

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)

1 / 187

2 / 187

## A rapid introduction to OCaml

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

`ocamlc`: Bytecode compiler

`ocamlopt`: Compiles to native code

3 / 187

4 / 187

## 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.

5/187

```
# 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>
```

7/187

## Declarations

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

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

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

6/187

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

8/187

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.

9/187

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
```

10/187

Functions in ML have exactly one parameter.

```
# let sumSqr x y = x*x + y*y;;  
val sumSqr : int -> int -> int = <fun>  
  
# let f = sumSqr 4;;  
val f : int -> int = <fun>
```

11/187

## 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)
```

12/187

Tuples are usually deconstructed using patterns.

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

13/187

Infix operators can be used in a prefix manner.

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

15/187

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.

14/187

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
```

16/187

## 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
```

17/187

18/187

## 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'
```

19/187

20/187

## 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).

21/187

22/187

```
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.

23/187

24/187

```

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

```

25/187

```

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.

## 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.

26/187

## 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 = ()

```

27/187

28/187

# 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.)

29/187

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

31/187

30/187

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

32/187



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`.

```
# 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}
```

## 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
```

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.

37 / 187

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

39 / 187

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.

38 / 187

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.

40 / 187

# The simply-typed lambda calculus

Abstract syntax:

$$\begin{aligned} t &::= x \\ &\quad | \lambda x : T . t \\ &\quad | t t \\ T &::= A \\ &\quad | T \rightarrow T \end{aligned}$$

41 / 187

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.)

43 / 187

42 / 187

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

44 / 187

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

The evaluation rule ( $\beta$ -reduction) is unchanged.

45 / 187

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

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

47 / 187

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

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

46 / 187

## A quick review of logic

48 / 187

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

49 / 187

For example (“modus ponens”):

$$\frac{\Gamma \vdash \alpha \rightarrow \beta \quad \Gamma \vdash \alpha}{\Gamma \vdash \beta}$$

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

51 / 187

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

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

50 / 187

Another example:

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

(“implication introduction” or  $\rightarrow_i$ ).

52 / 187

With the following “structural rule”:

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

we can now prove

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

$$\frac{\frac{\frac{a \rightarrow b \in \{a, a \rightarrow b\}}{a, a \rightarrow b \vdash a \rightarrow b} S}{a \in \{a, a \rightarrow b\}} S}{a, a \rightarrow b \vdash a} S \quad \frac{}{a, a \rightarrow b \vdash b} \rightarrow_e \quad \frac{}{a \vdash (a \rightarrow b) \rightarrow b} \rightarrow_i \quad \frac{}{\vdash a \rightarrow ((a \rightarrow b) \rightarrow b)} \rightarrow_i$$

53 / 187

54 / 187

## 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  $T$ .

55 / 187

56 / 187

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

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

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

57/187

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

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

58/187

Variables:

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

Abstraction:

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

59/187

60/187

Application:

$$\frac{\Gamma \vdash t_1 : (T_1 \rightarrow T_2) \quad \Gamma \vdash t_2 : T_1}{\Gamma \vdash t_1 t_2 : T_2}$$

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

61 / 187

62 / 187

$$\frac{\frac{\frac{y : A \rightarrow B \in \{x : A, y : A \rightarrow B\}}{x : A, y : A \rightarrow B \vdash y : A \rightarrow B} \text{VBL} \quad \frac{x : A \in \{x : A, y : A \rightarrow B\}}{x : A, y : A \rightarrow B \vdash x : A} \text{VBL}}{x : A, y : A \rightarrow B \vdash y x : B} \text{APP} \quad \frac{x : A \vdash (\lambda y : A \rightarrow B . y x) : (A \rightarrow B) \rightarrow B}{\vdash (\lambda x : A . \lambda y : A \rightarrow B . y x) : A \rightarrow ((A \rightarrow B) \rightarrow B)} \text{ABS}$$

$$\frac{\frac{\frac{a \rightarrow b \in \{a, a \rightarrow b\}}{a, a \rightarrow b \vdash a \rightarrow b} \text{S} \quad \frac{a \in \{a, a \rightarrow b\}}{a, a \rightarrow b \vdash a} \text{S}}{a, a \rightarrow b \vdash b} \rightarrow_e \quad \frac{a, a \rightarrow b \vdash b}{a \vdash (a \rightarrow b) \rightarrow b} \rightarrow_i \quad \frac{a \vdash (a \rightarrow b) \rightarrow b}{\vdash a \rightarrow ((a \rightarrow b) \rightarrow b)} \rightarrow_i$$

63 / 187

64 / 187



# The Curry-Howard correspondence

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

65 / 187

66 / 187

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

## Some definitions

67 / 187

68 / 187

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

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

A value is an abstraction.

69 / 187

## 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.

71 / 187

## Some theorems

70 / 187

## 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  $\Gamma, x : \sigma \vdash t : T$ ).

72 / 187

## 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  $t \rightarrow t'$ .

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

73 / 187

Progress + preservation = safety.

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

But there is trouble in paradise.

75 / 187

## Preservation

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

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

74 / 187

## Strong normalization

76 / 187

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...

77 / 187

**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.

79 / 187

78 / 187

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.

80 / 187

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

81 / 187

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.

83 / 187

# Extensions

82 / 187

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

The type `Nat` has constant `zero`.

84 / 187

$$\Gamma \vdash \text{true} : \text{Bool}$$
$$\Gamma \vdash \text{false} : \text{Bool}$$
$$\frac{\Gamma \vdash c : \text{Bool} \quad \Gamma \vdash t : T \quad \Gamma \vdash f : T}{\Gamma \vdash \text{if } c \text{ then } t \text{ else } f : T}$$

85/187

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.

87/187

$$\Gamma \vdash \text{zero} : \text{Nat}$$
$$\frac{\Gamma, t : \text{Nat}}{\Gamma \vdash \text{succ } t : \text{Nat}}$$
$$\frac{\Gamma \vdash t : \text{Nat}}{\Gamma \vdash \text{iszero } t : \text{Bool}}$$

86/187

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).

88/187

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.

89 / 187

In 11.3, Pierce shows that sequencing such as  $(t_1; t_2)$  can either be added as a syntactic form, or be desugared to  $(\lambda x. t_2) t_1$ .

91 / 187

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).

90 / 187

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.

92 / 187

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

But where does  $T_1$  come from?

We need an inference rule.

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

The type  $T_1$  is computed by the typechecker.

93 / 187

94 / 187

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  $\{t_1, t_2\}$  and projection functions  $t.1$  and  $t.2$ ;  
tuples (11.7) generalizing this.

95 / 187

96 / 187



Labelled records (11.8) with terms written as  $\{l_1=t_1, l_2=t_2\}$ , deconstructed via pattern matching.

97 / 187

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`.

99 / 187

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**.

98 / 187

But this would cause all sorts of problems.

What is the type of 3?

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

100 / 187

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

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

101 / 187

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

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

103 / 187

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”)

102 / 187

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**  $T_2$ .

104 / 187

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
```

105/187

106/187

New evaluation rules:

```
case inl v as T
  of inl x => s | inr y => t
  → [x ↦ v] s
```

```
case inr v as T
  of inl x => s | inr y => t
  → [y ↦ v] t
```

(Necessary evaluation rules to extend to subterms.)

107/187

108/187

New typing rules:

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

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

109/187

This can be extended to multiple variants (11.10).

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

111/187

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

110/187

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

But it can be hidden.

112/187

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).

113/187

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

`t ::= fix t`

115/187

## General recursion (11.11)

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

114/187

To write the factorial function:

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

116/187

In general, we want the behaviour

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

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

The evaluation rule:

$$\text{fix } \lambda x : T. t \rightarrow [x \mapsto \text{fix } \lambda x : T. t] t$$

The typing rule:

$$\frac{\Gamma \vdash t : T \rightarrow T}{\Gamma \vdash \text{fix } t : T}$$

117/187

118/187

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.

119/187

120/187

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.

121 / 187

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.

123 / 187

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

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

122 / 187

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}>
```

124 / 187

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]`.

125 / 187

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  $\lambda$ -calculus.

The resolution is similar.

127 / 187

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

(See the discussion in Pierce, 19.3.)

126 / 187

We introduce a recursion operator  $\mu$  on types binding a type variable such as  $X$ .

$T = \mu X. E$  means  $T$  satisfies the equation  $X = E$  (where  $E$  can use  $X$ ).

128 / 187

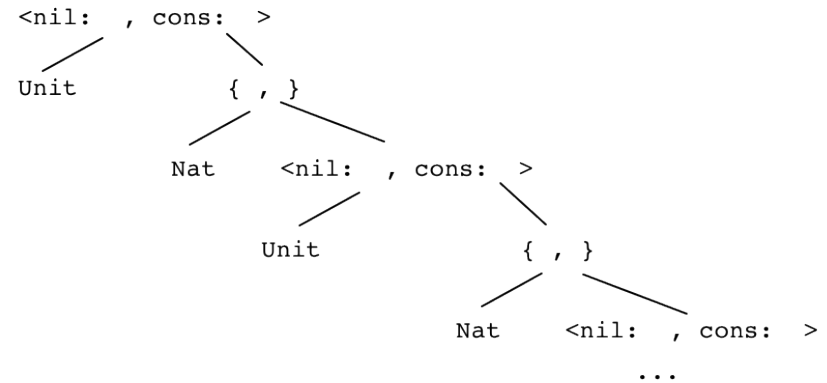


In the case of NatList:

NatList=  
 $\mu X. \langle \text{nil}:\text{Unit}, \text{cons}:\{\text{Nat}, X\} \rangle$

NatList satisfies the equation  
 $X = \langle \text{nil}:\text{Unit}, \text{cons}:\{\text{Nat}, X\} \rangle$

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



129/187

130/187

More practically, we need ways of working with a type such as  
 $\mu X. \langle \text{nil}:\text{Unit}, \text{cons}:\{\text{Nat}, X\} \rangle$ .

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

$\mu X. \langle \text{nil}:\text{Unit}, \text{cons}:\{\text{Nat}, X\} \rangle$

**unfolds** to

$\langle \text{nil}:\text{Unit}, \text{cons}:\{\text{Nat}, \mu X. \langle \text{nil}:\text{Unit}, \text{cons}:\{\text{Nat}, X\} \rangle \} \rangle$

131/187

132/187

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

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

133 / 187

134 / 187

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.

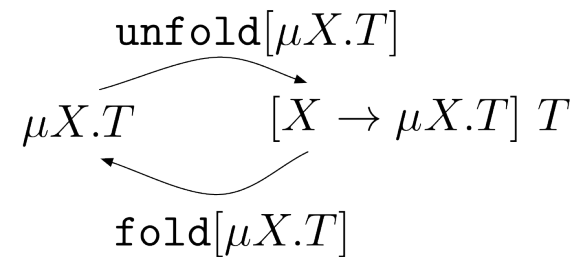
The simpler alternative is an **iso-recursive** system (Pierce, 20.2).

135 / 187

136 / 187

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.



137 / 187

138 / 187

`fold` creates a value with a  $\mu$ -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.

139 / 187

140 / 187

$T ::= \dots$   
 $X$   
 $\mu X.T$   
 $t ::= \dots$   
 $\text{fold}[T] \ t$   
 $\text{unfold}[T] \ t$   
 $v ::= \text{fold}[T] \ v$

Viewing a  $\text{fold}[T] \ v$  as a value means that this subexpression is not reduced further.

It stays around until an  $\text{unfold}[T]$  is applied to it, in which case the value  $v$  is exposed.

141 / 187

142 / 187

Evaluation:

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

Typing inference rules:

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

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

143 / 187

144 / 187

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

```
NatList =  
 $\mu$ X.<nil:Unit,cons:{Nat, X}>
```

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

145/187

```
nil = fold [NatList]  
      (<nil=unit> as NLBody)  
  
cons =  $\lambda$ n:Nat.  $\lambda$ l:NatList.  
      fold [NatList]  
      (<cons={n,l}> as NLBody)
```

147/187

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.

146/187

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

hd and tl are similar.

148/187

Exercise:

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

149 / 187

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.

151 / 187

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.

150 / 187

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

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

152 / 187

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
```

153 / 187

154 / 187

The typing rules for these constructs are straightforward.

$$\begin{array}{c}
 \frac{\Gamma \vdash t_1 : T_1}{\Gamma \vdash \text{ref } t_1 : \textit{Ref } T_1} \\
 \frac{\Gamma \vdash t_1 : \textit{Ref } T_1}{\Gamma \vdash !t_1 : T_1} \\
 \frac{\Gamma \vdash t_1 : \textit{Ref } T_1 \quad \Gamma \vdash t_2 : T_1}{\Gamma \vdash t_1 := t_2 : \textit{Unit}}
 \end{array}$$

155 / 187

156 / 187

The evaluation rules are more complicated.

What should the values of type  $\text{Ref } T$  be?

157 / 187

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.

159 / 187

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**.

158 / 187

Instead of our evaluation rules rewriting just a term  $t$ , they will now also rewrite a store  $\mu$ .

We put these together as  $t \mid \mu$ .

160 / 187



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

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

$$\frac{t_1 \mid \mu \rightarrow t'_1 \mid \mu'}{t_1 t_2 \mid \mu \rightarrow t'_1 t_2 \mid \mu'}$$

$$\frac{t_2 \mid \mu \rightarrow t'_2 \mid \mu'}{v_1 t_2 \mid \mu \rightarrow v_1 t'_2 \mid \mu'}$$

161 / 187

162 / 187

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:

$$\frac{t_1 \mid \mu \rightarrow t'_1 \mid \mu'}{!t_1 \mid \mu \rightarrow !t'_1 \mid \mu'}$$

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

163 / 187

164 / 187

Assignment:

$$\frac{t_1 \mid \mu \rightarrow t'_1 \mid \mu'}{t_1 := t_2 \mid \mu \rightarrow t'_1 := t_2 \mid \mu'}$$

$$\frac{t_2 \mid \mu \rightarrow t'_2 \mid \mu'}{v_1 := t_2 \mid \mu \rightarrow v_1 := t'_2 \mid \mu'}$$

$$\ell := v \mid \mu \rightarrow \text{unit} \mid [\ell \mapsto v]\mu$$

165 / 187

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

Creating references:

$$\frac{t_1 \mid \mu \rightarrow t'_1 \mid \mu'}{\text{ref } t_1 \mid \mu \rightarrow \text{ref } t'_1 \mid \mu'}$$

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

166 / 187

This looks reasonable, but is inadequate.

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

167 / 187

168 / 187

Since  $\mu$  keeps changing, we should include it with the context.

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

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.

$$\begin{aligned} \{ \ell_1 &\mapsto \lambda x. \text{Nat}.(!\ell_2) x, \\ \ell_2 &\mapsto \lambda x. \text{Nat}.(!\ell_1) x \} \end{aligned}$$

169 / 187

170 / 187

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

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

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.

171 / 187

172 / 187

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

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

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

173/187

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

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

175/187

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  $\Gamma, T$  such that  $\Gamma \mid \phi \vdash t : T$ .

174/187

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  $\Sigma$  and  $\mu$ .

176/187

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

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

Next try at preservation:

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

This is still not right.

177/187

178/187

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

This evaluation step grows the store, so  $\ell$  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  $t \mid \mu \rightarrow t' \mid \mu'$ , then for some  $\Sigma' \supseteq \Sigma$ ,  $\Gamma \mid \Sigma' \vdash \mu'$  and  $\Gamma \mid \Sigma' \vdash t' : T$ .

179/187

180/187

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.

181 / 187

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  $t \rightarrow t'$ .

182 / 187

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  $\Sigma, T$ ), then either  $t$  is a value or else, for any store  $\mu$  such that  $\phi \mid \Sigma \vdash \mu$ , there exist  $t', \mu'$  such that  $t \mid \mu \rightarrow t' \mid \mu'$ .

183 / 187

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.

184 / 187

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.

185 / 187

186 / 187

Next:  
type inference

187 / 187