Fundamentals of Functional Programming

Summer School
November 11, 2008

Christopher Dutchyn

University of Saskatchewan

Lists of Strings and Ints

```
  // node is EMPty, or payload is INT or
  STRing
typedef enum { EMP, INT, STR } node_tag;

// forward reference
typedef struct node *nodeptr;

// a node contains an int or a string
//                  and the next element
typedef struct node {
  node_tag t;
  union { int i;
          char *s; } payload;
  nodeptr next;
} node;
```

An example tree

```
node root;
root.tag = INT;      {INT,1}
root.payload = 1;
node next;
root.next = &next;
next.tag = INT;          {INT,2}
next.payload = 2;
node another;
next.next = another;
another.tag = STR;           {STR,"Hi"}
another.payload.s = "Hi"     {INT,3}
...                             {STR,"Aloha"}
                                   {INT,4}
                                      {STR,"Ola"}
```

Operations

```
// find the maximum integer
int max_int(node n) {
  switch (n.tag) {
    case EMP:
      assert (NULL == n.next);
      return 0;
    case INT:
      return max(max_int(*n.next),
                              n.payload.i);
    case STR:
      return max_int(*n.next);
    default:
      assert(false);
  }
}
```

Operations

```
// replace STR with INT == strlen
void str2int(node n) {
  switch (n.tag) {
    case EMP:
      assert (NULL == n.next);
      return;
    case STR:
      n.tag = INT;
      n.payload.i = strlen(n.payload.s);
    case INT:
      str2int(*n.next);
      return;
    default:
      assert(false);
  }
}
```

Operations

```
// find a STR
bool find_str(node n, char *s) {
  switch (n.tag) {
    case EMP:
      assert (NULL == n.next);
      return false;
    case INT:
        return (find_str(*n,next));
    case STR:
        return (0 == strcmp(s, n.payload.s)
                  || find_str(*n.next, s));
    default:
        assert(false);
  }
}
```

Main

```
int main() {
   node root = ...;   // make the tree

   int i = max_int(root);
   bool b = find_str(root, “Ola”);

   str2int(root);

   // !! old tree is gone !!
   bool nb = find_str(root, “Ola”);
   ...
}
```

Order of Execution

- implied by
  - changing state
  - reading input
  - generating output
  - throwing exceptions
  - ...
- in C (and other imperative languages)
  - the most common sequencing operator is

;

Main

```
int main() {
   node root = ...; // make the tree

   int i = max_int(root);
   bool b = find_str(root, "Ola");

   str2int(root);

   // !! old tree is gone !!
   bool nb = find_str(root, "Ola");
   ...
}
```

Is All This Sequencing Necessary?

```
// replace STR with INT == strlen
void str2int(node n) {
  switch (n.tag) {
  case EMP:
    assert (NULL == n.left);
    assert (NULL == n.right);
    return;
  case STR:
    n.tag = INT;
    n.payload.i = strlen(n.payload.s);
  case INT:
    str2int(*n.right);
    str2int(*n.left);
    return;
  default: assert(false);
```

Operations

```
// replace STR with INT == strlen
void str2int(node n) {
  switch (n.tag) {
    case EMP:
      assert (NULL == n.next);
      return;
    case STR:
      n.tag = INT;
      n.payload.i = strlen(n.payload.s);
    case INT:
      str2int(*n.next);
      return;
    default:
      assert(false);
  }
}
```

Mandatory Sequencing

- primitives:
  - **+** needs two (32-bit, 2's complement) integers
  - **[ ]** *array access* needs the array and index
  - **==** needs two values

  – depend on machine representation of values
    - expressions yielding those values must be computed before performing primitive operations on those values

Mandatory Sequencing

- conditionals (`if`, `switch`)
  - `if (x > y) then “higher” else “lower”`
- *strict* in their test expressions

Mandatory Sequencing

- function calls:
  - `max (x+1, y-3)`
  - need to have the function (`max`) before calling it
    - do arguments need to be computed before calling?
      - let's defer that question ...

Imperative Languages

- explicitly sequence everything
  - statement
  - expression
- many statements don't need sequencing
  - compilers (and compiler writers) undo much of it!
    - control-flow / data-flow analysis
    - code hoisting
    - speculative code execution in hardware
- sequential execution underperforms
  - on multi-core/thread hardware

Functional Languages

- adopt only the minimum sequencing
  - calls to primitives, conditionals
  - calls to user-written functions
  - explicit sequence statement
    - for when it is required: output ... later
- everything else can be computed whenever
  - at compilation time
  - once and cached
  - just when needed
  - in parallel
  - on a remote system

List of Strings and Ints

```
datatype Node = EMPNode

              | INTNode of Int
                         * Node

              | STRNode of String
                         * Node
```

Main *functionally* ...

```
fun main ()
  let root = ... (* make list *)
  in let i = max_int root
         b = find_str root "Ola"
     in (let newroot = str2int root
             nb = find_str newroot, "Ola"
         in ...
         end
     end
  end
```

Operations

```
fun max_int EMPNode = 0

    max_int INTNode (i,n) = max i (max_int n)

    max_int STRNode (_,n) = max_int n
```

Operations

```
// build new list, replacing STR with length
fun str2int EMPNode as n = n

    str2int STRNode (s,n) = INTNode (length s)
                                    (str2int n)

    str2int INTNode (i,n) = INTNode i
                                    (str2int n)
```

Operations

```
// find a STR
fun find_str s EMPNode = False

    find_str s INTNode (i,n) = find_str n

    find_str s STRNode (s',n) = (s == s')
                                or (find_str l)
```

Arguments -- Eager or Lazy

- Eager:
  - evaluate all arguments to values before calling
  - not just the procedure
  - faster
  - less memory
- Lazy:
  - keep argument expressions as expressions
  - evaluate these expressions when they're used
  - slower
  - more memory
- They have the same semantics!

Where are the type annotations?

- most (SML, Haskell, ...) infer them
  - without you writing them down
  - and check them at compile time
- type inference isn't a requirement of a functional language
  - but it's common!
- Scheme and LISP check at runtime
  - they're still safe and strong

Generics -- Polymorphic Types

```
type 'n GeneralNode = EMPNode

                    | INTNode of int
                                 * 'n

                    | STRNode of string
                                 * 'n

type Node = Node GeneralNode
```

Generics Operations

```
fun walk f EMPNode = (f EMPNode)

    walk f INTNode (i,n) =
                      (f (INTNode i
                                  (walk f n)))

    walk f STRNode (s,n) =
                      (f (STRNode s
                                  (walk f n)))
```

- notice f: it is a function
  - we can pass functions around as arguments
  - we can build functions wherever we want

Generics Operations

```
// build new tree, replacing STR with length
fun str2int n =
    let xform =
            (fn STRNode (s,n) =>
                              INTNode (length s) n
              | n => n)
    in
       walk xform n
    end
```

Generics Operations

```
// find a STR
fun find_str s n =
    let xform =
          (fn STRNode (s',n) => (s == s') || n
            | n => n)
    in
       walk xform n
    end
```

Lists and Comprehensions

- Lists are the fundamental language type
  - a number of standard transformations are known
    - map
      - » transform each element of a list in parallel
      - `map (fn x => x+1) [1,2,3,4]` ➛ `[2,3,4,5]`
    - fold (left and right variants)
      - » combine each sublist
      - `fold \op+ 0 [1,2,3,4]` ➛ `10`
- These generalize to comprehensions
    - `{ x+1 | x <- [1,2,3,4] }` ➛ `[2,3,4,5]`
  - which more concisely describe maps, folds

Costs of Functional Programming

- state and input/output are still required
- solution: Monads
  - thread state into each function
    - pass it along to next function

Threading State

```
type 'value SM = state -> ('value, state)


-- ^ :: value -> value SM

fun ^ val = fn state => (val,state)
```

- changes value into procedure that threads state

Threading State

```
-- >>= :: value SM -> (value -> value' SM)
                   -> value' SM

fun >>= f g =
  fn state =>
    let (result,newstate) = f state
     in g result newstate
     end
```

- acts as function composition threading state

Do Syntax

```
do {
   x <- 10          >>= ^10
   y <- f x               fn x => >>= f x
   y+5                                fn y => ^(y+5)
}
```

Monads prove Effects

- Input/Output
  - looks just like SM where state is
    - current open file handles
      - opening a file
        - » adds a file handle
      - closing a file
        - » removes a handle
      - reading/writing
        - » replaces/updates handle

Exceptions

```
type 'a Except = Val 'a
               | Exc string

fun ^ v = Val v


fun >>= f g = case f
                of Val v => g v
                 | Exc s => Exc s
```

Other Effects

- Monads generalize to all computational effects
  - partiality
  - parallelism
  - non-determinism

Costs of Functional Programming

- memory
  - consumption
  - unknown when to deallocate

- solution: automatic memory management
  - garbage collection

Core Idea is Sequencing

- ==> transformational programming
  - first-class functions
    - » abstraction of transformations
  - higher-order functions
    - » better code locality
    - » stronger modularity
  - pattern matching
  - shorthands
    - » comprehensions
    - » monads
  - generics
- ==> automatic memory management

Questions?