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

Problems:
- 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

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
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 \texttt{val} or \texttt{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 (‘\( \alpha \) \ldots’) for any unconstrained types.
   Inference and polymorphism are orthogonal; together = "sweet spot"

4. Enforce the \textit{value restriction}, discussed later.

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

Combine polymorphism and mutation:

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

\[
\begin{align*}
\text{val thing} & = \text{ref NONE (* : 'a option ref *)} \\
\text{val _} & = \text{thing} := \text{SOME "hi"} \\
\text{val i} & = 1 + \text{case !thing of NONE => 0 | SOME x} \Rightarrow x
\end{align*}
\]

Solution

Reject at least one of these lines

\[
\begin{align*}
\text{val thing} & = \text{ref NONE (* : 'a option ref *)} \\
\text{val _} & = \text{thing} := \text{SOME "hi"} \\
\text{val i} & = 1 + \text{case !thing of NONE => 0 | SOME x} \Rightarrow x
\end{align*}
\]

Cannot just special-case ref types. Abstract types!

\[
\begin{align*}
\text{signature HIDE} & = \text{sig} \\
\text{type 'a hidden} & \text{ val make : 'a -> 'a hidden} \\
\text{val thing} & = \text{'a hidden} \\
\text{end} \\
\text{structure Hide} & \text{ :> HIDE = struct} \\
\text{type 'a hidden} & = \text{'a ref} \\
\text{val make} & = \text{ref} \\
\text{val thing} & = \text{make NONE} \\
\text{end}
\end{align*}
\]

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.

\[
\begin{align*}
\text{val thing} & = \text{ref NONE (* : ?.X1 option ref *)} \\
\text{val _} & = \text{thing} := \text{SOME "hi"} \\
\text{val i} & = 1 + \text{case !thing of NONE => 0 | SOME x} \Rightarrow x
\end{align*}
\]

Value Restriction downside

Causes problems when unnecessary (no mutation) because:

\[
\begin{align*}
\text{val pairWithOne} & = \text{List.map (fn x} \Rightarrow (x,1)) \\
\text{(* does not get type 'a list -> ('a*int) list *)}
\end{align*}
\]

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

\[
\begin{align*}
\text{fun pairWithOne} & \text{ xs} = \text{List.map (fn x} \Rightarrow (x,1)) \text{ xs} \\
\text{(* 'a list -> ('a*int) list *)}
\end{align*}
\]

give up on polymorphism; write explicit non-polymorphic type

\[
\begin{align*}
\text{val pairWithOne} & : \text{int list} \rightarrow (\text{int * int}) \text{ list} = \\
\text{List.map (fn x} \Rightarrow (x,1)) \\
\text{val pairWithOne} & = \text{List.map (fn (x : int) \Rightarrow (x,1))}
\end{align*}
\]
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 (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.