Type Checking and Inference

Type-checking

- (Static) type-checking can reject a program before it runs to prevent the possibility of some errors
  - A feature of statically typed languages
- Dynamically typed languages do little (no?) such checking
  - So might try to treat a number as a function at run-time
- Part of language definition, not just an implementation detail.

Implicit static typing

Static typing and explicit typing are independent.

```
fun f x = (* infer val f : int -> int *)
    if x > 3
    then 42
    else x * 2

fun g x = (* report type error *)
    if x > 3
    then true
    else x * 2
```


Type inference

- **Type inference problem:** Give every binding/expression a type such that type-checking succeeds
  - Fail *if and only if* no solution exists
- Could be a pass before the type-checker
- But often implemented together
- Type inference/checking can be easy, difficult, or *impossible*
  - Easy: Accept all programs
  - Easy: Reject all programs
  - Subtle, elegant, and *not magic*: ML

Human type inference...

What is the type of x?

What is the type of f?

Describe your process.

```
val x = 42

fun f (y, z, w) =
    if y
    then z + x
    else 0
```

Next:

- More examples
  - General algorithm is a slightly more advanced topic
  - Supporting nested functions also a bit more advanced
- Enough to help you “do type inference in your head”
  - And appreciate it is not magic

5

Key steps

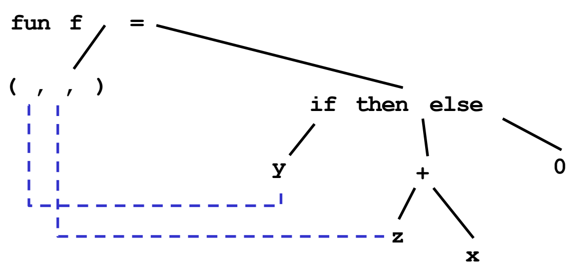
- Determine types of bindings in order
  - Cannot use later bindings.
- For each **`val`** or **`fun`** binding:
  - Analyze definition for all necessary facts (constraints)
  - Example: If see **`x > 0`**, then **`x`** must have type **`int`**
  - Type error if no way for all facts to hold (over-constrained)
- Afterward, use type variables (e.g., **`'a`**) for any unconstrained types
- (Finally, enforce the *value restriction*, discussed later)

6

```
val x = 42

fun f (y, z, w) =
    if y
    then z + x
    else 0
```

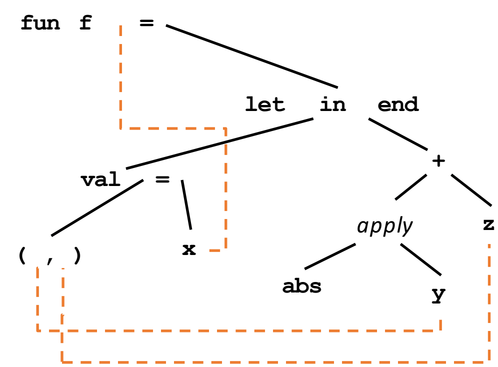
```
val x : int = 42
```



7

```
fun f x =
   let val (y,z) = x in
      (abs y) + z
   end
```

```
abs : int -> int
```



8

Type Inference and Polymorphism

- ML type inference can infer types with type variables
- Inference and polymorphism are orthogonal
  - Languages can have type inference without type variables
  - Languages can have type variables without type inference
  - But both together is a "sweet spot"

9

Key Idea

- Collect all the facts needed for type-checking
- These facts constrain the type of the function
- See code and/or reading notes for:
  - Two examples without type variables
  - And one example that does not type-check
  - Then examples for polymorphic functions
- See slides and notes on website for 2 optional more advanced topics:
  - Value restriction: mutation caused an occasionally annoying type issue.
  - ML type inference is in a sweet spot.

10

Two more (optional) topics

- ML type-inference story so far is too lenient
  - Value restriction limits where polymorphic types can occur
  - See why (mutation!) and then what
- ML is in a "sweet spot"
  - Type inference more difficult without polymorphism
  - Type inference more difficult with subtyping

Important to "finish the story" but these topics are:
- A bit more advanced
- A bit less elegant

11

The Problem

As presented so far, the ML type system is ***unsound***!
- Allows putting a value of type `t1` (e.g., `int`) where we expect a value of type `t2≠t1` (e.g., `string`)

A combination of polymorphism and mutation is to blame:

```
val r = ref NONE (* val r : 'a option ref *)
val _ = r := SOME "hi"
val i = 1 + case !r of NONE => 0 | SOME x => x
```

- Assignment type-checks because (infix) `:=` has type `'a ref * 'a -> unit`, so instantiate with `string`
- Dereference type-checks because `!` has type `'a ref -> 'a`, so instantiate with `int`

12

What to do

Must reject at least one of these three lines

```
val r = ref NONE (* val r : 'a option ref *)
val _ = r := SOME "hi"
val i = 1 + case !r of NONE => 0 | SOME x => x
```

Cannot make special rules for reference types because type-checker cannot know the definition of all type synonyms

- Module system coming up

```
type 'a foo = 'a ref
val f = ref (* val f : 'a -> 'a foo *)
val r = f NONE
```

13

The Value Restriction

```
val r = ref NONE (* val r : ?.X1 option ref *)
val _ = r := SOME "hi"
val i = let val SOME x = !r in 1 + x end
```

- A variable-binding can have a polymorphic type only if the expression is a variable or value
  - Function calls like `ref NONE` are neither
- Else get a warning and unconstrained types are filled in with dummy types (basically unusable)
- Not obvious this suffices to make type system sound, but it does

14

Value Restriction downside

Causes problems when unnecessary because (not using mutation):

```
val pairWithOne = List.map (fn x => (x,1))
(* does not get type 'a list -> ('a*int) list *)
```

The type-checker does not know `List.map` is not making a mutable reference.

Workarounds for partial application:

```
fun pairWithOne xs = List.map (fn x => (x,1)) xs
(* 'a list -> ('a*int) list *)
```

- give up on polymorphism; write explicit non-polymorphic type

```
val pairWithOne : int list -> (int * int) list =
   List.map (fn x => (x,1))
val pairWithOne = List.map (fn (x : int) => (x,1))
```

A local optimum

- Despite the value restriction, ML type inference is elegant and fairly easy to understand
- **More difficult *without* polymorphism**
  - What type should length-of-list have?
- **More difficult *with* subtyping**
  - Suppose pairs are supertypes of wider tuples
  - Then `val (y,z) = x` constrains `x` to have at least two fields, not exactly two fields
  - Depending on details, languages can support this, but types often more difficult to infer and understand
  - Will study subtyping later, but not with type inference

16