15-150 Spring 2018
Lecture 9
Types and Polymorphism
Span of Msort

fun Msort Empty = Empty 
  | Msort (Node(t1, x, t2)) = 
    Ins (x, Merge(Msort t1, Msort t2))

For a balanced tree with $d > 0$

$$S_{\text{Msort}}(d) \leq c_4 + \max(S_{\text{Msort}}(d-1), S_{\text{Msort}}(d-1)) + S_{\text{Merge}}(d_1, d_2) + S_{\text{Ins}}(d_3)$$

depths of trees produced by Msort  depth of tree produced by merge
Span of Msort

For a balanced tree with \( d > 0 \)

\[
S_{\text{Msort}}(d) \leq c_4 + \max(S_{\text{Msort}}(d-1), S_{\text{Msort}}(d-1)) + S_{\text{Merge}}(d-1,d-1) + S_{\text{Ins}}(2d -2) \\
\leq S_{\text{Msort}}(d-1) + k \cdot d^2
\]

\( S_{\text{Msort}}(d) \) is \( O(d^3) \)
• The balance assumptions are *not* realistic!

• But we could design a *rebalancing* function...

  ```
  fun Msort Empty = Empty
  | Msort (Node(t1, x, t2)) =
      Rebalance(Ins (x, Merge(Msort t1, Msort t2)))
  ```

• Or implement an *abstract type* of *balanced trees* with *Ins and Merge* functions that *preserve balance*
Types in Programming

• Program organization and documentation

• Making sure bit sequences in memory are interpreted correctly

• Providing information to the compiler
Goals for today

• Apply type-checking rules for ML expressions
• State what it means for a function to be polymorphic
• Determine the most general type for a given expression
• Define parameterized datatypes and use them correctly
Type safety

A static check provides a runtime guarantee (modulo termination)

<table>
<thead>
<tr>
<th>static guarantee</th>
<th>runtime guarantee</th>
</tr>
</thead>
<tbody>
<tr>
<td>$e$ has type $t$</td>
<td>if $e \Rightarrow v$ then $v : t$</td>
</tr>
<tr>
<td>$d$ declares $x : t$</td>
<td>if $d \Rightarrow [x : v]$ then $v : t$</td>
</tr>
</tbody>
</table>
Type Analysis

• There are *syntax-directed* rules for figuring out when \( e \) has type \( t \).

  \( e \) is well-typed, with type \( t \), if and only if this is *provable* from these rules.

We say “\( e \) has type \( t \)” or write “\( e : t \)”, possibly with assumptions like “\( x : \text{int} \) and \( y : \text{int} \)”
Polymorphism
fun rev(L : int list): int list =
  case L of
    [] => []
    |(x::R) => (rev R) @ [x]
Polymorphic \texttt{rev}

\begin{verbatim}
\textbf{fun} rev(L: 'a list): 'a list = 
  \textbf{case} L of [] => [] 
      | (x::R)  => (rev R) @ [x]
\end{verbatim}

In the scope of this declaration you can use \texttt{rev} with any list as an argument.
Parameterized datatypes

datatype 'a tree = Empty
  | Node of 'a tree * 'a * 'a tree

introduces a type constructor tree
  and polymorphic value constructors

  Empty: 'a tree
  Node: 'a tree * 'a * 'a tree -> 'a tree
Example

(* trav : 'a tree -> 'a list
  REQUIRES: true
  ENSURES: trav(t) returns a list consisting of the elements in t, in the same order as seen during an in-order traversal of t. *)

fun trav(Empty: 'a tree) : 'a list = []
| trav(Node(t1, x, t2)) = (trav t1) @ x :: (trav t2)
zip

(* zip : 'a list * 'b list -> ('a * 'b) list
REQUIRES: true
ENSURES:  zip([a1,a2,...,an],[b1,b2,...,bm]) ==
[(a1,b1), (a2,b2), ..., (ak,bk)]
with k = min(n,m) >= 0. *)

fun zip ([] : 'a list, B : 'b list) : ('a * 'b) list = []
  | zip (A, []) = []
  | zip (a::A, b::B) = (a,b)::zip(A,B)
options

datatype 'a option = NONE | SOME of 'a
lookup

(* lookup : ('a * 'a -> bool) * 'a * ('a * 'b) list -> 'b option
   REQUIRES: true
   ENSURES: lookup(eq, x, L) returns SOME(b) of the leftmost (a,b) in L for which eq(x,a) returns true, if there is such an (a,b); returns NONE otherwise. *)

fun lookup(_: 'a * 'a -> bool, _:'a, []: ('a * 'b) list): 'b option = NONE
    | lookup(eq, x, (a,b) :: L) = if eq(x,a) then SOME(b)
                                else lookup(eq, x, L)
Type Inference
Most General Types

Every well-typed expression has a **most general** type

t is a most general type for e if and only if every instance of t is a type for e and every type for e is an instance of t

ML determines if your code is well-typed and infers most general types, using a syntax-directed algorithm
Examples

1. fun f x = x + 1
2. fun g (x, y) = x + y
3. fun outer (x, y, z) = 2 * (x + z)
4. fun f x = f x
5. fun f x = f (f x)
6. fun id x = x
7. id id 42
Interesting examples

```haskell
let
  val x = rev []
in
  (true:: x, 7 :: x)
end
```

stdIn:8.47-9.62 Error: operator and operand don't agree [literal]
operator domain: int * int list
operand: int * bool list
in expression:
  7 :: x

If you used [] instead of rev [] there would be no error.
**lookup**

(* lookup : ('a * 'a -> bool) * 'a * ('a * 'b) list -> 'b option  
  REQUIRES: true  
  ENSURES: lookup(eq, x, L) returns SOME(b) of the 
    leftmost (a,b) in L for 
    which eq(x,a) returns true, if there is 
    such an (a,b);  
    returns NONE otherwise. 
  *)

fun lookup(_: 'a * 'a -> bool, _ :'a, []: ('a * 'b) list): 'b option = NONE  
   | lookup(eq, x, (a,b) :: L) = if eq(x,a) then SOME(b)  
   else lookup(eq, x, L)

In fact, if we omit the type annotations in our spec ML derives the following type 
lookup : ('a * 'b -> bool) * 'a * ('b * 'c) list -> 'c option