

15-150 Fall 2023

Lecture 10
Stephen Brookes

- Type checking
- Type inference

type benefits

... a *static check* provides a *runtime guarantee*

static property	runtime guarantee
e has type t	if $e \Rightarrow^* v$ then $v : t$
d declares $x : t$	if $d \Rightarrow^* [x : v]$ then $v : t$

advantages

Type analysis is **easy**, *static*, cheap

would be
expensive
to keep checking
at runtime

- A type error indicates a *bug* detected, *and prevented*, without running code
- An *unexpected* type may also indicate a *bug*!

Values of a given type have **predictable** form

- We can use **appropriate** patterns and design code accordingly

Type information can **guide** specs and proofs

Referential transparency

for types

How to tell *statically* when $e : t$

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

*... hence, the type of an expression depends on its **syntactic form** and the types of its free variables*

$x + x$ has type **int** if x has type **int**

(fn x:int => x+x) e has type **int** if e has type **int**

(fn x:int => x+x) **true** is not well typed

type analysis

can be done *statically*

- There are ***syntax-directed*** rules for figuring out when e has type t

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

We say “ e has type t ”
or write “ $e : t$ ”

... possibly with
assumptions like
“ $x:\text{int}$ and $y:\text{int}$ ”

Typing rules

Syntax-directed rules for “typing judgements”

e has type t

d declares $x_1 : t_1 \dots x_k : t_k$

p matches type t and binds $x_1 : t_1 \dots x_k : t_k$

under appropriate assumptions
about the free variables of e and d

intuition

- The typing rules embody *simple principles* that ensure runtime safety
 - **Functions** must be **applied** to arguments of an *appropriate* type
 - All items in a **list** must have the same type
 - All branches of **case** must have the same type and all patterns must match values of the same type
 - The test of an **if-then-else** must be **boolean**, and both branches must have same type

trade-off

- The rules ensure that well-typed expressions “don’t go wrong”
- As the rules are “only” based on *syntax*, some expressions deemed ill-typed by ML could actually be evaluated without error `if true then 42 else [1,2,3]`
- There’s a trade-off here: the *decidable* property (e has a type) gives an approximation to an *undecidable* property (e has a value)
- Learn to live with typing rules.
The pain is worth the gain.

type checking

- Use the typing rules to check that
(for a specific type **t**)
e has type **t**

fn x => x
has type **bool** -> **bool**

fn x => x
has type **int** -> **int**

type inference

- Use the typing rules to figure out
if **e** is well-typed, and — if so — its *most general* type


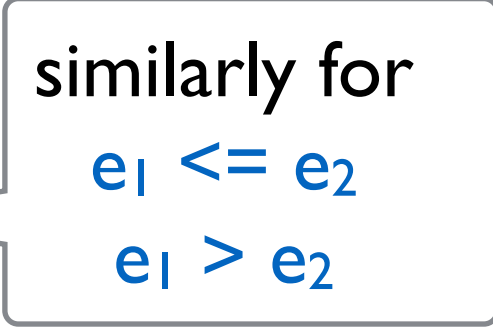
fn x => x
has most general type **'a** -> **'a**

arithmetic

- a numeral n has type **int**
- $e_1 + e_2$ has type **int**
if e_1 and e_2 have type **int**
- Similarly for $e_1 * e_2$ and $e_1 - e_2$

static property	runtime behavior
$21 + 21$ has type int	$21 + 21 \Rightarrow^* 42 : \text{int}$

booleans

- **true** and **false** have type **bool**
- e_1 **andalso** e_2 has type **bool**
if e_1 and e_2 have type **bool**

- $e_1 < e_2$ has type **bool**
if e_1 and e_2 have type **int**


static property	runtime behavior
$(3+4 < 1+7) : \text{bool}$	$(3+4 < 1+7) \Rightarrow^* \text{true} : \text{bool}$

conditional

(for each type t)

- **if** e **then** e_1 **else** e_2 has type t
if e has type **bool** and e_1, e_2 have type t

*test must be a boolean,
both branches must have the same type*

static

if $x < y$ **then** x **else** y has type **int**
if $x:\text{int}$ and $y:\text{int}$

runtime

if $x < y$ **then** x **else** $y \Rightarrow^* 4 : \text{int}$
if $x:4$ and $y:5$

tuples

(for all types t_1 and t_2)

- (e_1, e_2) has type $t_1 * t_2$
if e_1 has type t_1 and e_2 has type t_2

static

$(x+2, y)$ has type $\text{int} * \text{bool}$
when $x:\text{int}$ and $y:\text{bool}$

runtime

$(x+2, y) \Rightarrow^* (4, \text{true}) : \text{int} * \text{bool}$
when $x:2$ and $y:\text{true}$

Similarly for (e_1, \dots, e_k) when $k > 0$
Also $()$ has type **unit**

lists

(for each type t)

- $[e_1, \dots, e_n]$ has type t list
if for each i , e_i has type t
- $e_1 :: e_2$ has type t list
if e_1 has type t and e_2 has type t list
- $e_1 @ e_2$ has type t list
if e_1 and e_2 have type t list

*all items in a list
must have the same type*

$[1+2, 3+4]$ has type int list

$[1+2, 3+4] \Rightarrow^* [3, 7] : \text{int list}$

functions

- **fn** $x \Rightarrow e$ has type $t_1 \rightarrow t_2$
if e has type t_2 when $x : t_1$

when **applied**
to an argument of type t_1
the result will have type t_2

ensures type-safe
application

fn $x:\text{int} \Rightarrow x+x$ has type $\text{int} \rightarrow \text{int}$

fn $y:\text{int} \Rightarrow x+y$ has type $\text{int} \rightarrow \text{int}$ when $x:\text{int}$

application

- $e_1\ e_2$ has type t_2
if e_1 has type $t_1 \rightarrow t_2$ and e_2 has type t_1

functions must only be
applied to appropriately typed
arguments

$(\text{fn } x:\text{int} \Rightarrow x+x)\ (10+11)$ has type int

example

fn x => if x=0 then 1 else f(x-1)
has type **int -> int** if **f : int -> int**

by rules for

fn x => e
if-then-else
application

...

declarations

- **val** $x = e$ declares $x : t$
if e has type t

val $x = 42$ declares $x : \text{int}$

val $x = y + y$ declares $x : \text{int}$
if $y : \text{int}$

val $f = \text{fn } x \Rightarrow x + 1$ declares $f : \text{int} \rightarrow \text{int}$

declarations

If

d_1 declares $x_1:t_1$

and (with this type for x_1)

d_2 declares $x_2:t_2$

then

$d_1;d_2$ declares $x_1:t_1, x_2:t_2$

val $y = 21;$ val $x = y+y$	declares $y:\text{int}, x:\text{int}$
--	---------------------------------------

declarations

- **fun** $f\ x = e$ declares $f : t_1 \rightarrow t_2$
if, assuming $x : t_1$ and $f : t_1 \rightarrow t_2$, e has type t_2

assuming that
 f is applied to
an argument of type t_1

and
recursive calls to f in e
have type $t_1 \rightarrow t_2$

the result of
 e
will have type t_2

fun $f\ x = \text{if } x=0 \text{ then } 1 \text{ else } f(x-1)$

declares $f : \text{int} \rightarrow \text{int}$

... binds f to a function value of type $\text{int} \rightarrow \text{int}$

let expressions

- **let d in e end** has type **t**
if **d** declares **$x_1 : t_1, \dots, x_k : t_k$**
and, in the scope of these bindings
e has type **t**

let val x = 21 in x + x end has type **int**
and evaluates to **42 : int**

let
 fun f x = **if** x=0 **then** 1 **else** f(x-1)
in
 f 42
end has type **int**
and evaluates to **1 : int**

patterns

when p matches type t

- $_$ matches t always
- 42 matches t iff t is int
- x matches t always (binds $x : t$)
- (p_1, p_2) matches t iff
 t is $t_1 * t_2$, p_1 matches t_1 , p_2 matches t_2 (combine bindings from p_1 and p_2)
- $p_1 :: p_2$ matches t iff
 t is t_1 list, p_1 matches t_1 , p_2 matches t_1 list

examples

- Pattern **$x::R$** matches type **int list** and binds **$x:int$, $R:int\ list$**
- Pattern **$x::R$** matches type **bool list** and binds **$x:bool$, $R:bool\ list$**
- Pattern **$x::y::L$** matches type **'a list** and binds **$x:'a$, $y:'a$, $L:'a\ list$**
- Pattern **$42::R$** matches type **int list** and binds **$R:int\ list$**

clausal functions

- **fn** $p_1 \Rightarrow e_1 \mid \dots \mid p_k \Rightarrow e_k$ has type $t_1 \rightarrow t_2$
if for each i , p_i matches t_1
and produces bindings that give e_i type t_2

each clause $p_i \Rightarrow e_i$ must have *same* type $t_1 \rightarrow t_2$

fn $0 \Rightarrow 0 \mid n \Rightarrow f(n - 1)$ has type $\text{int} \rightarrow \text{int}$
if f has type $\text{int} \rightarrow \text{int}$

clausal declarations

- **fun** f $p_1 = e_1 \mid \dots \mid f$ $p_k = e_k$ declares $f : t_1 \rightarrow t_2$
if for each i ,
 p_i matches t_1 , giving type bindings for which,
assuming $f : t_1 \rightarrow t_2$, e_i has type t_2

each clause $p_i \Rightarrow e_i$ must have *same* type $t_1 \rightarrow t_2$
assuming recursive calls to f in e_i have this type

fun f $0 = 0 \mid f$ $n = f$ $(n - 1)$

declares $f : \text{int} \rightarrow \text{int}$

... and binds f to a **value** of type $\text{int} \rightarrow \text{int}$

example

fun f n = **if** n=0 **then** 1 **else** n + f (n - 1)

declares $f : \text{int} \rightarrow \text{int}$

because, assuming $n : \text{int}$ and $f : \text{int} \rightarrow \text{int}$,

if n=0 **then** 1 **else** n + f (n - 1)

has type int

them's the rules

- There's a typing rule for each program construct
application, fn, fun, let, val, ...
- A type derivation is a sequence of steps,
where each step follows from assumptions or
earlier steps, by an inference rule

(i)	$21 : \text{int}$	by <i>numeral</i> rule
(ii)	$21 : \text{int}$	by <i>numeral</i> rule
(iii)	$21+21 : \text{int}$	from (i), (ii) by $+\text{int}$ rule

We won't always be so fussy about numbering lines!

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***

$\text{int list} \rightarrow \text{int list}$	is a type for	rev
$\text{real list} \rightarrow \text{real list}$	is a type for	rev
$'a \text{ list} \rightarrow 'a \text{ list}$	is a type for	rev

Polymorphic types

- ML has ***type variables***

'a, 'b, 'c

- A type with type variables is ***polymorphic***

'a list -> 'a list

- 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

substitute
a type
for each type variable

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 t is a type for e
& every type for e is an instance of t

rev has most general type $'a\ list \rightarrow 'a\ list$

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
```

exercise

- Use the typing rules to show that `map` and `foldr` have the following most general types:

```
fun map f [ ] = [ ]  
|    map f (x::L) = (f x) :: ((map f) L)
```

`map : ('a -> 'b) -> 'a list -> 'b list`

```
fun foldr g z [ ] = z  
|    foldr g z (x::L) = g(x, foldr g z L)
```

`foldr : ('a * 'b -> 'b) -> 'b -> 'a list -> 'b`

exercise

- Find the most general types...

```
fun zap f [ ] = [ ]  
|   zap f (x::L) = (f x) @ ((zap f) L)
```

```
fun rap f [ ] = [ [ ] ]  
|   rap f (x::L) = (f x) :: ((rap f) L)
```

```
fun tap f [ ] = [ [ ] ]  
|   tap f (x::L) = (x f) :: ((tap f) L)
```

Standard ML of New Jersey (64-bit) v110.99 [built: Mon Jan 11 13:42:54 2021]

```
- fun zap f [ ] = [ ]  
|   zap f (x::L) = (f x) @ ((zap f) L);  
val zap = fn : ('a -> 'b list) -> 'a list -> 'b list  
  
- fun rap f [ ] = [ [ ] ]  
|   rap f (x::L) = (f x) :: ((rap f) L);  
val rap = fn : ('a -> 'b list) -> 'a list -> 'b list list  
  
- fun tap f [ ] = [ [ ] ]  
|   tap f (x::L) = (x f) :: ((tap f) L)= ;  
val tap = fn : 'a -> ('a -> 'b list) list -> 'b list list
```

benefits

- Types can guide program design
- Type errors may indicate bug in code
- An unexpected type may also indicate a bug

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
```

also (more generally!) declares

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

(the *most general* type is polymorphic)

To show **split** : int list -> int list * int list

Using type rules, show each clause “fits” with this type

First 2 clauses: easy. For the third clause...

- Assume (for recursive calls) **split** : int list -> int list * int list
- Need to show that RHS of clause gets type int list * int list when LHS pattern matches int list
 - Matching $x::y::L$ with int list produces $x:int, y:int, L:int\ list$ (by $::$ pattern rule)
 - **split** L : int list * int list (by *application* rule)
 - $val\ (A, B) = \text{split}\ L$ produces $A:int\ list, B:int\ list$ (by *val* rule)
 - $(x::A, y::B) : int\ list * int\ list$ (by $::$ rule, pair rule)
 - $let\ val\ (A, B) = \text{split}\ L\ in\ (x::A, y::B)\ end : int\ list * int\ list$ (by *let* rule)
- So RHS of third clause gets type int list * int list
- True for all clauses, so **split** : int list -> int list * int list (by *fun* rule)

To show $\text{split} : 'a \text{ list} \rightarrow 'a \text{ list} * 'a \text{ list}$

Using type rules, show each clause “fits” with this type

First 2 clauses: easy. For the third clause...

- Assume (for recursive calls) $\text{split} : 'a \text{ list} \rightarrow 'a \text{ list} * 'a \text{ list}$
- Need to show that RHS of clause gets type $'a \text{ list} * 'a \text{ list}$ when LHS pattern matches $'a \text{ list}$
 - Matching $x::y::L$ with $'a \text{ list}$ produces $x: 'a, y: 'a, L: 'a \text{ list}$ (by $::$ pattern rule)
 - $\text{split } L : 'a \text{ list} * 'a \text{ list}$ (by *application* rule)
 - $\text{val } (A, B) = \text{split } L$ produces $A: 'a \text{ list}, B: 'a \text{ list}$ (by *val* rule)
 - $(x::A, y::B) : 'a \text{ list} * 'a \text{ list}$ (by $::$ rule, pair rule)
 - $\text{let val } (A, B) = \text{split } L \text{ in } (x::A, y::B) \text{ end} : 'a \text{ list} * 'a \text{ list}$ (by *let* rule)
- So RHS of third clause gets type $'a \text{ list} * 'a \text{ list}$
- True for all clauses, so $\text{split} : 'a \text{ list} \rightarrow 'a \text{ list} * 'a \text{ list}$ (by *fun* rule)

sorting

Assuming $\text{split} : 'a \text{ list} \rightarrow 'a \text{ list} * 'a \text{ list}$
 $\text{merge} : \text{int list} * \text{int list} \rightarrow \text{int list}$

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

declares $\text{msort} : \text{int list} \rightarrow \text{int list}$

(earlier, we proved correctness of this function)

To show **msort** : int list -> int list

Assume (for recursive calls) **msort** : int list -> int list

For each clause, show that when LHS pattern matches type int list, RHS gets type int list.

- For first clause, LHS pattern `[]` matches int list, and RHS `[]` has type int list.
- For second clause, when LHS pattern `[x]` matches int list we get `x : int`, and RHS `[x] : int list`
- For third clause... LHS pattern is `L`, RHS is `let val (A, B) = split L in merge(msort A, msort B) end`
 - When `L : int list`, because `split : int list -> int list * int list` we get
`split L : int list * int list` (by *application* rule)
 - `val (A, B) = split L` produces `A : int list, B : int list` (by *val* rule)
 - Using type int list -> int list for the recursive calls, we get
`(msort A, msort B) : int list * int list` (by *application, pair* rules)
 - We already know `merge : int list * int list -> int list`, so
`merge (msort A, msort B) : int list` (by *application* rule)
 - `let val (A, B) = split L in merge(msort A, msort B) end : int list` (by *let* rule)
- Hence `msort : int list -> int list` (by *fun* rule)

sorting

Assuming $\text{split} : 'a \text{ list} \rightarrow 'a \text{ list} * 'a \text{ list}$
 $\text{merge} : \text{int list} * \text{int list} \rightarrow \text{int list}$

```
fun msort [] = []  
  | msort L = let  
    val (A, B) = split L  
  in  
    merge(msort A, msort B)  
  end
```

declares $\text{msort} : 'a \text{ list} \rightarrow \text{int list}$

An unexpected type... there's a bug in the code!

Reason: the *type guarantee*... tells us that $\text{msort } L$ doesn't terminate
when L is non-empty!

To show **msort** : 'a list -> int list

Assume (for recursive calls) **msort** : 'a list -> int list

For each clause, show that when LHS pattern matches type 'a list, RHS gets type int list.

- For first clause, LHS pattern `[]` matches 'a list, RHS `[]` has type int list.
- For the only other clause, LHS pattern is `L`, RHS is `let val (A, B) ... end`
 - When `L : 'a list`, because `split : 'a list -> 'a list * 'a list` we get
`split L : 'a list * 'a list` (by *application* rule)
 - `val (A, B) = split L` produces `A : 'a list, B : 'a list` (by *val* rule)
 - Using type 'a list -> int list for the recursive calls, we get
`(msort A, msort B) : int list * int list` (by *application, pair* rules)
 - We already know `merge : int list * int list -> int list`, so
`merge (msort A, msort B) : int list` (by *application* rule)
 - `let val (A, B) = split L in merge(msort A, msort B) end : int list` (by *let* rule)
- Hence `msort : 'a list -> int list` (by *fun* rule)

Without the clause `[x] => [x]`
the type rules don't force 'a to be int

polymorphic values?

Every type has a set of syntactic values.

What about a polymorphic type?

- 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!

- `[]` is the *only* value of type `'a list`

Reasons:

the *type guarantee*

being pedantic

about type variables

- Don't say things like “for all values of type 'a ” or “for all values of type 'a list ” or “e has type 'a”
- Instead you probably meant to say something like “for all types t, and all values of type t list” or you meant to assume some type t for e
- It is OK in talking about typing to say something like “if $f : 'a \rightarrow 'b$ and $x : 'a$ then $f\ x : 'b$ ” because this holds, for all instantiations of the type variables.

summary

- ML does type analysis based on syntax
- Tells you the (most general) type, or a type error message when not well typed
- Guarantee: a well-typed expression won't go wrong!
 - no runtime type errors like **true** + 42
- Although well-typed doesn't imply **correct**, an **unexpected type** is likely to mean **incorrect**.

annotations

- As the ML REPL does type inference, we don't usually *need* to annotate our functions with types
- Instead ML figures out what we meant!

```
fun sum ([ ]:int list) : int = 0  
| sum (x::L) = x + sum L
```

```
fun sum [ ] = 0  
| sum (x::L) = x + sum L
```

ML says: “val sum = fn - : int list -> int”

annotations

- You may *need* to annotate when there's ambiguity

fun add (x, y) = x+y

add : int * int -> int ?

add : real * real -> real ?

Actually,
ML assumes
int * int -> int

fun add (x:int, y:int):int = x+y

fun add (x:int, y:int) = x+y

fun add (x, y:int) = x+y

fun add (x:int, y) = x+y

add : int * int -> int

annotations

- You may *want* to annotate to check if your code has the *intended* type

```
fun isqrt (x:int) : int option =  
  if x<0 then NONE else  
    let  
      fun loop i = if x<i*i then i-1 else loop(i+1)  
    in  
      loop 1  
    end;
```

should be
SOME(i-1)

```
stdIn:6.1-7.64 Error: types of if branches do not agree [tycon mismatch]  
then branch: 'Z option  
else branch: int
```

lessons

- Expressions must be *well-typed* ...prevents bugs
- ML infers (*most general*) types ...less burden
- Can use expression at any *instance* ...re-use code
- Evaluation *respects* type ...predictable
- Design programs using *types* and *specifications* as a guide

well designed
= provably correct