CSE 3302
Programming Languages

Functional Programming Language
(Introduction and Scheme)

Chengkai Li
Fall 2007

Resources

• Textbook: Chapter 11
  • Tutorial:
      (Chapter 1-2)
  – Yet Another Haskell Tutorial http://www.cs.utah.edu/~hal/hutut
      (Chapter 1-4, 7)
  • Implementation:
    • (Optional) DrScheme http://www.drscheme.org/
    • (Required) Hugs http://www.haskell.org/hugs/ (download WinHugs)
  • (Optional) Further reading:
    – A Gentle Introduction to Haskell 98 http://www.haskell.org/tutorial/

History

<table>
<thead>
<tr>
<th>Lambda Calculus</th>
<th>formal model of computation</th>
</tr>
</thead>
<tbody>
<tr>
<td>Lisp</td>
<td>symbolic computations with lists</td>
</tr>
<tr>
<td>Scheme</td>
<td>algebraic programming with arrays</td>
</tr>
<tr>
<td>APL</td>
<td>let and where clauses equalational reasoning; birth of pure functional programming...</td>
</tr>
<tr>
<td>ISWIM</td>
<td>originally meta language for theorem proving</td>
</tr>
<tr>
<td>ML</td>
<td>lazy evaluation</td>
</tr>
<tr>
<td>Haskell</td>
<td>“Grand Unification” of functional languages</td>
</tr>
</tbody>
</table>

Disclaimer

• Many of the slides are based on “Introduction to Functional Programming" by Graham Hutton, lecture notes from Oscar Nierstrasz, and lecture notes of Kenneth C. Louden.

Functional Programming

• Functional programming is a style of programming:

  Imperative Programming:
  – Program = Data + Algorithms

  OO Programming:
  – Program = Object. message (object)

  Functional Programming:
  – Program = Functions Functions

• Computation is done by application of functions

Functional Programming Language

• A functional language supports and advocates for the style of

  • Everything is function (input -> function -> output)
  • No variables or assignments (only constant values, arguments, and returned values. Thus no notion of state, memory location)
  • No loops (only recursive functions)
  • No side-effect (Referential Transparency): the value of a function depends only on the values of its parameters. Evaluating a function with the same parameters gets the same results. There is no state. Evaluation order or execution path don’t matter. (random() and getchar()) are not referentially transparent.)
  • Functions are first-class values: functions are values, can be parameters and return values, can be composed.
We can use functional programming in imperative languages

- **Imperative style**
  ```c
  int sumto(int n)
  { int i, sum = 0;
    for(i = 1; i <= n; i++) sum += i;
    return sum;
  }
  ```

- **Functional style**
  ```c
  int sumto(int n)
  { if (n <= 0) return 0;
    else return sumto(n-1) + n;
  }
  ```

Why does it matter, anyway?

The advantages of functional programming languages:
- Simple semantics, concise, flexible
- "No" side effect
- Less bugs

It does have drawbacks:
- Execution efficiency
- More abstract and mathematical, thus more difficult to learn and use.

Even if we don’t use FP languages:
- Features of recursion and higher-order functions have gotten into most programming languages.

Functional Programming Languages in Use

Popular in prototyping, mathematical proof systems, AI and logic applications, research and education.

**Scheme:**

- Document Style Semantics and Specification Language (SGML stylesheets)
- GNU
- Guile (GNU’s official scripting language)
- Emacs
- Haskell
- Linspire (commercial Debian-based Linux distribution)
- Xmonad (X Window Manager)
- XSLT (Extensible Stylesheet Language Transformations)

Syntax (slightly simplified):

- `expression-sequence` → `expression` `expression-sequence` | `expression`
- `atom` → `number` | `string` | `identifier` | `character` | `boolean`
- `list` → `['expression-sequence']`

- Everything is an expression: programs, data, …
- Only 2 basic kinds of expressions:
  - atoms: unstructured
  - lists: the only structure (a slight simplification).

Expressions:

| 42 | —a number |
| "hello" | —a string |
| #T | —the Boolean value "true" |
| #\a | —the character ‘a’ |
| (2.1 2.2 3.1) | —a list of numbers |
| hello | —a identifier |
| (+ 2 3) | —a list (identifier "+" and two numbers) |
| (* (+ 2 3) (/ 6 2)) | —a list (identifier "*" and two lists) |
Eager Evaluation

- A list is evaluated by recursively evaluating each element in the list as an expression (in some unspecified order); the first expression in the list must evaluate to a function. This function is then applied to the evaluated values of the rest of the list. (prefix form).

  E.g.:
  \[
  \begin{align*}
  3 + 4 * 5 & \rightarrow (+ 3 (* 4 5)) \\
  (a \rightarrow b)(x \rightarrow 0) & \rightarrow (\text{let} \ a = b \ \text{in} \ 0) \\
  \text{gcd}(10, 35) & \rightarrow (\text{gcd} \ 10 \ 35)
  \end{align*}
  \]

- Most expressions use applicative order evaluation (eager evaluation): arguments are evaluated at a call site before they are passed to the called function.

Lazy Evaluation: Special Forms

- **if function** (if a b c):
  - a is always evaluated one of b and c is evaluated and returned as result.

  E.g.:
  \[
  \begin{align*}
  \text{gcd}(10, 35) & \rightarrow (\text{gcd} \ 10 \ 35)
  \end{align*}
  \]

Lazy Evaluation: Special Forms

- **define function**: (define a b): define a name
  (define (a ...) b1 b2 ...): define a function

  the first expression is never evaluated.

  E.g.:
  \[
  \begin{align*}
  \text{let} \ x = (\text{gcd} \ v \ u) & \rightarrow (\text{gcd} \ v \ \{\text{remainder} \ u \ v\})
  \end{align*}
  \]

Other Special Forms

- **let function**: create a binding list, then evaluate an expression

  \[
  \begin{align*}
  \text{let} \ ((n1 \ e1) (n2 \ e2) ...) \ v1 \ v2 ... & \rightarrow \text{evaluate expression}
  \end{align*}
  \]

Lists

- **List**: Only data structure
  - Used to construct other data structures.

  - **cons**: construct a list
    \[
    \begin{align*}
    (1 \ 2 \ 3) & \rightarrow ([1 \ 2 \ 3]) \\
    (\text{cons} \ 1 \ (\text{cons} \ 2 \ (\text{cons} \ 3 \ '()))) & \rightarrow ([1 \ 2 \ 3]) \\
    (\text{cons} \ 1 \ '(2 \ 3)) & \rightarrow ([1 \ 2 \ 3])
    \end{align*}
    \]

  - **car**: the first element (head)
    \[
    \begin{align*}
    \text{car} \ '(1 \ 2 \ 3) & \rightarrow 1
    \end{align*}
    \]

  - **cdr**: the tail
    \[
    \begin{align*}
    \text{cdr} \ '(1 \ 2 \ 3) & \rightarrow (2 \ 3)
    \end{align*}
    \]
Data structures

(define L '((1 2) 3 (4 (5 6))))
(car (car L))
(cdr (car L))
(car (car (cdr (cdr L)))))

Note:
car(car = caar
car(car = cdar
car(car = caaddr
Box diagrams

• L = ([1 2) 3 (4 5 6)]) looks as follows in memory

Lambda expressions /function values

• A function can be created dynamically using a lambda expression, which returns a value that is a function:
  (lambda (x) (* x x))
• The syntax of a lambda expression:
  [lambda list-of-parameters exp1 exp2 ...]
• Indeed, the "function" form of define is just syntactic sugar for a lambda:
  (define (f x) (* x x))
  is equivalent to:
  (define f (lambda (x) (* x x)))

Function values as data

• The result of a lambda can be manipulated as ordinary data:
  > (lambda (x) (* x x)) 5)
  25
  > (define (add-x x) (lambda(y)(+ x y)))
  > (define add-2 (add-x 2))
  > (add-2 15)
  17

Higher-order functions

• Higher-order function:
  a function that returns a function as its value
  or takes a function as a parameter
  or both
  E.g.:
  • add-x
  • compose (next slide)
let expressions as lambdas

• A let expression is really just a lambda applied immediately:
  (let ((x 2) (y 3)) (+ x y))
  is the same as
  ((lambda (x y) (+ x y)) 2 3)

• This is why the following let expression is an error if we
  want x = 2 throughout:
  (let ((x 2) (y (+ x 1))) (+ x y))

• Nested let (lexical scoping)
  (let ((x 2)) (let ((y (+ x 1))) (+ x y)))