Type-checking can reject a program before it runs to prevent the possibility of some errors.

Dynamically typed languages do little such checking.

So might try to treat a number as a function at run-time.

Part of language definition, not just an implementation detail.

Type inference

Type inference problem: Give every binding/expression a type such that type-checking succeeds.

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.

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

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 \( x > 0 \), 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)
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"

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.

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

The Problem

As presented so far, the ML type system is **unsound**:

• Allows putting a value of type \( t_1 \) (e.g., \( \text{int} \)) where we expect a value of type \( t_2 \neq t_1 \) (e.g., \( \text{string} \))

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 \text{string}"

• Dereference type-checks because ! has type
  \('a ref -> 'a, so instantiate with \text{int}"

```
What to do

Must reject at least one of these three lines

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

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

• Module system coming up

\[
\begin{align*}
  \text{type } 'a \text{ foo } & \text{=} 'a \text{ ref} \\
  \text{val } f & \text{ = ref (* val } f : 'a \rightarrow 'a \text{ foo *)} \\
  \text{val } r & \text{ = } f \text{ NONE}
\end{align*}
\]

The Value Restriction

\[
\begin{align*}
  \text{val } r & \text{ = ref } \text{NONE (* val } r : ?.\text{X1 option ref *)} \\
  \text{val } _- & \text{ = } r := \text{SOME "hi"} \\
  \text{val } i & \text{ = let val } \text{SOME } x \text{ = } !r \text{ in } 1 + x \text{ end}
\end{align*}
\]

• 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

Value Restriction downside

Causes problems when unnecessary because not using mutation:

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

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

Workarounds for partial application:

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

• Give up on polymorphism; write explicit non-polymorphic type

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

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
    • 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