PAUL G. ALLEN SCHOOL
OF COMPUTER SCIENCE & ENGINEERING

CSE341: Programming Languages

Lecture 11
Type Inference

Dan Grossman
Autumn 2017

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 (none?) such checking
  - So might try to treat a number as a function at run-time
- Will study relative advantages after some Racket
  - Racket, Ruby (and Python, Javascript, …) dynamically typed
- ML (and Java, C#, Scala, C, C++) is statically typed
  - Every binding has one type, determined "at compile-time"

Implicitly typed

- ML is statically typed
- ML is implicitly typed: rarely need to write down types

```
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
```

- Statically typed: Much more like Java than Javascript!

Type inference

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

Overview

- Will describe ML type inference via several 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

Key steps

- Determine types of bindings in order
  - (Except for mutual recursion)
  - So you cannot use later bindings: will not type-check
- 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
  - Example: An unused argument can have any type
- (Finally, enforce the *value restriction*, discussed later)

Very simple example

After this example, will go much more step-by-step
- Like the automated algorithm does

```
val x = 42 (* val x : int *)
fun f (y, z, w) =
    if y (* y must be bool *)
    then z + x (* z must be int *)
    else 0 (* both branches have same type *)
(* f must return an int
   f must take a bool * int * ANYTHING
   so val f : bool * int * 'a -> int
 *)
```


Relation to Polymorphism

- Central feature of ML type inference: it can infer types with type variables
  - Great for code reuse and understanding functions
- But remember there are two orthogonal concepts
  - Languages can have type inference without type variables
  - Languages can have type variables without type inference

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
    - Nothing changes, just under-constrained: some types can “be anything” but may still need to be the same as other types

Material after here is optional,
but is an important part of the full story

Two more topics

- ML type-inference story so far is too lenient
  - Value restriction limits where polymorphic types can occur
  - See why 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
- Will not be on the exam

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`** (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 + valOf (!r)
```

- 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`**

What to do

To restore soundness, need a stricter type system that rejects at least one of these three lines

```
val r = ref NONE (* val r : 'a option ref *)
val _ = r := SOME "hi"
val i = 1 + valOf (!r)
```

- And cannot make special rules for reference types because type-checker cannot know the definition of all type synonyms
  – Due to module system

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


The fix

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

- Value restriction: 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

The downside

As we saw previously, the value restriction can cause problems when it is unnecessary because we are 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

Saw workarounds in previous segment on partial application
  – Common one: wrap in a function binding

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


A local optimum

- Despite the value restriction, ML type inference is elegant and fairly easy to understand
- More difficult *without* polymorpism
  – 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