CSE399: Advanced Programming

Handout 23

Recap

Where We've Been

- recursive programming over lists, trees, etc.
- high-performance persistent data structures
  - Okasaki red-black trees
- higher-order programming ("functions as data")
  - continuation-passing style
  - pure lambda-calculus
- combinator libraries for parsing, XML transformation, pretty printing
- functional animation (`Behavior`s as values)
- monadic approaches to computational effects
  - e.g., monadic treatment of session state in WASH
- advanced aspects of type systems
  - polymorphism
  - type classes and qualified types
  - type inference
- Modularity and abstraction
- Testing (randomized and unit-testing)

Where We Haven't Been

There are many more topics that I wish we'd had time to cover...

- functional reactive programming
- concurrent functional programming
- monad transformers
- arrows
- advanced module systems (ML-style "functional programming with modules")
- more on "programming as writing"
- much, much more on abstraction and modularity

Functional Programming in the Real World

Back to the "Real World"

Of course, much as we might enjoy programming in Haskell, the fact is that most of the world uses other languages and other programming paradigms (e.g., OOP).

Back to the "Real World"

Of course, much as we might enjoy programming in Haskell, the fact is that most of the world uses other languages and other programming paradigms (e.g., OOP).

Q: How much of what we've seen this semester is transferrable to other settings?

Back to the "Real World"

Of course, much as we might enjoy programming in Haskell, the fact is that most of the world uses other languages and other programming paradigms (e.g., OOP).

Q: How much of what we've seen this semester is transferrable to other settings?

A: Not everything, but a lot...

Doing Without Assignment

We've gotten through a whole semester without using a single assignment statement!

Doing Without Assignment

We've gotten through a whole semester without using a single assignment statement!

There are lots of real-world settings that rely on purely functional programming

- Ericsson's flagship AXD301 ATM switch is implemented in Erlang, a purely functional distributed language
- Google's sawzall (designed by Unix luminary Rob Pike) is a purely functional language that allows operations such as `map` and `fold` over the entire contents of the web

Living with Assignment

- In Java, it is possible to program in a purely functional style
  - Cf. FeatherWeight Java
- However, in practice it is difficult; in particular, many libraries encourage an imperative style
  - e.g., Iterators, etc.
- However, keeping assignments to a minimum and making the scope of mutable variables as local as possible will make most programs much more modular and robust
  - The mostly functional style
- Particularly important in concurrent programs, where every piece of mutable state is an opportunity for race conditions and other nastiness.

Polymorphism

"Vanilla Java" provides only subtype polymorphism. Using libraries for generic structures such as lists involves awkward (and statically unsafe) downcasts.

```
interface Collection {
  public void add (Object x);
  public Iterator iterator ();
}
interface Iterator {
  public Object next ();
  public boolean hasNext ();
}

...

LinkedList ys = new LinkedList();
ys.add("zero"); ys.add("one");
String y = (String)ys.iterator().next();
```

Polymorphism

However, Java 1.5 (and new releases of .NET) include support for generics that are in many ways very close to the parametric polymorphism of Haskell.

```
interface Collection<A> {
  public void add(A x);
  public Iterator<A> iterator();
}
interface Iterator<A> {
  public A next();
  public boolean hasNext();
}

...

LinkedList<String> ys = new LinkedList<String>();
ys.add("zero"); ys.add("one");
String y = ys.iterator().next();
```

Abstract Data Types

Java does not provide modules as such. However, one of the most important idioms involving modules, abstract data types, can often be simulated using Java interfaces.

```
-- Haskell interface:
module Stack ( T, push, pop, empty ) where

data T a = EmptyStk | Stk a (T a)
empty :: T a
push  :: a -> T a -> T a
pop   :: T a -> T a
```

```
-- Hasekll implementation:

push x s = Stk x s
pop (Stk _ s) = s
empty = EmptyStk
```

```
// Java interface:
interface Stack<A>
{
    A push(A item);
    A pop();
    A peek();
    boolean empty();
}
```

```
// Java implementation:
//
// (too hideous to show :-)
```

Higher-Order Programming in Java

Insight: "object" $\sim$ "function with multiple entry points"

I.e., objects can be used (often in somewhat clunky and verbose ways) to approximate many of the programming idioms we are used to in Haskell.

- inner classes (lexical closures)
- inheritance (the way it is used in lots of libraries)
- Subject/Observer pattern (and generalizations such as Model-View-Controller)
- see any design patterns book for many more examples

Subject/Observer

```
/* interface Subject is implemented by classes that
   encapsulate data that may change */
public interface Subject {
  public void register(Observer obs);
}

/* interface Observer is implemented by classes that
   need to know about changes to the data */
public interface Observer {
  public void notify(String msg);
}
```

E.g., a graphical widget such as a slider could implement the `Subject` interface, allowing anyone who is interested to register callbacks in the form of `Observer` objects.

Caveat: Tail Calls

Unfortunately, higher-order programming idioms cannot be used with complete freedom in Java, because tail calls are not guaranteed to be implemented as jumps by most compilers (e.g., javac).

Why?

1. compiler writers are troglodytes
2. tricky interactions with stack inspection

Caveat: Tail Calls

Unfortunately, higher-order programming idioms cannot be used with complete freedom in Java, because tail calls are not guaranteed to be implemented as jumps by most compilers (e.g., javac).

Why?

1. compiler writers are troglodytes
2. tricky interactions with stack inspection

However, all is not lost:

- in many situations, stack depth will still be bounded
- more modern compilers (e.g., .NET) do a better job
- there are techniques (e.g., trampolining) for getting around these compiler limitations

Datatypes and Pattern Matching

Haskell's mechanisms for datatype definitions and compact pattern matching allow complex symbolic manipulations to be written very compactly and readably.

This makes Haskell (and OCaml, Standard ML, etc.) excellent tools for certain tasks.

- E.g., you'd be crazy to write a serious compiler in a language without pattern matching!

The Visitor Pattern

Data types and pattern matching can be approximated (unfortunately in a somewhat verbose way) in Java.

Cf. the so-called visitor pattern.

What's Left

A few important aspects of Haskell are, sadly, hard to approximate in other languages.

- Type classes (Java's overloading is a poor substitute)
- Explicit encapsulation of computational effects (monads)
  - This is the feature of Haskell that makes it possible to define "domain-specific imperative languages"
  - E.g., it is what ensures that WASH can capture all IO actions in its session log

Fortunately, both of these ideas are getting a lot of interest in other parts of the programming languges community, so we can hope that they'll show up more widely at some point.

- cf. garbage collection, type safety, polymorphism

Closing Thoughts

- Programming is writing
- Functions are data
- Abstraction is good
- The type checker is your friend

Have a great summer!