CSE583:
Programming Languages

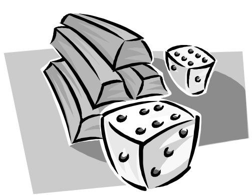
David Notkin
18 January 2000
notkin@cs.washington.edu
http://www.cs.washington.edu/education/courses/583

Functional programming: two weeks

Last Week's Lecture!

- ***Scheme***
  - ***Gives a strong, language-based foundation for functional programming***
  - ***May be mostly review for some of you***
- **Some theory**
  - **Theoretical foundations and issues in functional programming**
- **ML**
  - **A modern basis for discussing key issues in functional programming**

Flash!



- **We won't get through functional languages tonight**
- **Next week, I'll try to finish before the break**
  - **A little more on ML types**
  - **A little on Haskell, as a comparison to ML**
- **Then we'll cover some basic information on type systems in general**

Shift: From Scheme to theory

- **What are the underpinnings of the functional programming paradigm?**
- **We've seen a lot of the basics**
- **Now for a bit more depth on the theory behind them**

Functions and their combination

- **Functional programs deal with structured data, are often nonrepetitive and nonrecursive, are hierarchically constructed, do not name their arguments, and do not require the complex machinery of procedure declarations to become generally applicable. Combining forms can use high level programs to build still higher level ones in a style not possible in conventional languages. – Backus**
- **Functions are first-class data objects: they may be passed as arguments, returned as results, and stored in variables. The principal control mechanism in ML is recursive function application. –Harper**

The lambda calculus

- This prevalence of functions demands a basic understanding of the lambda calculus
  - Other models of computation (such as Turing machines) don't give us much insight into functional computation
- Why care about the model of computation underlying a programming language?
  - It helps us deeply understand a language, and thus be more effective in using it

Function definition

- A function definition in the lambda calculus has a single basic form
  - λx,y,z • expr
  - where the x,y,z are identifiers representing the function's arguments
- The value of the lambda expression is a mathematical function (not a number, or a set of numbers, or a float, or a structure, or a C procedure, or anything else)

Function application

- Application consists of applying this function given values for the arguments
- We show this by listing the values after the lambda expression
  - (λx,y •x+y) 5 6
  - (λx,y •if x > y then x else y) 5 6
  - (λx •if x > 0 then 1 else
    if x < 0 then -1 else 0) -9

That it!

- **That's just about the full definition of the λ-calculus**

Currying

- **Many definitions of the lambda calculus restrict functions to having a single argument**
- **But that's OK, since there exists a function that can curry other functions, yielding only functions of one argument**
- **So we can use a multiple argument lambda calculus without loss of generality**

Beta reduction rule

- Application is defined using the beta-reduction rule
  - This in essence replaces the formals with the values of the applied arguments
  - It defines the actual computation of the machine
  - More details to come
- (λx,y •if x > y then x else y) 5 6
  if 5 > 6 then 5 else 6
  6

Representation

- **For the λ-calculus to be Turing-complete, we need to address some representational issues**
- **Indeed, it only really has identifiers that don't really represent anything**
- **Even writing + is kind of a cheat**
  - **You had to define addition for a Turing machine, too**

(Non-negative) integers

- Here's a scheme for representing non-negative integers
  - **0** ≡ λf • λx • x
  - **1** ≡ λf • λx • f x
  - **2** ≡ λf • λx • f (f x)
  - **3** ≡ λf • λx • f (f (f x))
- That is, every time we see λf • λx • f (f x), we think "Oh, that's 2"
  - If you prefer to "uncurry" you can think of this as λf,x • f (f x),
- You can think of f as "increment by 1"

Defining addition

- add ≡ λa • λb • λf • λx • a f (b f x)
  - Or λa,b,f,x • a f (b f x)
- To add 1 and 2 we write
  - (λa • λb • λf • λx • a f (b f x))
    (λf • λx • f x) (λf • λx • f (f x))

Reduction

- Applying β-reduction to both arguments
  - λf • λx • (λf • λx • f x) f
    ((λf • λx • f (f x)) f x)
  - λf • λx • (λf • λx • f x) f (f (f x))
  - λx • (λf • λx • f x) (f (f x))
  - λf • λx • f (f (f x))
- Whew, 1 + 2 is 3
  - This is much like adding in unary on a Turing machine

Representing booleans

- true ≡ λt • λf • t
- false ≡ λt • λf • f

Defining cond

- cond ≡ λb • λc • λa • b c a
  - cond true 2 1
    (λb • λc • λa • b c a) true 2 1
  - (λc • λa • true c a) 2 1
  - (λa • true 2 a) 1
  - true 2 1
  - (λt • λf • t) 2 1
  - (λf • 2) 1
  - 2
- Of course, we could represent 1 and 2 explicitly as functions, too

Normal form

- A lambda expression has reached normal form if no reduction other than renaming variables can be applied
  - Not all expressions have such a normal form
- The normal form is in some sense the value of the computation defined by the function
  - One Church-Rosser theorem in essence states that for the lambda calculus the normal form (if any) is unique for an expression

Reduction order

- A *normal-order reduction* sequentially applies the leftmost available reductions first
- An *applicative-order reduction* sequentially applies the leftmost innermost reduction first
- This is a little like top-down vs. bottom-up parsing and choosing what to reduce when

Example

- (λx • y) ((λx • x x) (λx • x x))
  - never reduces in applicative-order
- (λx • y) ((λx • x x) (λx • x x))
  - reduces to y directly in normal-order

High-level view

- Normal-order defines a kind of lazy (non-strict) semantics, where values are only computed as needed
  - This is not unlike short-circuit boolean computations
- Applicative-order defines a kind of eager (strict) semantics, where values for functions are computed regardless of whether they are needed

A different high-level view

- **To the first order, you can think of normal-order as substituting the actual parameter for the formal parameter rather than evaluating the actual first**
  - **It's closely related to call-by-name in Algol**
- **To the first order, you can think of applicative-order (also called eager-order) as evaluating each actual parameter once and passing its value to the formal parameter**

An aside: call by name

- **You should be aware of standard parameter passing mechanisms**
  - **Call by value**
    - **Make a copy of the actual to pass to the formal parameter**
  - **Call by reference**
    - **Have the formal actually point to the actual**
  - **Call by result, call by value-result, etc.**
- **ALGOL defined a rich, expensive and confusing mechanism, Call by Name**

Call by name

- **The formal is reevaluated at each use**
  - **Uses a *thunk* to implement it**
- `swap(i, a[i])`
  - `i = 1`
  - `a[1] = 3`
  - `a[3] = 17`
- **Doesn't swap, and you can't fix it**

```
proc swap(a, b : int);
  var temp : int;
  begin
    temp := a;
    a := b;
    b := temp
end;
```

Jensen's device

- Why define Call by Name?
- Highly expressive
  - At least based on this one classic example!

```
real proc SUM (k, low, up, ak);
value low, up;
integer k, low, up; real ak;
begin real s;
  s := 0;
  for k := low step 1 until up do
    s := s + ak; sum := s
end;

• sum(i, 1, m, A[i])
• sum(i, 1, m, sum(j, 1, n, B[i,j]))
```


Comparing reduction orders

- For many functions, the reduction order is immaterial to the computation
- `fun sqr n = n * n;` // from D. Watt
- `sqr (p+q) [say, p = 2, q = 5]`
- For applicative-order, we compute p+q=7, bind n to 7, then compute 49
- For normal-order, we pass in "p+q" and evaluate 2+5 each time sqr uses n
- But we get the same answer regardless

Strict functions

- A strict function requires all its parameters in order to be evaluated
- `sqr` is strict, since it requires its (only) parameter
  - `(define one (x) 1)` is not-strict
  - `(one 92)`

More strictness

- `fun cand (b1, b2) = if b1 then b2 else false`
- `cand(n>0,t/n>0.5) with n=2 and t=0.8`
  - Eagerly, n>0 evaluates to true, t/n>0.5 evaluates to false, and therefore the function evaluates to false
  - Normal-order also evaluates to false.
- But what if n=0
  - Eagerly, n>0 evaluates to false but t/n>0.5 fails due to division-by-zero; so the function call fails.
  - But with normal-order, the division isn't needed nor done, so it's fine.
- This function is considered to be strict in its first argument but non-strict in its second argument

Fixed-points (or fixpoints)

- The idea of defining the semantics of lambda calculus by reducing first every expression to normal form (for which a simple mathematical denotation exists) by a sequence of contractions is attractive but, unfortunately, does not work as simply as suggested… The problem is that, since every contraction step … removes a λ, we have deduced a bit hastily that it decreases the overall number of λs. We have neglected the possibility for a contraction step actually to *add* one λ, or even more, while it removes another. **– Meyer**

Example

- SELF ≡ λx • (x (x))
- SELF (λx • (x (x)))
- λx • (x (x)) (λx • (x (x)))
- What does this application of SELF to itself produce?
  - λx • (x (x)) (λx • (x (x)))
  - Itself, with no reduction in lambda's.

The good news

- However, we're still not in trouble
- Church proved a theorem that shows that any recursive function can be written non-recursively in the lambda calculus
  - So we can use recursion without (this) danger in defining programs in functional languages

But its complicated

- **Theorem: If there is a normal form for a lambda expression, then it is unique**
  - **There isn't always a normal form, however**
- **Theorem: If there is a normal form, then normal-order reduction will get to it**
  - **Applicative-order reduction might not**
- **So, it seems pretty clear that you want to define a functional language in terms of normal-order reductions, right?**

In theory, there is no difference between theory and practice

- **Nope, since efficiency shows it's ugly head**
  - **Even for sqr above, we had to recompute values for expressions more than once**
  - **And there are lots of examples that arise in practice where "unnecessary" computations arise regularly**
- **So, applicative-order evaluation looks better again**

But…

- **But there are two problems with this, too**
  - **The "magic" approach to representing recursion without recursion falls apart for applicative-order evaluation; a special reduction rule for recursion must be introduced**
  - **It isn't always faster to evaluate**

Example

- **(λx•1)(* 5 4) in normal-order and in applicative-order**
- **(λx•1)((λx•x x) (λx•x x)) in normal-order and in applicative-order, as we know still stands as a problem**
- **Even with this, most early functional languages used applicative-order evaluation: pure Lisp, FP, ML, Hope, etc.**

What do to?

- **The basic approach to doing better lies in representing reduction as a graph reduction process, not a string reduction process; this allows sharing of computations not allowed in string reductions (Wadsworth)**
- **A graph-based approach to normal-order evaluation in which recomputation is avoided (by sharing) is called lazy evaluation, or call-by-need**
  - **One can prove it has all the desirable properties of normal-order reduction and it more efficient than applicative order evaluation.**
  - **Still, performance of the underlying mechanisms isn't that great, although it's improved a ton**

Theory

- OK, that's all the theory we'll cover for functional languages
  - There's tons more (typed lambda-calculus, as one example)
- It's not intended to make you theoreticians, but rather to give you some sense of the underlying mathematical basis for functional programming

ML

- Same core concepts as Scheme
  - Strongly typed
  - Expression-oriented, mostly side-effect-free
  - List-oriented, garbage-collected, heap-based
  - Highly regular and expressive
- Designed as a *M*eta *L*anguage for automatic theorem proving system in the mid-1970s by Milner et al.
- Standard ML in 1980; SML'97 in 1997

What's different from Scheme?

- Statically typed
  - Polymorphic type system
  - Automatic type inference
- Pattern matching
  - To define alternate cases
- Exceptions
- Modules

Basic datatypes

- `unit` (like `void`)

```
- ()
() : unit
```

- `bool, int, real, string`

```
- 7 > 5;
true : bool
- ~24 + 1;
~23 : int;
- 3.14159;
3.14159 : real;
- "hi there"
"hi there" : string
```


Lists

- Variable number of elements but homogeneous element type
- `[…]` constructor notation, like `(list …)`
- `::` and `nil` constructor notation, like `cons` and `()`
  - `::` is infix and left associative

List examples

```
- [3,4,5];
[3,4,5] : int list
- 3::4::nil;
[3,4] : int list
- (3::nil)::(4::5::nil)::nil;
[[3],[4,5]] : int list list;
- [3,"hi there"];
Error: operator and operand don't agree
  operator domain: int * int list
  operand:         int * string list
  in expression: (3 : int) :: "hi there" :: nil
```

Tuples

- **Heterogeneous element types**
  - **Fixed number of elements**
  - **Positional components**

```
- ("wow",6);
("wow",6) : string * int;
- (1.0,2,true,"s");
(1.0,2,true,"s") :
     real * int * bool * string
```


Records

- **Heterogeneous element types**
  - **Fixed number of elements**
  - **Unordered, named components**

```
- {name="J",age=1};
{name="J",age=1} : {age:int,name:string}
- {name="J",age=1}={age=3 mod 2,name="J"};
true : bool
```


Bindings

- **Global variables**

```
- val x = 23;
val x = 23 : int;
```

- **Local variables for declarations**

```
- local
    val x = 10
    val q = x
  in
    val u = 230 * x
    val z = (true, 22+q)
  end;
val u = 2300 : int;
val z = (true,32) : bool * int
```


Local variables for expressions

```
- local
    val x = 5
    val q = x + 1
  in
    x * q
  end;
30 : int
```


Named function definitions

```
- fun succ x = x + 1;
val succ = fn : int -> int
- succ 8;
9 : int
- fun fact n =
        if n <= 1 then 1
                  else n*fact (n-1);;
val fact = fn : int -> int
- fact 4;
24 : int
```


Anonymous functions

```
- fn x => x + 1;
fn : int -> int
- (fn x => x + 1) 8;
9 : int
- val succ = fn x => x + 1;
val succ = fn : int -> int
```

- **`fun` is syntactic sugar in ML, just like `define` is in Scheme**

Function types

- Types of functions in ML are always written as `input -> output`
  - Only one argument (and type) as input
  - Only one argument (and type) as output
- To “pass multiple arguments”
  - Use tuples
  - Or use currying

Using tuples

```
- fun plus(x,y) = x + y;
fn : int*int -> int
- fun all_to_all(v,lst) =
      if null lst then nil
      else (hd lst + v)::
               add_to_all(val, tl lst);
val : add_to_all = fn int*int list ->
                      int list
```


Minor notes

- You can “return multiple results” the same way, using tuples
  - Ex: returning a point in two-space using either Cartesian or polar coordinates
- Precedence rules for expressions
  - Juxtaposition, then prefix, then infix
- Precedence rules for types
  - `list` suffix, then `*`, then `->`

Another function type example

```
- fun quad(a,b,c) =
      let b_sqr = b*b
          sq_4_a_c = sqrt(4.0*a*c)
          two_a = 2.0*a
      in
          ((b_sqr + sq_4_a_c)/two_a),
           (b_sqr - sq_4_a_c)/two_a))
  end;
val : quad = fn real*real*real ->
                      real*real
```


Pattern matching: on tuples

- Means of decomposing compound values
- Reuse constructor syntax to take values apart

```
- val x = (false,17);
- val (a,b) = x;
val a = false : bool;
val b = 17 : int;
- val (false,c) = x;
val c = 17 : int;
- val (true,w) = x;
exception: nonexhaustive binding failure
```


Another couple of examples

```
- val s = ["1","2","3"];
- val hd::tl = s;
val hd = "1" : string
val tl = ["2","3"] : string list
- val hd::_ = s;
val hd = "1" : string
```

Pattern matching with functions

- Means of defining function behavior by cases

```
- fun fib 0 = 0
    | fib 1 = 1
    | fib n = fib(n-1)+fib(n-2);
val fib = fn : int -> int
```

- Could use conditionals, but pattern matching is often much clearer

Another example

```
- fun is_empty nil = true
    | is_empty (_::_) = false;
val is_empty = fn : 'a list -> bool
```

- What's going on here?
  - What's `'a`?
- This is a polymorphic function
  - It must take a list
  - But it can take a list of anythings
  - Those "anythings" are represented by the type variable `'a`
  - We'll see examples where a type variable appears twice in the function type, and they must be the same

And here is such an example

```
- fun append(nil,l) = l
   | append(hd:tl,l) = hd::append(tl,l);
val append = fn : 'a list * 'a list
                 -> 'a list
```

- All three `'a` type variables must have the same type

```
- append([1,2],[3]);
[1,2,3] : int list;
- append(["w","us"],["s"]);
["w","us","s"] : string list;
- append([1,2],["s"]);
…type error…
```


Polymorphic functions

- Some functions are general
  - That is, they can be used on arguments of different types
  - Ex: `length, append, is_empty, first_of_pair`
- In Scheme: easy using dynamic typing
- How can you do this with a static typing?
  - In C, you can't
    - or you need to cheat using casts
  - In ML, functions can have polymorphic types

Polymorphic types

- A polymorphic type contains one or more type variables
  - `'a list`
  - `'a * 'b`
  - `(('a * 'b) list 'a) -> 'b`
- To make a regular type, replace the type variables with regular types
  - Each occurrence of a type variable must be replaced with the same type (cf. unification in logic programming)

On invocation

- When calling a polymorphic function, the caller knows the real type
  - So replace the type variable with the regular type

```
val length = fn : 'a list -> int
- length([3,4,5]); (* replaces 'a with int *)
- length([(3.0,"hi"),(2.0,"there")]);
        (* replaces 'a with int*string *)
```

Examples

```
- fun map(f,nil)    = nil
    | map(f,x::xs) = f x :: map(f,xs);
val map = fn : ('a -> 'b) * 'a list
                  -> 'b list
val square = fn : int -> int
- map square [3,4,5];
[9,16,25] : int list
        (* 'a is int and 'b is int *)
val length = fn : 'a list -> int
- map length [[3,4],[5,6,7],[]];
[2,3,0] : int list
        (* for length: 'a is int
           for map: 'a is int list, 'b is int   *)
```


Currying

- **Simple definition**

```
- fun map f =
    fn lst => if null lst then nil
              else f (hd lst)::
                     (map f) (tl lst);
val map = fn : ('a -> 'b) -> 'a list
                 -> 'b list
```

- **Simple application**

```
- (map square) [3,4,5];
[9,16,25] : int list
```


Exploiting juxtaposition

```
- map square [3,4,5];
[9,16,25] : int list
```

- **Juxtaposition associates left-to-right**

A pattern-based curried `map`

```
- fun map f =
    fn nil => nil
     | x::xs => f x:: map f xs;
val map = fn : ('a -> 'b) -> 'a list
                 -> 'b list
```


Clean syntactic sugar for currying

- **ML allows multiple formal argument patterns, which implies a curried function**
  - In essence, ML always curries for you
  - You never see functions of multiple arguments in ML
    - Even with tuples, it's really a single argument

```
- fun map f nil      = nil
    | map f (x::xs) = f x:: map f xs;
val map = fn : ('a -> 'b) -> 'a list
                  -> 'b list
```


Reminder: currying is

- **syntactically cleaner**
- **semantically more flexible, since each function of one-argument can be used alone, if desired**

Polymorphism vs. overloading

- With a polymorphic function, the exact same function can be used with many different argument types
- In contrast, *overloading* gives several different functions the same name

```
- 3 + 4;
7 : int
- 3.0 + 4.5;
7.5 : real
```


How does ML handle overloading?

- Resolves overloading based on static argument types
  – OO languages more generally use dynamic argument classes
  – Ada also uses return types
- If you want to specify a particular operator in ML, you can

```
– (op +): real*real->real
fn : real*real -> real
```


Equality types

- ML defines a built-in = function that is polymorphic over all types that "admit equality"
  – In ML, this is any type except those containing functions or reals
  – Why? Are there other examples of types that shouldn't be compared for equality?

Examples

A type variable that is restricted to types that admit equality

```
- fun member x nil      =   false
    | member x (y::ys) =
        x = y orelse member x ys;
val member = fn :''a -> ''a list -> bool
- member 3 [4,5,6,3,4,5];
true : bool
- member 4.5 [3.4,5.6,7.8];
…type error…
- member ("hi",3,square)
         [("there",6,double)];
…type error…
```


Type inference: infer types of expressions automatically

- Assign each declared variable a fresh type variable
  – Result of function is an implicit variable
  – Share argument and result type variables across function cases
  – Each reference to a let-bound polymorphic identifier (roughly, a named function) gets separate type variables
- Each expression in construct places constraints on the types of its operands
- Solve those constraints

Type inference: partial refinement

- Constraint solving can partially refine types: that is, it can replace some, but not necessarily all, type variables with more constrained values
  – **'a => 'b list**
  – **'a => ''b**

Properties of ML's type system

- Hindley-Milner type system/inference
  - Universal parametric polymorphism
    - No constrained polymorphism
    - Except for equality types
  - Only let-bound polymorphism
    - Cannot pass polymorphic value that is polymorphic inside callee (cf. next slide)
- Type inference yields *principal type* for expression
  - Single most general type that can be inferred
- Worse case time complexity: exponential
- Average case complexity: linear

Examples

```
- fun id x = x;
val id = fn : 'a -> 'a
- fun g f = (f 3, f "hi");
(* type error in ML, but in
  SuperML++: *)
val g = fn : (∀'a.'a->'a) ->
  int*string
```


Next time

- ML types
  - user-defined datatypes, variant records, recursive types, polymorphic types, exceptions, streams, …
- Haskell
  - lazy evaluation
    - purely side-effect free, infinite lists
  - type classes for added flexibility in polymorphism