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?

<table>
<thead>
<tr>
<th>Language</th>
<th>Passed as arguments</th>
<th>Returned from functions</th>
<th>Nested scopes</th>
</tr>
</thead>
<tbody>
<tr>
<td>Java</td>
<td>No</td>
<td>No</td>
<td>No</td>
</tr>
<tr>
<td>C++</td>
<td>Yes</td>
<td>Yes</td>
<td>No</td>
</tr>
<tr>
<td>Pascal</td>
<td>Yes</td>
<td>No</td>
<td>Yes</td>
</tr>
<tr>
<td>Modula-3</td>
<td>Yes</td>
<td>No</td>
<td>Yes</td>
</tr>
<tr>
<td>Scheme</td>
<td>Yes</td>
<td>Yes</td>
<td>Yes</td>
</tr>
<tr>
<td>ML</td>
<td>Yes</td>
<td>Yes</td>
<td>Yes</td>
</tr>
</tbody>
</table>

Function Types

- A new type constructor
  \[(T_1, T_2, ..., T_n) \rightarrow T_0\]
  Takes \(n\) arguments of type \(T_1, T_2, ..., T_n\) and returns a value of type \(T_0\)

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

plotQ (a, b, c: float) \{ 
p (x: float) \{ return a*x^2 + 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();
```

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 e₁, e₂, and e are expressions in lambda calculus, so are
  - Variable: v
  - Application: e₁ e₂
  - Abstraction: λv.e
- Bound vs free variables
- Beta reduction:
  - (λv.e₁) e₂ → e₁[e₂/v]
  - (e₁ but with all free occurrences of v in e₁ replaced by e₂)
  - need to be careful to avoid the variable capturing problem (name clashes)
**Church encoding: Integers**

- Integers:
  - 0 = \( \lambda s. \lambda z. z \)
  - 1 = \( \lambda s. \lambda z. s z \)
  - 2 = \( \lambda s. \lambda z. s s z \)
  - ... they correspond to successor (s) and zero (z)

- Simple arithmetic:
  - \( -a \) = \( \lambda n. \lambda m. \lambda s. \lambda z. n s (m s z) \)

**Other Types**

- Booleans
  - \( \text{true} = \lambda t. \lambda f. t \)
  - \( \text{false} = \lambda t. \lambda f. f \)
  - if pred \( e_1. e_2 = \text{pred} \ e_1 \ ? \ e_2 \)

- Lists
  - nil = \( \lambda c. \lambda n. n \)
  - \([2,5,8] = \lambda c. \lambda n. c 2 (c 5 (c 8 n)) \)

**Reductions**

- REDucible EXression (redex)
  - an application expression is a redex
  - abstractions and variables are not redexes

- Use beta reduction to reduce
  - \( (\lambda x. \text{add} x x) \ 5 \) 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 form (WF)
  - no redexes outside of abstraction bodies

- Call by value (eager evaluation): WNF + leftmost innermost reductions

- 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

**Second-Order Polymorphic Lambda Calculus**

- Types are:
  - Type variable: \( v \)
  - Universal quantification: \( \forall v. t \)
  - Function: \( t_1 \rightarrow t_2 \)

- Lambda terms are:
  - Variable: \( v \)
  - Application: \( e_1 e_2 \)
  - Abstraction: \( \lambda v : t_1. e \)
  - Type abstraction: \( \Lambda v. e \)
  - Type instantiation: \( e[t] \)

- Integers
  - int = \( \forall a. (a \rightarrow a) \rightarrow a \rightarrow a \)
  - succ = \( \lambda x : \text{int}. \lambda a : (a \rightarrow a). a \lambda c. a \ (x[a] \ a \ c) \)
  - plus = \( \lambda x : \text{int}. \lambda y : \text{int}. x [\text{int}] \ x \) succ y

**Recursion**

- Infinite reduction: \( (\lambda x. x x) (\lambda x. x x) \)
  - no normal form; no termination

- A fixpoint combinator \( Y \) satisfies:
  - \( Y \ f \) is reduced to \( f (Y \ f) \)

  \( Y \ = \ (\lambda g. (\lambda x. g (x x)) (\lambda x. g (x x))) \)

  \( Y \) is always built-in

- Implements recursion
  - factorial = \( Y (\lambda f. \lambda n. \text{if} (= n 0) 1 (* n (f (- n 1)))) \)

**Type Checking**

- \( \Gamma \vdash e : t \)

  - Type variable: \( e, e_1, e_2 \) t = \( x \)
  - Type instantiation: \( e[t] \)
  - Universal quantification: \( \forall t. t \rightarrow t \)
  - Function: \( t_1 \rightarrow t_2 \)
  - Application: \( \lambda v. e \)
  - Type abstraction: \( \Lambda v. e \)
  - Type instantiation: \( e[t] \)

- (app)
  - \( \Gamma \vdash e : t_1 \wedge e : t_2 \)

- (bind)
  - \( \Gamma, e : t \vdash e \rightarrow t' \)

- (pair)
  - \( \Gamma, e : t \vdash e : t_1 \times t_2 \)
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 ∀a. (List a) → (List a) → (List a)
- Parametric polymorphism vs ad-hoc polymorphism (overloading)

Type Inference

- Functional languages need type inference rather than type checking
  - λv:t. e requires type checking
  - λv. e requires type inference (need to infer the type of v)
- Type inference is undecidable in general
- Solution: type schemes (shallow types):
  - ∀t1. ∀t2. ... ∀tn. t
  - ( ∀t. t → int ) → ( ∀t. t → 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 ∀a. a → 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 f [] = []
  map f [a] = f a : map f [a]
  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 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