Types

- **What is a type?**
  - An equivalence class of objects/values
  - **Denotational view:** a type is a set (of values): Pascal
    - `type weekday = (sun, mon, tue, wed, thu, fri, sat);`
    - `Weekday = {sun, mon, tue, wed, thu, fri, sat}`
  - **Constructive view:** a type is the result of an expression consisting of primitive types operated upon by type constructors: Ada
    - `type computer is record
      serial : array (1..10) of integer;
      age: integer;
    end record;`
  - **Abstraction view:** a type is an interface, providing a set operations on objects of the type; an abstract data type (ADT):
    - Pascal: `pred(), succ(), <, =, >`
    - Class declaration
Types (cont.)

- What has a type?
  - Literals 1.25 ‘abc’
  - Variables `var x: integer;`
  - Expressions `x:int + y` ML type `int`
    - Induced by types of variables, literals, operators (casting ops included), and any implicit coercion (conversion) rules
  - Objects `Stack<int> s;`
  - Functions `-fun area(r) = 3.141582818*r*r;`
    `val area = fn : real -> real;`
  - References `int& x;` `x: ref int;`
  - Pointers `int i = 3; int& r = i; int* p = &r;`
Type System

• Type definition rules
  ■ Declaration (naming): introduce new name & bind to scope
  ■ Definition (description):
    ◦ Primitive: booleans, characters, integers, fixedpoint, floating point
    ◦ Enumeration: Ada type weekday is (sun, ... , sat);
    ◦ Subtype: subtype weekend is weekday range sat..sun;
    ◦ Composite: record, union, array, reference, list (type “operators”)
    ◦ Function: C++: int max(int a, int b){return a>b?a:b;}
    ◦ Derived: Ada: type mass is new REAL;

• Type equivalence rules
  ■ Name equivalence
    ◦ Each definition a new type
    ◦ Equivalent only if declared as same primitive or pre-defined type
      Ada distinct types: a, b: array(1..10) of BOOLEAN;
  ■ Declaration equivalence
    ◦ Same declaration implies same type (example above is dec. equiv.)
Type System (cont.)

- Structural Equivalence: have same type-operator expression
- **Type compatibility rules**
  - Argument/parameter compatibility; assignment compatibility
  - Types might be different but compatible; rules differ widely
    - Ada: a subtype is compatible with a supertype & arrays of same size & base type are compatible
    - C: short int s; unsigned long int l; ... ; s = l;
    - C: void* p; int* q; ... ; q = p;
  - *Coercion*: implicit type conversion defined by language (≠ cast)
- **Type Checking**: verifying a program adheres to type compatibility rules (e.g. lint a type checker for a weakly typed C)
  - Strong typing: prohibits an op when incompatibility exists
    - Ada strongly typed. Bliss untyped. ANSI C in middle
  - Static type checking: compile time (Ada, C++)
  - Dynamic type checking: late binding (Lisp, Scheme, Smalltalk)
Type System (cont.)

• Type Inference Rules
  - Rules for typing an expression given the types of its components
    - Type of \( x = y \); is type of \( x \)
    - Type of \( b?a:b \) is the (common) type of \( a, b \) etc, etc
    - Ada: “con” & “cat” (both array[1..3] of char) returns array[1..6] of char
  - Subranges
    - \( x:INTEGER\ range 0..40; \ y:INTEGER\ range 10..20; \) type of \( x + y \) ?
  - Can be complex, and involve coercion
    - Recall PL/I example with fixed bin and fixed dec operands
  - Some inferences impossible at compile time
  - Inference is a kind of “evaluation” of expressions having coarse values; types have their own arithmetic
Polymorphism

- A polymorphic subroutine is one that can accept arguments of different types for the same parameter
  - \( \text{max}(x, y) \{ \text{max} = x > y \? x : y \} \) could be reused for any type for which \( > \) is well-defined

- A polymorphic variable (parameter) is one that can refer to objects of multiple types. ML: \( x : 'a \)

- True (or “pure”) polymorphism always implies code reuse: the same code is used for arguments of different types.

- What polymorphism is not:
  - Not overloading.
  - Not generics.
  - Not coercion.
  - All 4 aim at off-loading effort from programmer to translator, but in different ways
Polymorphism (cont.)

• Overloading
  - An overloaded name refers to several distinct objects in the same scope; the name’s reference (denotation) is resolved by context. Unfortunately sometimes called “ad hoc polymorphism”!
  - C++
    ```cpp
    int j, k; float r, s;
    int max(int x, int y) { return x <= y ? y : x }
    float max(float x, float y) { return y > x ? y : x }
    ...
    max(j, k); // uses int max
    max(r, s); // uses float max
    - Even constants can be overloaded in Ada:
      ```ada
      type weekday is (sun, mon, ...);
      type solar is (sun, merc, venus, ...);
      planet: solar; day: weekday;
      day := sun; planet := sun; -- compatible
      day := planet; -- type error
    ```
Polymorphism (cont.)

- **Generic** subroutines
  - A generic subroutine is a syntactic *template* containing a type parameter that can be used to generate different code for each type instantiated
  - Ada
    ```
generic
    type T is private;
    with function "<="(x, y : T) return Boolean;

    function max(x, y : T) return T is
    begin
        if x <= y then return y;
        else return x;
    end if;
end max;

function bool_max is new max(BOOLEAN, implies);
function int_max is new max(INTEGER, "<=");
```
Polymorphism (cont.)

- **Coerced subroutine arguments**
  - A coercion is a built-in compiler conversion from one type to another
  - Fortran
    
    ```fortran
    function rmax(x, y)
    real x
    real y
    rmax = x
    if (y .GT. x) rmax = y
    return
    end
    ```
  - In `k = rmax(i, j)` causes args to be coerced to floating point & return value truncated to integer
  - Although *same code* is used for both arg types, this is not true polymorphism
Kinds of Polymorphism

- *Pure polymorphism*: a single subroutine can be applied to arguments of a variety of types
  - *Parametric* polymorphism: the type value is passed explicitly as an argument. There is a type called `type` in CLU:
    ```
sorted_bag = cluster[t: type] is create, insert, ...  
  where t has lt, eq: proctype(t,t) returns (bool);  
...
wordbag := sorted_bag[string]; -- create cluster
wb: wordbag := wordbag$create(); -- instance
...
wordbag$insert(wb, word); -- mutate instance
```
Kinds of Polymorphism (cont.)

- **Type variable** polymorphism: a type signature with type variables is derived for each subroutine that is as general as possible (unification). An applied subroutine has its type variables instantiated with particular types.

```ml
- fun length(nil) = 0
-   | length(a ::: y) = 1 + length(y);
val length = fn : 'a list -> int

- val a = ["a", "b", "c"];
val a = ["a","b","c"] : string list

- length(a);
val it = 3 : int

- val b = [1,3,5,7,21,789];
val b = [1,3,5,7,21,789] : int list

- length(b);
val it = 6 : int
```
Kinds of Polymorphism (cont.)

- `val d = [35, 3.14];`
  
  std_in: 0.0-0.0 Error: operator and operand don't agree (tycon mismatch)
  
  operator domain: int * int list  operand: int * real list in
  expression: 35 :: 3.14 :: nil

- `val e = [3.14, 2.71828, 1.414];`
  
  val e = [3.14, 2.71828, 1.414] : real list

- `length(e);`
  
  val it = 3 : int
Kinds of Polymorphism (cont.)

- **Late Binding** polymorphism: deferral of type checks to run-time allows polymorphic code to be written once and used with different types

```scheme
caslon> cat length.scm
;;; length - return length of a list
(define length
  (lambda (x)
    (if (null? x)
      0
      (1+ (length (cdr x))))))
caslon> scheme
1 ]=> (load "length.scm")

1 ]=> (define a (list 2 7 1 8 28 1 8))
A
1 ]=> (length a)
7
1 ]=> (define a (list "foo" "baz" "snafu"))
A
1 ]=> (length a)
3
```
Inheritance polymorphism: one class method executed on objects of distinct subclasses; common code is “inherited”.

Ex: in Little Smalltalk the subclasses of Magnitude are
Kinds of Polymorphism (cont.)

- An implementation of class Magnitude

```
Class: Magnitude
Instance variables:
Instance methods:

<n  ^self implementedBySubclass
=\n  ^self implementedBySubclass
<=\n  ^(self < n) or: (self = n)
>n  ^(self <= n) not
>=\n  ^(self < n) not
between: min and: max
  ^ (min <= self) and: (self <= max)
max: n
  (self > n)
    ifTrue: [^self]
    ifFalse: [^n]
min: n
  (self < n)
    ifTrue: [^self]
    ifFalse: [^n]

^name = value of name
```
Kinds of Polymorphism (cont.)

- Invocation with different classes (types)
  - Char: \(x\) between: \$a\ and: \$z\)
    - If \(x\) is a Char, method is not found at Char. Search proceeds up to superclass Magnitude. \(\wedge (\$a \leq x) \& (x \leq \$z)\) invoked. First \(\leq x\) sent to \$a\ of class Char, where method \(\leq\) is not found, ..., in Magnitude invoke \(\wedge (\$a < x) \) or \(\$a = x\). This sends message \(< x\) to \$a\ and this method is found in class Char. Suppose \(x\) is actually \$b\. Eventually the ob \text{true}\ is sent message \text{or: false}, resulting in value \text{true}. So result of \(\$a \leq x\) is now effectively determined at \$a, returning a \text{true} ob. ... Eventually, by similar process this \text{true} will be sent and:\text{true}, so it returns itself.
  - Point: \(x\) between: \(2@4\) and: \(5@6\)

- All use same code!
ML: Strong Typing & Polymorphism

lec> sml
Standard ML of New Jersey, Version 110.0.6, October 31, 1999
val use = fn : string -> unit
- fun succ n = n+1;
val succ = fn : int -> int
- succ "zero";
stdIn:7.1-7.12 Error: operator and operand don't agree [tycon mismatch]
  operator domain: int operand: string in expression: succ "zero"
- succ 3;
val it = 4 : int
- fun add(x,y) = x + y;
val add = fn : int * int -> int
- add 3 5;
stdIn:9.1-9.8 Error: operator and operand don't agree [literal]
  operator domain: int * int operand: int in expression: add 3
- add (3,5);
val it = 8 : int
- fun I x = x;
GC #0.0.0.0.1.5: (0 ms)
val I = fn : 'a -> 'a
ML (cont.)

- fun self = (x x);

stdIn:11.14-11.20 Error: operator is not a function [circularity]
  operator: 'Z in expression: x x
- fun apply f x = (f x);
val apply = fn : ('a -> 'b) -> 'a -> 'b
- apply succ 7;
val it = 8 : int
- add 3;

stdIn:13.1-13.6 Error: operator and operand don't agree [literal]
  operator domain: int * int operand: int in expression: add 3
- fun plus x y = x + y;
val plus = fn : int -> int -> int
- plus 3;
val it = fn : int -> int
- plus 3 5;
val it = 8 : int
- val add3 = plus 3;
val add3 = fn : int -> int
- add3 5;
val it = 8 : int
- fun K x y = x;
val K = fn : 'a -> 'b -> 'a
- K I;

stdIn:19.1-19.4 Warning: type vars not generalized because of
value restriction are instantiated to dummy types (X1,X2,...)
val it = fn : ?.X1 -> ?.X2 -> ?.X2
- K I 3;

stdIn:20.1-20.6 Warning: type vars not generalized because of
value restriction are instantiated to dummy types (X1,X2,...)
val it = fn : ?.X1 -> ?.X1
- K I 3 24;
val it = 24 : int
- K I "foo" 24;
val it = 24 : int
- K succ;

stdIn:23.1-23.7 Warning: type vars not generalized because of
value restriction are instantiated to dummy types (X1,X2,...)
val it = fn : ?.X1 -> int -> int
- K succ 3;
val it = fn : int -> int
- K succ 3 15;
val it = 16 : int
- ^D
ML: Polymorphic Reference Types

- (* can have refs to variable types *)
- val a = ref 7;
val a = ref 7 : int ref
- val b = ref 11;
val b = ref 11 : int ref
- !a;
val it = 7 : int
- !b;
val it = 11 : int
- fun swap (x, y) =
  let val temp = !x
  =  in x := !y; y := temp
  =  end;
val swap = fn : 'a ref * 'a ref -> unit
- swap(a,b);
val it = () : unit
- !a;
val it = 11 : int
- !b;
val it = 7 : int
ML: Reference Types (cont.)

- val c = ref true;
val c = ref true : bool ref
- val d = ref false;
val d = ref false : bool ref
- swap(c,d);
val it = () : unit
- !c;
val it = false : bool
- !d;
val it = true : bool
- swap(a,c);

std_in:29.1–29.9 Error: operator and operand don't agree (tycon mismatch)
  operator domain: int ref * int ref operand: int ref * bool ref in
  expression: swap (a,c)
ML: Static Typing

<table>
<thead>
<tr>
<th>opu&gt; scheme</th>
</tr>
</thead>
<tbody>
<tr>
<td>1 ]=&gt; ;;;;;; a function acceptable to Scheme but not type-correct in ML</td>
</tr>
<tr>
<td>(define applyto</td>
</tr>
<tr>
<td>(lambda (f) (cons (f 3) (f &quot;hi&quot;))) )</td>
</tr>
<tr>
<td>APPLYTO</td>
</tr>
<tr>
<td>1 ]=&gt; (applyto (lambda (x) x) )</td>
</tr>
<tr>
<td>(3 . &quot;hi&quot;)</td>
</tr>
<tr>
<td>1 ]=&gt; (applyto (lambda (x) (cons 'glurg x)))</td>
</tr>
<tr>
<td>((GLURG . 3) GLURG . &quot;hi&quot;) ;;; ((GLURG . 3) (GLURG . &quot;hi&quot;))</td>
</tr>
<tr>
<td>opu&gt; sml</td>
</tr>
<tr>
<td>- (* ML type-inference algorithm unwilling to accept APPLYTO *)</td>
</tr>
<tr>
<td>- val applyto = fn f =&gt; ( f(3), f(&quot;hi&quot;) );</td>
</tr>
<tr>
<td>std_in:11.23-11.39 Error: operator and operand don't agree (tycon mismatch)</td>
</tr>
<tr>
<td>operator domain: int operand: string in expression: f (&quot;hi&quot;)</td>
</tr>
<tr>
<td>- (* Below there are two insances of I x = x that take distinct types. = Why?? *)</td>
</tr>
<tr>
<td>- let fun I x = x in ( I(3), I(&quot;hi&quot;) ) end;</td>
</tr>
<tr>
<td>val it = (3,&quot;hi&quot;) : int * string</td>
</tr>
</tbody>
</table>
ML & \( \lambda \)-Calculus

lec> script skk
Script started on Tue Feb 19 09:01:20 200
lec> sml
Standard ML of New Jersey, Version 110.0.6, October 31, 1999
val use = fn : string -> unit
- fun I x = x;
val I = fn : 'a -> 'a
- fun add x y = x + y;
val add = fn : int -> int -> int
- fun add1 z = add 1 z;
val add1 = fn : int -> int
- add1 10;
GC #0.0.0.0.1.4: (0 ms)
val it = 11 : int
- fun twice f x = f (f x);
val twice = fn : ('a -> 'a) -> 'a -> 'a
- twice add1 5;
val it = 7 : int
- fun mul2 x = 2*x;
val mul2 = fn : int -> int
- mul2 10;
val it = 20 : int
ML & $\lambda$-Calculus (cont)

- `fun S x y z = x z (y z);`
  `val S = fn : ('a -> 'b -> 'c) -> ('a -> 'b) -> 'a -> 'c`
- `S add mul2 5;`
  `val it = 15 : int`
- `S add I 5;`
  `val it = 10 : int`
- `fun K x y = x;`
  `val K = fn : 'a -> 'b -> 'a`
- `fun T z = S K z;`
  `val T = fn : ('a -> 'b) -> 'a -> 'a`
- `fun V w = T K w;`
  `val V = fn : 'a -> 'a`
- `V 3;`
  `val it = 3 : int`
- `V 10;`
  `val it = 10 : int`
- `V 20;`
  `val it = 20 : int`
- `S K K 10;`
  `val it = 10 : int`
- `S K K 21;`
  `val it = 21 : int`
ML & $\lambda$-Calculus (cont)

- $S \ I \ I$
  
  stdIn: 26.1-26.6 Error: operator and operand don't agree [circularity]
  
  operator domain: ('Z -> 'Y) -> 'Z operand: ('Z -> 'Y) -> 'Z -> 'Y in
expression: (S I) I

- $S \ K \ I$
  
  stdIn: 27.1-27.6 Warning: type vars not generalized because of
value restriction are instantiated to dummy types (X1,X2,...)

```
val it = fn : ?.X1 -> ?.X1
- S K I 1;
val it = 1 : int
- S K I 21;
val it = 21 : int
- val T = S K;
stdIn: 31.1-31.12 Warning: type vars not generalized because of
value restriction are instantiated to dummy types (X1,X2,...)
val T = fn : (?.X1 -> ?.X2) -> ?.X1 -> ?.X1
- val U = S K add1;
val U = fn : int -> int
- U 21;
val it = 21 : int
- U 234;
val it = 234 : int
- ^D
```