Functional programming in mainstream languages

CS 3110 Lecture 26
Andrew Myers
Some “functional” language features

- Higher-order functions
- Freedom from side effects
- Polymorphism
- Pattern matching
- Modules (structures, signatures, and functors)

This lecture: how to program “functionally” in “mainstream” languages.
Higher-order fun 1

- Higher-order functions can be encoded using objects (in Java, C#, C++)

```java
class C {
    T doit(T x) {
        return e;
    }
}
...
C f = new C();
... f.doit(...)...
... g(f)...
```

```plaintext
let f =
    fun (x:T) :T' -> e
in
  ... f(...) ... First-order use
  ... g(f) ... Higher-order use
```
Higher order fun 2

Higher-order functions can be encoded in C by passing free variables as an extra argument.

(Ugh)

```plaintext
let y = 137 in
let f =
  fun (x:T) :T' -> e
in
... f(...) ...
... g(f) ...
```

```c
int y = 137;

struct f_env { int y; }
int f(T x, f_env *env) {
    return e;
}
struct f_env f_env1;
f_env1.y = y;
... f(x, &f_env1); ...

struct T_T'_closure {
    T' (*)(T, void *) funptr;
    void *env;
};
struct T_T'_closure f_closure;
fClosure.funptr = (...f;
f_closure.env = (void *)&f_env1;
... g(&f_closure);
```

Type system doesn't understand

use env->y
Iterators

- A major use of higher-order functions: iterators

 Three functional iterator patterns:

- **map**: iteration with each element handled independently
  - Use and implementation are both easy. Iteration must complete.
  - In imperative setting, can use side effect instead, like
    ```
    Array.iter: (\'a->unit) -> \'a array -> unit
    ```

- **fold**: iteration with state carried across elements
  - Easy to implement and use (once you’re used to it) but iteration must complete.
  - Threading state via fold is essentially the same as imperatively updating loop variables, but with updated state explicit rather than implicit.

- **streams**: “don’t call us, we’ll call you”
  - Nice to use; painful to implement imperatively or functionally.
  - Example: Java Iterator interface.
Fold vs. iterators

Fold abstractions are pretty easy to write:

type 'a tree = Empty
  | Node of ('a tree) * 'a * ('a tree)

let rec fold f i t =
  match t with Empty -> i
  | Node (left, val, right) ->
    let lacc = fold f i left in
    let vacc = f lfold val in
    let racc = fold f vfold right in
    racc
Implementing iterators in Java

```java
class TreeIterator implements Iterator {
    Iterator subiterator;
    boolean hasNext;
    Object current;
    int state;
    // states:
    //   1. Iterating through left child.
    //   2. Just yielded current node value
    //   3. Iterating through right child

    TreeIterator() {
        subiterator = RBTree.this.left.iterator();
        state = 1;
        preloadNext();
    }

    public boolean hasNext() {
        return hasNext;
    }

    public Object next() {
        if (!hasNext) throw new NoSuchElementException();
        Object ret = current;
        preloadNext();
        return ret;
    }
}

private void preloadNext() {
    loop: while (true) {
        switch (state) {
        case 1:
        case 3:
            hasNext = true;
            if (subiterator.hasNext()) {
                current = subiterator.next();
                return;
            } else {
                hasNext = false;
                return;
            }
        case 2:
            subiterator = RBTree.right.iterator();
            state = 3;
            continue loop;
        }
    }
```

- Iterators (and streams) are painful to implement.
- Result: Java programmers don’t provide iteration abstractions.
Coroutine iterators (C# 2.0)

- Best of both worlds: easy to use and to implement. Loop body and iterator are coroutines.

```csharp
class Tree {
    Tree left, right;
    Elem elem;
    public IEnumerator<Elem> elements() {
        foreach (Elem e in left.elements()) {
            yield return e;
        }
        yield return elem;
        foreach (Elem e in right.elements()) {
            yield return e;
        }
    }
}
```
Avoiding side effects

- Keep side effects local to methods
- Methods are only creators/observers/mutators
  - Update object/struct fields only in constructors/initializers
  - Final fields prevent side effects

```java
class Nat {
    final int num, den;
    Nat(int n, int d) {
        int g = gcd(n, d);
        num = n/g;
        den = d/g;
    }
    ...
}
```

```c
struct nat {
    int num, den;
};
struct nat *
create_nat(int n, int d) {
    struct nat *ret=(struct nat *)
        malloc(sizeof(struct nat));
    int g = gcd(n,d);
    ret->num = n/g;
    ret->den = n/g;
    return ret;
}
```
Polymorphism and parameterized types

Java/C#/C++ have some support.

let f (x:'a array)
    (print_x:'a -> unit) = ...
    print_x(x.(i)) ...

type 'a pair = 'a * 'a ...
let z: foo pair = ...

Warning: may fail.
C++ doesn’t restrict how A is used in Pair. Modularity failure.

public<A> void f(A[] x, Printer<A> p) {
    ... p.doit(x[i]);
}
class Pair<A> {
    A left; A right;
}
Pair<Foo> z = ...

template<class A>
    void f(A x[], void (*print_x)(A)) {
        ... print_x(x[i])...
    }
template<class A> class Pair {
    A left; A right;
}
Pair<Foo> z = ...
Polymorphism in C

- C option 1: use void * and lots of run-time casts.
  
  ```c
  f(void *x[], void (*print_x)(void *)) {
  ...
  }
  ```

  (void * is a bit like “Object”)

- C option 2: use preprocessor to macro-expand code
  with desired type (really what C++ does)

```c
#define A_TYPE int
#include "pair.t"
#undef A

f_int(x, p);

f_##A(x[],
     void (*print_x)(A));
```
Modular programming

- Originally an OO feature...
  - C++, Java, C# : classes (packages) are modules

- Classes have AF and RI -- document!
  - Fields should be private.
  - Packages may have invariants too.

- Java: public interface is the HTML from javadoc.
  - Javadoc automatically generates interface descriptions from public methods and properly formatted comments. Supports clauses corresponding to Returns/Requires/Effects.
  - Pro: don’t have to write signature twice.
  - Con: implementer can accidentally change the contract!
  - Clients should be able to use code by only looking at javadoc web pages.
Modular programming in C

- C has header files (foo.h) and source files (foo.c)
- Source files use #include (like CL) to read in header files -- no real modularity support.
- Usage pattern: .h is the interface, .c is the implementation.

⇒ Headers should contain specs. Clients should not need to read source files (.c)
⇒ Headers should not declare representations.

typedef struct nat_s *Nat; // an abstract type!
Nat create_nat(int num, int den);
Pattern matching

- No real pattern matching in mainstream languages.

- Java, C#: use `instanceof` to figure out which class an object belongs to:

  ```java
  if (o instanceof Foo) {
    Foo asFoo = (Foo)o;
    int x = asFoo.x;
    int y = asFoo.y;
    ...
  } else if (o instanceof Bar) {
    Bar asBar = (Bar)o;
    ...
  }
  ```

- C: use tagged unions (awkward **and** not type-safe)

  ```c
  struct variant_s {
    int tag;
    union {
      struct foo f; // if tag==1
      struct bar b; // if tag==2
    }
  }
  ```