CS 222: Programming Languages

Introduction to Functional Programming

and

Haskell

Functional Programming

- The Functional Programming Paradigm is one of the major programming paradigms.
  - Functional Programming is a type of declarative programming paradigm – you describe the logic of the computation, not the flow of control
  - Also known as *applicative programming*
- Idea: everything is a function
- Based on sound theoretical frameworks (e.g., **λ**-calculus)
- Examples of FP languages
  - First FP language: Lisp
  - Other important FPs: ML, Haskell, Miranda, Scheme, Logo

Imperative versus Functional

- The design of the imperative languages is based directly on the von Neumann architecture
  - Efficiency is the primary concern, rather than the suitability of the language for software development
- The design of the functional languages is based on mathematical functions
  - A solid theoretical basis that is also closer to the user, but relatively unconcerned with the architecture of the machines on which programs will run

Characteristics of Pure FPLs

Pure FP languages tend to

- Have no side-effects
- Have no assignment statements
- Often have no variables!
- Be built on a small, concise framework
- Have a simple, uniform syntax
- Be implemented via interpreters rather than compilers
- Be mathematically easier to handle

Imperative versus Functional

- Summing integers from 1 to 10 in Java

```
total = 0;
for (i = 1; i ≤ 10; ++i)
    total = total+i;
```

The computation method is *variable assignment* with *side effects* (change *state*)

- Summing integers from 1 to 10 in Scheme (Lisp)

```
(define sum (lambda (l)
    (if (null? l) 0
    (+ (car l) (sum (cdr l))))))
```

The computation method is *function application*

- Summing integers from 1 to 10 in Haskell

```
sum [1..10]
```

Importance of FP

- In their pure form, FPLs dispense with the notion of assignment (claim: it's easier)

- FPLs encourage thinking at higher levels of abstraction
  - support modifying and combining existing programs
  - thus, FPLs encourage programmers to work in units larger than statements of conventional languages: "programming in the large"

- FPLs provide a paradigm for parallel computing
  - absence of assignment (or single assignment)
  - independence of evaluation order

Lisp

- Defined by John McCarthy ~1958 as a language for AI.
- Originally, LISP was a typeless language with only two data types: atom and list
- LISP's lists are stored internally as single-linked lists
- Lambda notation was used to specify functions
- Function definitions, function applications, and data all have the same form

  If the list (A B C) is interpreted as data it is a simple list of three atoms, A, B, and C but if interpreted as a function application, it means that the function named A is applied to the two parameters, B and C

Scheme

- Mid 70's: Sussman and Steele (MIT) defined Scheme as a new LISP-like Language

- Goal: return Lisp to its simpler roots and incorporate ideas which had been developed in the PL community since 1960
  - Uses only static scoping
  - Treat functions as *first-class objects* that can be the values of expressions & elements of lists, assigned to variables and passed as parameters
  - Includes the ability to create and manipulate *closures* and *continuations*
    - A *closure* is a data structure that holds an expression & an environment of variable bindings in which it is to be evaluated. It is used to represent unevaluated expressions when implementing FPLs with *lazy evaluation*
    - A *continuation* is a data structure which represents "the rest of a computation"

- Scheme has mostly been used for teaching programming concepts whereas Common Lisp is widely used as a practical language

ML (Meta Language)

- ML is a strict, static-scoped functional language with a Pascal-like syntax that was defined by Robin Milner et. al. in 1973
- It was the first language to include statically checked polymorphic typing
  - Uses type declarations, but also does type inferencing to determine the types of undeclared variables
  - Strongly typed (whereas Scheme is essentially typeless) and has no type coercions
- Includes exception handling and a module facility for implementing abstract data types, and garbage collection
- Most common dialect is Standard ML (SML)

Haskell

- Similar to ML (syntax, static scoped, strongly typed, type inferencing)
- Different from ML and most other FPLs in that it is purely functional -- no variables, no assignment statements, and no side effects
- Some key features:
  - Uses lazy evaluation (evaluate no subexpression until the value is needed)
  - Has "list comprehensions" to deal with infinite lists

Some FP Concepts

- A number of interesting programming language concepts have arisen, including:
  - Curried functions
  - Type inferencing
  - Polymorphic functions
  - Higher-order functions
  - Functional abstraction
  - Lazy evaluation

- **For next class – look up and define each of the bulleted items in your own words. Submit to Moodle**