CSE 3302
Programming Languages
Final Review

Leonidas Fegaras
University of Texas at Arlington

Control: Applicative Order Evaluation

- Also known as strict evaluation
- Evaluates the operands first, then applies operators
  - bottom-up evaluation
  - subexpressions are always evaluated, no matter whether they are needed

```
*  
+  
3  4  5  6
```

Normal Order Evaluation

- Also known as lazy evaluation
  - Operation evaluated before the operands are evaluated;
  - Operands evaluated only when necessary

```
int double (int x) { return x+x; }
int square (int x) { return x*x; }
```

Applicative order evaluation:
- square(double(2)) = square(4) = 16
- Normal order evaluation:
- square(double(2)) = double(2)*double(2) = (2+2)*(2+2)

Runtime Environment

- Environment: binding from names to their attributes

```
static(global) area
stack
(unallocated)
heap
```

- automatically-allocated space for local variables; under the control of runtime system
- manually-allocated space; under the control of programmer
**Activation Records for Nested Blocks**

```c
int x; //global
{
  int x, y;
  x = y + 10;
  int i;
  i = x / 2;
}
```

1: nonlocal variable, in the surrounding activation record

**Pass by Value**

- Caller:
  - `... (i);` ...

- Callee:
  - `int f(int a) { ... }`

- This is the most common form
- Replace formal parameters by the values of actual parameters
- Actual parameters: No change
- Formal parameters: Local variables (C, C++, Java, Pascal)

**Activation Records for Procedures**

```c
int x; //global
void B(void) { int i; i = x / 2; }
void A(void) {
  int x, y;
  x = y + 10;
  B();
}
main() {
  A();
  return 0;
}
```

x: global variable in the defining environment

**Pass-by-Value: Pointers**

- C:
  - `void f(int *p) { *p = 0; }`
  - `main() {`
  - `int *q;`
  - `q = (int *) malloc(sizeof(int));`
  - `*q = 1;`
  - `f(q);`
  - `printf("\d\n", q[0]);`
Pass by Reference

- Caller:
  ```
  _
  f(i);
  _
  ```

- Callee:
  ```
  int f(int a){
  ...a...;
  }
  ```

- Formal parameters become alias of actual parameters
- Actual parameters: changed by changes to formal parameters
- Examples:
  - Fortran: the only parameter passing mechanism
  - C++ (reference type, $\$}$/Pascal (var)

Pass by Name

- Caller:
  ```
  _
  f(i);
  _
  ```

- Callee:
  ```
  int f(int a){
  ...a...;
  }
  ```

- The actual parameters are only evaluated when they are needed
- The same parameter can be evaluated multiple times
- They are evaluated in the calling environment
- Essentially equivalent to normal order evaluation
- Example:
  - Algol 60
  - Not adopted by any major languages due to implementation difficulty

Algebraic Specification of ADT

- Syntactic specification (signature, interface):
  the name of the type, the prototype of the operations

- Semantic specification (axioms, implementation):
  required properties of the implementation
  mathematical properties of the operations

They don’t specify:
- data representation
- implementation details

Syntactic Specification

```java
type queue(element) imports boolean
operations:
  createq: queue
  enqueue: queue x element -> queue
  dequeue: queue -> queue
  frontq: queue -> element
  emptyq: queue -> boolean
```

- imports: the definition queue needs boolean
- Parameterized data type (element)
- createq: not a function (viewed as a function with no parameters)
Algebraic Specification

variables: \( q \): queue; \( x \): element

axioms:
- \( \text{empty}(\text{create}(q)) \) = true
- \( \text{empty}(\text{enqueue}(q,x)) \) = false
- \( \text{front}(\text{create}(q)) \) = error
- \( \text{front}(\text{enqueue}(q,x)) \) = if empty(q) then x else front(q)
- \( \text{dequeue}(\text{create}(q)) \) = error
- \( \text{dequeue}(\text{enqueue}(q,x)) \) = if empty(q) then \( q\) else enqueue dequeue(q), x

* error axiom (actually an exception)

Inheritance

```java
public abstract class Shape
{
    private Point center;

    public Shape(Point c)
    { center = c; }

    public abstract double area();
}
```

Abstract Class

- **Abstract classes**
  - Cannot be instantiated
  - Incomplete: subclasses fill in "missing pieces"

- To make a class abstract
  - public abstract class Shape { .. }
  - Contain one or more abstract methods
    - No implementation
    - E.g., public abstract void draw();

- Subclasses:
  - fill in "missing pieces" (i.e., overriding the abstract methods)
  - E.g., Circle, Triangle, Rectangle extends Shape
    - Each must implement draw it is concrete

```java
Test.java
package p2;
import p1.Base;

public class Test {
    public static void main(String[] args) {
        Base b = new Base();
        b.a = 1;
        // Compilation error
        b.b = 2;
        // Compilation error
        b.b = 3;
        // OK
        b.b = 4;
        // Compilation error
        Derived d = new Derived();
        d.f();
        // OK
        d.f();
        // Compilation error
    }
}
```

```java
Derived.java
package p2;
import p1.Base;

public class Derived extends Base
{
    void k()
    { f();
    }
    void k()
    { g();
    }
}
```
Example of Abstract Class

```java
public class Base {
    public String m1() {
        return "Base.m1";
    }
}

public abstract class Derived extends Base {
    public abstract String m2();
}

public class Derived2 extends Derived {
    public String m2() {
        return "Derived2.m2";
    }
}
```

Interface

- No method implementations
- Java doesn’t allow multiple inheritance:
  - E.g., … C extends A, B …
- Instead, use **Interface**
  - E.g., … C implements I₁, I₂ …
- One class may implement multiple interfaces
  - Must implement all functions in those interfaces if class is concrete

Interface Example

```java
public class Base {
    public String m1() {
        return "Base.m1";
    }
}

interface Interface1 { String m2(); }

interface Interface2 { String m3(); }

public class Derived extends Base implements Interface1, Interface2 {
    public String m2() {
        return "Derived.m2";
    }

    public String m3() {
        return "Derived.m3";
    }
}
```

Interfaces vs. Abstract Classes

- A class may implement several interfaces
- An interface cannot provide any code at all
- Static final constants only
- A class may extend only one abstract class
- An abstract class can provide partial code
- Both instance and static constants are possible
Haskell

Constructing lists:

- The Empty List \([\ ]\)
- The "Cons" (:) Constructor
  \[ ? 3 : [3,4,5] \\
  [3, 3, 4, 5] \]
- The Dot Dot notation
  \[ ? [1 .. 4] \\
  [1, 2, 3, 4] \]
- The Comprehension Notation
  \[ ? [x + 1 | x <- [2..4]] \\
  [3, 4, 5] \]
  \[ ? [ (x,y) | x <- [1..2], y <- [3,5,7]] \\
  [(1,3), (1,5), (1,7), (2,3), (2,5), (2,7)] \]
  \[ ? [ x * 2 | x <- [1..10], even x] \\
  [4, 8, 12, 16, 20] \]

Rules for Patterns

- All the patterns (on the left) should have compatible types
- The cases should (but are not required to) be exhaustive
- There should be no ambiguity as to which case applies
- Ordering fixes ambiguity if there is any (the first match is chosen)

- A Pattern is:
  - A variable \( x \)
  - A constructor applied to patterns \( x:xs \) or \( \text{Branch}(x,y,z) \)
  - A constant \( 3 \) or \([\ ]\)
  - A tuple of patterns \( (x,3,y:ys) \)

Ways to Create Functions

- By defining: `pluseone x = x+1`
  \[ ? \text{pluseone } 3 \\
  4 \]
- By operator section
  \[ ? (3+) 5 \\
  8 \]
  \[ ? \text{map } (3+) [2,3,4] \\
  [5, 6, 7] \]
- By lambda expression
  \[ ? (\lambda x \rightarrow x+2) 5 \\
  7 \]
  \[ ? \text{map } (\lambda x \rightarrow x*2) [2,3,4] \\
  [4, 6, 8] \]

Creating Functions (cont.)

- By currying (partial application)
  \[ ? \text{plus } 3 \\
  \text{plus } 3 \]
  \[ ? :\text{type } (\text{plus } 3) \\
  \text{plus } 3 :: \text{Int} \rightarrow \text{Int} \]
  \[ ? \text{map } (\text{plus } 3) [3,4] \\
  [6, 7] \]
- By composition
  \[ ? \text{map } (\text{head . tail}) [[2,3,4],[4,5,6]] \\
  [3, 5] \]
**Polymorphism**

- Consider: `tag x = (1,x)`
  ```
  tag :: a -> (Int,a)
  ```
- Other functions have types like this consider `(++)`
  ```
  (+) :: [a] -> [a] -> [a]
  ```
- What are some other polymorphic functions and their types?
  - `id ::`
  - `reverse ::`
  - `head ::`
  - `tail ::`
  - `() ::`
  - `split ::`

**Defining New Datatypes**

- Kinds of datatypes
  - enumerated types
  - records (or products or struct)
  - variant records (or sums)
  - pointer types
  - arrays
- Haskell’s `data` declaration provides many of these kinds of types in a uniform way which abstracts from their implementation details
- The `data` declaration defines new functions and constants, which provide an abstract interface to the newly defined type

**Shape types from the Text**

```haskell
data Shape = Rectangle Float Float
           | Ellipse Float Float
           | RtTriangle Float Float
           | Polygon (Float,Float)

deriving Show
```

**Recursive Data Types**

- Lists:
  ```
  data List a = Nil
              | Cons a (List a)
  ```
  Cons 1 (Cons 2 Nil)
  Cons “a” (Cons “b” Nil)
- Lists are already defined in Haskell:
  ```
  Data [a] = [ ]
          | () a [a]
  ```
  `[1,2,3] = 1:2:3:[]`
- Trees:
  ```
  data Tree a b = Leaf a
                 | Node b (Tree a b) (Tree a b)
  ```
Higher-Order Functions

* Map a function f over every element in a list
  map:: (a → b) → [a] → [b]
  map f [ ] = [ ]
  map f (a:s) = (f a):(map f s)

  e.g. map (x → x+1) [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 → b → b) → b → [a] → 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 (a r → (f a)r) [ ] x