CSCI 334:
Principles of Programming Languages

Lecture 5: Fundamentals III & ML

Instructor: Dan Barowy

Williams

Announcements

Claire Booth Luce info session tonight

for women interested in summer research

(hopefully in CS!),

TBL 211

(also: pizza and ice cream)

midterm: before or after spring break?

```
M ::= x        variable
   |  λx.M     abstraction
   |  MM       function application
```

`MMM`

`(MM)M` or `M(MM)`?

Order does not matter

$$\begin{tikzcd}
 & M \arrow[dl] \arrow[dr] & \\
M_1 \arrow[dr, dotted] & & M_2 \arrow[dl, dotted] \\
 & N &
\end{tikzcd}$$

If $M \rightarrow M_1$ and $M \rightarrow M_2$
then $M_1 \rightarrow^* N$ and $M_2 \rightarrow^* N$
for some N

"confluence"

Reduction strategies

$$\underbrace{\texttt{(}\lambda\texttt{x.y)}}_{\text{function}}\underbrace{\texttt{((}\lambda\texttt{x.x x)(}\lambda\texttt{x.x x))}}_{\text{argument}}$$

Reduction strategies

$$\texttt{(}\lambda\texttt{x.y)(}\underbrace{\texttt{(}\lambda\texttt{x.x x)}}_{\text{function}}\underbrace{\texttt{(}\lambda\texttt{x.x x)}}_{\text{argument}}\texttt{)}$$

Reduction strategies

$$\underbrace{\texttt{(}\lambda\texttt{x.y)}}_{\text{function}}\underbrace{\texttt{((}\lambda\texttt{x.x x)(}\lambda\texttt{x.x x))}}_{\text{argument}}$$

Normal-order reduction:
Choose the left-most redex first.

1. `([(λx.x x)(λx.x x)/x]y)`
2. `y`

Reduction strategies

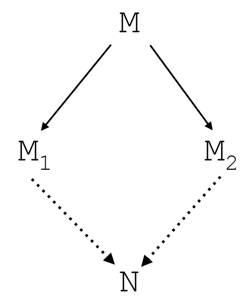
```
(λx.y)((λx.x x)(λx.x x))
```

function argument

Applicative-order reduction:
Choose the right-most redex first.

```
1.  (λx.y)(([(λx.x x)/x]x x))
2.  (λx.y)((λx.x x)(λx.x x))
3.  (λx.y)(([(λx.x x)/x]x x))
4.  uh oh…
```

Order does not matter

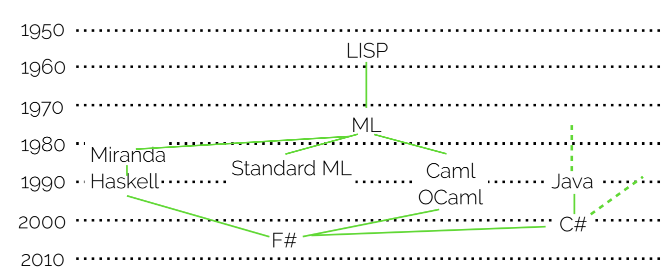


If $M \to M_1$ and $M \to M_2$
then $M_1 \to^* N$ and $M_2 \to^* N$
for some N

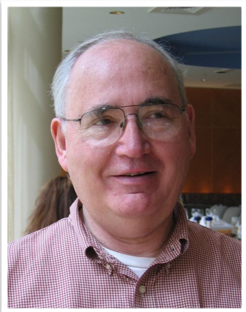
- Normal-order reduction will always find a normal form *if one exists*.
- Applicative-order reduction will always find a normal form *if reduction terminates*.

```
(λf.λx.f(f x))(λz.x+z)2 =
(λf.λa.f(f a))(λz.x+z)2 ->        (rename x)
([(λz.x+z)/f]λa.f(f a))2 =        (reduce f)
(λa.(λz.x+z)((λz.x+z)a))2 =       (substitution)
(λa.(λb.x+b)((λz.x+z)a))2 ->      (rename z)
(λa.(λb.x+b)([a/z](x+z))2 =       (reduce z)
(λa.(λb.x+b)(x+a))2 ->            (substitution)
(λa.[(x+a)/b](x+b))2 =            (reduce b)
(λa.x+(x+a))2 ->                  (substitution)
[2/a](x+(x+a))=                   (reduce a)
x+x+2                             (substitution)
```

ML



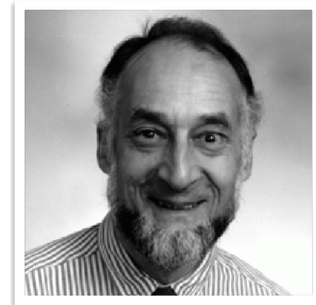
ML



- Dana Scott
- Logic of Computable Functions (LCF)
  - Automated proofs!
  - Theorem proving is essentially a "search problem".
  - It is (essentially) NP-Complete
  - But works "in practice" with the right "tactics"

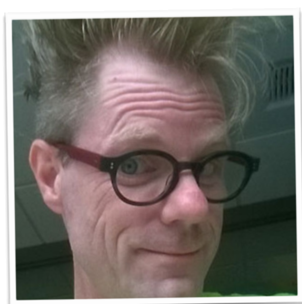
ML

- Robin Milner
- How to program tactics?
- A "meta-language" is needed
- ML is born



LCF/ML influence: Dafny

```
method MultipleReturns(x: int, y: int) returns
(more: int, less: int)
   ensures less < x
   ensures x < more
{
   more := x + y;
   less := x - y;
}
```



- K. Rustan Leino
- Dafny programs can often be *proven* correct (wrt spec)
- Rustan also famous for his hair :)

ML Features: static types

- Core: LISP + "static types"
  - types are checked *before program runs*
- Static types guarantee correctness of programs
  - Why does this not violate halting problem?
  - All "well-typed" programs do not fail at runtime

ML Features: parametric polymorphism

```
fun swapInt(x: int, y: int): int*int = (y,x)
fun swapReal(x: real, y: real): real*real = (y,x)
fun swapString(x: string, y: string): string*string = (y,x)
```

- "abstract types" allow programmers to write generic programs; reveal underlying idea without boilerplate

```
fun swap(x: 'a, y: 'b): 'b * 'a = (y,x)
```

ML Features: type inference

```
fun swap(x: 'a, y: 'b): 'b * 'a = (y,x)
```

- writing types is hard (and sometimes ugly!)

```
fun swap(x, y) = (y,x)
```

ML Features: exceptions

- Milner: it's hard to write well-typed programs
  - mechanism to allow programs to signal error
  - *and correct for them at runtime*

```
fun foo =
 exception DivByZero of string
 if x = 0 then raise DivByZero("no zeros!")

…

foo
handle DivByZero msg => do something else
```

ML Features: side effects; mutability

- These are features?
- For real-world programs, yes.

```
fun foo() =
  let val name = "Dan"
  in
    print (name ^ "\n")
  end;
```

side effect

```
let val x : int ref = ref 3
    val y : int = !x
in
    x := (!x) + 1;
    y + (!x)
end
```

mutability

- Both are often essential for speed
- But can be largely avoided in many programs for safety

Running ML

- Type `sml` on Unix machines
- `Ctrl-D` to quit
- Enter expression or declarations to evaluate:

```
- 3 + 5;
val it = 8 : int
- it * 2;
val it = 16 : int
- val six = 3 + 3;
val six = 6 : int
```

- Or "`sml < file.ml`"

Defining Functions

No type info given- compiler infers it

- Example

```
- fun succ x = x + 1;
val succ = fn : int -> int
- succ 12;
val it = 13 : int
- 17 * (succ 3);
val it = 68 : int;
```

- Or:

```
- val succ = fn x => x + 1;
val succ = fn : int -> int
```

Recursion

- All functions written using recursion and `if.. then.. else` (and patterns):

```
- fun fact n =
    if n = 0 then 1 else n * fact (n-1);
```

- `if..then..else` is an expression:

```
- if 3<4 then "moo" else "cow";
val it = "moo" : string
```

  - types of branches must match

Local Declarations

```
- fun cylinderVolume diameter height =
    let val radius = diameter / 2.0;
        fun square y = y * y
    in
        3.14 * square radius * height
    end;

val cylinderVolume = fn : real -> real -> real

- cylinderVolume 6.0 6.0;
val it = 169.56 : real
```

Built-in Data Types

- unit
  - only value is `()`
- bool
  - `true, false`
  - `operators not, andalso, orelse`
- int
  - `..., ~2, ~1, 0, 1, 2, ...`
  - `+,-,*,div,mod,abs`
  - `=,<,<=, etc.`

Built-in Data Types

- real
  - `3.17, 2.2,` …
  - `+, -, *, /`
  - `<, <=,` etc.
  - no implicit conversions from int to real: `2 + 3.3` is bad
  - no equality (test that `-0.001 < x-y < 0.001`, etc.)
- strings
  - `"moo"`
  - `"moo" ^ "cow"`

Overloaded Operators

- +,-,etc. defined on both int and real
- Which variant inferred depends on operands:

```
- fun succ x = x + 1
val succ = fn : int -> int

- fun double x = x * 2.0
val double = fn : real -> real

- fun double x = x + x
val double = fn : int -> int
```

Type Declarations

- Can add types when type inference does not work

```
- fun double (x:real) = x + x;
val double = fn : real -> real

- fun double (x:real) : real = x + x;
val double = fn : real -> real
```

Compound Types

- Tuples, Records, Lists
- Tuples

```
(14, "moo", true): int * string * bool
```

- Functions can take tuple argument

```
- fun power (exp,base) =
    if exp = 0 then 1
      else base * power(exp-1,base);
val power = fn : int * int -> int
- power(3,2);
```

Curried Functions (named after Haskell Curry)

- Previous power

```
- fun power (exp,base) =
      if exp = 0 then 1
         else base * power(exp-1,base);
val power = fn : int * int -> int
```

- Curried power function

```
- fun cpower exp =
     fn base =>
       if exp = 0 then 1
          else base * cpower (exp-1) base;

val cpower = fn : int -> (int -> int)
```

Curried Functions (named after Haskell Curry)

- Why is this useful?

```
- fun cpower exp base =
      if exp = 0 then 1
         else base * cpower (exp-1) base;
val cpower = fn : int -> (int -> int)
```

- Can define

```
- val square = cpower 2
val square = fn : int -> int
- square 3;
val it = 9 : int
```

Records

- Like tuple, but with labeled elements:

```
- val x =
     { name="Gus", salary=3.33, id=11 };
```

- Selector operator:

```
- #salary(x);
val it = 3.33 : real
- #name(x);
val it = "Gus" : string
```

Lists

- Examples
  - `[1, 2, 3, 4]`, `["wombat", "numbat"]`
  - `nil` is empty list (sometimes written `[]`)
  - all elements must be same type
- Operations
  - `length` `length [1,2,3]` ⇒ `3`
  - `@` - append `[1,2]@[3,4]` ⇒ `[1, 2, 3, 4]`
  - `::` - prefix `1::[2,3]` ⇒ `[1, 2, 3]`
  - `map` `map succ [1,2,3]` ⇒ `[2,3,4]`

Lists

- Functions on Lists

```
- fun product (nums) =
  if (nums = nil)
    then 1
    else (hd nums) * product(tl nums);

val product = fn : int list -> int

- product([5, 2, 3]);
val it = 30 : int;
```

Pattern Matching

- List is one of two things:
  - `nil`
  - `"first elem" :: "rest of elems"`
  - `[1, 2, 3] = 1::[2,3] = 1::2::[3]`
    `= 1::2::3::nil`
- Can define function by cases

```
fun product (nil) = 1
  | product (x::xs) = x * product (xs);
```

Patterns on Integers

- Patterns on integers

```
fun listInts 0 = [0]
  | listInts n = n::listInts(n-1);

listInts 3 ⇒ [3, 2, 1, 0];
```

- More on patterns for other data types next time

Many Types Of Lists

- `1::2::nil : int list`
  `"wombat"::"numbat"::nil : string list`
- What type of list is `nil`?

  ```
  - nil;
  val it = [] : 'a list
  ```
- Polymorphic type
  - `'a` is a type variable that represents any type
  - `1::nil : int list`
    `"a"::nil : string list`

The Length Function

- Another Example

```
fun length (nil) = 0
  | length (x::xs) = 1 + length (xs);
```

- What is the type of length?
- How about this one:

```
fun id x = x;
```

Polymorphism

```
fun length (nil) = 0
  | length (x::xs) = 1 + length (xs);
- val it = fun 'a list -> int
```

```
fun id x = x;
- val it = fun 'a -> 'a
```

Type variable represents any type

Patterns and Other Declarations

- Patterns can be used in place of variables
- Most basic pattern form

  ```
  - val <pattern> = <exp>;
  ```
- Examples

  ```
  - val x = 3;
  - val tuple = ("moo", "cow");
  - val (x,y) = tuple;
  - val myList = [1, 2, 3];
  - val w::rest = myList;
  - val v::_ = myList;
  ```

Datatype

```
public static final int NORTH = 1;
public static final int SOUTH = 2;
public static final int EAST = 3;
public static final int WEST = 4;

public move(int x, int y, int dir) {
  switch (dir) {
    case NORTH: ...
    case ...
  }
}
```

Datatype

```
datatype Direction =
     North | South | East | West;

fun move((x,y),North) = (x,y-1)
  | move((x,y),South) = (x,y+1)
  ;
```

- Above is an “incomplete pattern”
- ML will warn you when you’ve missed a case!
  - “proof by exhaustion”