Type Inference

Consider the expression \{0, true\}.1:

\[
\begin{array}{c}
\emptyset \vdash 0 : \text{Nat} \\
\emptyset \vdash \text{true} : \text{Bool} \\
\emptyset \vdash \{0, \text{true}\} : \text{Nat} \times \text{Bool} \\
\emptyset \vdash \{0, \text{true}\}.1 : \text{Nat}
\end{array}
\]
Type Reconstruction – Example 1

Consider the expression \{x, y\}.1, and the problem of determining the most general typing context under which the expression is well-typed.

\[
\begin{array}{c}
\text{x} & \alpha \\
\downarrow & \downarrow \\
\text{\{x, y\}} & \alpha \times \beta \\
\downarrow & \\
\text{1} & \alpha \\
\end{array}
\]

\[
\frac{[x : \alpha] \vdash x : \alpha \quad \text{T-VAR} \quad [y : \beta] \vdash y : \beta \quad \text{T-VAR}}{[x : \alpha, y : \beta] \vdash \{x, y\} : \alpha \times \beta \quad \text{T-PAIR}} \quad \text{T-PAIR} \quad \text{T-Proj1}
\]

Key Idea: Permit type variables in type expressions.

Type Reconstruction — Example 2

Can we infer the types of the subexpressions (or at least, some constraints on their types) assuming the full expression is well-typed?

- Consider \(\lambda x. \lambda y. x\)
- Treat this as \(\lambda x : \alpha. \lambda y : \beta. x\)
Type Reconstruction —Example 3

Consider \( \lambda x \text{ if } x.1 \text{ then } 0 \text{ else } x.2 \)

- Since the entire expression is of the form \( \lambda x. e \) its type is \( \alpha \to \beta \), where \( x : \alpha \) and \( e : \beta \).
- Determine the type of \( e \) under the type environment \([x : \alpha]\).
- Since \( e \) is an “if” expression, we have the following:
  1. \( x.1 : Boolean \)
  2. \( 0 : \beta \)
  3. \( x.2 : \beta \)
- From \( 0 : \beta \), we get \( \beta = \text{Nat} \).
- From \( x : \alpha \) and \( x.1 : \text{Bool} \), and from the rule T-PAIR, we get \( \alpha = \text{Bool} \times \gamma \).
- From \( x : \alpha \) and \( x.2 : \text{Nat} \), and from the rule T-PROJ2, we get, \( \alpha = \delta \times \text{Nat} \).
- From the last two steps, we get \( \alpha = \text{Bool} \times \text{Nat} \) and hence the original expressions type is:
  \( \text{Bool} \times \text{Nat} \to \text{Nat} \)
Type Unification — Motivation

- In the previous example, we initially assigned types denoted by type variables to certain identifiers.
- As type inference proceeded, we discovered constraints on those type variables, e.g. $\alpha = \text{bool} \times \gamma$.
- The constraints are all equality constraints, hence if $\alpha_1 = \alpha_2$ and $\alpha_2 = \alpha_3$ then $\alpha_1 = \alpha_3$.
- From $\text{bool} \times \gamma = \delta \times \text{int}$, we get $\gamma = \text{int}$ and $\delta = \text{bool}$.
- More formally, the constraints are solved to obtain the most general types for the type variables, using unification.
- The most general type of an expression is called its principal type.

Type Unification

- Two type expressions $T_1$, $T_2$ are said to unify iff there is some substitution $\theta$ to the type variables in $T_1$ and $T_2$ such that $T_1\theta = T_2\theta$.
- Examples:
  - A type variable $\alpha$ unifies with any type expression $T$ that does not contain $\alpha$.
    - Note: The above illustrates the use of occur check.
    - $\alpha$ and $\alpha \times \beta$ do not unify.
  - $\alpha_1 \times \alpha_2$ and $\beta_1 \times \beta_2$ unify if $\alpha_1 = \beta_1$ and $\alpha_2 = \beta_2$.
  - $\alpha_1 \rightarrow \alpha_2$ and $\beta_1 \rightarrow \beta_2$ unify if $\alpha_1 = \beta_1$ and $\alpha_2 = \beta_2$.
- Type unification is the core of Hindley-Milner type inference.
ML’s Type Inference

ML’s type inference proceeds very similar to our definition of types, with two main changes.

1. ML’s `let` is polymorphic: every use of a let-defined identifier may have a different, incompatible type. Our rules have (implicitly) demanded that all uses of an identifier have the same consistent set of types.

2. ML has user-defined types and uses pattern matching in function definitions.

Example of `let`-polymorphism

```
let f = fun x -> x
in (f 0, f true)
```

ML’s response:
```
- : int * bool = (0, true)
```

In our type system:

- `f`’s type is reconstructed as \( \alpha \rightarrow \alpha \).
- From `f 0`, we will get a constraint \( \alpha = \text{Nat} \).
- From `f true`, we will get a constraint \( \alpha = \text{Bool} \).
- When taken together, these constraints are unsatisfiable!
let-polymorphism

- The type of $\lambda x. \ x$ is not just $\alpha \to \alpha$, but $\forall \alpha \ \alpha \to \alpha$.

- In $f \ 1$, we should consider an instance of $f$, i.e. a specific type $\alpha' \to \alpha'$.

- In $f \ true$, we should consider another instance of $f$, i.e. a specific type $\alpha'' \to \alpha''$.

- Constraints on $\alpha'$ do not interact with those on $\alpha''$. 