15-150 Fall 2020

Lecture 10
Stephen Brookes

- Type checking
- Type inference
- Polymorphism

type benefits

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

static property	runtime guarantee
e has type t	if e =>* v then v : t
d declares x : t	if d =>* [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`

`x + x` has type `real` if `x` has type `real`

(**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*, at ***parse time***

- 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:int and y:int"

Typing rules

There are syntax-directed rules for "judgements"

e *has type* t

d *declares* $x_1 : t_1 \ldots x_k : t_k$

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

under appropriate assumptions
about the free variables of e and d

type checking

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

type inference

- Use the typing rules to figure out
  (given partial information about variables)
  if **e** is well-typed, and — if so — its most general type

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 $\Longrightarrow^*$ 42 : **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**

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

  similarly for $e_1 <= e_2$ $e_1 > e_2$

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

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	runtime
if x<y **then** x **else** y has type **int** if x:**int** and y:**int**	**if** x<y **then** x **else** y $\Longrightarrow^*$ 4 : 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	**runtime**
$(x+2, y)$ has type int * bool when x:int and y:bool	$(x+2, y) \Longrightarrow^* (4, \text{true})$: int * bool when x:2 and y:true

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

lists (for each type t)

- $[e_1, ..., e_n]$ has type t list
  if for each i, e_i has type t

all items in a list
must have the same type

- $e_1 :: e_2$ 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

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

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

functions

- **fn** x => e has type t_1 -> t_2
  if e has type t_2 when x : t_1

the type of a
function
ensures type-safe
application

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

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

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

fn y => x+y has type **int -> int** when **x:int**

application

- e_1 e_2 has type t_2
  if e_1 has type t_1 -> t_2 and e_2 has type t_1

argument e_2 must have
correct type for function e_1

(fn x => x+x) (10+11) has type **int**

(fn x => x+x) (1.0+1.1) has type **real**

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 : int

val x = y+y declares x : int
if y : int

val f = **fn** x => x + 1 declares f : int -> 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:int, x:int

declarations

- **fun** f x = e declares f : t_1 -> t_2
  if, assuming x : t_1 and f : t_1 -> 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 -> t_2

the result of
e
will have type t_2

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

declares f : int -> int

… binds f to a function value of type int -> int

let expressions

- **let** d **in** e **end** has type t
  if d declares $x_1 : t_1$, ..., $x_k : t_k$
  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 **42::R** matches type **int list**
  and binds **R:int list**

clausal functions

- **fn** p_1 => e_1 | ... | p_k => e_k has type t_1 -> 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 => e_i must have *same* type t_1 -> t_2

- each p_i must *match* type t_1
  - each e_i must have type t_2

fn 0 => 0 | n => f(n - 1) has type int -> int
if f has type int -> int

clausal declarations

- **fun** f p_1 = e_1 | ... | f p_k = e_k declares f : t_1 -> t_2
  if for i = 1 to k,
  p_i matches t_1, giving type bindings for which,
  assuming f : t_1 -> t_2, e_i has type t_2

each clause p_i => e_i must have *same* type t_1 -> t_2
assuming recursive calls to f in e_i have this type

fun f 0 = 0 | f n = f (n - 1)

declares f : int -> int

… and binds f to a value of type int -> int

example

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

declares f : int -> int

because, assuming n : int and f : int -> int,

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

has type int

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***

  substitute a type for each type variable

  int list -> int list

  real list -> real list

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

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

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 `t` is a type for `e`
> & *every* type for `e` is an instance of `t`

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

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)

sorting

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

declares msort : int list -> int list

(earlier, we proved correctness of this function)

sorting

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

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

declares msort : 'a list -> int list

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

Reason: the *type guarantee…* tells us that msort L doesn't terminate when L is non-empty!

polymorphic values?

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

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*

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”

- You’d be quantifying over a very small set of values

- 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

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 inord Empty = [ ]
  |  inord (Node(T1, x, T2)) = (inord T1) @ x :: (inord T2)
```

declares inord : 'a tree -> 'a list

options

datatype 'a option = NONE | SOME **of** 'a

```
fun try (f, [ ]) = NONE
|   try (f, x::L) = case (f x) of
                      NONE => try (f, L)
                    |    y   => y
```

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

```
fun mem (x, [ ])  = false
  |    mem (x, y::L) = (x=y) orelse mem (x, L)
```

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

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.***

demo

- Type rules may seem a bit formal, arcane, fussy…
- … But they embody **simple principles**
  that help you to avoid bugs
- If there's time in class, we'll look at some examples
- See how the ML system gives type information,
  and type error or warning messages;
  learn what they mean

demo

- Look at the file type-programs-lecture10.rtf

exercise

- Check the types of these functions:

```
fun flatten L = foldr (op @) [ ] L

val flatten = foldr (op @) [ ]
```

```
fun splits [ ] = [([ ], [ ])]
|   splits (x::L) =
      ([ ], x::L) :: (map (fn (L1, L2) => (x::L1, L2)) (splits L))

fun perms [ ] = [[ ]]
|     perms (x::L) =
        let
          val S = map splits (perms L)
          val P = flatten S
        in
          map (fn (L1, L2) => L1 @ [x] @ L2) P
        end
```

exercise

- Check these versions

```
fun bad_perms [ ] = [ ]
|     bad_perms (x::L) =
       let
  val S = map splits (bad_perms L)
  val P = flatten S
in
  map (fn (L1, L2) => L1 @ [x] @ L2) P
end;
```

well typed
but
useless

```
fun bad_perms [ ] = [ [ ] ]
|     bad_perms (x::L) =
       let
  val S = map splits (bad_perms L)
in
  map (fn (L1, L2) => L1 @[x]@ L2) S
end;
```

not well typed

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

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 ?

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;

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

should be SOME(i-1)

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