CS538 Lecture 12: Core ML

March 1997

James R. Larus
Computer Sciences Department
University of Wisconsin–Madison
larus@cs.wisc.edu

Outline:

- Overview of core ML

ML Atomic Types

- Integers (**`int`**)
  - **`1`**, **`100001`**, **`~1`**, **`~38`**
- Real
  - 1.0, 2.6
- Boolean (**`bool`**)
  - **`true`**, **`false`**
  - Operators: **`not`**, **`andalso`**, **`orelse`**
- String
  - **`"a"`**, **`"dog"`**
  - **`"a " ^ "dog"`** = **`"a dog"`**

Types, cont'd

- Unit
  - `()`
  - Domain of function with no arguments

Compound Types 1: Tuples

- If T_1, T_2, ... T_n are types, so is T_1 * T_2 * ... * T_n
  - *Product type*
  - Constructor is ( )

```
- (1, 2, 3);
val it = (1,2,3) : int * int * int

- ("A", 1, 1.0);
val it = ("A",1,1.0) : string * int * real

- val x = ("A", ("big", "dog"), true);
val x = ("A",("big","dog"),true) :
    string * (string * string) * bool
```

  - Accessor is **#n**

```
- #1(x);
val it = "A" : string
- #2(x);
val it = ("big","dog") : string * string
```

Compound Type 2: Lists

- If `t` is a type, so is `t list`
  - Either empty (`nil`) or value of type `t` followed by `t list`
  - Constructors `::` and `[]`

```
- nil;
val it = [] : 'a list
- 1 :: nil;
val it = [1] : int list
- 1 :: 2 :: 3 :: nil;
val it = [1,2,3] : int list
```

```
- [];
val it = [] : 'a list
- [1];
val it = [1] : int list
- [1,2,3];
val it = [1,2,3] : int list
```

  - Accessors

```
- hd([1,2,3]);
val it = 1 : int
- tl([1,2,3]);
val it = [2,3] : int list
```


Lists, cont'd

- Homogeneous type

```
- ["dog", 1.0];
std_in:0.0-0.0 Error: operator and operand don't agree (tycon
mismatch)
  operator domain: string * string list
  operand:         string * real list
  in expression:
    "dog" :: 1.0 :: nil

- [("dog", 1.0)]
val it = [("dog",1.0)] : (string * real) list
```

Compound Types 3: Records

Optional

- Set (unordered!) of labeled fields with values of arbitrary types

```
- val r = {name="jim", job="prof", age=37,
married=true};
val r = {age=37,job="prof",married=true,name="jim"}
  : {age:int, job:string, married:bool, name:string}
- #name(r);
val it = "jim" : string
- #age(r);
val it = 37 : int
```

- No constructor

Type Abbreviations

- type <name> = <type>;

```
- type intpair = int * int;
type  intpair = int * int
- type seq = int list;
type  seq = int list
- type `a seq = `a list;
type  `a seq = `a list
```

Compound Type 4: Datatypes

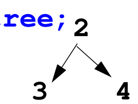
- datatype <name> = constructor of <type>
  | constructor of <type>
  ....

```
datatype tree = empty
              | node of tree * tree;
datatype  tree
  con empty : tree
  con node : tree * tree -> tree
- node(empty,empty);
val it = node (empty,empty) : tree
- node(node(empty,empty),empty);
val it = node (node (empty,empty),empty) : tree
```


Datatypes, cont'd

- More elaborate trees

```
- datatype `a tree = empty
                   | node of `a * `a tree * `a tree;
datatype  `a tree
  con empty : `a tree
  con node : `a * `a tree * `a tree -> `a tree
- node(2, node(3, empty, empty), node(4, empty,
empty));
val it = node (2,node (3,empty,empty),node (4,empty,empty)) :
int tree
```



- Nullary constructors form enumerated types

```
- datatype color = red | blue | green;
datatype  color
  con blue : color
  con green : color
  con red : color
- red;
val it = red : color
```


Copyright © 1996–97 by James R. Larus. All rights reserved.

Expressions

- Everything is expression (produces a value)
  - No statements
- Literals
  - **`"a"`**, **`1`**, **`true`**, **`[1,2,3]`**
- Identifier
  - **`x`**, **`a_variable`**, **`jim's`**, **`'a`**
- Built-in operators
  - **`1 + 2`**, **`"a " ^ "dog"`**, **`is_here andalso is_there`**
- Function application
  - **`fact 3 or fact(6)`**

If Expression

- if <expr> then <expr2> else <expr3>
  - Always an else clause
  - ```
    if 8 > 9 then 8 else 9
    if l = nil then "cat" else "dog"
    ```
    ```

Case Expression

```
case <expr> of
       <value1> => <expr1>
     | <value2> => <expr2>
       ...
     | _        => <exprn>    (*optional default*)
```

- expr and values have same type
- expr_1 ... expr_n have same type

```
- val x = 5;
val x = 5 : int
- case x of 1 => "tiny"
  | 2 => "small"
  | 3 => "medium"
  | 4 => "ok"
  | 5 => "big";
std_in:47.1-51.12 Warning: match nonexhaustive
...
val it = "big" : string
```


Let Expression

- let <declaration> in <expr> end
  - <declaration> ::= val <pattern> = <expr>
  - Binds identifiers in pattern to values

```
- let val x=2 in x+4 end;
val it = 6 : int
- let val y = [1,2,3] in
    let val [a,b,c] = y in
      a+b+c
    end
  end;
std_in:55.5-55.19 Warning: binding not exhaustive
          a :: b :: c :: nil = ...
val it = 6 : int
- let val L = [1,2,3] in
    let val hd::tl = L in
      hd
    end
  end;
```

(Let) Patterns, cont'd

```
- let val L = [1,2,3] in
    let val hd::_ = L in
      hd
    end
  end;
std_in:70.5-70.17 Warning: binding not exhaustive
          hd :: _ = ...
val it = 1 : int

- let val T = node(2, node(3, empty, empty), node(4,
empty, empty)) in
    let val node(k, _, _) = T in
      k
    end
    end;
std_in:75.5-75.32 Warning: binding not exhaustive
          node (k,left,right) = ...
val it = 2 : int
```


Functions

- fun <name> (<params>) = <expr>;

```
- fun inc(x) = x + 1;
val inc = fn : int -> int
- inc 1;
val it = 2 : int

- fun divX(a,b) =
  if b=0 then 0
  else a div b;
val divX = fn : int * int -> int
- divX(10,0);
val it = 0 : int

- fun divX(a,0)=0
  | divX(a,b) = a div b;
val divX = fn : int * int -> int
- divX(20,1);
val it = 20 : int
```

Functions Details

- Cases
  - Tried in order
  - Warning if not mutually exclusive
- May need to declare argument and result type

```
- fun sq (x) = x * x;
std_in:94.16 Error: overloaded variable cannot be resolved: *

- fun sq (x : real) = x * x;
val sq = fn : real -> real
- sq 1.0;
val it = 1.0 : real
```

  - Compiler infers the most general type
  - Type may not exist for overloaded operators

Function Factoid

- Functions have only 1 arg
  - Parentheses form tuple
  - Tuple deconstructed into formal parameter pattern

Currying

- Construct new functions by applying function to subset of args

```
- fun exp1(x, 0) = 1.0
  | exp1(x, y) = x * exp1(x, y-1);
val exp1 = fn : real * int -> real
- fun exp2 x 0 = 1.0
  | exp2 x y = x * exp2 x (y-1);
val exp2 = fn : real -> int -> real
- exp1(3.0, 4);
val it = 81.0 : real
- exp2 3.0 4;
val it = 81.0 : real
```

Note different types

- Curry

```
- val g = exp2 3.0;
val g = fn : int -> real
- g 4;
val it = 81.0 : real
- g 5;
val it = 243.0 : real
```


Function Examples

```
- fun sum(nil) = 0
  | sum(hd::tl) = hd + sum(tl);
val sum = fn : int list -> int
- sum [1,2,3,4,5];
val it = 15 : int

- fun sum(empty) = 0
  | sum(node(k, left, right))
     = k + sum(left) + sum(right);
val sum = fn : int tree -> int
- val t = node(2, node(3, empty, empty), node(4, empty,
empty));
val t = node (2,node (3,empty,empty),node (4,empty,empty)) :
int tree
- sum t;
val it = 9 : int
```

Accumulating Parameters

- Reverse order of operations by passing result as argument
  - O(N^2) algorithm

```
- fun append(nil, v) = [v]
  | append(hd::tl, v) = hd::append(tl, v);
val append = fn : 'a list * 'a -> 'a list
- fun reverse(nil) = nil
  | reverse(hd::tl) = append(reverse(tl), hd);
val reverse = fn : 'a list -> 'a list
- reverse [1, 2, 3];
val it = [3,2,1] : int list
```

  - O(N) algorithm

```
- fun reverse(L) =
  let fun revl(nil, A) = A
        | revl(hd::tl, A) = revl(tl, hd::A)
  in revl(L, nil) end;
val reverse = fn : 'a list -> 'a list
- reverse [5,6,7,8];
val it = [8,7,6,5] : int list
```

Note: Local Function

Tail Recursion

- No for loop
  - Does have a while loop (useless so far: no side-effects)
- Iteration expressed as recursion
  - More general construct
- Tail-recursive functions are efficient

```
- fun append(nil, v) = [v]
  | append(hd::tl, v) = hd::append(tl, v);

fun revl(nil, A) = A
  | revl(hd::tl, A) = revl(tl, hd::A);
```

Tail Recursive

  - No stack growth
  - Compiled to same code as loop