Today

- We have learned an interesting subset of the ML expression language
- But we have been really informal about some aspects of the type system:
  - Type inference (what types do bindings implicitly have)
  - Type variables (what do \( \tau a \) and \( \tau b \) really mean)
  - Type constructors (why is \( \texttt{int} \texttt{list} \) a type but not \( \texttt{list} \texttt{int} \))
- Note: Type inference and parametric polymorphism are separate concepts that end up intertwined in ML. A different language could have one or the other.

Type Inference

Some languages are untyped or dynamically typed.

ML is \textit{statically typed}; every binding has one type, determined during type-checking (compile-time).

ML is \textit{implicitly typed}; programmers rarely need to write the types of bindings.

The type-inference question: Given a program without explicit types, produce types for all bindings such that the program type-checks, or reject (only) if it is impossible.

Whether type inference is easy, hard, or impossible depends on details of the type system: Making it more or less powerful (i.e., more programs typecheck) may make inference easier or harder.

ML Type Inference

- Determine types of bindings in order (earlier first) (except for mutual recursion)
- For each \texttt{val} or \texttt{fun} binding, analyze the binding to determine necessary facts about its type.
- Afterward, use \textit{type variables} (e.g., \( \tau a \)) for any unconstrained types in function arguments or results.
- Some extra details for type variables and references we’ll mention later.

Amazing fact: For the ML type system, "going in order" this way never causes unnecessary rejection.
Example 1

fun f x = 
  let val (y, z) = x in
    (Real.abs y) + z
  end

Example 2

fun sum lst = 
  case lst of
    [] => 0
  | hd::tl => hd + (sum tl)

Example 3

fun compose (f,g,x) = f (g x)

Comments on ML type inference

- If we had subtyping, the "equality constraints" we generated would be unnecessarily restrictive.
- If we did not have type variables, we would not be able to give a type to compose until we saw how it was used.
  - But type variables are useful regardless of inference.
- Inference is why the following aren't really equivalent:
  - let val x = e1 in e2 end
  - (fn x => e2) e1
E.g., let's try e2 = (x 0, x "foo") and something simple for e1 like fn y => y:
  - let val x = (fn y => y) in (x 0, x "foo") end
  - (fn x => (x 0, x "foo")) (fn y => y)
The latter gives a type error ...
Parametric polymorphism

Fancy words for "forall-types". Coming to next version of Java, C#,
VB, etc. Sometimes called generics. A bit like C++ templates if C++
didn’t have operator-overloading.

In principle, just two new kinds of types:

\[ \text{ty} ::= \forall t \mid \text{int} \mid \text{string} \mid \text{bool} \mid t \to t \mid \{ \text{li:ti}, \ldots, \text{ln:tn} \} \]

\[ \text{t} ::= \text{dtname} \mid \text{ty} \mid \text{forall } \forall \text{tv. } t \]

Given an expression of type \( \forall \text{tv. } t \), we can instantiate it at
type \( t_2 \) to get an expression of type \( t \) with \( \forall \text{tv} \) replaced by \( t_2 \)

Example: We can instantiate
\[ \text{forall } \forall \text{a. forall } \forall \text{b. (\text{\textquoteleft a \times \textquoteleft b}) \to (\text{\textquoteleft b \times \textquoteleft a})} \]

with \text{string for } \forall \text{a} \text{ and int->int for } \forall \text{b} \text{ to get}
\[ \text{string * (int->int)) \to ((int->int) * string) \]

ML-style polymorphism

The ML type system is actually more restrictive:

- "forall" must appear "all the way on the outside-left"
- So it’s implicit; no way to write the words "for all"

Example: \( (\forall \text{a } \times \forall \text{b}) \to (\forall \text{b } \times \forall \text{a}) \)

forall \( \forall \text{a. forall } \forall \text{b. (\text{\textquoteleft a } \times \text{\textquoteleft b}) } \to (\text{\textquoteleft b } \times \text{\textquoteleft a}) \)

Non-example: There’s no way to have a type like
\[ \text{(forall } \forall \text{a. } \forall \text{a } \to \text{int) } \to \text{int} \]

Versus Subtyping

Compare

fun \( \text{swap } (x,y) = (y,x) \) \((\forall \text{a } \times \text{\textquoteleft b}) \to (\text{\textquoteleft b } \times \text{\textquoteleft a}) \)

with

\[
\text{class Pair } \{ \text{Object x; Object y; } \ldots \} \nu\\
\text{Pair swap(Pair x,y) } \{ \text{return new Pair(y,x); } \}
\]

ML wins in two ways (for this example):

- Caller instantiates types, so doesn’t need to cast result
- Callee cannot return a pair of any two objects.

Containers

Parametric polymorphism (forall types) are also the right thing for
containers (lists, sets, hashtables, etc.) where elements have the same
type.

Example: ML lists

val \( : : \) \((\forall \text{a } \times \text{\textquoteleft a list\textquoteleft}) \to \text{\textquoteleft a list\textquoteleft} \times \text{\textquoteleft a list\textquoteleft}) \to \text{\textquoteleft a list\textquoteleft} \)

val \( \text{map } : ((\text{\textquoteleft a } \to \text{\textquoteleft b}) \times \text{\textquoteleft a list\textquoteleft}) \to \text{\textquoteleft b list\textquoteleft} \)

val \( \text{sum } : \text{int list } \to \text{int} \)

val \( \text{fold } : ((\forall \text{b } \to \text{\textquoteleft b}) \to \text{\textquoteleft b}) \to \text{\textquoteleft b } \to \text{\textquoteleft a list\textquoteleft} \to \text{\textquoteleft b}\)

list is a type constructor, not a type; if \( \text{t} \) is a type, then \( \text{t list} \) is a
type.
User-defined type constructors

Language-design: don't provide a fixed set of a useful thing.
Let programmers declare type constructors.

Examples:

```
datatype 'a non_mt_list = One of 'a
            | More of 'a * ('a non_mt_list)

datatype 'a rope = Empty
            | Cons of 'a * ('a rope)
            | Rope of ('a rope) * ('a rope)
```

You can have multiple type-parameters (not shown here).
And now, finally, everything about lists is syntactic sugar!