Chapter 11 :: Functional Languages

Historical Origins

• The imperative and functional models grew out of work undertaken by 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

• Turing’s model of computing was the Turing machine a sort of pushdown automaton using an unbounded storage “tape”
  – the Turing machine computes in an imperative way, by changing the values in cells of its tape – like variables just as a high level imperative program computes by changing the values of variables

Historical Origins

• Church’s model of computing is called the lambda calculus
  – based on the notion of parameterized expressions (with each parameter introduced by an occurrence of the letter \( \lambda \)—hence the notation’s name.
  – Lambda calculus was the inspiration for functional programming
  – one uses it to compute by substituting parameters into expressions, just as one computes in a high level functional program by passing arguments to functions

Historical Origins

• Mathematicians established a distinction between
  – constructive proof (one that shows how to obtain a mathematical object with some desired property)
  – nonconstructive proof (one that merely shows that such an object must exist, e.g., by contradiction)
• Logic programming is tied to the notion of constructive proofs, but at a more abstract level
  – the logic programmer writes a set of axioms that allow the computer to discover a constructive proof for each particular set of inputs

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
  – serious polymorphism
  – powerful list facilities
  – structured function returns
  – fully general aggregates
  – garbage collection

• So how do you get anything done in a functional language?
  – Recursion (especially tail recursion) takes the place of iteration
  – In general, you can get the effect of a series of assignments
    \[
    \begin{align*}
    x &:= 0 \quad \ldots \\
    x &:= \text{expr1} \quad \ldots \\
    x &:= \text{expr2} \quad \ldots
    \end{align*}
    \]
    from \( f_3(f_2(f_1(0))) \), where each \( f \) expects the value of \( x \) as an argument, \( f_1 \) returns \( \text{expr1} \), and \( f_2 \) returns \( \text{expr2} \).

• Recursion even does a nifty job of replacing looping
  \[
  \begin{align*}
  x &:= 0; \ i := 1; \ j := 100; \\
  \text{while } i < j \text{ do} \\
  &x := x + i * j; \\
  &i := i + 1; \\
  &j := j - 1 \\
  \text{end while} \\
  \text{return } x
  \end{align*}
  \]
  becomes \( f(0,1,100) \), where
  \[
  f(x,i,j) = \begin{cases} 
  i < j & \text{then} \\
  f(x+i*j, i+1, j-1) & \text{else } x 
  \end{cases}
  \]

• Thinking about recursion as a direct, mechanical replacement for iteration, however, is the wrong way to look at things
  – One has to get used to thinking in a recursive style
  – Even more important than recursion is the notion of higher-order functions
  – Take a function as argument, or return a function as a result
  – Great for building things

• Lisp also has the following (which are not necessarily present in other functional languages)
  – homo-iconography
  – self-definition
  – read-evaluate-print

• Variants of LISP
  – Pure (original) Lisp
  – Interlisp, MacLisp, Emacs Lisp
  – Common Lisp
  – Scheme

• Pure Lisp is purely functional; all other Lisps have imperative features
  • All early Lisps dynamically scoped
    – Not clear whether this was deliberate or if it happened by accident
  • Scheme and Common Lisp statically scoped
    – Common Lisp provides dynamic scope as an option for explicitly-declared special functions
    – Common Lisp now THE standard Lisp
  • Very big, complicated (The Ada of functional programming)
Functional Programming Concepts

- Scheme is a particularly elegant Lisp
- Other functional languages
  - ML
  - Miranda
  - Haskell
  - FP
- Haskell is the leading language for research in functional programming

A Bit of Scheme

- As mentioned earlier, Scheme is a particularly elegant Lisp
  - Interpreter runs a read-eval-print loop
  - Things typed into the interpreter are evaluated (recursively) once
  - Anything in parentheses is a function call (unless quoted)
  - Parentheses are NOT just grouping, as they are in Algol-family languages
    - Adding a level of parentheses changes meaning
      (+ 3 4) ⇒ 7
      ((+ 3 4)) ⇒ error
      (the '⇒' arrow means 'evaluates to')

A Bit of Scheme

- Scheme:
  - Boolean values #t and #f
  - Numbers
  - Lambda expressions
  - Quoting
    (+ 3 4) ⇒ 7
    (quote (+ 3 4)) ⇒ (+ 3 4)
    '(+ 3 4) ⇒ (+ 3 4)
  - Mechanisms for creating new scopes
    (let ((square (lambda (x) (* x x))) (plus +))
     (sqrt (plus (square a) (square b))))

A Bit of Scheme

- Scheme:
  - conditional expressions
    (if (< x 0) (- 0 x))
    (if (< x y) x y)
    (if (< 2 3) 4 5) ⇒ 4
  (cond
   (< 3 2) 1)
   (< 4 3) 2)
  (else 3)
  ⇒ 3
  - case selection
    (case month
     (((sep apr jun nov) 30)
      (feb) 28)
     (else 31)
    )

A Bit of Scheme

- Scheme:
  - Imperative stuff
    - assignments
    - sequencing (begin)
    - iteration
    - I/O (read, display)

A Bit of Scheme

- Scheme standard functions (this is not a complete list):
  - arithmetic
  - boolean operators
  - equivalence
  - list operators
  - symbol?
  - number?
  - complex?
  - real?
  - rational?
  - integer?
A Bit of Scheme

- expressions
  - Cambridge prefix notation for all Scheme expressions:
    \[(f \ x_1 \ x_2 \ ... \ x_n)\]
    
    \[(+ \ 2) \quad ; \text{evaluates to 4}\]
    \[(+ \ (* \ 5 \ 4) \ (- \ 6 \ 2)) \quad ; \text{means 5*4 + (6-2)}\]
    \[(\text{define } \text{Square} \ x \ (* \ x \ x)) \quad ; \text{defines a fn}\]
    \[(\text{define } f \ 120) \quad ; \text{defines a global}\]

  - Note: Scheme comments begin with ;


A Bit of Scheme

- expression evaluation
  - three steps:
    1. Replace names of symbols by their current bindings.
    2. Evaluate lists as function calls in Cambridge prefix.
    3. Constants evaluate to themselves.

  - e.g.,
    \[x \quad ; \text{evaluates to 5}\]
    \[ (+ \ (* \ x \ 4) \ (- \ 6 \ 2)) \quad ; \text{evaluates to 24}\]
    \[5 \quad ; \text{evaluates to 5}\]
    \[\text{"red"} \quad ; \text{evaluates to "red"}\]


A Bit of Scheme

- lists
  - series of expressions enclosed in parentheses
  - represent both functions and data
  - empty list written as ()
  - e.g., (0 2 4 6 8) is a list of even numbers

- stored as


A Bit of Scheme

- list transforming functions
  - using cons (construct):
    \[(\text{cons } 8 ()) \quad ; \text{gives } (8)\]
    \[(\text{cons } 6 \ (\text{cons } 8 ())) \quad ; \text{gives } (6 \ 8)\]
    \[(\text{cons } 4 \ (\text{cons } 6 \ (\text{cons } 8 ()))) \quad ; \text{gives } (4 \ 6 \ 8)\]
    \[(\text{cons } 4 \ (\text{cons } 6 \ (\text{cons } 8 9))) \quad ; \text{gives } (4 \ 6 \ 8 \ . \ 9)\]

  - Note: the last element of a list should be a null list


A Bit of Scheme

- more on car/cdr

  \[(\text{car } (\text{cdr } \text{evens})) \quad ; \text{gives 2}\]
  \[(\text{cdr } \text{evens}) \quad ; \text{gives 2}\]
  \[(\text{cadr } \text{evens}) \quad ; \text{gives } (1 \ 2 \ 4 \ 6 \ 8)\]
  \[(\text{caddr } \text{evens}) \quad ; \text{gives } (6 \ 8)\]
  \[(\text{car } \text{evens}) \quad ; \text{gives 1, or true}\]
  \[(\text{equal? } 5 \ (* \ 5)) \quad ; \text{gives #t, or true}\]
  \[(\text{append } (1 \ 3 \ 5) \text{ evens}) \quad ; \text{gives } (1 \ 3 \ 5 \ 0 \ 2 \ 4 \ 6 \ 8)\]
  \[(\text{list } (1 \ 3 \ 5) \text{ evens}) \quad ; \text{gives } ((1 \ 3 \ 5) \ (0 \ 2 \ 4 \ 6 \ 8))\]

  Note: the last two lists are different!

A Bit of Scheme

• defining functions

  (define (name arguments) function-body)

  (define (min x y) (if (< x y) x y))
  (define (abs x) (if (< x 0) (- 0 x) x))

  (define (factorial n)
         (if (< n 1) 1 (* n (factorial (- n 1))))
  )

  Note: be careful to match all parentheses


A Bit of Scheme

• even simple tasks are accomplished recursively

  (define (mystery1 alist)
         (if (null? alist) 0
             (+ (car alist) (mystery1 (cdr alist))))
  )

  (define (mystery2 alist)
         (if (null? alist) 0 (+ 1 (mystery2 (cdr alist))))
  )


A Bit of Scheme

• subst function

  (define (subst y x alist)
         (if (null? alist) '()
             (if (equal? x (car alist))
                 (cons y (subst y x (cdr alist)))
                 (cons (car alist) (subst y x (cdr alist))))
  )

  e.g., (subst 'x 2 '(1 (2 3) 2)) returns (1 (2 3) x)


A Bit of Scheme

• let expressions allow simplification of function definitions by defining intermediate expressions

  (define (subst y x alist)
         (if (null? alist) '()
             (let ((head (car alist)) (tail (cdr alist)))
                 (if (equal? x head)
                     (cons y (subst y x tail))
                     (cons head (subst y x tail))))
  )


A Bit of Scheme

• functions as arguments

  • mapcar applies the function to each member of a list

  (define (mapcar func alist)
         (if (null? alist) '()
             (cons (func (car alist))
                 (mapcar func (cdr alist))))
  )

  e.g., if (define (square x) (* x x)) then
     (mapcar square '(2 3 5 7 9)) returns (4 9 25 49 81)


A Bit of Scheme

Example program - Symbolic Differentiation

• Symbolic Differentiation Rules

  \[
  \frac{d}{dx}(c) = 0 \quad \text{c is a constant}
  \]

  \[
  \frac{d}{dx}(x) = 1
  \]

  \[
  \frac{d}{dx}(u+v) = \frac{du}{dx} + \frac{dv}{dx} \quad \text{u and v are functions of x}
  \]

  \[
  \frac{d}{dx}(u-v) = \frac{du}{dx} - \frac{dv}{dx}
  \]

  \[
  \frac{d}{dx}(uv) = u \frac{dv}{dx} + v \frac{du}{dx}
  \]

  \[
  \frac{d}{dx}\left(\frac{u}{v}\right) = \frac{v \frac{du}{dx} - u \frac{dv}{dx}}{v^2}
  \]

A Bit of Scheme
Example program - Symbolic Differentiation

• Scheme encoding
  1. Uses Cambridge Prefix notation
e.g., \(2x + 1\) is written as \((+ (* 2 x) 1)\)
  2. Function \(\text{diff}\) incorporates these rules.
e.g., \(\text{diff }'x'(+(+ (* 2 x) 1))\) should give an answer.
  3. However, no simplification is performed.
e.g. the answer for \(\text{diff }'x'(+(+ (* 2 x) 1))\) is
\(+ (+ (* 2 1) (* x 0)) 0\)
which is equivalent to the simplified answer, 2

A Bit of Scheme
Example program - Symbolic Differentiation

• Scheme program

```scheme
(define (diff x expr)
  (if (not (list? expr))
      (if (equal? x expr) 1 0)
      (let ((u (cadr expr)) (v (caddr expr)))
        (case (car expr)
          ((+) (list '+ (diff x u) (diff x v)))
          ((-) (list '-' (diff x u) (diff x v)))
          ((*) (list '+ (list '* u (diff x v))
                  (list '* v (diff x u))))
          ((/) (list '/ (list '-' (list '* v (diff x u))
                      (list '* u (diff x v)))
                      (list '* u v)))
        )))
```


A Bit of Scheme
Example program - Simulation of DFA

• We'll invoke the program by calling a function called 'simulate', passing it a DFA description and an input string
  – The automaton description is a list of three items:
  • start state
  • the transition function
  • the set of final states
  – The transition function is a list of pairs
  • the first element of each pair is a pair, whose first element is a state
    and whose second element in an input symbol
  • if the current state and next input symbol match the first element of a pair, then the finite automation enters the state given by the second element of the pair

A Bit of Scheme
Example program - Simulation of DFA

```scheme
(define zero-one-even-dfa
  '((0 0) (0 1) (1 0) (1 1)) ; start state
  (((0 0) 0) (0 1) (1 0) (1 1)) ; transition fn
  ((0 0) 0) (0 1) (1 0) (1 1) (0) ; final states
)
```

Figure 11.2 DFA to accept all strings of zeros and ones containing an even number of each. At the bottom of the figure is a representation of the machine as a Scheme data structure, using the conventions of Figure 11.1.
A Bit of OCaml

- OCaml is a descendent of ML, and cousin to Haskell, F#
  - “O” stands for objective, referencing the object orientation introduced in the 1990s
  - Interpreter runs a read-eval-print loop like in Scheme
  - Things typed into the interpreter are evaluated (recursively) once
  - Parentheses are NOT function calls, but indicate tuples

A Bit of OCaml

- OCaml:
  - Boolean values
  - Numbers
  - Chars
  - Strings
  - More complex types created by lists, arrays, records, objects, etc.
  - (+, *, /) for ints, (+., *, ./) for floats
  - let keyword for creating new names

```ocaml
let average = fun x y -> (x +. y) /. 2.;;
```

A Bit of OCaml

- Ocaml:
  - Variant Types
    ```ocaml
type 'a tree = Empty | Node of 'a * 'a tree * 'a tree;;
```
  - Pattern matching
    ```ocaml
    let atomic_number (s, n, w) = n;;
    let mercury = ("Hg", 80, 200.592);;
    atomic_number mercury;;
    ⇒ 80
    ```

A Bit of OCaml

- OCaml:
  - Different assignments for references ‘:=’ and array elements ‘<-’

```ocaml
let insertion_sort a =
  for i = 1 to Array.length a - 1 do
    let t = a.(i) in
    let j = ref i in
    while !j > 0 && t < a.(!j - 1) do
      a.(!j) <- a.(!j - 1);
      j := !j - 1;
    done;
    a.(!j) <- t;
  done;;
```

A Bit of OCaml

Example program - Simulation of DFA

- We’ll invoke the program by calling a function called ‘simulate’, passing it a DFA description and an input string
  - The automaton description is a record with three fields:
    - start state
    - the transition function
    - the list of final states
  - The transition function is a list of triples
    - the first two elements are a state and an input symbol
    - if these match the current state and next input, then the automaton enters a state given by the third element

Example program - Simulation of DFA

```ocaml
let simulate (start, trans, finals) input =
  let current = start in
  let rec next state char =
    match (trans state, char) with
    | (None, _) -> (current, None)
    | (Some (next, new state), _) -> (new state, Some next)
  in
  List.fold_left next (current, None) input
  |> List.map (fun (state, _) -> String.of_int state)
  |> List.member finals
```
Evaluation Order Revisited

- Applicative order
  - what you're used to in imperative languages
  - usually faster
- Normal order
  - like call-by-name: don't evaluate arg until you need it
  - sometimes faster
  - terminates if anything will (Church-Rosser theorem)

High-Order Functions

- Higher-order functions
  - Take a function as argument, or return a function as a result
  - Great for building things
  - Currying (after Haskell Curry, the same guy Haskell is named after)
    - For details see Lambda calculus on CD
    - ML, Miranda, OCaml, and Haskell have especially nice syntax for curried functions

Evaluation Order Revisited

- In Scheme
  - functions use applicative order defined with lambda
  - special forms (aka macros) use normal order defined with syntax-rules
- A strict language requires all arguments to be well-defined, so applicative order can be used
- A non-strict language does not require all arguments to be well-defined; it requires normal-order evaluation

Evaluation Order Revisited

- Lazy evaluation gives the best of both worlds
- But not good in the presence of side effects.
  - delay and force in Scheme
  - delay creates a "promise"

Functional Programming in Perspective

- Advantages of functional languages
  - lack of side effects makes programs easier to understand
  - lack of explicit evaluation order (in some languages) offers possibility of parallel evaluation (e.g. MultiLisp)
  - lack of side effects and explicit evaluation order simplifies some things for a compiler (provided you don't blow it in other ways)
  - programs are often surprisingly short
  - language can be extremely small and yet powerful
Functional Programming in Perspective

- Problems
  - difficult (but not impossible!) to implement efficiently on von Neumann machines
  - lots of copying of data through parameters
  - (apparent) need to create a whole new array in order to change one element
  - heavy use of pointers (space/time and locality problem)
  - frequent procedure calls
  - heavy space use for recursion
  - requires garbage collection
  - requires a different mode of thinking by the programmer
  - difficult to integrate I/O into purely functional model