FP Foundations, Scheme

In Text: Chapter 15

Outline

• Mathematical foundations
• Functional programming
• $\lambda$-calculus
• LISP
• Scheme

Imperative Languages

• We have been discussing imperative languages
  – C/C++, Java, and Pascal are imperative languages
  – They follow the von Neumann architecture [1]

Functional Programming

• A different way of looking at things
  – It is based on mathematical functions
  – It is supported by functional, and applicative, programming languages
    • LISP, ML, Haskell

Mathematical Foundations

• A mathematical function is a mapping of members from one set to another set
  – The "input" set is called the domain
  – The "output" set is called the range

Mathematical Foundations

• The evaluation order of mapping expressions is controlled by recursion and conditional expressions, rather than by the sequencing and iterative repetition
• Functions do not have states
  – They have no side effects
  – They always produce the same output given the same input parameters
Simple Functions

• Usual form:
  function name + a list of parameters in parentheses + mapping expression

  E.g., cube(x) = x * x * x, where
  - both the domain and range sets are real numbers, and
  - x can represent any member of the domain set, but it is fixed to represent one specific element during the expression evaluation

Function Application

• It is specified by paring the function name with a particular element of the domain set
• The range element is obtained by evaluating the function-mapping expression with the domain element substituted for the particular element
  - Cube(2.0) = 2.0 * 2.0 * 2.0 = 8.0

Functional Forms

• A higher-order function, or functional form, is one that either takes functions as parameters, or yields a function as its result, or both
• Two common functional forms
  – Function composition
  – Apply-to-all

Function Composition

• Function composition has two functional parameters and yields a function whose value is the first function applied to the result of the second
• It is written as an expression, using a \( \circ \) operator (called "circle" or "round")
  - E.g., \( h = f \circ g \)
    if \( f(x) = x + 2 \), and
    \( g(x) = 3 \times x \)
    then \( h(x) = f(g(x)) = (3 \times x) + 2 \)

Apply-to-all

• Apply-to-all takes a single function as a parameter
• If applied to a list of arguments, apply-to-all applies its functional parameter to each element of the list, and then collects results in a list or sequence
• It is denoted by \( \alpha \)
  - E.g., \( h(x) = x \times x \times x \), then
    \( \alpha(h, (2, 3, 4)) = (4, 9, 16) \)

Lambda expression

• Early theoretical work on functions separated the task of defining a function from that of naming the function
• Lambda notation, \( \lambda \), provides a method for defining nameless functions
• A lambda expression is a function, which specifies the parameters, and the mapping expression
  - E.g., \( \lambda(x) \times x \times x \times x \)
Lambda-Calculus

• In the mid 1960s, Peter Landin observed that a complex programming language can be understood by formulating it as a tiny core calculus capturing the language's essential mechanisms, together with a collection of convenient derived forms whose behavior is understood by translating them into the core.

factorial Example

• factorial(n) =
  if n = 0 then 1 else n * factorial(n - 1)
• The corresponding λ-calculus term is:
  factorial(n) =
  \( \lambda n. \text{if } n=0 \text{ then } 1 \text{ else } n * \text{factorial} (n - 1) \)
• Meaning
  – For each nonnegative number n, instantiating the function with the argument n yields the factorial of n as a result.

Syntax of λ-calculus

\[ t ::= x \quad (a \text{ variable}) \]
\[ | \lambda x. t \quad (a \text{ function}) \]
\[ | t \ t \quad (\text{function application}) \]
• The syntax of lambda-calculus comprises three sorts of terms
  – Variable itself is a term
  – The abstraction of a variable x from a term t is a term
  – The application of term t₁ to another term t₂, is a term.

Two conventions of writing lambda-terms

• Application is left associative
  – Given s t u, the calculation is (s t) u
  \[ \text{apply} \]
  \[ \text{apply} \quad u \]
  \[ s \quad t \]
Two Conventions

- The body of abstraction is extended to right as much as possible
  - Given \( \lambda x. \lambda y. x \, y \, x \), the calculation is \( \lambda x. (\lambda y. ((x \, y) \, x)) \).

<table>
<thead>
<tr>
<th>Two Conventions</th>
</tr>
</thead>
<tbody>
<tr>
<td>• The body of abstraction is extended to right as much as possible</td>
</tr>
<tr>
<td>• Given ( \lambda x. \lambda y. x , y , x ), the calculation is ( \lambda x. (\lambda y. ((x , y) , x)) ).</td>
</tr>
</tbody>
</table>

Scope

- An occurrence of the variable \( x \) is said to be bound when it occurs in the body \( t \) of an abstraction \( \lambda x. t \).
- An occurrence of \( x \) is free if it appears in a position where it is not bound by an enclosing abstraction on \( x \).
  - In \( x \, y \), and \( \lambda y. x \, y \), \( x \) is free.
  - In \( \lambda x. x \), and \( \lambda z. \lambda x. \lambda y. (y \, z) \), \( x \) is bound.

<table>
<thead>
<tr>
<th>Scope</th>
</tr>
</thead>
<tbody>
<tr>
<td>• An occurrence of the variable ( x ) is said to be bound when it occurs in the body ( t ) of an abstraction ( \lambda x. t ).</td>
</tr>
<tr>
<td>• An occurrence of ( x ) is free if it appears in a position where it is not bound by an enclosing abstraction on ( x ).</td>
</tr>
<tr>
<td>• In ( x , y ), and ( \lambda y. x , y ), ( x ) is free.</td>
</tr>
<tr>
<td>• In ( \lambda x. x ), and ( \lambda z. \lambda x. \lambda y. (y , z) ), ( x ) is bound.</td>
</tr>
</tbody>
</table>

Operational Semantics

- \( (\lambda x. t_{12}) t_2 \rightarrow (x_1 \rightarrow t_2) t_{12} \)
  - Evaluate the term \( t_{12} \) by replacing every occurrence of \( x \) with \( t_2 \).
  - What is the reduction result of \( (\lambda x. x) \, y \) ?
  - What is the evaluation result of the term \( (\lambda x. x) ((\lambda x. x) \, x)) \,(u \, r) \) ?
  - All terms of the form \( (\lambda x. t_{12}) t_2 \) is called redex (reducible expression).
  - The operation of rewriting a redex according to the above rule is called beta-reduction.

<table>
<thead>
<tr>
<th>Operational Semantics</th>
</tr>
</thead>
<tbody>
<tr>
<td>• ( (\lambda x. t_{12}) t_2 \rightarrow (x_1 \rightarrow t_2) t_{12} )</td>
</tr>
<tr>
<td>• Evaluate the term ( t_{12} ) by replacing every occurrence of ( x ) with ( t_2 ).</td>
</tr>
<tr>
<td>• What is the reduction result of ( (\lambda x. x) , y ) ?</td>
</tr>
<tr>
<td>• What is the evaluation result of the term ( (\lambda x. x) ((\lambda x. x) , x)) ,(u , r) ) ?</td>
</tr>
<tr>
<td>• All terms of the form ( (\lambda x. t_{12}) t_2 ) is called redex (reducible expression).</td>
</tr>
<tr>
<td>• The operation of rewriting a redex according to the above rule is called beta-reduction.</td>
</tr>
</tbody>
</table>

An Example of Reduction

- \((\lambda x. x) ((\lambda x. x)(\lambda z. (\lambda x. x) \, z)) \)
  - \( (\lambda x. x)(\lambda z. (\lambda x. x) \, z) \)
  - \( \lambda z. (\lambda x. x) \, z \)

<table>
<thead>
<tr>
<th>An Example of Reduction</th>
</tr>
</thead>
<tbody>
<tr>
<td>• ((\lambda x. x) ((\lambda x. x)(\lambda z. (\lambda x. x) , z)) )</td>
</tr>
<tr>
<td>• ( (\lambda x. x)(\lambda z. (\lambda x. x) , z) )</td>
</tr>
<tr>
<td>• ( \lambda z. (\lambda x. x) , z )</td>
</tr>
</tbody>
</table>

Programming in the Lambda-Calculus

- Multiple arguments
  - Lambda-calculus provides no built-in support for multi-argument functions.
  - But we can use higher-order functions to achieve the same effect.
Multiple Arguments

- Suppose
  - s is a term involving two free variables x and y
  - We want to write a function f, such that for each pair of arguments (v, w), f yields the result of substituting v for x, and w for y
  - f = λx. λy. s
  - Applying f to (v, w): f v w

Multiple Arguments

- The transformation of multi-argument functions into higher-order functions is called currying