

15-150 Fall 2025

Lecture 9

Types and Polymorphism

Announcement

Sections A-I go to **PH 100**

Sections J-L go to **MM Breed Hall**

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)

static guarantee	runtime guarantee
e has type t	if $e \Rightarrow v$ then $v : t$

Polymorphism

Monomorphic rev

```
fun rev ([ ]:int list) :int list = [ ]  
  | rev (x::xs) = rev xs @ [x]
```

$(x::xs): t \text{ list}$ if $x:t$ and $xs: t \text{ list}$

datatype $_ \text{list} = \text{nil} \mid :: \text{ of } _ * _ \text{list}$

$(x::xs): t \text{ list}$

if $x:t$ and $xs: t \text{ list}$

datatype 'a list = nil |:: **of** 'a * 'a list



“alpha”

infixr ::

[]: 'a list

[]: 'a list

[1]



int list

1 :: []



int 'a list

'a specialized to/instantiated as int, giving us int list

[]: 'a list

[true] true :: []



bool list bool 'a list

'a specialized to/instantiated as bool, giving us bool list

'a list
is an instance of 'b

[[]]



'a list list

[] :: []



'a list 'b list

'b specialized to/instantiated as 'a list, giving us list 'a list list

Polymorphic rev

```
fun rev ([ ]:'a list) : 'a list = [ ]  
  | rev (x::xs) = rev xs @ [x]
```

In the scope of this declaration you can use `rev` with any list as an argument.

`([ ],[ ]):` `'a list * 'b list`

`[1]::[[ ]]` : `int list list`

`1::[[ ]]:` `not well-typed`

Parameterized datatypes

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

introduces a type constructor (tree)

and polymorphic value constructors Empty and Node:

Empty: 'a tree

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

Parameterized datatypes

```
datatype ('a,'b) mixed = A of 'a  
                        | B of 'b
```

introduces a type constructor (mixed)
and polymorphic value constructors A and B:

```
A: 'a -> ('a, 'b) mixed
```

```
B: 'b -> ('a, 'b) mixed
```

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))
```

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))
```

```
trav (Node(Empty, 1, Empty)):  int list
```

zip

(* zip : 'a list * 'b list -> ('a * 'b) list

REQUIRES: true

ENSURES: zip([a1,a2,...,an],[b1,b2,...,bm]) $\cong$
 [(a1,b1), (a2,b2), ..., (ak,bk)]
 with k = min(n,m) $\geq$ 0.

*)

(* zip : 'a list * 'b list -> ('a * 'b) list
REQUIRES: true
ENSURES: zip([a1,a2,...,an],[b1,b2,...,bm]) $\cong$
 [(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)
```

(* zip : 'a list * 'b list -> ('a * 'b) list
REQUIRES: true
ENSURES: zip([a1,a2,...,an],[b1,b2,...,bm]) $\cong$
 [(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)
```

zip ([1,2,3,4,5],["a","b","c","d"]): (int * string) list

evaluates to [(1,"a"),(2,"b"),(3,"c"),(4,"d")]

options

datatype 'a option = NONE | SOME of 'a

lookup

(* lookup : _____
REQUIRES:
ENSURES:

*)

(* lookup : _____
REQUIRES: true
ENSURES:

*)

(* lookup : _____ * 'a * ('a * 'b) list -> _____

REQUIRES: true

ENSURES:

*)

(* lookup : ('a * 'a -> bool) * 'a * ('a * 'b) list -> _____

REQUIRES: true

ENSURES:

*)

(* lookup : ('a * 'a -> bool) * 'a * ('a * 'b) list -> 'b option

REQUIRES: true

ENSURES:

*)


```
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)
```

```
val L = [(1,"a"),(2,"b"),(3,"c"),(4,"d")] : (int * string) list
```

```
lookup ((op =), 2, L): string option
```

evaluates to SOME "b"

```
lookup ((op =), ? , L)           evaluates to NONE
```

```
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)
```

```
val L = [(1,"a"),(2,"b"),(3,"c"),(4,"d")] : (int * string) list
```

```
lookup ((op =), 2, L): string option
```

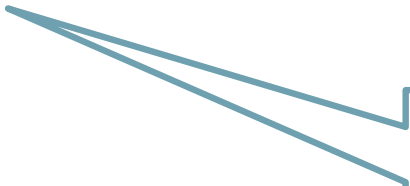
evaluates to SOME "b"

```
lookup ((op =), 5 , L)           evaluates to  NONE
```

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** square x = x * x * 1

square: int -> int

2. **fun** first(x,y) = x

first: 'a * 'b -> 'a

3. **fun** sqrf (f, x) = square (f(x))

('a -> int)*'a ->int

4. **fun** f x = f x

('a -> 'b)

5. **fun** h x = h (h x)

('a -> 'a)

6. **fun** id x = x

('a -> 'a)

7. id id 42

int

```
fun id x = x : ('a -> 'a)
```

Function application is left-associative

f g x means (f g) x

id square 7 : int

square id 7 not well typed

square (id 7) : int

```
(* 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
```