Chapter 15

Functional Programming

Topics

- Introduction
- Functional programs
- Mathematical functions
- Functional forms
- Lambda calculus
- Eager and lazy evaluation
- Haskell

Introduction

- Emerged in the early 1960s for Artificial Intelligence and its subfields:
  - Theorem proving
  - Symbolic computation
  - Rule-based systems
  - Natural language processing
- The original functional language was Lisp, developed by John McCarthy (1960)

Functional Programs

- A program is a description of a specific computations.
  - A program can be seen as a “black box” for obtaining outputs from inputs.
  - From this point of view, a program is equivalent to a mathematical function.

Mathematical Functions

- A ***function*** is a rule that associates to each x from some set X of values a unique y from another set Y of values.
  - In mathematical terminology, if f is the name of the function

    `y = f(X)` or

    `f: X → Y`
  - The set `X` is called the *domain* of `f`.
  - The set `Y` is called the *range* of `f`.

Mathematical Functions

- The `x` in `f(x)`, which represents any value from `X` (domain), is called *independent variable*.
- The `y` from the set `Y` (range), defined by the equation `y = f(x)` is called *dependent variable*.
- Sometimes `f` is not defined for all `x` in `X`, it is called a *partial function*. Otherwise it is a *total function*.

- Example: `square(x) = x * x`

  function name — parameters — mapping expressions

Mathematical Functions

- Everything is represented as a mathematical function:
  - *Program*: x represents the input and y represents the output.
  - *Procedure* or *function*: x represents the parameters and y represents the returned values.
- No distinction between a program, a procedure, and a function. However, there is a clear distinction between input an output values.

Mathematical Functions: variables

- In imperative programming languages, variables refer to memory locations as well as values.

```
x = x + 1
```

  - Means "update the program state by adding 1 to the value stored in the memory cell named x and then storing that sum back into that memory cell"
  - The name x is used to denote both a value (as in x+1), often called an *r-value*, and a memory address, called an *l-value*.

Mathematical Functions: variables

- In mathematics, variables always stand for actual values, there is no concept of memory location (l-values of variables).
  - Eliminates the concept of variable, except as a name for a value.
  - Eliminates assignment as an available operation.

Mathematical Functions: variables

- Consequences of the lack of variables and assignment
  1. No loops.
     - The effect of a loop is modeled via recursion, since there is no way to increment or decrement the value of variables.
  2. No notation of the internal state of a function.
     - The value of any function depends only on the values of its parameters, and not on any previous computations, including calls to the function itself.

Mathematical Functions: variables

- The value of a function does not depend on the order of evaluation of its parameters.
- The property of a function that its value depend only on the values of its parameters is called *referential transparency*.

3. No state.
   - There is no concept of memory locations with changing values.
   - Names are associated to values which once the value is set it never changes.

Mathematical Functions

- Functional Forms
  - Def: A higher-order function, or functional form, is one that either takes functions as parameters or yields a function as its result, or both

Functional Forms

1. Function Composition
   - A functional form that takes two functions as parameters and yields a function whose value is the first actual parameter function applied to the application of the second

   Form: `h ≡ f ° g`

   which means `h (x) ≡ f ( g ( x ))`

   For `f (x) ≡ x * x * x` and
   `g (x) ≡ x + 3,`

   `h ≡ f ° g` yields `(x + 3)* (x + 3)* (x + 3)`

Functional Forms

2. Construction
   - A functional form that takes a list of functions as parameters and yields a list of the results of applying each of its parameter functions to a given parameter

   Form: `[f, g]`

   For `f (x) ≡ x * x * x` and
   `g (x) ≡ x + 3,`

   `[f, g] (4)` yields `(64, 7)`

Functional Forms

3. Apply-to-all
   - A functional form that takes a single function as a parameter and yields a list of values obtained by applying the given function to each element of a list of parameters

   Form: α

   For `h (x) ≡ x * x * x`

   `α( h, (3, 2, 4))` yields `(27, 8, 64)`

Pure Functional Programming

- In pure functional programming there are no variables, only constants, parameters, and values.
- Most functional programming languages retain some notation of variables and assignment, and so are “impure”
  - It is still possible to program effectively using the pure approach.

Lambda Calculus

- The foundation of functional programming developed by Church (1941).
- A *lambda expression* specifies the parameters and definition of a function, but not its name.
  - Example: lambda expression that defined the function `square`:

    `(λx·x*x)`
  - The identifier x is a parameter for the (unnamed) function body `x*x`.

Lambda Calculus

- Application of a lambda expression to a value: `((λx·x*x)2)` which evaluates to `4`
- What is a lambda expression?
  1. Any identifier is a lambda expression.
  2. If `M` and `N` are lambda expressions, then the *application* of `M` to `N`, written `(MN)` is a lambda expression.
  3. An *abstraction*, written `(λx·M)` where `x` is an identifier and `M` is a lambda expression, is also a lambda expression.

Lambda Expressions: BNF

- A simple BNF grammar for the syntax of the lambda calculus

    LambdaExpression → ident | (M N) | (λ ident · M)
    M → LambdaExpression
    N → LambdaExpression

- Examples:

    `x`
    `(λx·x)`
    `((λx·x)(λy·y))`

Lambda Expressions: free and bound variables

- In the lambda expression `(λx·M)`
  - The identifier `x` is said to be *bound* in the subexpression `M`.
  - Any identifier not bound in `M` is said to be *free*.
  - Free variables are like globals and bound variables are like locals.
  - Free variables can be defined as:

```
free(x) = x
free(MN) = free(M) ∪ free(N)
free(λx·M) = free(M) – {x}
```


Lambda Expressions: substitution

- A substitution of an expression `N` for a variable `x` in `M`, written `M[N/x]`, is defined:
  1. If the free variable of `N` have no bound occurrences in `M`, then the term `M[N/x]` is formed by replacing all free occurrences of `x` in `M` by `N`.
  2. Otherwise, assume that the variable `y` is free in `N` and bound in `M`. Then consistently replace the binding and corresponding bound occurrences of `y` in `M` by a new variable, say `u`. Repeat this renaming of bound variables in `M` until the condition in Step 1 applies, then proceed as in Step 1.

Lambda Expressions: substitution

- Examples:

```
x[y/x] = y
(xx)[y/x] = (yy)
(zw)[y/x] = (zw)
(zx)[y/x] = (zy)
[λx·(zx))[y/x] = (λu·(zu))[y/x] = (λu·(zu))
```


Lambda Expressions: beta-reduction

- The meaning of a lambda expression is defined by the *beta-reduction* rule:

    `((λx·M)N) ⇒ M[N/x]`

- An *evaluation* of a lambda expression is a sequence `P ⇒ Q ⇒ R ⇒ …`
  - Each expression in the sequence is obtained by the application of a beta-reduction to the previous expression.

    `((λy·((λx·xyz)a))b) ⇒ ((λy·ayz)b) ⇒ (abz)`

Functional Programming vs. Lambda Calculus

- A functional programming languages is essentially an applied lambda calculus with constant values and functions build in.
  - The pure lambda expression `(xx)` can be written as `(x times x)` or `(x*x)` or `(* x x)`
  - When constants, such as numbers, are added (with their usual interpretation and definitions for functions, such as *), then *applied lambda calculi* is obtained

Eager Evaluation

- An important distinction in functional languages is usually made in the way they define function evaluation.
- *Eager Evaluation or call by value:* In languages such as Scheme, all arguments to a function are normally evaluated at the time of the call.
  - Functions such as `if` and `and` cannot be defined without potential run-time error

Eager Evaluation

```
(if ( = x 0 ) 1 ( / 1 x ))
```

- Defined the value of the function to be 1 when x is zero and 1/x otherwise.
- If all arguments to the if functions are evaluated at the time of the call, division by zero cannot be prevented.

Lazy Evaluation

- An alternative to the eager evaluation strategy is *lazy evaluation or call by name*, in which an argument to a function is not evaluated (it is deferred) until it is needed.
  - It is the default mechanism of Haskell.

Eager vs. Lazy Evaluation

- An advantage of eager evaluation is efficiency in that each argument passed to a function is only evaluated once,
  - In lazy evaluation, an argument to a function is reevaluated each time it is used, which can be more than once.
- An advantage of lazy evaluation is that it permits certain interesting functions to be defined that cannot be implemented as eager languages

Haskell

Haskell

- The interactive use of a functional language is provided by the HUGS (Haskell Users Gofer System) environment developed by Mark Jones of Nottingham University.
- HUGS is available from
  http://www.haskell.org/hugs/
- The Haskell web page is
  http://www.haskell.org/

Haskell: sessions

- Expressions can be typed directly into the Hugs/Haskell screen.
  - The computer will respond by displaying the result of evaluating the expression, followed by a new prompt on a new line, indicating that the process can begin again with another expression

```
? 6 * 7
42
```

- This sequence of interactions between user and computer is called a *session*.

Haskell: scripts

- Scripts are collections of definitions supplied by the programmer.

```
square  :: Integer → Integer
square x = x * x
smaller :: (Integer,Integer) → Integer
smaller (x,y)= if x ≤ y then x else y
```

- Given the previous script, the following session is now possible:

```
? square 3768       ? square(smaller(5,3+4))
14197824            25
```


Haskell: scripts

- The purpose of a definition of a function is to introduce a *binding* associating a given name with a given definition.
  - A set of bindings is called an *environment* or *context*.
    - Expressions are always evaluated in some context and can contain occurrences of the names found in that context.
    - The Haskell evaluator uses the definitions associated with those names as rules for simplifying expressions.

Haskell: scripts

- Some expressions can be evaluated without having to provide a context.
  - Those operations are called *primitives* (the rules of simplification are build into the evaluator).
    - Basic operations of arithmetic.
    - Other libraries can be loaded.
- At any point, a script can be modified and resubmitted to the evaluator.

Haskell: first things to remember

- Scripts are collections of definitions supplied by the programmer.
- Definitions are expressed as equations between certain kinds of expressions and describe mathematical functions.
  - Definitions are accompanied by type signatures.
- During a session, expressions are submitted for evaluation
  - These expressions can contain references to the functions defined in the script, as well as references to other functions defined in libraries.

Haskell: evaluation

- The computer evaluates an expression by reducing it to its simplest equivalent form and displaying the result.
  - This process is called *evaluation*, *simplification*, or *reduction*.
  - Example: `square(3+4)`
  - An expression is *canonical* or in *normal form* If it cannot be further reduced.

Haskell: evaluation

- A characteristic feature of functional programming is that if two different reduction sequences terminate, they lead to the same result.
  - For some expressions some ways of simplification will terminate while other do not.
  - Example: `three infinity`
  - Lazy evaluation guarantees termination whenever termination is possible

Getting Started with Hugs

```
% hugs
Type : ? for help
Prelude> 6*7
42
Prelude> square(smaller(6,9))
ERROR – Undefined variable “smaller”
Prelude> sqrt(16)
4.0
Prelude> :load example1.hs
Reading file “example1.hs”
Main> square(smaller(6,9))
36
```


Getting Started with Hugs

Typing `:?` In Hugs will produce a list of possible commands.
Typing `:quit` will exit Hugs
Typing `:reload` will repeat last load command
Typing `:load` will clear all files