Principles of Programming Languages V

Wael Aboelsaadat
wael@cs.toronto.edu
http://www.dgp.toronto.edu/~wael/324.html

Acknowledgement: Some of the material in these lectures are based on material by Diane Horton.

Today

• Introduction to Typing
• Introduction to ML

Typing: introduction

• A name for a set of values and some operations which can be performed on that set of values.

• Give some semantic meaning to what is ultimately just mere bits

Typing: categories

• Primitive types
  – Simplest kind of type e.g. integer and floating-point number

• Composite Types
  – Consisting of basic types
  – E.g.
    ```
    struct Person {
      int age;
      char *name;
    };
    ```

• Object types
  – A datatype that is used in object-oriented programming to wrap a non-object type to make it look like an object
  – E.g. class Integer in Java

• Class types
  – Classes describe the rules by which objects behave; those objects, described by a particular class, are known as “instances” of said class.
Typing: categories

• Subtypes
  – If given type A is compatible with type B, then
    A is a subtype of B
  – Hence, one datatype can be more than one subtype
  – Polymorphism
    • E.g.

```java
public class Shape{
    public void draw( int x, int y){
        // do nothing
    }
}

public class Rectangle extends Shape{
    public void draw( int x, int y){
    }
}

public class Circle extends Shape{
    public void draw( int x, int y){
    }
}
```

... .

```java
Shape myShape;
myShape = new Rectangle( );
myShape .draw( );
myShape = myCircle;
myShape .draw( );
```

Typing: categories

• Interfaces/protocols
  – A definition of methods and values which the objects agree upon in order to cooperate.
  – A specification of those properties of a software component that other components may rely upon
  – E.g.

```java
public interface Shape{
    public abstract void draw( int x, int y);
}

public class Circle extends Shape{
    public void draw( int x, int y){
        myShape=myShape;
        myShape = new Rectangle( );
        myShape = myCircle;
        myShape.draw( );
    }
}
```

Typing: why?

• Efficiency & Optimization
  – Compiler can generate better code if it knows exactly what is in each variable.
  – Historically that's where typing comes from
  – In Fortran
    • INTEGER I
    • REAL R
  – Subroutines have parameters of specified type
    – Compiler can generate efficient code for exactly the type of data present
      without any kind of checks.
  – Types match machine data types
    • For example: float(32bit-IEEE754), double(64bit-IEEE754)
Typing: why?

• Error Checking & Prevention
  – In a freely typed system (e.g. assembler)
    • Can add a float to an integer?!?
    • Can compare a string to a float?!?
    • Can use a float as a pointer?!?
    • Can assume a buffer is 1K? (when really it is only 512 bytes?!?)
  – Typing prevents errors of this kind
  – Stronger typing capabilities mean that more errors can be detected.

• Documentation & Readability
  – Using types in languages also improves documentation of code.

Typing: why?

• Modularity
  – Allow programmers to express the interface between two subsystems.
  – This localizes the definitions required for interoperability of the
    subsystems and prevents inconsistencies when those subsystems
    communicate.

Type-checking: introduction

• The process of verifying and enforcing the constraints of types
  is called type checking.

• Type checking may either occur at compile-time (static check)
  or at run-time (dynamic check)
Type Checking: static vs. dynamic

- **Static type checking**
  - Part of semantic analysis carried by compiler
  - E.g.:
    - Code contains explicit declarations
    - E.g. A : Integer;
    - Compiler knows the type
    - Compiler checks the type at compile time

- **Dynamic type checking**
  - Type is not determined till run-time
  - Variables can be differently typed according to execution path
  - E.g.:
    ```
    if( !bFlag ) {
        String strClassFilePath;
        strClassFilePath = readfromConfigurationFile();
        myShape = (Shape)class.forName( strClassFilePath );
    }
    myShape.draw();
    ```

Type Checking: type inference

- Compiler/interpreter infer type from code!
  - E.g. A3 := B4 + 1;
  - Q: What type is A3 and B4 ?
    - A: Must be integer
  - E.g. if test then ...
  - Q: What type is test ?
    - A: Must be Boolean
  - E.g. A3 := Func(Test1);
  - Q: What is Func?
    - A: Must be array/function Boolean->Integer

- A language with a sound type system is one in which all types can always be inferred in any valid program.
- Example of language designed this way is Standard ML.
- Note that we still have strong static typing and many of the benefits that come from that.

ML: introduction

- Developed at Edinburgh (early ’80s) as Meta-Language for a program verification system
  - Now a general purpose language
  - There are two basic dialects of ML
    - Caml (including Objective Caml, or OCaml)

- A pure functional language
  - Based on typed lambda calculus
  - Grew out of frustration with Lisp!
  - Serious programs can be written without using variables
  - 1970’s Lisp (McCarthy)
  - 1960’s semantic, deconstruction
  - 1970’s FP (Backer)
  - 1980’s Miranda (Turner), ML (Milner)
  - 1990’s Haskell

- Widely accepted
  - reasonable performance (claimed)
  - can be compiled
  - syntax not as arcane as LISP (nor as simple …)

ML: main features

- Strong, static typing
  - Quite a fancy type system?
- Parametric polymorphism
  - Similar to OOP (in fact, it influenced OO)
- Pattern matching
  - Function as a template
- Exception handling
  - Allow you to handle errors/exception
- Elaborate module system
  - Most highly developed of any language
- Type inference
- Recursive data type
ML: how far have PL advanced?

- Writing a gcd implementation

\[
gcd(m,n) = \begin{cases} 
    n & m = 0 \\
    gcd(r \mod m, m) & m > 0
\end{cases}
\]

```
define gcd(m,n) = 
    if (zero? n) m
    (gcd n (remainder m n))
```

Pascal
```
function gcd(m,n : integer) : integer;
begin
    while n <> 0 do begin
        temp := m; m := n; n := temp \mod n;
    end;
    gcd := m;
end;
```

Scheme
```
fun gcd(m,n) =
    if m=0 then
        n
    else
        gcd(n mod m, m);
```

ML: types & expressions

- **Primitive types**
  - bool, int, real, string

- **Constructors**
  - list, tuple, array, record, function

- Each ML expression has a type associated with it.
  - Interpreter builds the type expression for each input

- Cannot mix types in expressions 2+3.0 : real \text{error!}
  - Must explicitly coerce/type: e.g. real(2) + 3.0 : real

ML: Primitive Types

- **int** e.g. x : int,
  - Negative sign uses ~
  - Operators: +/- \text{div} mod

- **real** e.g. x : real
  - 3.45 or using e notation (3.45E7)
  - Operators: +/- *.2
  - Conversion functions: real(integer), floor(real), abs(x)

- **string** e.g. x : string
  - Delimited by double quotes
  - Caract ^ is concatenation e.g. "house" ^ "cat"
  - Function size returns length of string
  - Special characters: \n \t " \\n
- **bool** e.g. b : bool;
  - \text{true} and \text{false}

ML: operators

- All operators are infix

- **Numeric operators**
  - The usual <, >, <=, >= and <> are available
  - For reals, = and <> are not available (a <= b andalso a>= b)
  - For strings, these can be used for lexicographic ordering

- **Logical operators**
  - Short-circuit evaluation
  - if .. then .. else .. is an expression, not a control structure...

- Operator overloading:
  - Same symbol could be used for operations that are internally dissimilar
  - <, <=, >, => are all overloaded
  - When an overloaded operator is used, the leftmost argument is inspected first to decide on type
ML: assignment

- Use `val` to assign value to variables
  - Syntax: `val <constant-name> = <expression>;`
  
- Examples:
  - `val seconds = 60;`
  - `val minutes = 60;`
  - `val tm = seconds * minutes;`
  - `val shout = "aaa" ^ "rgh" ^ "!!!!";`

ML: constructor types - lists

- Syntax: `[ obj1, obj2, ... ]`
  
- Objects in a list must be homogenous (same type)
  
- Examples:
  - `[1,2,3] : int list -- a list of integers
  - `["dog","cat","moose"] : string list -- a list of strings
  - `[1.0,2.0,3.0] : real list -- a list of reals
  - `[(1,"a"), (3,"bc"), (7,"efg") ] : (int * string) list
  - The empty list is written `[]` or `nil`

- Operations:
  - `@` operator is used to concatenate two lists of the same type
  
- Examples:
  - `2 :: [3,4]` returns `[2,3,4]`
  - `hd returns the first element of a list`
  
- Examples:
  - `hd [1,2,3] returns 1`
ML: constructor types - functions

- Anonymous functions
  - E.g.: `fn n => n*20(n),`
- The following declarations are identical:
  - `fun fn x => 2*x,`
  - `val fn = fn x => 2*x,`
- ML figures out the input and/or output types for simple expressions, constant declarations, and function declarations
  - Type checking requires that type expression of functions and their arguments match, and that type expression of context match result of function.
  - If the default isn’t what you want, you can specify the input and output types e.g.: `fun divideBy2 (y : real) = y / 2.0,`
- What is this doing?
  - `fun foo (m, n) = if m > n then [ ] else m :: foo(m+1, n);`
    - `foo(1,6); // [1,2,3,4,5,6]`

Examples:

- Factorial (n!)
  - `fun fact n = if n = 0 then 1 else n * fact(n-1);`
- List reverse
  - `fun reverse L = if L = nil then nil else reverse(tl L) @ [hd L];`
    - `fun 4 = 4 * (fact 3) = 4 * 3 * (fact 2) = 4 * 3 * 2 * (fact 1) = 4 * 3 * 2 * 1 * (fact 0) = 4 * 3 * 2 * 1 * 1 = 24`
    - `reverse [1,2,3] = reverse[2,3] @ 1 = reverse[3] @ 2 @ 1 = reverse[] @ 3 @ 2 @ 1 = [3,2,1]`

ML: local environment using let

- Recall syntax
  - `fun <func-name> <input-param> = <expression>;`
- Functions without parameters
  - E.g.: `fun message () = "hello world";`
    - `val message = fn () = "hello world";
- Functions with more than one parameter
  - E.g.: `fun birthday = (date = (1, "Jan",1900));`
    - `val birthday = fn : int * string * int = bool`
- Functions returning more than 1 result
  - E.g.: `fun quotient (x,y) = (x div y), (x mod y));`
    - `val quotient = fn : int * int => int * int;
      val (quot, rem) = quotient(x,y);`

ML: local environment using let

- Syntax
  - `let variable1 = expression1;`
    - `val variable1 = expression1;`
  - `in expression2 end;`
- Let allows declarations to be used in expressions
- Example:
  - Compute hundredth power of a number
    - `fun hundredthPower (n : real) = let`
      - `val four = n * n * n * n;`
      - `val twenty = four * four * four * four * four * four * four * four * four * four * twenty * twenty * twenty * twenty;`
    - `end;`
ML: pattern matching

- Syntax
  - \[ \text{fun} \ <\text{func}> <\text{pattern}> = <\text{expression}> \]
  - \[ \text{fun} \ <\text{func}> <\text{pattern}1> = <\text{expression}1> \]
  - \[ \ldots \]
  - \[ \text{fun} \ <\text{func}> <\text{pattern}n> = <\text{expression}n> \]

- Define a function by a series of equations, LHS is a pattern.
  - Always put the most specific pattern first
  - ML interpreter will use the first equation whose LHS matches

- Example:
  - Fibonacci function \(a_n = a_{n-1} + a_{n-2} \) \(0, 1, 1, 2, 3, 5, 8, 13, 21, \ldots\)
  - \[ \text{fun} \ fib \ n = \]
    - if \( n = 0 \) then 0
    - else if \( n = 1 \) then 1
    - else \( \text{fib}(n-1) + \text{fib}(n-2) \)

- Pattern matching is powerful:
  - Allows the programmer to see the arguments
  - No more hd's and tl's sprinkled all over the place

ML: pattern matching – cont’d

- Patterns may consist of constants (integers, true, false, …), tuples, and variables. Arithmetic or logical expressions are invalid.
  - E.g. \[ \text{fun} \ wrong(x,y) = \ldots \]

- No duplicates in patterns
  - E.g. \[ \text{fun} \ \text{wrong_equal}(x,y) = \text{true} \]
  - \[ \text{fun} \ \text{wrong_equal}(x,y) = \text{false} \]

- Pattern matching with wild cards
  - E.g. \[ \text{fun} \ \text{first}(x,\_\_\_) = x; \]
  - Matches anything like a variable. Binds nothing.
  - Avoid need to name every pattern

- ML does extensive pattern checking
  - E.g. \[ \text{fun} \ \text{reverse}(\_) = \text{reverse}() @ [\_]; \]
  - Warning: match nonexhaustive

ML: pattern matching – cont’d

- Examples:
  - Sum all the elements in a list of integers
    - \[ \text{fun} \ \text{listsum} L = \]
      - if \( L = \text{nil} \) then 0
      - else \( \text{hd} L + \text{listsum} t L \)
  - \[ \text{fun} \ \text{listsum} L = \]
    - if \( \text{null} L \) then 0
    - else \( \text{hd} L + \text{listsum} t L \)
  - \[ \text{fun} \ \text{listsum} L = \]
    - if \( \text{null} L \) then 0
    - else \( \text{hd} L + \text{listsum} t L \)
  - Reversing a list
    - \[ \text{fun} \ \text{reverse} L = \]
      - if \( L = \text{nil} \) then \( \text{nil} \)
      - else \( \text{reverse} t L @ [\text{hd} L] \)
  - Return first \( n \) elements of a list
    - \[ \text{fun} \ \text{take} ([], I) = [] \]
    - \[ \text{take} (h::t, I) = \]
      - if \( I > 0 \) then \( \text{take} (t, I-1) @ [h] \)

ML: pattern matching – cont’d

- How ML matches patterns?

- Diagram: 
  - \[ \text{fun} \ x . y . z (w) \]
  - \[ \ldots \]
  - \[ \text{fun} \ (1,2,3,4) \]
  - \[ \ldots \]
  - \[ \text{fun} \ ([1,2,3,4],5) \]

- 3
- 4
- nil
ML: Read-Evaluate-Print Cycle

- The system response \texttt{val it =} indicates that the built-in name \texttt{it} always holds the result of the last evaluated command.

Midterm Review: introduction

- Abstraction levels of programming languages
  - Machine language
  - Assembly language
  - High-level

- Language translation:
  - Compilation
  - Interpretation

- Language Paradigms:
  - Imperative
  - Object-oriented
  - Functional
  - Logic-based

ML: Read-Evaluate-Print Cycle – cont’d

Midterm Review: language spec.

- Language Specification:
  - Syntax vs Semantics
  - Semantics-informal descriptions

- Context-free grammars:
  - How are CFG’s descriptions of languages?
  - Derivations
  - Parse trees
  - Ambiguity
  - Fixing an ambiguous grammar:
    - Introduce definitions
    - Impose associativity/precedence
    - Inherently ambiguous grammar
Consider the following BNF grammar for a new language called Cork:

```
<sequence> ::= <statement> | <sequence> <statement>
<statement> ::= <assignment>  | <alternation> | <iteration>
<assignment> ::= <variable> = <constant>
<alternation> ::= if <variable> <sequence> fi | if <variable> else <sequence> fi
<iteration> ::= do <variable> <sequence> od
<branch> ::= A | B | C | ... | Z
<constant> ::= 0 | 1 | 2 | ... | 9
```

For each of the following Cork code fragments, indicate whether a Cork parser would produce a syntax error or not. Explain how the parser will determine that.

- a) if A = 1 fi
  - Answer: syntax error
- b) A = 1
  - Answer: valid
- c) A = B = C = 3
  - Answer: syntax error
- d) do if A = 1 B = 2 else C = 3 fi od
  - Answer: syntax error
- e) if A do Z if B C = 3 fi od fi
  - Answer: valid

Using the following grammar:

```
<assign> => <id> = <expr>
<id> => A | B | C
<expr> => <id> + <expr> | <id> * <expr> | ( <expr> ) | <id>
```

Show a parse tree and a leftmost derivation for

```
A = A * (B + C A)
```

Characteristics

- Basic constructs
- Lists
- Anonymous functions
- map, reduce, for-each...
- let, let*
Midterm Review: Scheme questions

• What is a high-order function? Give an example.

• What does it mean for a language to be referentially transparent?

• What is the manifest interface principle?

• Suppose we have the following definitions:
  
  (define w 42)
  (define x 17)
  (define y (lambda (n) (+ n w)))
  (define z (lambda (x) (+ x 2)))

  What is the result of the following expressions?

  \[
  \Rightarrow (map (lambda (x) (z x)) '(1 2 3 4))
  \]
  \[
  \Rightarrow (let ((w (map y '(1 2 3 4)))) w)
  \]

  Show a trace for (within? 'c '(a (b c))) Your trace should explain the sequence of calls in the recursion tree.

Midterm Review: Scheme questions

• The following function finds if a particular item is found anywhere in an expression:

  \[
  (define within? (lambda (item lst)
      (or (equal? item lst)
          (and (pair? lst)
               (or (within? item (car lst))
                   (within? item (cdr lst))))))
  \]

  => (within? 'b '(a b (c) d))
  #t

  => (within? 'd '(a b c))
  #f