Type Reconstruction

Principles of Programming Languages

CSE 526
Consider the expression `{0, true}.1:`
Consider the expression \(\{0, \text{true}\}.1\):

\[
\begin{array}{c}
\emptyset \vdash 0 : \text{Nat} \\
\text{T-ZERO} \\
\emptyset \vdash \text{true} : \text{Bool} \\
\text{T-TRUE} \\
\hline
\emptyset \vdash \{0, \text{true}\} : \text{Nat} \times \text{Bool} \\
\text{T-PAIR} \\
\emptyset \vdash \{0, \text{true}\}.1 : \text{Nat} \\
\text{T-PROJ1}
\end{array}
\]
Consider the expression \( \{0, \text{true}\}.1 \):

\[
\begin{align*}
\emptyset & \vdash 0 : \text{Nat} & \quad \text{T-ZERO} & \quad \emptyset & \vdash \text{true} : \text{Bool} & \quad \text{T-TRUE} \\
\emptyset & \vdash \{0, \text{true}\} : \text{Nat} \times \text{Bool} & \quad \text{T-PAIR} & \quad \emptyset & \vdash \{0, \text{true}\}.1 : \text{Nat} \\
\end{align*}
\]

Diagram:

```
  0  Nat
   \downarrow
\{ \}
   \uparrow
   \text{Nat} \times \text{Bool}
     \downarrow
   .1  Nat
```
Consider the expression \( \{x, y\}.1 \), and the problem of determining the most general typing context under which the expression is well-typed.
Consider the expression \( \{x, y \}.1 \), and the problem of determining the most general typing context under which the expression is well-typed.

Key Idea: Permit type variables in type expressions.
Consider the expression \{x, y\}.1, and the problem of determining the most general typing context under which the expression is well-typed.

![Type Reconstruction Diagram](image)

Key Idea:
Permit type variables in type expressions.

\[
\frac{\frac{\frac{\frac{\frac{\text{[x : } \alpha]}{\vdash x : \alpha}}{} \text{T-VAR}}{} \frac{\frac{\frac{\frac{\frac{\text{[y : } \beta]}{\vdash y : \beta}}{} \text{T-VAR}}{} \text{T-PAR}}{} \text{T-PROJ1}}{\text{[x : } \alpha, y : \beta] \vdash \{x, y\} : \alpha \times \beta}{} \text{T-VAR}}{} \frac{\frac{\frac{\frac{\frac{\frac{\text{[x : } \alpha, y : \beta]}{\vdash \{x, y\} : \alpha \times \beta}}{} \text{T-PAR}}{} \text{T-PROJ1}}{\text{[x : } \alpha, y : \beta] \vdash \{x, y\}.1 : \alpha}}{\text{T-VAR}}{} \text{T-PAR}}{} \text{T-PROJ1}}
\]
Consider the expression \( \{x, y\}.1 \), and the problem of determining the most general typing context under which the expression is well-typed.

\[
\begin{align*}
\Gamma & : \alpha \\
\Delta & : \beta \\
\{ \alpha \times \beta \} & : \alpha \\
\{ \alpha \} & : \alpha \\
\end{align*}
\]

**Key Idea:** Permit type variables in type expressions.
Can we infer the types of the subexpressions (or at least, some constraints on their types) assuming the full expression is well-typed?

- Consider $\lambda x. \lambda y. x$
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 $\lambda x : \alpha. \lambda y : \beta. x$
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 $\lambda x : \alpha. \lambda y : \beta. x$
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 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 $\lambda x : \alpha. \lambda y : \beta. x$

Diagram:

- $\lambda x : \alpha. \lambda y : \beta. x$
- $\text{var}(x) : \alpha$
- $y : \beta$
- $\lambda : \beta \rightarrow \alpha$
- $x : \alpha$

Programming Languages

Type Reconstruction

CSE 526 4 / 11
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 $\lambda x : \alpha. \lambda y : \beta. x$

```
\lambda x : \alpha \\
\lambda y : \beta \\
\text{var}(x) : \alpha
```

```
\lambda : \alpha \rightarrow \beta \rightarrow \alpha
```
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. $x.1 : \text{Bool}$
  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:
  \[ \text{Bool} \times \text{Nat} \rightarrow \text{Nat} \]
Type Reconstruction

—Example 3

\[ \lambda x \text{ if } .1 \cdot .2 \]

\[ x \quad \text{if} \quad .1 \quad 0 \quad .2 \]

\[ x \quad \lambda \]

\[ x \quad \text{Nat} \quad \text{Nat} \quad \text{Nat} \quad \text{Nat} \]

\[ \text{Nat} \mid \text{Bool} \times \text{Nat} \]

\[ \text{Bool} \times \text{Nat} \rightarrow \text{Nat} \]
Type Reconstruction — Example 3

\[ \lambda x \text{ if } .1 \quad 0 \quad .2 \]

\[ x : \alpha \quad \text{if} \quad x \]

\[ \lambda \]

\[ \alpha : \text{Bool} \quad \text{Nat} \]

\[ \alpha = \delta \times \text{Nat} \]

\[ \text{Nat} \mid \alpha = \text{Bool} \times \text{Nat} \]

\[ \text{Bool} \times \text{Nat} \rightarrow \text{Nat} \]
Type Reconstruction

—Example 3

\[ \lambda \quad x \quad \text{if} \quad x \quad x \]

\[ \cdot 1 : \text{Bool} \quad 0 \quad \cdot 2 \]

\[ x : \alpha \]

\[ \lambda \]

\[ \alpha = \text{Bool} \times \gamma \]

\[ \text{Nat} \quad | \quad \alpha = \text{Bool} \times \text{Nat} \]

\[ \text{Nat} \rightarrow \text{Nat} \]
Type Reconstruction — Example 3

\[ x : \alpha = \text{Bool} \times \gamma \]

\[ .1 : \text{Bool} \quad 0 \quad .2 \]

\[ \lambda \quad \text{if} \quad x \]

\[ x : \alpha = \text{Bool} \times \text{Nat} \]

\[ \text{Nat} \mid x : \text{Bool} \times \text{Nat} \rightarrow \text{Nat} \]
Type Reconstruction — Example 3

\[ \lambda x \text{ if } \begin{align*} x & : \alpha = \text{Bool} \times \gamma \\ .1 & : \text{Bool} \\ .2 & \cdot \text{Nat} \end{align*} \]
Type Reconstruction — Example 3

\[ x : \alpha = \text{Bool} \times \gamma \]

\[ \lambda \]

\[ .1 : \text{Bool} \quad 0 : \text{Nat} \quad .2 : \text{Nat} \]

\[ x : \alpha \quad \text{if} \]
Type Reconstruction — Example 3

\[ \lambda \ x \ \text{if} \ x \ \text{then} .1 : \text{Bool} \ \text{else} \ 0 : \text{Nat} \rightarrow \text{Nat} \]

\[ x : \alpha = \text{Bool} \times \gamma \quad \alpha = \delta \times \text{Nat} \]

\[ x : \alpha = \text{Bool} \times \gamma \]

\[ .1 : \text{Bool} \\
.2 : \text{Nat} \]
Type Reconstruction — Example 3

\[
x : \alpha = \text{Bool} \times \gamma \quad x : \alpha = \delta \times \text{Nat}
\]

\[
.1 : \text{Bool} \quad 0 : \text{Nat} \quad .2 : \text{Nat}
\]

\[
x : \alpha \\
\text{if} : \text{Nat} \mid \alpha = \text{Bool} \times \text{Nat}
\]

\[
\lambda
\]
Type Reconstruction — Example 3

\[
\begin{align*}
\lambda & : \text{Bool} \times \text{Nat} \\
\text{x : } \alpha & = \text{Bool} \times \gamma \\
\text{.1 : } \text{Bool} & \\
\text{.2 : } \text{Nat} & \\
\text{0 : } \text{Nat} & \\
\text{if : } \text{Nat} \mid \alpha & = \text{Bool} \times \text{Nat} \\
\text{x : } \alpha & = \delta \times \text{Nat} \\
\text{λ : } \text{Bool} \times \text{Nat} \rightarrow \text{Nat}
\end{align*}
\]
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 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

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

ML’s response:

```
- : int * bool = (0, true)
```
Example of let-polymorphism

```ml
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!
The type of $\lambda x. x$ is not just $\alpha \rightarrow \alpha$, but

\[ \forall \alpha \alpha \rightarrow \alpha \]

In $f \text{true}$, we should consider another instance of $f$, i.e. a specific type $\alpha' \rightarrow \alpha'$.

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

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

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

In $\texttt{f 1}$, we should consider an \textit{instance} of $\texttt{f}$, i.e. a specific type $\alpha' \rightarrow \alpha'$.
let-polymorphism

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

- In $\text{if } 1$, we should consider an instance of $\text{if}$, i.e. a specific type $\alpha' \rightarrow \alpha'$.

- In $\text{if } \text{true}$, we should consider another instance of $\text{if}$, i.e. a specific type $\alpha'' \rightarrow \alpha''$. 
The type of $\lambda x. x$ is not just $\alpha \rightarrow \alpha$, but $\forall \alpha \; \alpha \rightarrow \alpha$.

In $\text{if} \; 1$, we should consider an instance of $\text{if}$, i.e. a specific type $\alpha' \rightarrow \alpha'$.

In $\text{if} \; \text{true}$, we should consider another instance of $\text{if}$, i.e. a specific type $\alpha'' \rightarrow \alpha''$.

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