \lambda z. s\ ((\lambda s. \lambda z. s\ (s\ z))\ s\ z)) \\
& \rightarrow (\lambda s. \lambda z. s\ (s\ (s\ z)) = c3
\end{align*}
\]