Functional Languages and
Higher-Order Functions

Leonidas Fegaras

First-Class Functions

- Data values are first-class if they can
  - be assigned to local variables
  - be components of data structures
  - be passed as arguments to functions
  - be returned from functions
  - be created at run-time
- How functions are treated by programming languages?

language	passed as arguments	returned from functions	nested scope
Java	No	No	No
C	Yes	Yes	No
C++	Yes	Yes	No
Pascal	Yes	No	Yes
Modula-3	Yes	No	Yes
Scheme	Yes	Yes	Yes
ML	Yes	Yes	Yes

Function Types

- A new type constructor
  (T1,T2,...,Tn) -> T0
  Takes n arguments of type T1, T2, ..., Tn and returns a value of type T0
  Unary function: T1 -> T0   Nullary function: () -> T0
- Example:

```
sort ( A: int[], order: (int,int)->boolean ) {
    for (int i = 0; i<A.size; i++)
        for (int j=i+1; j<A.size; j++)
            if (order(A[i],A[j]))
                switch A[i] and A[j];
  }
  boolean leq ( x: int, y: int ) { return x <= y; }
  boolean geq ( x: int, y: int ) { return x >= y; }
  sort(A,leq)
  sort(A,geq)
```


How can you do this in Java?

```
interface Comparison {
  boolean compare ( int x, int y );
}
void sort ( int[] A, Comparison cmp ) {
    for (int i = 0; i<A.length; i++)
        for (int j=i+1; j<A.length; j++)
            if (cmp.compare(A[i],A[j]))
                ...
}
class Leq implements Comparison {
  boolean compare ( int x, int y ) { return x <=y; }
}
sort(A,new Leq());
```


... or better

```
class Comparison {
  abstract boolean compare ( int x, int y );
}

sort(A,new Comparison()
        { boolean compare ( int x, int y ) { return x <=y; } })
```


Nested Functions

- Without nested scopes, a function may be represented as a pointer to its code
- Functional languages (Scheme, ML, Haskell), as well as Pascal and Modula-3, support nested functions
  - They can access variables of the containing lexical scope

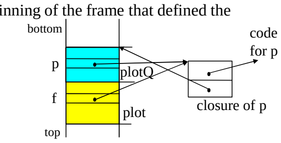
```
plot ( f: (float)->float ) { ... }

plotQ ( a, b, c: float ) {
   p ( x: float ) { return a*x*x + b*x + c; }
   plot(p);
}
```

- Nested functions may access and update free variables from containing scopes
- Representing functions as pointers to code is not good any more

Closures

- Nested functions may need to access variables in previous frames in the stack
- Function values is a closure that consists of
  - a pointer to code
  - an environment (dictionary) for free variables
- Implementation of the environment:
  - It is simply a static link to the beginning of the frame that defined the function

```
plot ( f: (float)->float ) { ... }
plotQ ( a, b, c: float ) {
   p ( x: float ) { return a*x*x + b*x + c; }
   plot(p);
}
```



Run-time stack

What about Returned Functions?

- If the frame of the function that defined the passing function has been popped out from the run-time stack, the static link will be a dangling pointer

```
()->int  make_counter () {
    int count = 0;
    int inc () { return count++; }
    return inc;
}
make_counter()() + make_counter()();
()->int  c = make_counter();
c()+c();
```


Frames in Heap!

- Solution: heap-allocate function frames
  - No need for run-time stack
  - Frames of all lexically enclosing functions are reachable from a closure via static link chains
  - The GC will collect unused frames
- Problem: Frames will make a lot of garbage look reachable

Escape Analysis

- Local variables need to be
  - stored in heap **only if** they can escape
  - accessed after the defining function returns
- It happens **only if**
  - the variable is referenced from within some nested function
  - the nested function is returned or passed to some function that might store it in a data structure
- Variables that do not escape are allocated on a stack frame rather than on heap
- No escaping variable => no heap allocation
- Escape analysis must be global
  - Often approximate (conservative analysis)

Functional Programming Languages

- Programs consist of functions with no side-effects
- Functions are first class values
- Build modular programs using function composition
- No accidental coupling between components
- No assignments, statements, for-loops, while-loops, etc
- Supports higher-level, declarative programming style
- Automatic memory management (garbage collection)
- Emphasis on types and type inference
  - Built-in support for lists and other recursive data types
  - Type inference is like type checking but no type declarations are required
    - Types of variables and expressions can be inferred from context
  - Parametric data types and polymorphic type inference
- Strict vs lazy functional programming languages

Lambda Calculus

- The theoretical foundation of functional languages is lambda calculus
  - Formalized by Church in 1941
  - Minimal in form
  - Turing-complete
- Syntax: if e_1, e_2, and e are expressions in lambda calculus, so are
  - Variable: v
  - Application: $e_1\ e_2$
  - Abstraction: $\lambda v.\ e$
- Bound vs free variables
- Beta reduction:
  - $(\lambda v.\ e_1)\ e_2 \rightarrow e_1[e_2/v]$

    (e_1 but with all free occurrences of v in e_1 replaced by e_2)
  - need to be careful to avoid the variable capturing problem (name clashes)

Church encoding: Integers

- Integers:
  - 0 = λs. λz. z
  - 1 = λs. λz. s z
  - 2 = λs. λz. s s z
  - 6 = λs. λz. s s s s s s z
  - ... they correspond to successor (s) and zero (z)
- Simple arithmetic:
  - add = λn. λm. λs. λz. n s (m s z)

```
add 2 3   = (λn. λm. λs. λz. n s (m s z)) 2 3
          = λs. λz. 2 s (3 s z)
          = λs. λz. (λs. λz. s s z) s ((λs. λz. s s s z) s z)
          = λs. λz. (λs. λz. s s z) s (s s s z)
          = λs. λz. s s s s s z
          = 5
```


Other Types

- Booleans
  - true = λt. λf. t
  - false = λt. λf. f
  - if pred e_1 e_2 = pred e_1 e_2
    - eg, if pred is true, then (λt. λf. t) e_1 e_2 = e_1
- Lists
  - nil = λc. λn. n
  - [2,5,8] = λc. λn. c 2 (c 5 (c 8 n))
  - cons = λx. λr. λc. λn. c x (r c n)
    - cons 2 (cons 5 (cons 8 nil)) = … = λc. λn. c 2 (c 5 (c 8 n))
  - append = λr. λs. λc. λn. r c (s c n)
  - head = λs. s (λx. λr. x) ?
- Pairs
  - pair = λx. λy. λp. p x y
  - first = λs. s (λx. λy. x)

Reductions

- REDucible EXpression (*redex*)
  - an application expression is a redex
  - abstractions and variables are not redexes
- Use beta reduction to reduce
  - (λx. add x x) 5 is reduced to 10
- Normal form = no reductions are
- Reduction is confluent (has the Church-Rosser property)
  - normal forms are unique regardless of the order of reduction
- Weak normal forms (WNF)
  - no redexes outside of abstraction bodies
- Call by value (*eager evaluation*): WNF + leftmost innermost reductions
- Call by name: WNF + leftmost outermost reductions (normal order)
- Call by need (*lazy evaluation*): call by name, but each redex is evaluated at most once
  - terms are represented by graphs and reductions make shared subgraphs

Recursion

- Infinite reduction: (λx. x x) (λx. x x)
  - no normal form; no termination
- A fixpoint combinator Y satisfies:
  - Y f is reduced to f (Y f)
  - Y = (λg. (λx. g (x x)) (λx. g (x x)))
  - Y is always built-in
- Implements recursion
- factorial = Y (λf. λn. if (= n 0) 1 (* n (f (- n 1))))

Second-Order Polymorphic Lambda Calculus

- Types are:
  - Type variable: v
  - Universal quantification: ∀v. t
  - Function: $t_1 \rightarrow t_2$
- Lambda terms are:
  - Variable: v
  - Application: e_1 e_2
  - Abstraction: λv:t. e
  - Type abstraction: Λv. e
  - Type instantiation: e[t]
- Integers
  - int = ∀a. (a → a) → a → a
  - succ = λx:int. Λa. λs:(a → a). λz:a. s (x[a] s z)
  - plus = λx:int. λy:int. x[int] succ y

Type Checking

$$\Gamma \;::=\; \langle\rangle \;\mid\; x : t, \Gamma$$

$$\begin{array}{lcll} t, t_1, t_2 & ::= & v & \text{Type variable} \\ & \mid & \forall v.\, t & \text{Universal quantification} \\ & \mid & t_1 \rightarrow t_2 & \text{Function} \end{array}$$

$$\begin{array}{lcll} e, e_1, e_2 & ::= & x & \text{Variable} \\ & \mid & e_1\ e_2 & \text{Application} \\ & \mid & \lambda x : t.\, e & \text{Abstraction} \\ & \mid & \Lambda v.\, e & \text{Type abstraction} \\ & \mid & e[t] & \text{Type instantiation} \end{array}$$

$$(var) \quad \Gamma, x : t, \Gamma' \vdash x : t$$

$$(abs) \quad \frac{\Gamma, x : t_1 \vdash e : t_2}{\Gamma \vdash (\lambda x : t_1.\, e) : t_1 \rightarrow t_2} \qquad (appl) \quad \frac{\Gamma \vdash e_1 : t_1 \rightarrow t_2 \qquad \Gamma \vdash e_2 : t_1}{\Gamma \vdash (e_1\ e_2) : t_2}$$

$$(poly) \quad \frac{\Gamma \vdash e : t}{\Gamma \vdash (\Lambda v.\, e) : \forall v.\, t} \qquad (inst) \quad \frac{\Gamma \vdash e : \forall v.\, t_1}{\Gamma \vdash (e[t_2]) : [t_2/v]t_1}$$

Functional Languages

- Functional languages = typed lambda calculus + syntactic sugar
- Functional languages support parametric (generic) data types

```
data List a = Nil
            | Cons a (List a)
data Tree a b = Leaf a
              | Node b (Tree a b) (Tree a b)
Cons 1 (Cons 2 Nil)
Cons “a” (Cons “b” Nil)
```

- Lists are built-in Haskell: [1,2,3] = 1:2:3:[]
- Polymorphic functions:

```
append (Cons x r) s    = Cons x (append r s)
append Nil s           = s
```

The type of append is $\forall a.\ (\text{List } a) \to (\text{List } a) \to (\text{List } a)$

- Parametric polymorphism vs ad-hoc polymorphism (overloading)

Type Inference

- Functional languages need type inference rather than type checking
  - $\lambda v{:}t.\ e$ requires type checking
  - $\lambda v.\ e$ requires type inference (need to infer the type of v)
- Type inference is undecidable in general
- Solution: *type schemes* (shallow types):
  - $\forall a_1.\ \forall a_2.\ \ldots\ \forall a_n.\ t$ no other universal quantification in t
  - $(\forall b.\ b \to \text{int}) \to (\forall b.\ b \to \text{int})$ is not shallow
- When a type is missing, then a fresh type variable is used
- Type checking is based on type equality; type inference is based on type unification
  - A type variable can be unified with any type
- Example in Haskell:

  let f = λx. x in (f 5, f “a”)

  λx. x has type $\forall a.\ a \to a$
- Cost of polymorphism: polymorphic values must be *boxed* (pointers to heap)

Higher-Order Functions

- Map a function f over every element in a list

map:: $(a \to b) \to [a] \to [b]$
map f [] = []
map f (a:s) = (f a):(map f s)

e.g. map (+) [1,2,3,4] = [2,3,4,5]

- Replace all cons list constructions with the function c and the nil with the value z

foldr:: $(a \to b \to b) \to b \to [a] \to b$
foldr c z [] = z
foldr c z (a:s) = c a (foldr c z s)

e.g. foldr (+) 0 [1,2,3] = 6
e.g. append x y = foldr (++) y x
e.g. map f x = foldr (λa r. (f a):r) [] x