Programming Languages (Langages Evolués)

Roel Wuyts

Functional Programming

Note

- We will see:
  - functional programming in general
  - one functional programming language: Scheme
- Later on in the course we'll see some finer points in more detail
  - ...and compare them to other paradigms or languages

Functional Programming (FP)

- Key concepts:
  - Everything is a function: +, -, *, if, >, <, myfunc, ...
- Control Structure: function evaluation and application
- No need for variables or side effects
- First Class functions
- Formal foundation: lambda calculus

Concept: First class

- A programming language element is called First class if
  - it can be assigned to a variable
  - it can be passed as argument to a function/procedure/method
  - it can be returned as result in a function/procedure/method
- Examples: functions in Scheme, objects in Java, classes in Smalltalk, ...

FP Theory

- Functional Programming Foundations Overview:
  - Lambda calculus (Church, 1941)
  - Recursive functions (Kleene, and Church) is equivalent with universal machines (Turing)
    - So functional programming languages have the same “power” as imperative languages
  - Church-Rosser Theorem: Result is independent of order of evaluation
    - when there is no side effects!

Lambda Calculus

- Reduction technique
- Uses λ-expressions (lambda expressions)
- λ-expressions are reduced with β-reductions
- Example:

  $(\lambda.x\ x+1)\ 4$ (β-reduction)

  $\Rightarrow 4+1$

Lambda Calculus

- Lambda calculus has:
  - variable references
  - lambda expressions with a single parameter
  - procedure calls
- Grammar:

```
<exp> ::= <varref>
        | (lambda (<var>) <exp>)
        | (<exp> <exp>)
<exp> ::= <number>
```
(not strictly necessary)

Concept: Free and Bound

- Variable references can be free or bound:
  - a variable reference is said to be bound in an expression if it refers to a formal parameter introduced in the expression
  - a reference that is not bound to a formal parameter in the expression is said to be free
- Example

```
((lambda (x) x) y)
```

  reference to x is bound, reference to y is free

α-conversion

- exp[y / x]: substitutes an expression y for all free occurences of a variable x in expression exp
  (lambda (var) exp) = (lambda (var') exp[var' / var])
- So

```
(lambda (x)
     ((lambda (x) (cons x '()))
     (cons x '())))
```

[y / x]

```
(lambda (y)
     ((lambda (x) (cons x '()))
     (cons y '())))
```

Concept: Name capture

- Naming conflict that arises when a name is the same as an already existing free name
- Example: α-conversion with existing name
- Other examples: package and module systems
  - will encounter this later on

β-reduction: idea

- ```
  ((lambda (x)
      (lambda (y) (x y)))
    (y w))
  ```

  α-reduction to rename y

  ```
  ((lambda (x)
      (lambda (z) (x z)))
    (y w))
  ```

  reduce

  ```
  (lambda (z) ((y w) z))
  ```
  ```

β-reduction: definition

- So: need to avoid capture of vree variables.
- Will give inductive definition for substitution.
- Subsitution of M for x in E: $E[M / x]$:
  - $x[M / x] = M$
  - $y[M / x] = y$, with y a variable or a constant, $x \neq y$
  - $(F\ G)[M / x] = (F[M / x]\ G[M / x])$
  - ...

β-reduction: definition (ctd)

- E = (lambda (y) E')
  - y = x, or : (lambda (x) E')[M / x] = (lambda (x) E')
  - x not free in E':
    (lambda (y) E')[M / x] = (lambda (y) E')
  - y ≠ x, y not free in M:
    (lambda (y) E')[M / x] = (lambda (y) E'[M / x])
  - y ≠ x, x free in E', y free in M, z not free in E'/M:
    (lambda (y) E')[M / x]
    = (lambda (z) (E'[z / y])[M / x])

Concept: Operational Semantics

- Operational Semantics:
  - A reduction rule expresses a semantic equivalence between two expressions.
  - describe computation as a rewriting process where an expression is transformed by the application of primitive computation rules.
- Other approaches: denotational semantics, axiomatic semantics, ...

Church-Rosser Theorem

- If an expression E can be reduced to either E1 or E2, using different reduction sequences, then there is some expression N that can be reached from both E1 and E2.

$$
\begin{tikzcd}
 & E \arrow[dl] \arrow[dr] & \\
E1 \arrow[dr] & & E2 \arrow[dl] \\
 & N &
\end{tikzcd}
$$

- Consequence: Result is independent of order of evaluation

Computation Strategies

- Example: reduce the following expression:

```
((lambda (x) (x (x y)))
  ((lambda (w) w) z))
```

- A computation strategy:
  - determines in which order to apply primitive computation steps.
  - The strategy defines rules to identify within an abstract syntax tree the first candidate node on which to apply a reduction rule.
- Will see 2 strategies, but more exist.

Applicative-Order Reduction

- no reduction inside the body of a lambda

```
(reduce-once-appl exp succeed fail) =
  (succeed exp')
    if exp contains an applicative β-redex,
    in which case exp' is the result of performing it
    on exp
  (fail)
    if exp has no applicative β-redex
```

Applicative-Order Reduction

```
(define (reduce-once-appl e success fail)
  (cases exp e
    (varref (var) (fail))
    (lambda (formal body) (fail))
    (app (rator rand)
      (if (and (lambda? rator) (not (app? rand)))
        (success (beta-reduce e))
        (reduce-once-appl rator
          (lambda (reduced-rator)
            (success (make-app reduced-rator rand)))
          (lambda ()
            (reduce-once-appl rand
              (lambda (reduced-rand)
                (success (make-app rator reduced-rand)))
              fail)))))))
```

Applicative Order Example

- ```
  ((lambda (x) (x (x y)))
   ((lambda (w) w) z))
  ```

```
((lambda (x) (x (x y))) z)
```

```
(z (z y))
```
```

Concept: Continuation Passing

- Procedures *success* and *fail* are called continuations
  - determine how the computation continues
- When continuations are passed from the outside: continuation passing style

Leftmost Reduction

- reduce the Beta-redex whose left paren comes first
- expression can have at most one normal form
  - leftmost reduction always finds it
  - a.k.a. *normal order reduction* or *lazy evaluation*
- Price to pay? efficiency...

Leftmost Reduction Example

- ```
  ((lambda (x) (x (x y)))
   ((lambda (w) w) z))
  ```

```
(((lambda (w) w) z)
  (((lambda (w) w) z)
   y))
```

```
(z (((lambda (w) w) z) y))
```

```
(z (z y))
```
```

Note

- Example:
```
(define (try a b)
   (if (= a 0) 1 b))
```
- Now evaluate: `(try 0 (/ 1 0))`
- Result in Scheme? `Error`.
  - Scheme uses applicative order
- Result in normal-order language? `1`
  - Division is never evaluated

Functional Languages: Overview

Language	Typing	Scoping	Evaluation	Side effects
Lisp	dynamic	dynamic	eager	yes
Scheme	dynamic	static	eager + lazy (continuations)	yes
Standard ML	strong,static	static	eager	yes
Haskell	strong,static	static	lazy (combinator reduction)	no

Lisp: History

- LISP = LISt Processing
- First functional programming language
- Developed by John McCarthy at MIT in 1958
- Primary usage: artificial intelligence:
  - needed lists (not arrays)
  - symbolic calculation (not numeric calculation)

Lisp: Characteristics

- Dynamic typing
- Uses functions throughout
- Higher Order Functions
- Automatic garbage collection
- Call by reference (variables are references)
- Formal foundation (lambda-calculus)
- Strong resemblance of code and data
- Dynamic scoping

Scheme: History

- Developed by Steele and Sussman in 1975
- Successor of LISP
  - Simpler
  - More standardized

Scheme: Characteristics

- "First Class" functions
  - In LISP, a lambda expression can be applied
  - In Scheme, a lambda expression returns a closure
- Typage dynamique
- Static scoping
- Continuations: data structure representing "the rest of the computation"
- Side effects

Concept: Static/Dynamic Scoping

- ```
  (define f
      (lambda (a b) (+ a b c)))
  (define g
      (lambda (c) (f 1 2)))
  ```

- Evaluate `(g 3)`:
  - in LISP: when f is evaluated, the value of c in the environment is 3

    ➞ dynamic scoping

  - in Scheme: error: there is no value for c

    ➞ static scoping
  ```

Standard ML: History

- Developed by Robin Milner in 1973
- The first language to include static typing of polymorphic functions

Standard ML

- Syntax resembling Pascal
- Formal semantics
- Exception Handling
- Static Scoping
- Module system
- Strong static typing, no type coercion
- Combination of explicit type declarations and type inferencing to determine type of undeclared variables
- Garbage collection

Haskell: History

- Developed by a group in 1987 as a pure functional programming language
  - standard
  - non-strict
- Current version: Haskell 98

Haskell: Characteristics

- Lazy evaluation
- Strong static typing
- Type inferencing
- Static Scoping
- Pure functional language
- list comprehension
  - mathematical notation
  - mechanisms to manipulate infinite lists

Wrap-up

- Functional programming:
  - formal foundation: lambda calculus
    - operational semantics
  - no side effects
  - computation strategies
- Several examples of functional programming languages
  - differences in typing and variable scoping

References

- Friedman, Wand and Haynes, *Essentials of Programming Languages*, MIT Press, 1992.
- H. Abelson, G.J. Sussman, J. Sussman. *Structure and Interpretation of Computer Programs*. MIT Press, 1984.