Functional Languages and Higher-Order Functions

Leonidas Fegaras
First-Class Functions

- Values of some type are first-class if
  - They can be assigned to local variables
  - They can be components of data structures
  - Passed as arguments to functions
  - Returned from functions
  - 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 scope</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>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
  
  \((T_1, T_2, \ldots, T_n) \rightarrow T_0\)

  Takes \(n\) arguments of type \(T_1, T_2, \ldots, T_n\) and returns a value of type \(T_0\)

  Unary function: \(T_1 \rightarrow T_0\)  
  Nullary function: \(() \rightarrow T_0\)

- Example:

  ```java
  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?

```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);
```
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

```cpp
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
  - Pointer to code
  - An environment for free variables
- Implementation of environment:
  - It is simply a static link to the beginning of the frame that defined the function

```plaintext
plotQ (a, b, c: float)
   p (x: float) { return a*x*x + b*x + c; }
   { plot(p); }
```
What about Returned Functions?

- If the frame of the function that defined the passing function has been exited, the static link will be a dangling pointer

```c
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
- Frames make a lot of garbage look reachable
Escape Analysis

• Local variables need to be stored on heap if they can escape and be 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 programs by 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)
  – careful to avoid the variable capturing problem (name clashes)
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 z
  - like successor (s) and zero (z)

- Simple arithmetic:
  - add = λn. λm. λs. λz. n s (m s z)

\[
\begin{align*}
\text{add 2 3} &= (λn. λm. λs. λz. n s (m s z)) 2 3 \\
&= λs. λz. 2 s (3 s z) \\
&= λs. λz. (λs. λz. s s z) s ((λs. λz. s s z) s z) \\
&= λs. λz. (λs. λz. s s z) s (s s z) \\
&= λs. λz. s s s s s z \\
&= 5
\end{align*}
\]
Other Types

- Booleans
  - true = λt. λf. t
  - false = λt. λf. f
  - if pred e₁ e₂ = pred e₁ e₂
    - eg, if pred is true, then (λt. λf. t) e₁ e₂ = e₁

- 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 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
- 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 redexes are evaluated at most once
  - terms are represented by graphs and reductions make shared subgraphs
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 = \(Y (\lambda f. \lambda n. \text{if } (= n 0) 1 (* n (f (- n 1))))\)
Second-Order Polymorphic Lambda Calculus

- Types are:
  - Type variable: $v$
  - Universal quantification: $\forall v. \ t$
  - Function: $t_1 \to t_2$

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

- Integers
  - $\text{int} = \forall a. \ (a \to a) \to a \to a$
  - $\text{succ} = \lambda x : \text{int}. \ \Lambda a. \ \lambda s : (a \to a). \ \lambda z : a. \ s \ (x[a] \ s \ z)$
  - $\text{plus} = \lambda x : \text{int}. \ \lambda y : \text{int}. \ x[\text{int}] \ \text{succ} \ y$
Type Checking

\[
\Gamma ::= \emptyset \mid x : t, \Gamma
\]

\[
t, t_1, t_2 ::= v \quad \text{Type variable}
\mid \forall v. \ t \quad \text{Universal quantification}
\mid t_1 \rightarrow t_2 \quad \text{Function}
\]

\[
e, e_1, e_2 ::= x \quad \text{Variable}
\mid e_1 \ e_2 \quad \text{Application}
\mid \lambda x : t. \ e \quad \text{Abstraction}
\mid \Lambda v. \ e \quad \text{Type abstraction}
\mid e[t] \quad \text{Type instantiation}
\]

\[
\begin{align*}
\text{(var)} & \quad \Gamma, x : t, \Gamma' \vdash x : t \\
\text{(abs)} & \quad \frac{\Gamma, x : t_1 \vdash e : t_2}{\Gamma \vdash (\lambda x : t_1. e) : t_1 \rightarrow t_2} \\
\text{(poly)} & \quad \frac{\Gamma \vdash e : t}{\Gamma \vdash (\Lambda v. e) : \forall v. t} \\
\text{(appl)} & \quad \frac{\Gamma \vdash e_1 : t_1 \rightarrow t_2 \quad \Gamma \vdash e_2 : t_1}{\Gamma \vdash (e_1 \ e_2) : t_2} \\
\text{(inst)} & \quad \frac{\Gamma \vdash e : \forall v. t_1}{\Gamma \vdash (e[t_2]) : [t_2/v]t_1}
\end{align*}
\]
Functional Languages

- Functional languages = typed lambda calculus + syntactic sugar
- Functional languages support parametric (generic) data types
  
  \[
  \text{data List } a = \text{Nil} \\
  \quad \mid \text{Cons } a \text{ (List } a) \\
  \]

  \[
  \text{data Tree } a \ b = \text{Leaf } a \\
  \quad \mid \text{Node } b \text{ (Tree } a \ b) \text{ (Tree } a \ b) \\
  \]

  (Cons 1 (Cons 2 Nil))
  (Cons “a” (Cons “b” Nil))

- Polymorphic functions:
  
  append (Cons x r) s = Cons x (append r s)
  append Nil s = s

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

- 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 \( t \)
  - (\( \forall b. b \rightarrow \text{int} \) \( \rightarrow \) (\( \forall b. b \rightarrow \text{int} \)) 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 = \( x \rightarrow x \) in (f 5, f “a”)

  \( x \rightarrow x \) has type \( \forall a. a \rightarrow a \)

- Cost of polymorphism: polymorphic values must be boxed (pointers to heap)