Type Checking and Type Inference

Type checking

Static:
Can reject a program before it runs to prevent possibility of some errors.

Dynamic:
Little/no static checking.
May try to treat a number as a function during evaluation. Report error then.

Part of language definition,
not an implementation detail.

Type inference

static types ≠ explicit 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

Problem:
– Give every binding/expression a type such that type checking succeeds.
– Fail if and only if no solution exists

Implementation:
– Could be a pass before type checker
– Often implemented in type checker

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.

Next:
• More examples, but:
  – General algorithm is a slightly more advanced topic
  – Supporting nested functions also a bit more advanced

• Enough to “do type inference in your head”
  – And appreciate it is not magic

Key steps

1. Determine types of bindings in order
   – Cannot use later bindings.

2. For each val or fun binding:
   – Analyze definition for all necessary facts (constraints).
     • Example: \( x > 0 \Rightarrow x : \text{int} \)
     – Type error if no way for all facts to hold (over-constrained)

3. Use type variables (\( 'a \ ... \)) for any unconstrained types.
   Inference and polymorphism are orthogonal; together = “sweet spot”.
   Results in most general feasible type.

4. Enforce the value restriction, discussed later.

See code examples in inf.sml.
Problem: unsoundness!

Combine polymorphism and mutation:

- Assignment type-checks:
  - (op:=) :'a ref * 'a -> unit
  - instantiate string for 'a
  - use as string ref * string -> unit
- Dereference type-checks:
  - !: 'a ref -> 'a
  - instantiate int for 'a
  - use as int ref -> int
- \texttt{val i : int = "hi"}

Solution

Reject at least one of these lines

\texttt{val thing = ref NONE (* : 'a option ref *)}
\texttt{val _ = thing := SOME "hi"
val i = 1 + case !thing of NONE => 0 | SOME x => x
}

Cannot just special-case ref types. Abstract types!

\texttt{signature HIDE = sig}
\texttt{type 'a hidden}
\texttt{val make : 'a -> 'a hidden}
\texttt{val thing : 'a hidden}
\texttt{end}
\texttt{structure Hide :> HIDE = struct}
\texttt{type 'a hidden = 'a ref}
\texttt{val make = ref}
\texttt{val thing = make NONE}
\texttt{end}

The Value Restriction

A variable-binding can have a polymorphic type only if the expression is a variable or value.

- Function calls like \texttt{ref NONE} are neither

Otherwise

Warning: type vars not generalized because of value restriction are instantiated to dummy types (Basically unusable)

Not obvious: suffices to make type system sound.

Value Restriction downside

Causes problems when unnecessary (no mutation) because:

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

Type-checker does not know \texttt{List.map} is not making a mutable ref.

Workarounds for partial application:

wrap in a function binding to keep it polymorphic

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

give up on polymorphism; write explicit non-polymorphic type

\texttt{val pairWithOne : int list -> (int * int) list =}
\texttt{List.map (fn x => (x,1))}
\texttt{val pairWithOne = List.map (fn (x : int) => (x,1))}
A local optimum

ML type inference is elegant and fairly easy to understand, despite the value restriction.

More difficult without polymorphism
  – What type should length-of-list have?

More difficult with subtyping (later)
  – Suppose pairs are supertypes of wider tuples
  – Then `val (y, z) = x` constrains `x` to have at least two fields, not exactly two fields.
  – Sometimes languages can support this, but types are often more difficult to infer and understand.