OCaml

ML

- Stands for “Meta Language”
- Originally designed as metalanguage for LCF theorem prover
  (modern successors HOL/Isabelle)
- Robin Milner, 1978
- Standardized (SML): 1990
- Implementation: SML/NJ
- Basis library: 1997
- Fork: Caml (1985), OCaml (1996),
  F# (2002)

A rapid introduction to OCaml

`ocaml`: Interpreter similar to DrRacket's Interactions window.

`ocamlc`: Bytecode compiler
`ocamlopt`: Compiles to native code

Expressions

```
# 1 + 2;;
- : int = 3
```

Double semicolons are used to tell the `ocaml` interpreter to evaluate the expression. They are used much more infrequently in programs.

Note the inference of the type.

Declarations

```
# let x = 1;;
> val x : int = 1

# let y = 1.2;;
val y : float = 1.2

# x + 3;;
- : int = 4
```

```
# let f = fun x -> x + 1;;
val f : int -> int = <fun>

# let rec fact n =
    if n=0 then 1 else n * fact (n-1);;
val fact : int -> int = <fun>
```

```
# let sqr x = x * x;;
val sqr : int -> int = <fun>

# let sqrf x = x *. x;;
val sqrf : float -> float = <fun>
```

We can add type annotations as needed (ascription):

```
# let sqr (x : int) = x * x;;
val sqr : int -> int = <fun>
```

Ascription is useful in clarifying intent and debugging type errors.

ML will use type variables for polymorphic functions.

```
# let id x = x;;
val id : 'a -> 'a = <fun>

# let y = id 3;;
val y : int = 3
```

Functions in ML have exactly one parameter.

```
# let sumSqrs x y = x*x + y*y;;
val sumSqrs : int -> int -> int = <fun>

# let f = sumSqrs 4;;
val f : int -> int = <fun>
```

Tuples

```
# let x = (true, 'z');;
val x : bool * char = (true, 'z')
# let y = (4, x);;
val y : int * (bool * char) = (4, (true, 'z'))
# let w = (4, 5, 6);;
val w : int * int * int = (4, 5, 6)
```

Tuples are usually deconstructed using patterns.

```
# let fst (x,_) = x
val fst : 'a * 'b -> 'a = <fun>
```

There aren't 1-tuples, but the 0-tuple `()` comes in handy.

```
# ();;
- : unit = ()
```

`()` is the sole value of the type `unit`, the equivalent of `#<void>` in Racket.

Infix operators can be used in a prefix manner.

```
# (+) 3 4;;
- : int = 7
# (+) 3;;
- : int -> int = <fun>
```

New infix operators can be made from the characters used for built-in infix operators.

```
# let ( *+* ) x y = x*x + y*y;;
val ( *+* ) : int -> int -> int = <fun>
# 3 *+* 4;;
- : int = 25
```

Lists

```
# [1; 2; 3];;
- : int list = [1; 2; 3]
# 1 :: 2 :: 3 :: [];;
- : int list = [1; 2; 3]
# [1] @ [2; 3];;
- : int list = [1; 2; 3]
```

Lists can be deconstructed using `List.hd` and `List.tl`, or by patterns.

```
let rec merge lst1 lst2 =
 match (lst1, lst2) with
    ([], ns) -> ns
  | (ms, []) -> ms
  | (n::ns, (m::_ as mms)) when n < m
     -> n :: merge ns mms
  | (nns, m::ms)
     -> m :: merge nns ms
```

Local definitions

```
let rec split = function
   [] -> ([],[])
| [a] -> ([a],[])
| (a::b::cs) ->
    let (ms,ns) = split cs
    in  (a::ms, b::ns)
```

```
let rec msort = function
   [] -> []
| [x] -> [x]
| xs ->
     let (ms,ns) = split xs
     in let  ms' = msort ms
     in let  ns' = msort ns
     in merge ms' ns'
```

Type synonyms

```
type intpair = int * int;
```

(like `typedef` in C).

```
type suit = Heart | Spade | Diamond | Club
type value = Ace | Num of int | Jack | Queen |
King
type card = suit * value

type ('a,'b) tree =
  Empty
| Node of 'a * 'b * ('a,'b) tree * ('a,'b) tree
```

`tree` is a type constructor (as are `*`, `->`, `list`).

`Empty` and `Node` are data constructors (not functions).

```
let rec kmem x = function
  Empty -> false
| Node (y,_,lf,rt) ->
  x = y || kmem x lf || kmem x rt

val kmem : 'a -> ('a, 'b) tree -> bool = <fun>
```

Option types

This type is defined in the basis library:

```
type 'a option = None | Some of 'a
```

It is used when a value might not be produced.

```
let rec kmap x = function
  Empty -> None
| Node (y,v,lf,rt) ->
   if x = y then Some v
   else if x < y then kmap x lf
   else kmap x rt
```

Exceptions

```
exception DivByZero;;

let safeDiv = function
  (_,0)   -> raise DivByZero
  | (x,y) -> x / y

let quot = try safeDiv (3,0) with DivByZero -> 0
```

The handler must produce a value of the same type as the expression to which it is attached.

```
exception DivByZero of int * string

let safeDiv = function
    (x,0) -> raise (DivByZero(x,"bad!"))
  | (x,y) -> x / y

let quot =
 try safeDiv (3,0)
 with DivByZero(x,_) -> x
```

A similar mechanism is provided in Racket and as a standard library in R6RS Scheme.

Input/output

OCaml provides many I/O functions. We will mention only one here:

```
# Printf.printf "The answer is %d\n" 3;;
The answer is 3
- : unit = ()
```

Mutation

Reference types

Like Racket, Scheme, and Lisp, the ML languages are not pure, but mutation is used sparingly.

The data constructor `ref` provides the equivalent of boxes in Racket.

(In OCaml, references are a special case of mutable records.)

```
# let x = ref 5;;
val x : int ref = {contents = 5}
# !x;;
- : int = 5
# x.contents;;
- : int = 5
```

```
# x := 6;;
- : unit = ()
# !x;;
- : int = 6
# x.contents <- 7;;
- : unit = ()
# !x;;
- : int = 7
```

Expression sequencing is handled by single semicolon, which is an operator that expects a left operand of type `unit` and produces the value of its right operand.

Sequences are delimited by parentheses or by `begin` and `end`.

Records

```
# type point = {x : int; y : int};;
type point = { x : int; y : int; }
# let p = {x = 3; y = 4};;
val p : point = {x = 3; y = 4}
# p.x;;
- : int = 3
# let {y = yv} = p in yv;;
- : int = 4
```

```
# type grade = {name : string; mutable mark : int};;
type grade = { name : string; mutable mark : int; }
# let s = {name = "Toad"; mark = 49};;
val s : grade = {name = "Toad"; mark = 49}
# s.mark <- 100;;
- : unit = ()
# s;;
- : grade = {name = "Toad"; mark = 100}
```

OCaml also provides mutable arrays and strings, and `for` and `while` loops, so it can be used to program in a more imperative style.

OCaml also provides object-oriented features (surprise) which we may return to if time permits.

It's time to learn some type theory.

Type errors in ML languages often appear incomprehensible.

Sometimes we are prevented from writing code in a fashion that seems natural to us because of restrictions in the type system.

In order to understand how ML implementations approach type inference, we will add types to the lambda calculus.

Types were first added to the lambda calculus by Church in the 1930's.

We use a style closer to that of Curry's work in the same period.

The simply-typed lambda calculus

Abstract syntax:

```
t ::= x
    | λ x : T . t
    | t t
T ::= A
    | T -> T
```

Pierce starts with the concrete base types `Nat` and `Bool` in chapter 9, and moves to general uninterpreted base types in chapter 11.

We will use base types `A`, `B`, etc.

(Read chapter 8 on typed arithmetic expressions as required.)

Again, to avoid excessive parentheses, we will adopt the convention that the type constructor $\rightarrow$ is right-associative.

The only change to the syntax of terms is to introduce a type annotation on the variable of an abstraction.

The evaluation rule (β-reduction) is unchanged.

Our goal is to be able to make **type judgments**.

Example: what is the type of $\lambda x : A \,.\, x$?

What about
$\lambda x : A \,.\, \lambda y : A \to B \,.\, y\ x$?

To formalize this process, we use inference rules again. We also borrow some additional notation and ideas from mathematical logic.

A quick review of logic

Propositional logic builds propositions out of variables such as a, b and logical connectives such as $\rightarrow$ (implication or “if-then”).

We denote by $\Gamma \vdash \alpha$ the statement "From the set of propositions Γ, we can prove the proposition α."

A proof is a tree built from applications of inference rules.

For example (“modus ponens”):

$$\frac{\Gamma \vdash \alpha \to \beta \qquad \Gamma \vdash \alpha}{\Gamma \vdash \beta}$$

(“implication elimination” or $\to_e$).

Another example:

$$\frac{\Gamma, \alpha \vdash \beta}{\Gamma \vdash \alpha \to \beta}$$

("implication introduction" or $\to_i$).

With the following “structural rule”:

$$\frac{\alpha \in \Gamma}{\Gamma \vdash \alpha}$$

we can now prove

$$\vdash a \to ((a \to b) \to b).$$

$$\dfrac{\dfrac{\dfrac{\dfrac{a \to b \in \{a, a \to b\}}{a, a \to b \vdash a \to b}\,\mathsf{S}}{a, a \to b \vdash a}\,\mathsf{S}}{a, a \to b \vdash b}\,{\to_e}}{a \vdash (a \to b) \to b}\,{\to_i}}{\vdash a \to ((a \to b) \to b)}\,{\to_i}$$

Back to types

To type $\lambda x : T \,.\, t$, we need to type t.

But t contains occurrences of x.

In typing t, we need to remember that x has type T.

We store variable-type bindings of this form in a **type environment** (denoted by Γ).

Pierce: uses "context", notation $x : T \in \Gamma$ (assumes α-conversion to avoid name clashes).

$\Gamma, x : T$ notation for type environment extension (as in logic).

The type judgment $\Gamma \vdash t : T$ means that, using type environment Γ, we can demonstrate that t has type T.

We have three ways of forming terms, and three corresponding inference rules.

Variables:

$$\frac{x : T \in \Gamma}{\Gamma \vdash x : T}$$

Abstraction:

$$\frac{\Gamma, x : T_1 \vdash t : T_2}{\Gamma \vdash (\lambda x : T_1 . t) : (T_1 \to T_2)}$$

Application:

$$\frac{\Gamma \vdash t_1 : (T_1 \to T_2) \qquad \Gamma \vdash t_2 : T_1}{\Gamma \vdash t_1\ t_2 : T_2}$$

Let's use these inference rules to construct a proof tree verifying the type of $\lambda x : A \,.\, \lambda y : A \to B \,.\, y\ x$.

$$\dfrac{\dfrac{\dfrac{\dfrac{\dfrac{y : A \to B \in \{x : A, y : A \to B\}}{x : A, y : A \to B \vdash y : A \to B}\ \textsc{Vbl} \quad \dfrac{x : A \in \{x : A, y : A \to B\}}{x : A, y : A \to B \vdash x : A}\ \textsc{Vbl}}{x : A, y : A \to B \vdash y\ x : B}\ \textsc{App}}{x : A \vdash (\lambda y : A \to B\ .\ y\ x) : (A \to B) \to B}\ \textsc{Abs}}{\vdash (\lambda x : A\ .\ \lambda y : A \to B\ .\ y\ x) : A \to ((A \to B) \to B)}\ \textsc{Abs}$$

$$\dfrac{\dfrac{\dfrac{\dfrac{a \to b \in \{a, a \to b\}}{a, a \to b \vdash a \to b}\,\mathrm{S}}{a, a \to b \vdash a}\,\mathrm{S}}{a, a \to b \vdash b}\,{\to_e}}{a \vdash (a \to b) \to b}\,{\to_i}}{\vdash a \to ((a \to b) \to b)}\,{\to_i}$$

The Curry-Howard correspondence

Logic	typed LC
propositions	types
$\rightarrow_i$	Abs
$\rightarrow_e$	App
S	Var
proof of α	term of type α

For more:
Philip Wadler,
“Proofs are Programs”.

Some definitions

A term is **closed** if it has no free variables.

A term $t : T$ is **well-typed** if there exists Γ such that $\Gamma \vdash t : T$.

A value is an abstraction.

Some theorems

Unique types

Thm: If $\Gamma \vdash t : T_1$ and $\Gamma \vdash t : T_2$, then $T_1 = T_2$ (and the derivation is also unique).

Proof: Follows from the fact that there is one typing rule for each term-creating rule.

Substitution

Thm: If $\Gamma, x : S \vdash t : T$ and $\Gamma \vdash s : S$, then $\Gamma \vdash [x \mapsto s]t : T$.

Proof: Structural induction on t (or, equivalently, on the depth of the derivation of $\Gamma, x : \sigma \vdash t : T$).

Progress

Thm: If t is a closed, well-typed term (that is, $\phi \vdash t : T$ for some T), then either t is a value or there exists t' such that $t \longrightarrow t'$.

Proof: Structural induction on t (or, equivalently, on the depth of the derivation of $\phi \vdash t : T$).

Preservation

Thm: (Preservation) If $\Gamma \vdash t : T$ and $t \to t'$, then $\Gamma \vdash t' : T$.

Proof: Induction on the depth of a derivation of $\Gamma \vdash t : T$ (alternately, on length of reduction $t \to t'$).

Progress + preservation = safety.

Robin Milner: “Well-typed programs cannot *go wrong*”.

But there is trouble in paradise.

Strong normalization

When we try to type the
Y-combinator, we run into problems.

Consider typing $\lambda x.x\ x$.

Just because we cannot type self-application doesn’t mean we can’t do recursion in some other fashion.

But...

Thm: Every reduction sequence of every well-typed term of the simply-typed lambda calculus is of finite length.

Proof: Complicated induction. See Pierce (chapter 12) for a simpler version under simplifying assumptions.

The strong normalization theorem suggests that the simply-typed lambda calculus is a weak model of computation.

Schwichtenberg showed that if numbers are encoded as Church numerals, then the only functions definable on them are polynomials extended with conditionals.

To gain more power, we must extend the simply-typed lambda calculus with a construct for recursion, which breaks strong normalization.
(Pierce, 12.11)

Extensions

Pierce (chapter 8) shows how to use Bool and Nat as base types, with inference rules for constants and primitive functions.

The rules for variables, abstractions, and applications are unchanged.

The type Bool has constants `true` and `false`.

The type Nat has constant `zero`.

$$\Gamma \vdash \texttt{true} : \mathsf{Bool}$$

$$\Gamma \vdash \texttt{false} : \mathsf{Bool}$$

$$\frac{\Gamma \vdash c : \mathsf{Bool} \qquad \Gamma \vdash t : T \qquad \Gamma \vdash f : T}{\Gamma \vdash \texttt{if } c \texttt{ then } t \texttt{ else } f : T}$$

$$\Gamma \vdash \texttt{zero} : \mathsf{Nat}$$

$$\frac{\Gamma, t : \mathsf{Nat}}{\Gamma \vdash \texttt{succ}\ t : \mathsf{Nat}}$$

$$\frac{\Gamma \vdash t : \mathsf{Nat}}{\Gamma \vdash \texttt{iszero}\ t : \mathsf{Bool}}$$

Since the inference rules are all structural in nature, it is fairly straightforward to write a recursive typechecking algorithm.

For example, to type `(if c t f)`, recursively type `c`, `t`, and `f`, then apply the appropriate inference rule.

One goal of static typing is to limit the use of types at run time.

Types are checked at compile time (and perhaps used in compilation).

But then they are erased (do not appear in the compiled code).

Pierce's Chapter 11 contains many other extensions to the simply-typed lambda calculus.

You should read and understand them (you will implement some in assignment questions).

Progress and preservation theorems can be proved for these extensions.

We will examine some of these in a little more detail and just mention others, leaving details for reading.

The unit type (11.2); sequencing and dummy variables (11.3); ascription (11.4).

In 11.3, Pierce shows that sequencing such as `(t1; t2)` can either be added as a syntactic form, or be desugared to `(λx.t2)t1`.

Let bindings are treated in 11.5.

Recall that `let` can be desugared to an immediately applied abstraction.

But when types are involved, `let` is no longer just syntactic sugar.

`let x =` t_1 `in` t_2
desugars to $(\lambda x : T_1.t_2)\ t_1$.

But where does T_1 come from?

We need an inference rule.

$$\frac{\Gamma \vdash t_1 : T_1 \qquad \Gamma, \mathtt{x} : T_1 \vdash t_2 : T_2}{\Gamma \vdash \mathtt{let\ x} = t_1 \mathtt{\ in\ } t_2 : T_2}$$

The type T_1 is computed by the typechecker.

So `let` is desugared for evaluation purposes, but left alone for typechecking purposes.

We will see an additional reason to leave `let` alone when we consider type inference and polymorphism.

Pairs (11.6) with terms written as `{t1, t2}` and projection functions `t.1` and `t.2`;
tuples (11.7) generalizing this.

Labelled records (11.8) with terms written as `{l1=t1,l2=t2}`, deconstructed via pattern matching.

These are examples of what are known as **product types**.

For example, the type of a pair whose components have type T_1 and T_2 is typically written as $T_1 \times T_2$.

Variant types are examples of what are known as **sum types**.

In the Curry-Howard correspondence, product types map to logical AND; sum types map to logical OR.

We might be tempted to say that a value of type `Int+Bool` could be either an `Int` or a `Bool`.

But this would cause all sorts of problems.

What is the type of `3`?

Is it `Int`? `Int+Bool`? `Int+String`?

This is the reason that OCaml requires a unique constructor for each variant of a type.

Another name for this is **labelled union**.

We start by treating binary sum types.

To form an element of type $T_1 + T_2$, we either apply `inl` to an element of T_1, or `inr` to an element of T_2.

("Inject left" and "inject right")

To use an element of type $T_1 + T_2$, we use a `case` construct:

```
case e1 of
    inl x => e2
  | inr x => e3
```

Adding syntax, evaluation, and typing inference rules is quite straightforward, but there is a problem.

Our typechecker will need to compute the type of expressions such as `inl x`.

If `x` types as `T_1`, `inl x` could be of type `T_1+T_2` for **any** `T_2`.

The simplest solution is for the programmer to annotate the expression `inl x` with the intended sum type.

```
inl 5 as Nat+Bool
```

New syntax:

```
T ::= T+T
t ::= inl t as T
      inr t as T
      case t of
       inl x=>t | inr x=>t
v ::= inl v
      inr v
```

New evaluation rules:

$$\texttt{case inl v as T of inl x => s | inr y => t} \longrightarrow [\texttt{x} \mapsto \texttt{v}]\,\texttt{s}$$

$$\texttt{case inr v as T of inl x => s | inr y => t} \longrightarrow [\texttt{y} \mapsto \texttt{v}]\,\texttt{t}$$

(Necessary evaluation rules to extend to subterms.)

New typing rules:

$$\frac{\Gamma \vdash t_1 : T_1}{\Gamma \vdash \texttt{inl}\ t_1\ \texttt{as}\ T_1 + T_2 : T_1 + T_2}$$

$$\frac{\Gamma \vdash t_2 : T_2}{\Gamma \vdash \texttt{inl}\ t_2\ \texttt{as}\ T_1 + T_2 : T_1 + T_2}$$

$$\frac{\Gamma \vdash t_0 : T_1 + T_2 \qquad \Gamma, x : T_1 \vdash s : T \qquad \Gamma, y : T_2 \vdash t : T}{\Gamma \vdash \texttt{case}\ t_0\ \texttt{of}\ \texttt{inl}\ x \Rightarrow s \mid \texttt{inr}\ y \Rightarrow t : T}$$

This can be extended to multiple variants (11.10).

$T_1 + T_2$ becomes $\langle l_1 : T_1, l_2 : T_2 \rangle$.
`inl` t becomes
$\langle l_1 = t \rangle$ as $\langle l_1 : T_1, l_2 : T_2 \rangle$.

The annotation on the formation of sum-type terms (but not on their deconstruction) is annoying.

But it can be hidden.

OCaml types must be declared before use, and each data constructor must be unique, not just within the type, but across all types.

This lets the typechecker infer the necessary annotations (though the uniqueness requirement is still annoying).

General recursion (11.11)

We cannot type the Y_v combinator, or any fixed-point combinator, in the simply-typed λ-calculus, because a fixed-point combinator would let us write an expression with no normal form.

But we can add `fix` as a primitive, with appropriate evaluation and typing rules.

```
t ::= fix t
```

To write the factorial function:

$$\texttt{fix}\ \lambda f : \texttt{Nat} \to \texttt{Nat}.$$
$$\lambda n : \texttt{Nat}.$$
$$\text{if iszero } n \text{ then } 1$$
$$\text{else mult } n\ (f\ (\text{pred } n))$$

In general, we want the behaviour

$$\texttt{fix}\ f = f\ (\texttt{fix}\ f)$$

and if $\texttt{fix}\ f$ has type T, then f needs to have type $T \to T$.

The evaluation rule:

$$\texttt{fix}\ \lambda x : T.t \to [x \mapsto \texttt{fix}\ \lambda x : T.t]t$$

The typing rule:

$$\frac{\Gamma \vdash t : T \to T}{\Gamma \vdash \texttt{fix}\ t : T}$$

We can provide `letrec` as syntactic sugar.

```
letrec x:T = t in s
=
let x = fix (λx:T.t) in s
```

In `letrec x:T = t in s`, the type `T` is not restricted to be a function type.

Pierce shows how mutually recursive functions can be defined by letting `T` be a record type.

We can similarly “bake” lists into the type system by adding syntax for `cons`, `head`, `tail` (or a deconstruction pattern). See Pierce 11.12 for details.

But this does not address how recursive datatypes in OCaml are handled.

```
type 'a tree =
  Empty
| Node of 'a * 'a tree * 'a tree
```

We handle this in two steps.

First, we introduce **recursive types** (Pierce, chapter 20).

Then, in the next lecture module, we deal with the **parametric polymorphism** caused by the use of the type variable.

Let's consider the simpler example of defining lists of natural numbers.

In OCaml:

```
type NatList = Empty
             | Cons of Nat * NatList
```

In Pierce's type notation:

```
NatList = <nil:Unit,
           cons:{Nat, NatList}>
```

In a **nominative** type system, where equality is based on names, this definition presents no issues.

Nominative type systems do not deal well with abstractions such as `List[T]`.

OCaml's type system, and the ones we have been discussing so far, are **structural**.

(See the discussion in Pierce, 19.3.)

If we want names to be inessential in the type system, we are faced with a dilemma similar to that of defining recursion in the λ-calculus.

The resolution is similar.

We introduce a recursion operator μ on types binding a type variable such as `X`.

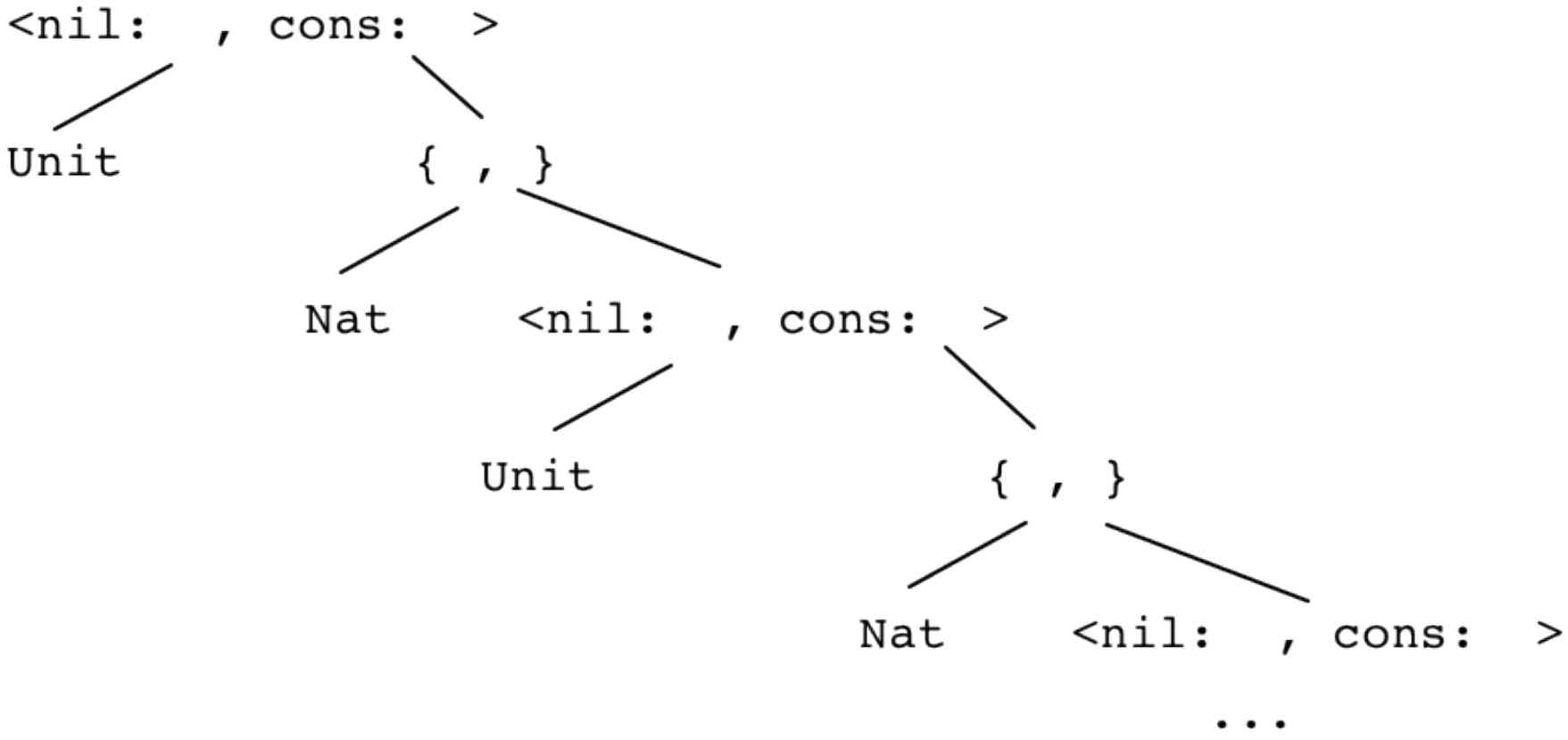
`T`=μ`X.E` means `T` satisfies the equation `X=E` (where `E` can use `X`).

In the case of `NatList`:

`NatList=`
μ`X.<nil:Unit,cons:{Nat, X}>`

`NatList` satisfies the equation
`X = <nil:Unit,cons:{Nat, X}>`

By repeatedly substituting for `X`, we can view `NatList` as an infinite tree:



More practically, we need ways of working with a type such as μX.<nil:Unit,cons:{Nat, X}>.

We need to be able to create values of such a type, and deconstruct them to get at their components.

By doing one step of substitution, we can say that the type

μX.<nil:Unit,cons:{Nat, X}>

unfolds to

```
<nil:Unit,
 cons:{Nat,μX.<nil:Unit,
                  cons:{Nat, X}>}}>
```

This unfolding action exposes the sum type so that, for example, a `case` statement can be used on it.

More generally,

$\mu X.T$

unfolds to

$[X \mapsto \mu X.T]T$

In an `equi-recursive` type system, these two types are treated as equal.

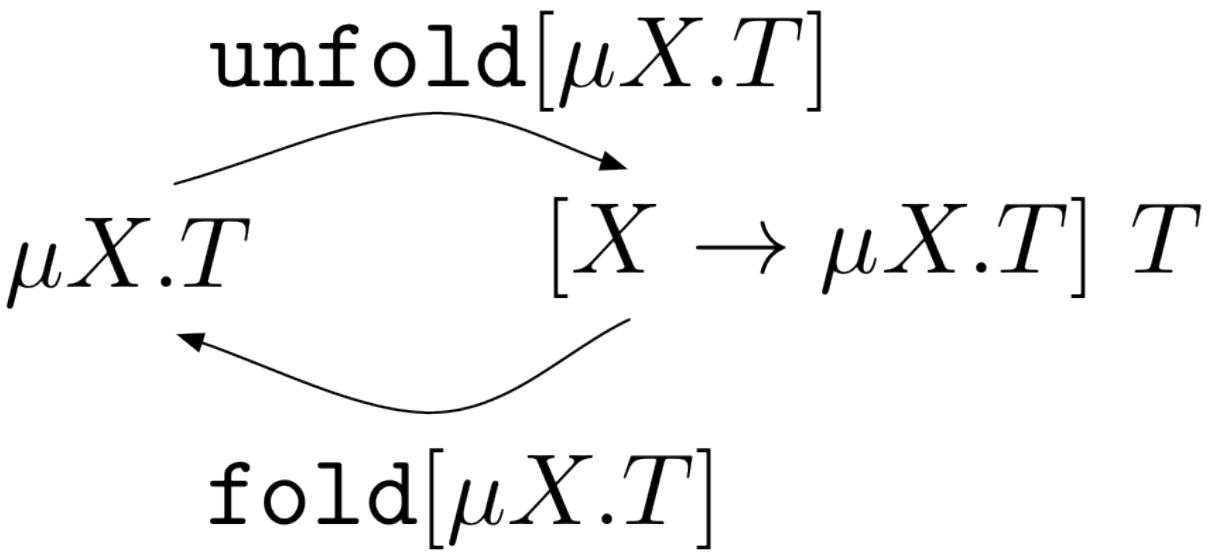
This requires mechanisms for managing the kinds of infinite trees that arise using this kind of definition (many details in Pierce, chapter 21).

An equi-recursive system extends well to structural subtyping and object systems.

The simpler alternative is an **iso-recursive** system (Pierce, 20.2).

In an iso-recursive system, a recursive type and its unfolding are viewed as isomorphic, but not equal.

Two functions are provided to map between these two types.


unfold[μX.T]
μX.T
[X → μX.T] T
fold[μX.T]

`fold` creates a value with a μ-type, and `unfold` changes its type to the isomorphic unfolded type, which exposes one level of structure (without changing the value).

To accomplish this, we add `fold` and `unfold` to the language as primitives, and provide evaluation and typing rules to cover their use that have the properties we want.

`fold` is used in a term when constructing a value of the recursive type, and `unfold` is used when destructuring such a value.

```
T ::= ...
      X
      μX.T
t ::= ...
      fold[T] t
      unfold[T] t
v ::= fold[T] v
```

Viewing a `fold[T] v` as a value means that this subexpression is not reduced further.

It stays around until an `unfold[T]` is applied to it, in which case the value `v` is exposed.

Evaluation:

$$\texttt{unfold[S] (fold[T] v)} \rightarrow \texttt{v}$$

Typing inference rules:

$$\frac{U = \mu X.T \qquad \Gamma \vdash t : [X \mapsto U]\ T}{\Gamma \vdash \texttt{fold}[U]\ t : U}$$

$$\frac{U = \mu X.T \qquad \Gamma \vdash t : U}{\Gamma \vdash \texttt{unfold}[U] t :\ [X \mapsto U]\ T}$$

As an example, let's see how to handle `NatList`.

```
NatList =
μX.<nil:Unit,cons:{Nat, X}>

NLbody =
<nil:Unit,cons:{Nat, NatList}>
```

By our rules,
if `v` has type `NLbody`,
then `fold[NatList] v` has type `NatList`.

Similarly,
if `v` has type `NatList`,
then `unfold[NatList] v` has type `NLBody`.

```
nil = fold [NatList]
        (<nil=unit> as NLBody)

cons = λn:Nat. λl:NatList.
         fold [NatList]
           (<cons={n,l}> as NLBody)
```

```
isnil =
λl:NatList.
 case unfold [NatList] l of
   <nil=u> => true
 | <cons=p> => false;
```

`hd` and `tl` are similar.

Exercise:
Check the types of `isnil nil` and `isnil (cons 1 nil)`, and then evaluate the expressions.

The explicit appearance of the `fold` and `unfold` syntax might place a burden on the programmer.

In practice, these are hidden in constructors and destructuring patterns respectively.

We will not properly deal with polymorphism until the next lecture module.

As it stands, we need many versions of, e.g., the identity function.

```
let idNat (fun (x: Nat) x)
    in ...

let idBool (fun (x: Bool) x)
    in ...
```

The last subject we will deal with in this lecture module is a proper treatment of reference types (Pierce, chapter 13).

We'll add OCaml-like syntax for references.

```
t ::= ...
      unit
      ref t
      !t
      t := t
```

The typing rules for these constructs are straightforward.

$$\frac{\Gamma \vdash t_1 : T_1}{\Gamma \vdash \texttt{ref}\ t_1 : \mathit{Ref}\ T_1}$$

$$\frac{\Gamma \vdash t_1 : \mathit{Ref}\ T_1}{\Gamma \vdash !t_1 : T_1}$$

$$\frac{\Gamma \vdash t_1 : \mathit{Ref}\ T_1 \qquad \Gamma \vdash t_2 : T_1}{\Gamma \vdash t_1 := t_2 : \mathit{Unit}}$$

The evaluation rules are more complicated.

What should the values of type `Ref T` be?

Our knowledge of the machine leads us to think of references as pointers, or memory addresses.

We abstract this idea to the notion of a **store** addressed by **locations**.

More precisely, references are elements of some uninterpreted set $\mathcal{L}$ of locations, and a store is a partial function from $\mathcal{L}$ to values.

Instead of our evaluation rules rewriting just a term `t`, they will now also rewrite a store μ.

We put these together as $\texttt{t} \mid \mu$.

First, we rewrite our three existing evaluation rules (function application and extension to subterms) to carry along stores.

$$(\lambda x.t_1)t_2 \mid \mu \to [x \mapsto t_2]\ t_1 \mid \mu$$

$$\frac{t_1 \mid \mu \to t_1' \mid \mu'}{t_1\ t_2 \mid \mu \to t_1'\ t_2 \mid \mu'}$$

$$\frac{t_2 \mid \mu \to t_2' \mid \mu'}{v_1\ t_2 \mid \mu \to v_1\ t_2' \mid \mu'}$$

We need to regard locations as values, and therefore as terms, though our language will not expose locations to the programmer.

They will appear as terms during the evaluation process, for example, when a term that is a reference type is fully evaluated.

Dereferencing:

$$\frac{t_1 \mid \mu \to t_1' \mid \mu'}{!t_1 \mid \mu \to !t_1' \mid \mu'}$$

$$\frac{\mu(\ell) = v}{!\ell \mid \mu \to v \mid \mu}$$

Assignment:

$$\frac{t_1 \mid \mu \to t_1' \mid \mu'}{t_1 := t_2 \mid \mu \to t_1' := t_2 \mid \mu'}$$

$$\frac{t_2 \mid \mu \to t_2' \mid \mu'}{v_1 := t_2 \mid \mu \to v_1 := t_2' \mid \mu'}$$

$$\ell := v \mid \mu \to \texttt{unit} \mid [\ell \mapsto v]\mu$$

Creating references:

$$\frac{t_1 \mid \mu \to t_1' \mid \mu'}{\texttt{ref}\ t_1 \mid \mu \to \texttt{ref}\ t_1' \mid \mu'}$$

$$\frac{\ell \notin \mathit{dom}(\mu)}{\texttt{ref}\ v_1 \mid \mu \to \ell \mid (\mu, \ell \mapsto v_1)}$$

Since locations show up as terms and values, we need to be able to type them.

This looks reasonable, but is inadequate.

$$\frac{\Gamma \vdash \mu(\ell) : T_1}{\Gamma \vdash \ell : \mathit{Ref}\ T_1}$$

Since μ keeps changing, we should include it with the context.

$$\frac{\Gamma \mid \mu \vdash \mu(\ell) : T_1}{\Gamma \mid \mu \vdash \ell : \mathit{Ref}\ T_1}$$

But this is still inadequate.

If the store contains a cyclic set of references (e.g. a doubly-linked list), this rule will not let us type them.

$$\{\ell_1 \mapsto \lambda x.\mathtt{Nat}.(!\ell_2)\ x,$$
$$\ell_2 \mapsto \lambda x.\mathtt{Nat}.(!\ell_1)\ x\}$$

The solution is to add a **store typing** Σ, which is a function from locations to types.

$$\frac{\Sigma(\ell) = T_1}{\Gamma \mid \Sigma \vdash \ell : \mathit{Ref}\ T_1}$$

The other typing rules for the new constructs remain the same, with the addition of the store typing to the context.

Let's see what happens to the type safety theorems we discussed for the simply-typed lambda calculus.

Here's the old definition of *well-typed*:

A term $t : T$ is **well-typed** if there exists Γ such that $\Gamma \vdash t : T$.

We need to add a store typing. But which one?

Typechecking can be done with respect to the empty store, since locations do not show up in what the programmer writes.

A term t is **well-typed** if there exist Γ, T such that $\Gamma \mid \phi \vdash t : T$.

Here's the preservation theorem for the simply-typed lambda calculus.

Thm: If $\Gamma \vdash t : T$ and $t \longrightarrow t'$, then $\Gamma \vdash t' : T$.

We could try just adding in the store and store typing.

Thm: If $\Gamma \mid \Sigma \vdash t : T$ and $t \mid \mu \rightarrow t' \mid \mu'$, then $\Gamma \mid \Sigma \vdash t' : T$.

But this expresses no relationship between Σ and μ.

We write $\Gamma \mid \Sigma \vdash \mu$ to indicate the store μ is well-typed with respect to Γ, Σ.

This means $\mathit{dom}(\mu) = \mathit{dom}(\Sigma)$ and for every $\ell \in \mathit{dom}(\mu)$, $\Gamma \mid \Sigma \vdash \mu(\ell) : \Sigma(\ell)$.

Next try at preservation:

Thm: If $\Gamma \mid \Sigma \vdash \mu$, $\Gamma \mid \Sigma \vdash t : T$ and $t \mid \mu \to t' \mid \mu'$, then $\Gamma \mid \Sigma \vdash t' : T$.

This is still not right.

$$\frac{\ell \notin \mathit{dom}(\mu)}{\texttt{ref}\ v_1 \mid \mu \rightarrow \ell \mid (\mu, \ell \mapsto v_1)}$$

This evaluation step grows the store, so ℓ is not typeable under the old store typing.

We need to allow store typings to grow.

Thm: If $\Gamma \mid \Sigma \vdash \mu$, $\Gamma \mid \Sigma \vdash t : T$ and $t \mid \mu \to t' \mid \mu'$, then for some $\Sigma' \supseteq \Sigma$, $\Gamma \mid \Sigma' \vdash \mu'$ and $\Gamma \mid \Sigma' \vdash t' : T$.

Now that we have the statement of the preservation theorem right, the proof goes through pretty much as before, with the help of a few simple technical lemmas about how stores behave.

Here's the progress theorem for the simply-typed lambda calculus.

Thm: If t is a closed, well-typed term (that is, $\phi \vdash t : T$ for some T), then either t is a value or there exists t' such that $t \rightarrow t'$.

Here's the new progress theorem.

Thm: If t is a closed, well-typed term (that is, $\phi \mid \Sigma \vdash t : T$ for some Σ, T), then either t is a value or else, for any store μ such that $\phi \mid \Sigma \vdash \mu$, there exist t', μ' such that $t \mid \mu \rightarrow t' \mid \mu'$.

The proof of the new progress theorem is very similar to that of the old one; we just need to add new cases for the new constructs.

Our evaluation semantics only add to the store.

It is possible to give a principled treatment of garbage collection using these ideas.

It is harder to ensure type safety with explicit memory allocation and deallocation.

Static type checks cannot prevent dangling references.

Next:
type inference