15-150 Fall 2013

Lecture 9
Stephen Brookes

- Type checking
- Polymorphism
- Type inference

... a static check provides a runtime guarantee

- ML only evaluates well-typed expressions
- Well-typed expressions don't go wrong!

If e has type t and e =>* v, then v is a value of type t.

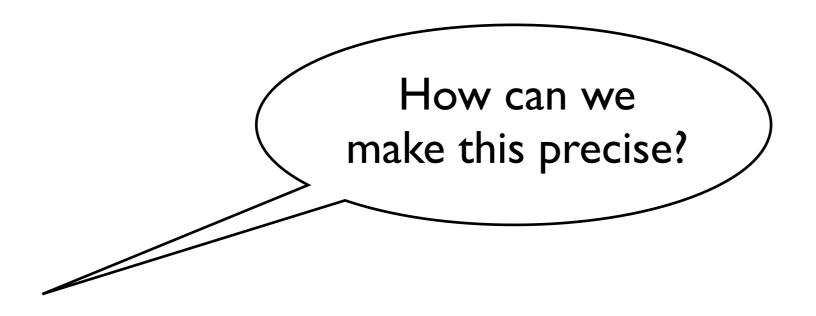
How can we check if e has type t?

- ML only elaborates well-typed declarations
- Well-typed declarations don't go wrong

If d declares x of type t, then d binds x to a value of type t

How can we check if d declares x:t?

- ML only performs well-typed pattern matches
- Well-typed patterns don't go wrong



 The type of an expression depends on the types of its sub-expressions

 The type of an expression depends on the types of its sub-expressions

... hence, the type of an expression depends on the types of its free variables

 The type of an expression depends on the types of its sub-expressions

... hence, the type of an expression depends on the types of its free variables

x + x has type int if x has type int

 The type of an expression depends on the types of its sub-expressions

... hence, the type of an expression depends on the types of its free variables

```
x + x has type intx + x has type intx + x has type realif x has type real
```

can be done at compile time

- There are **syntax-directed** rules for figuring out when e has type t
 - with assumptions about free variables of e
- Rules are based on the syntax of e and t

can be done at compile time

- There are **syntax-directed** rules for figuring out when e has type t
 - with assumptions about free variables of e
- Rules are based on the syntax of e and t

e is well-typed, with type t, if and only if **provable** from these rules

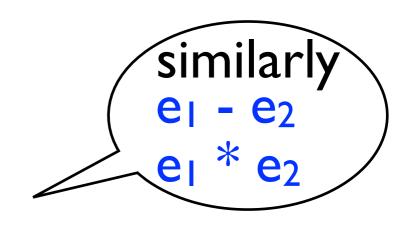
Typing rules

There are syntax-directed rules for

```
e has type t
d declares x : t
p fits type t and binds x : t'
under appropriate assumptions
```

Arithmetic

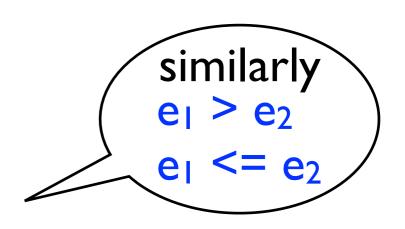
- $0, 1, 2, \sim 1, \dots$ have type int
- 0.0, 1.1, ~2.0, ... have type real
- e₁ + e₂ has type int
 if e₁ and e₂ have type int
- e₁ + e₂ has type real
 if e₁ and e₂ have type real



e₁ + e₂ is not well-typed, otherwise

Comparison

- e₁ < e₂ has type bool
 if e₁ and e₂ have type int
- e₁ < e₂ has type bool
 if e₁ and e₂ have type real



e₁ < e₂ is not well-typed, otherwise

Conditional

(for all types t)

• if e then e₁ else e₂ has type t if e has type bool and e₁, e₂ have type t

> both branches must have the same type

if e then e₁ else e₂ is not well-typed, otherwise

Tuples

(for all types t₁ and t₂)

(e₁, e₂) has type t₁ * t₂
 if e₁ has type t₁ and e₂ has type t₂

Tuples

(for all types t₁ and t₂)

(e₁, e₂) has type t₁ * t₂
 if e₁ has type t₁ and e₂ has type t₂

Similarly for $(e_1, ..., e_k)$ when k>0

Tuples

(for all types t₁ and t₂)

(e₁, e₂) has type t₁ * t₂
 if e₁ has type t₁ and e₂ has type t₂

Similarly for $(e_1, ..., e_k)$ when k>0

() has type unit

Lists

(for all types t)

- [e_I, ..., e_n] has type t list
 if for each i, e_i has type t
- (for all $n \ge 0$)
- e₁::e₂ has type t list
 if e₁ has type t and e₂ has type t list
- e₁@e₂ has type t list
 if e₁ and e₂ have type t list

Functions

fn x => e has type t₁ -> t₂
 if, assuming x : t₁, e has type t₂

fn x => e is not well-typed,
if no such t₁ and t₂ exist

fn x => x + 1.0 has type real -> real

Application

e₁ e₂ has type t₂
 if e₁ has type t₁ -> t₂ and e₂ has type t₁

e₁ e₂ is not well-typed, otherwise

if e_1 does not have a function type, or e_1 has type $t_1 \rightarrow t_2$ but e_2 doesn't have type t_1

Declarations

- val x = e declares x : t
 if e has type t
- fun f x = e declares f:t₁ -> t₂
 if, assuming x:t₁ and f:t₁ -> t₂,
 e has type t₂

assuming that argument has type to and recursive calls have type to -> t2, body has type t2

(also rules for combining declarations)

Declarations

- val x = e declares x : tif e has type t
- fun f x = e declares f:t₁ -> t₂
 if, assuming x:t₁ and f:t₁ -> t₂,
 e has type t₂

assuming that argument has type to and recursive calls have type to -> t2, body has type t2

(also rules for combining declarations)

val
$$x = 42$$
 declares
fun $f(y) = x+y$ $x : int and f : int -> int$

let d in e end has type t₂
 if d declares x : t₁, ...,
 and, assuming x : t₁, ...,
 e has type t₂

let d in e end has type t₂ if d declares x : t₁, ..., and, assuming x : t₁, ..., e has type t₂

let val x = 21 in x + x end

let d in e end has type t₂ if d declares x : t₁, ..., and, assuming x : t₁, ..., e has type t₂

let val x = 21 in x + x end has type int

let d in e end has type t₂ if d declares x : t₁, ..., and, assuming x : t₁, ..., e has type t₂

let val x = 21 in x + x end has type int

```
let
    val x = 2!
    fun f(y) = x+y
in
    x + (f x)
end
```

let d in e end has type t₂ if d declares x : t₁, ..., and, assuming x : t₁, ..., e has type t₂

let val x = 21 in x + x end has type int

Patterns

When does pattern p fit type t?

- _ fits t always
- 42 fits t iff t is int
- x fits t always
- (p₁, p₂) fits t iff
 t is t₁*t₂, p₁ fits t₁, p₂ fits t₂
- p₁::p₂ fits t iff
 t is t₁ list, p₁ fits t₁, p₂ fits t₁ list

Patterns

When p fits t, what type bindings does it produce ?

- Fitting _ to t produces no bindings
- Fitting x to t produces x:t
- Fitting (p₁, p₂) to t₁ * t₂ produces the bindings from fitting p₁ to t₁ and p₂ to t₂
- Fitting p₁::p₂ to t₁ list produces the bindings from fitting p₁ to t₁ and p₂ to t₁ list

functions

• fn p₁ => e₁ | ... | p_k => e_k has type t₁ -> t₂ if for each i, fitting p_i to t₁ succeeds, with type bindings for which e_i has type t₂

fn
$$0 \Rightarrow 0 \mid n \Rightarrow n - 1$$

has type int -> int

recursive functions

• fun f p₁ = e₁ | ... | f p_k = e_k declares f: t₁ -> t₂ if for each i, matching p_i to t₁ succeeds, with type bindings for which, assuming f:t₁ -> t₂, e_i has type t₂

fun f 0 = 0 | f n = f (n - 1)declares f : int -> int

example

Polymorphic types

ML has type variables

A type with type variables is polymorphic

A polymorphic type has instances

```
int list -> int list
    real list -> real list
(int * real) list -> (int * real) list
```

... instances of 'a list -> 'a list

split

```
fun split [ ] = ([ ], [ ])
    | split [x] = ([x], [ ])
    | split (x::y::L) =
    let val (A,B) = split L in (x::A, y::B) end
```

split

```
fun split [] = ([], [])
  | split [x] = ([x], [])
  | split (x::y::L) =
  let val (A,B) = split L in (x::A, y::B) end
```

declares

split: int list -> int list * int list

split

```
fun split [ ] = ([ ], [ ])
    | split [x] = ([x], [ ])
    | split (x::y::L) =
    let val (A,B) = split L in (x::A, y::B) end
```

split

```
fun split [] = ([], [])
  | split [x] = ([x], [])
  | split (x::y::L) =
  let val (A,B) = split L in (x::A, y::B) end
```

declares

split: 'a list -> 'a list * 'a list

sorting

```
Assuming split: int list -> int list * int list merge: int list * int list -> int list -> int list
```

declares msort: int list -> int list

sorting

```
split: 'a list -> 'a list * 'a list
Assuming
               merge: int list * int list -> int list
  fun msort [] = []
      msort[x] = [x]
      msort L = let
                   val(A,B) = split L
                  in
                   merge (msort A, msort B)
                  end
      is not well-typed
                                    (because the sub-expression
                                              split L
                                         is not well-typed)
```

sorting

```
Assuming split: 'a list -> 'a list * 'a list merge: int list * int list -> int list
```

declares msort: 'a list -> int list

... there's a bug in the code!

- Every type has a set of syntactic values
- What are the values of type 'a -> 'a ?

• What are the values of type 'a?

- Every type has a set of syntactic values
- What are the values of type 'a -> 'a ?
 (all are equivalent to) fn x => x or fn x => loop()
- What are the values of type 'a?

- Every type has a set of syntactic values
- What are the values of type 'a -> 'a ?
 (all are equivalent to) fn x => x or fn x => loop()
- What are the values of type 'a?

There are none!

- Every type has a set of syntactic values
- What are the values of type 'a -> 'a ?
 (all are equivalent to) fn x => x or fn x => loop()
- What are the values of type 'a?

There are none!

Reason: the type guarantee

typability

- t is a type for e
 iff (e has type t) is provable
- In the scope of d, x has type t
 iff (d declares x:t) is provable

```
int list -> int list is a type for rev real list -> real list is a type for rev 'a list -> 'a list is a type for rev
```

Instantiation

If e has type t, and t' is an instance of t,
 then e also has type t'

An expression can be used at any instance of its type

Most general types

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

t is a most general type for e iff every instance of t is a type for e & every type for e is an instance of t

rev has most general type 'a list -> 'a list

Most general types

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

t is a most general type for e iff every instance of t is a type for e & every type for e is an instance of t

rev has most general type 'a list -> 'a list



Most general types

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

t is a most general type for e iff every instance of t is a type for e & every type for e is an instance of t

rev has most general type 'a list -> 'a list



: bean list



type inference

- ML computes most general types
 - statically, using syntax as guide

Standard ML of New Jersey v110.75

type inference

- ML computes most general types
 - statically, using syntax as guide

```
Standard ML of New Jersey v110.75
- fun rev [] = [] | rev (x::L) = (rev L) @ [x];
```

type inference

- ML computes most general types
 - statically, using syntax as guide

```
Standard ML of New Jersey v110.75

- fun rev [] = [] | rev (x::L) = (rev L) @ [x];

val rev = fn : 'a list -> 'a list
```

datatypes

ML allows parameterized datatypes

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

a type constructor tree

and **polymorphic** value constructors

Empty: 'a tree

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

example

```
fun trav Empty = []
| trav (Node(t1, x, t2)) = (trav t1) @ x :: (trav t2)
```

declares trav: 'a tree -> 'a list

options

datatype 'a option = NONE | SOME of 'a

options

datatype 'a option = NONE | SOME of 'a

try: ('a -> 'b) -> ('a option -> 'b option)

equality

- ML allows use of = only on certain types
- These are called equality types
 - int
 - tuples and lists built from equality types
 - not real and not function types
- ML uses type variables "a, "b, "c to stand for equality types
 - must be instantiated with an equality type

example

declares mem: "a * "a list -> bool

OK instances include

```
int * int list -> bool
    (int list) * (int list) list -> bool
but not real * real list -> bool
```