CSc 520

Principles of Programming Languages

24: Functional Programming — Conclusion

Christian Collberg

collberg@cs.arizona.edu

Department of Computer Science

University of Arizona

Functional Programming Languages

Functional Programming

In contrast to procedural languages, functional programs don't concern themselves with state and memory locations. Instead, they work exclusively with values, and expressions and functions which compute values.

- Functional programming is not tied to the von Neumann machine.
- It is not necessary to know anything about the underlying hardware when writing a functional program, the way you do when writing an imperative program.
- Functional programs are more declarative than procedural ones; i.e. they describe what is to be computed rather than how it should be computed.

Functional Languages

Common characteristics of functional programming languages:

1. Simple and concise syntax and semantics.
2. Repetition is expressed as recursion rather than iteration.
3. Functions are first class objects. I.e. functions can be manipulated just as easily as integers, floats, etc. in other languages.
4. Data as functions. I.e. we can build a function on the fly and then execute it. (Some languages).

Functional Languages...

5. Higher-order functions. I.e. functions can take functions as arguments and return functions as results.
6. Lazy evaluation. Expressions are evaluated only when needed. This allows us to build infinite data structures, where only the parts we need are actually constructed. (Some languages).
7. Garbage Collection. Dynamic memory that is no longer needed is automatically reclaimed by the system. GC is also available in some imperative languages (Modula-3, Eiffel) but not in others (C, C++, Pascal).

Functional Languages...

8. Polymorphic types. Functions can work on data of different types. (Some languages).
9. Functional programs can be more easily manipulated mathematically than procedural programs.

Pure vs. Impure FPL

- Some functional languages are pure, i.e. they contain no imperative features at all. Examples: Haskell, Miranda, Gofer.
- Impure languages may have assignment-statements, goto:s, while-loops, etc. Examples: LISP, ML, Scheme.

Scheme

Scheme

- Functions and data share the same representation: S-Expressions.
- Scheme is an impure functional language.
- I.e., Scheme has imperative features.
- I.e., in Scheme it is possible to program with side-effects.
- S-expressions are constructed using dotted pairs.
- Scheme is homoiconic, self-representing, i.e. programs and data are both represented the same (as S-expressions).

Scheme — Evaluation Order

- To evaluate an expression `(op arg1 arg2 arg3)` we first evaluate the arguments, then apply the operator op to the resulting values. This is known as applicative-order evaluation.
- This is not the only possible order of evaluation
- In normal-order evaluation parameters to a function are always passed unevaluated.
- Both applicative-order and normal-order evaluation can sometimes lead to extra work.
- Some special forms (`cond`, `if`, etc) must use normal order since they need to consume their arguments unevaluated.t

Scheme — Metacircular Interpreter

- One way to define the semantics of a language (the effects that programs written in the language will have), is to write a metacircular interpreter.
- I.e, we define the language by writing an interpreter for it, in the language itself.
- A metacircular interpreter for Scheme consists of two mutually recursive functions, `Eval` and `Apply`.

Scheme — Lists

- Lists are heterogeneous, they can contain elements of different types, including other lists.
- `(equal? L1 L2)` does a structural comparison of two lists.
- `(eqv? L1 L2)` does a "pointer comparison".
- This is sometimes referred to as deep equivalence vs. shallow equivalence.

```
> '(1 a "hello")
(1 a "hello")
> (eqv? '(a b c) '(a b c))
false
> (equal? '(a b c) '(a b c))
true
```


Scheme — Typing

- Unlike languages like Java and C which are statically typed (we describe in the program text what type each variable is) Scheme is dynamically typed. We can test at runtime what particular type of number an atom is:
  - `(complex? arg)`, `(real? arg)`
  - `(rational? arg)`, `(integer? arg)`

Scheme — Higher-Order Functions

- A function is higher-order if
  1. it takes another function as an argument, or
  2. it returns a function as its result.
- Functional programs make extensive use of higher-order functions to make programs smaller and more elegant.
- We use higher-order functions to encapsulate common patterns of computation.

Haskell

What is Haskell?

- Haskell is statically typed (the signature of all functions and the types of all variables are known prior to execution);
- Haskell uses lazy rather than eager evaluation (expressions are only evaluated when needed);
- Haskell uses type inference to assign types to expressions, freeing the programmer from having to give explicit types;
- Haskell is pure (it has no side-effects).

Haskell — Lazy evaluation

- No expression is evaluated until its value is needed.
- No shared expression is evaluated more than once; if the expression is ever evaluated then the result is shared between all those places in which it is used.
- No shared expression should be evaluated more than once.

Haskell — Infinite data structures

- Lazy evaluation makes it possible for functions in Haskell to manipulate 'infinite' data structures.
- The advantage of lazy evaluation is that it allows us to construct infinite objects piece by piece as necessary
- Consider the following function which can be used to produce infinite lists of integer values:

```
countFrom n = n : countFrom (n+1)
? countFrom 1
[1, 2, 3, 4, 5, 6, 7, 8,^C{Interrupted!}]
(53 reductions, 160 cells)
?
```


Haskell — Currying

- Haskell only supports one-argument functions. Multi-argument functions are constructed by successive application of arguments, one at a time.
- Currying is the preferred way of constructing multi-argument functions.
- The main advantage of currying is that it allows us to define specialized versions of an existing function.
- A function is specialized by supplying values for one or more (but not all) of its arguments.

Referential Transparency

Referential Transparency

- The most important concept of functional programming is referential transparency.
- RT means that the value of a particular expression (or sub-expression) is always the same, regardless of where it occurs.
- RT makes functional programs easier to reason about mathematically.
- Pure functional programming languages are referentially transparent.

Referential Transparency...

- We can evaluate it by substitution. I.e. we can replace a function application by the function definition itself.
- Expressions and sub-expressions always have the same value, regardless of the environment in which they're evaluated.
- The order in which sub-expressions are evaluated doesn't effect the final result.
- Functions have no side-effects.
- There are no global variables.

Side Effects — Bad

- Programs with side effects are hard to read and understand.
- Referential transparency —expressions without side-effects can be executed in any order.
- equational reasoning —if two expressions are ever the same, they are always the same.

Side Effects — Good

- Interacting with the real world (file IO, terminal IO, GUI, networking, etc) doesn't seem to fit in well in the functional paradigm.
- Since, in these cases, we are actually "changing the state of the world", side-effect free programming is problematic.
- Haskell uses monads to sequence IO operations. See Scott, pp. 607-609.
- A monad is an Abstract Data Type that supports sequencing.

Trivial Update Problem

- In pure functional languages variables never change.
- If we want to change the second element of a list `[1,2,3]` to 4, we have to create a new list, copying the elements from the original list.
- If we want to sort a list in a functional language we have to create new lists, rather than sorting in-place, which is more efficient.
- Adding two matrices $A \leftarrow A + B$ will create a new matrix, even if we're throwing away A, and could do the addition in-place.
- Similarly, how do we construct an array of 0s without copying the entire array for every new element?

Sisal

- Sisal is a functional language intended to be used for high-performance codes (scientific programming, think FORTRAN).
- The Sisal compiler tries to verify that an updated array will never be used again, if so, a copy need not be made.
- The Sisal compiler can remove 99-100% of all unnecessary copy operations this way.

http://tamanoir.ece.uci.edu/projects/sisal/sisaltutorial/Tutorial.html

Sisal. . .

```
type OneDim = array [ real ];
type TwoDim = array [ OneDim ];

function generate( n : integer returns TwoDim, TwoDim )
   for i in 1, n cross j in 1, n
   returns array of real(i)/real(j)
           array of real(i)*real(j)
   end for
end function % generate
```


Sisal. . .

```
function doit( n : integer; A, B : TwoDim returns TwoDim )
   for i in 1, n cross
       j in 1, n
      c := for k in 1, n
              t := A[i,k] * B[k,j]
            returns value of sum t
            end for
   returns array of c
   end for
end function % doit

function main(  n : integer returns TwoDim )
let A, B := generate( n )
in  doit( n, A, B )
end let
end function % main
```


Sisal. . .

- The Sisal compiler will automatically parallelize the code on the previous slide.
- Although the code looks imperative, it is actually functional. The compiler makes the necessary transformations of the loops into tail-recursion.

Lambda Calculus

Lambda Calculus

- Branch of mathematical logic. Provides a foundation for mathematics. Describes —like Turing machines —that which can be effectively computed.
- In contrast to Turing machines, lambda calculus does not care about any underlying “hardware” but rather uses simple syntactic transformation rules to define computations.
- A theory of functions where functions are manipulated in a purely syntactic way.
- Lambda calculus is the theoretical foundation of functional programming languages.

Lambda Calculus — Reductions

- To evaluate a lambda expression we reduce it until we can apply no more reduction rules. There are four principal reductions that we use:
  1. α-reduction —variable renaming to avoid name clashes in β-reductions.
  2. β-reduction —function application.
  3. η-reduction —formula simplification.
  4. δ-reduction —evaluation of predefined constants and functions.

Lambda Calculus — Termination

- **Question:**

  Can every lambda expression be reduced to a normal form?
- **Answer:** No.

$$((\lambda x.(x\ x))\ (\lambda x.(x\ x)))$$

- Lambda calculus contains non-terminating reductions.

Lambda Calculus — Paths

- **Question:**
  Is there more than one way to reduce a lambda expression?
- **Answer:** Yes.

Lambda Calculus — Application Order

- The leftmost redex is that redex whose λ is textually to the left of all other redexes within the expression.
- An outermost redex is defined to be a redex which is not contained within any other redex.
- An innermost redex is defined to be a redex which contains no other redex.
- A normal order reduction always reduces the leftmost outermost β-redex (or δ-redex) first.
- A applicative order reduction always reduces the leftmost innermost β-redex (or δ-redex) first.

Church-Rosser Theorem

- **Question:**
  If there is more than one reduction strategy, does each one lead to the same normal form expression?
- **Answer:** Yes, if a lambda expression is in normal form, it is unique, except for changes in bound variables.

Church-Rosser Theorem…

- **Theorem:**
  For any lambda expressions E, F and G, if $E \stackrel{*}{\Rightarrow} F$ and $E \stackrel{*}{\Rightarrow} G$, there is a lambda expression Z such that $F \stackrel{*}{\Rightarrow} Z$ and $G \stackrel{*}{\Rightarrow} Z$.
- **Corollary:**
  For any lambda expressions E, M and N, if $E \stackrel{*}{\Rightarrow} M$ and $E \stackrel{*}{\Rightarrow} N$, where M and N are in normal form, M and N are variants of each other (except for changes in variables, using α-reductions).

Church-Rosser Theorem II

- **Question:**

  Is there a reduction strategy that will guarantee that a normal form expression will be produced, if one exists?

- **Theorem:**

  For any lambda expressions E and N, if $E \stackrel{*}{\Rightarrow} N$ where N is in normal form, there is a normal order reduction from E to N.

- **Answer:** Yes, normal order reduction will produce a normal form lambda expression, if on exists.

Church's Theses

The effectively computable functions on the positive integers are precisely those functions definable in the pure lambda calculus.

- Turing Machines and lambda calculus are equivalent.
- Since it's not possible to determine whether a Turing Machine will terminate, it's not possible to determine whether a normal order reduction will terminate.
- Pure lambda calculus has no constant —everything is a function.
- Data structures (lists), numbers, booleans, control structures (if-expressions, recursion) can all be constructed within a pure lambda calculus.

Readings and References

- Read Scott, pp. 622-623.