CS520, Handout 18, 2 Nov 2006

Type Reconstruction

Reading: [Pierce, Chapter 22]

Scope of [Pierce, Chapter 22]

[Pierce, Chapter 22] considers **type reconstruction** (also called **type inference**) for the language of the lambda-calculus with arithmetic expressions, possibly augmented with **`let-in`** (the latter in [Pierce, Sect 22.7]).

This handout follows the general organization of [Pierce, Chapt 22]. But it also changes many of the technical details, hopefully to make the presentation simpler and more transparent.

Scope of [Pierce, Chapter 22]

[Pierce, Chapter 22] considers **type reconstruction** (also called **type inference**) for the language of the lambda-calculus with arithmetic expressions, possibly augmented with **`let-in`** (the latter in [Pierce, Sect 22.7]).

This handout follows the general organization of [Pierce, Chapt 22]. But it also changes many of the technical details, hopefully to make the presentation simpler and more transparent.

Syntax of Terms [Handout 06, page 2] + [Handout 07, page 2], i.e., lambda-calculus augmented with arithmetic expresssions.

Evaluation Rules [Handout 02, pages 4-5] + [Handout 07, page 4].

Syntax of Types

Different from the syntax in [Pierce, Ch 9 and 11] and [Handouts 07, 08 and 09]:

$$
\begin{array}{lll}
X & ::= \ \mathtt{A} \mid \mathtt{B} \mid \cdots \mid \mathtt{X} \mid \mathtt{Y} \mid \cdots & \textit{type variables} \\
T & ::= & \textit{types} \\
 & \quad \mathtt{Nat} \mid \mathtt{Bool} \mid \cdots & \quad \textit{base type} \\
 & \quad X & \quad \textit{type variable} \\
 & \quad T \to T & \quad \textit{type of functions}
\end{array}
$$

We added type variables to the syntax of types. A type variable can be instantiated with other types via substitutions.

Syntax of Types

Different from the syntax in [Pierce, Ch 9 and 11] and [Handouts 07, 08 and 09]:

$$\begin{array}{lll} X & ::= & \mathtt{A} \mid \mathtt{B} \mid \cdots \mid \mathtt{X} \mid \mathtt{Y} \mid \cdots \qquad \textit{type variables} \\ T & ::= & \qquad\qquad\qquad\qquad\qquad \textit{types} \\ & & \mathtt{Nat} \mid \mathtt{Bool} \mid \cdots \qquad\qquad \textit{base type} \\ & & X \qquad\qquad\qquad\qquad\quad \textit{type variable} \\ & & T \to T \qquad\qquad\qquad\quad \textit{type of functions} \end{array}$$

We added type variables to the syntax of types. A type variable can be instantiated with other types via substitutions.

Type contexts [Handout 07, p 3].

Typing rules [Handout 06, page 3] + [Handout 07, page 3].

Example of a Typing Derivation

$$\dfrac{\dfrac{\dfrac{\dfrac{(\mathtt{x}:\mathtt{A}) \in (\mathtt{x}:\mathtt{A}, \mathtt{y}:\mathtt{B}, \mathtt{z}:\mathtt{A})}{\mathtt{x}:\mathtt{A}, \mathtt{y}:\mathtt{B}, \mathtt{z}:\mathtt{A} \vdash \mathtt{x}:\mathtt{A}}\,\text{T-Var}}{\mathtt{x}:\mathtt{A}, \mathtt{z}:\mathtt{A} \vdash (\lambda \mathtt{y}:\mathtt{B}.\,\mathtt{x}) : \mathtt{B} \to \mathtt{A}}\,\text{T-Abs}}{\mathtt{z}:\mathtt{A} \vdash (\lambda \mathtt{x}:\mathtt{A}.\,\lambda \mathtt{y}:\mathtt{B}.\,\mathtt{x}) : \mathtt{A} \to \mathtt{B} \to \mathtt{A}}\,\text{T-Abs} \qquad \dfrac{(\mathtt{z}:\mathtt{A}) \in (\mathtt{z}:\mathtt{A})}{\mathtt{z}:\mathtt{A} \vdash \mathtt{z}:\mathtt{A}}\,\text{T-Var}}{\mathtt{z}:\mathtt{A} \vdash (\lambda \mathtt{x}:\mathtt{A}.\,\lambda \mathtt{y}:\mathtt{B}.\,\mathtt{x})\,\mathtt{z} : \mathtt{B} \to \mathtt{A}}\,\text{T-App}$$

Example of a Typing Derivation

$$\dfrac{\dfrac{\dfrac{\dfrac{(\mathtt{x}:\mathtt{A}) \in (\mathtt{x}:\mathtt{A}, \mathtt{y}:\mathtt{B}, \mathtt{z}:\mathtt{A})}{\mathtt{x}:\mathtt{A}, \mathtt{y}:\mathtt{B}, \mathtt{z}:\mathtt{A} \vdash \mathtt{x}:\mathtt{A}}\,\text{T-Var}}{\mathtt{x}:\mathtt{A}, \mathtt{z}:\mathtt{A} \vdash (\lambda \mathtt{y}:\mathtt{B}.\,\mathtt{x}) : \mathtt{B} \to \mathtt{A}}\,\text{T-Abs}}{\mathtt{z}:\mathtt{A} \vdash (\lambda \mathtt{x}:\mathtt{A}.\,\lambda \mathtt{y}:\mathtt{B}.\,\mathtt{x}) : \mathtt{A} \to \mathtt{B} \to \mathtt{A}}\,\text{T-Abs} \qquad \dfrac{(\mathtt{z}:\mathtt{A}) \in (\mathtt{z}:\mathtt{A})}{\mathtt{z}:\mathtt{A} \vdash \mathtt{z}:\mathtt{A}}\,\text{T-Var}}{\mathtt{z}:\mathtt{A} \vdash (\lambda \mathtt{x}:\mathtt{A}.\,\lambda \mathtt{y}:\mathtt{B}.\,\mathtt{x})\,\mathtt{z} : \mathtt{B} \to \mathtt{A}}\,\text{T-App}$$

Substituting `Nat` for `A`, we get another valid typing derivation:

$$\dfrac{\dfrac{\dfrac{\dfrac{(\mathtt{x}:\mathtt{Nat}) \in (\mathtt{x}:\mathtt{Nat}, \mathtt{y}:\mathtt{B}, \mathtt{z}:\mathtt{Nat})}{\mathtt{x}:\mathtt{Nat}, \mathtt{y}:\mathtt{B}, \mathtt{z}:\mathtt{Nat} \vdash \mathtt{x}:\mathtt{Nat}}}{\mathtt{x}:\mathtt{Nat}, \mathtt{z}:\mathtt{Nat} \vdash (\lambda \mathtt{y}:\mathtt{B}.\,\mathtt{x}) : \mathtt{B} \to \mathtt{Nat}}}{\mathtt{z}:\mathtt{Nat} \vdash (\lambda \mathtt{x}:\mathtt{Nat}.\,\lambda \mathtt{y}:\mathtt{B}.\,\mathtt{x}) : \mathtt{Nat} \to \mathtt{B} \to \mathtt{Nat}} \qquad \dfrac{(\mathtt{z}:\mathtt{Nat}) \in (\mathtt{z}:\mathtt{Nat})}{\mathtt{z}:\mathtt{Nat} \vdash \mathtt{z}:\mathtt{Nat}}}{\mathtt{z}:\mathtt{Nat} \vdash (\lambda \mathtt{x}:\mathtt{Nat}.\,\lambda \mathtt{y}:\mathtt{B}.\,\mathtt{x})\,\mathtt{z} : \mathtt{B} \to \mathtt{Nat}}$$

Example of a Typing Derivation

$$\dfrac{\dfrac{\dfrac{\dfrac{(\mathtt{x}:\mathtt{A}) \in (\mathtt{x}:\mathtt{A}, \mathtt{y}:\mathtt{B}, \mathtt{z}:\mathtt{A})}{\mathtt{x}:\mathtt{A}, \mathtt{y}:\mathtt{B}, \mathtt{z}:\mathtt{A} \vdash \mathtt{x}:\mathtt{A}}\,\text{T-Var}}{\mathtt{x}:\mathtt{A}, \mathtt{z}:\mathtt{A} \vdash (\lambda \mathtt{y}:\mathtt{B}.\,\mathtt{x}) : \mathtt{B} \to \mathtt{A}}\,\text{T-Abs}}{\mathtt{z}:\mathtt{A} \vdash (\lambda \mathtt{x}:\mathtt{A}.\,\lambda \mathtt{y}:\mathtt{B}.\,\mathtt{x}) : \mathtt{A} \to \mathtt{B} \to \mathtt{A}}\,\text{T-Abs} \qquad \dfrac{(\mathtt{z}:\mathtt{A}) \in (\mathtt{z}:\mathtt{A})}{\mathtt{z}:\mathtt{A} \vdash \mathtt{z}:\mathtt{A}}\,\text{T-Var}}{\mathtt{z}:\mathtt{A} \vdash (\lambda \mathtt{x}:\mathtt{A}.\,\lambda \mathtt{y}:\mathtt{B}.\,\mathtt{x})\,\mathtt{z} : \mathtt{B} \to \mathtt{A}}\,\text{T-App}$$

Substituting `Nat` for `A` and `Bool` for `B`, we get another valid typing derivation:

$$\dfrac{\dfrac{\dfrac{\dfrac{(\mathtt{x}:\mathtt{Nat}) \in (\mathtt{x}:\mathtt{Nat}, \mathtt{y}:\mathtt{Bool}, \mathtt{z}:\mathtt{Nat})}{\mathtt{x}:\mathtt{Nat}, \mathtt{y}:\mathtt{Bool}, \mathtt{z}:\mathtt{Nat} \vdash \mathtt{x}:\mathtt{Nat}}}{\mathtt{x}:\mathtt{Nat}, \mathtt{z}:\mathtt{Nat} \vdash (\lambda \mathtt{y}:\mathtt{Bool}.\,\mathtt{x}) : \mathtt{Bool} \to \mathtt{Nat}}}{\mathtt{z}:\mathtt{Nat} \vdash (\lambda \mathtt{x}:\mathtt{Nat}.\,\lambda \mathtt{y}:\mathtt{Bool}.\,\mathtt{x}) : \mathtt{Nat} \to \mathtt{Bool} \to \mathtt{Nat}} \qquad \dfrac{(\mathtt{z}:\mathtt{Nat}) \in (\mathtt{z}:\mathtt{Nat})}{\mathtt{z}:\mathtt{Nat} \vdash \mathtt{z}:\mathtt{Nat}}}{\mathtt{z}:\mathtt{Nat} \vdash (\lambda \mathtt{x}:\mathtt{Nat}.\,\lambda \mathtt{y}:\mathtt{Bool}.\,\mathtt{x})\,\mathtt{z} : \mathtt{Bool} \to \mathtt{Nat}}$$

Type Substitutions

Equivalent to the definition in [Pierce, Sect 22.1] but not identical:

$$\begin{array}{lll} \sigma & ::= & & \textit{substitutions} \\ & & \varnothing & \textit{empty substitution} \\ & & \sigma, X \mapsto T & \textit{type variable binding} \end{array}$$

Instead of writing

$$\sigma \;=\; X_1 \mapsto T_1, X_2 \mapsto T_2, \ldots, X_n \mapsto T_n$$

we write

$$\sigma \;=\; [X_1 \mapsto T_1, X_2 \mapsto T_2, \ldots, X_n \mapsto T_n]$$

Define

$$dom(\sigma) = \{X_1, X_2, \ldots, X_n\} \quad \text{and} \quad range(\sigma) = \{T_1, T_2, \ldots, T_n\}$$

Applying Type Substitutions

By induction on the definition of types:

$$\begin{array}{ll} \sigma(X) & = \begin{cases} T & \text{if } X \in dom(\sigma) \text{ and } \sigma(X) = T \\ X & \text{if } X \notin dom(\sigma) \end{cases} \\ \sigma(\texttt{Nat}) & = \texttt{Nat} \\ \sigma(\texttt{Bool}) & = \texttt{Bool} \\ \sigma(T_1 \to T_2) & = \sigma(T_1) \to \sigma(T_2) \end{array}$$

Instead of $\sigma(T)$, write $\sigma\, T$ whenever possible.

Applying Type Substitutions

By induction on the definition of types:

$$\begin{array}{ll} \sigma(X) & = \begin{cases} T & \text{if } X \in dom(\sigma) \text{ and } \sigma(X) = T \\ X & \text{if } X \notin dom(\sigma) \end{cases} \\ \sigma(\texttt{Nat}) & = \texttt{Nat} \\ \sigma(\texttt{Bool}) & = \texttt{Bool} \\ \sigma(T_1 \to T_2) & = \sigma(T_1) \to \sigma(T_2) \end{array}$$

Instead of $\sigma(T)$, write $\sigma\, T$ whenever possible. Extend type substitutions to contexts, and similarly to terms:

$$\sigma(x_1 : T_1, x_2 : T_2, \ldots, x_n : T_n) = (x_1 : \sigma\, T_1, x_2 : \sigma\, T_2, \ldots, x_n : \sigma\, T_n)$$

Applying Type Substitutions

By induction on the definition of types:

$$\begin{array}{ll} \sigma(X) & = \begin{cases} T & \text{if } X \in dom(\sigma) \text{ and } \sigma(X) = T \\ X & \text{if } X \notin dom(\sigma) \end{cases} \\ \sigma(\texttt{Nat}) & = \texttt{Nat} \\ \sigma(\texttt{Bool}) & = \texttt{Bool} \\ \sigma(T_1 \to T_2) & = \sigma(T_1) \to \sigma(T_2) \end{array}$$

Instead of $\sigma(T)$, write $\sigma\, T$ whenever possible. Extend type substitutions to contexts, and similarly to terms:

$$\sigma(x_1 : T_1, x_2 : T_2, \ldots, x_n : T_n) = (x_1 : \sigma\, T_1, x_2 : \sigma\, T_2, \ldots, x_n : \sigma\, T_n)$$

Preservation of Typing Under Type Substitution [Pierce, Theorem 22.1.2]
If $\Gamma \vdash t : T$ is derivable and σ is a type substitution, $\sigma\Gamma \vdash \sigma t : \sigma\, T$ is derivable.

Constraint Typing Rules [Pierce, Fig 22-1, somewhat different]

$$\frac{(x : T) \in \Gamma}{\Gamma \vdash x : T \mid \{\}} \qquad \text{(CT-VAR)}$$

$$\frac{\Gamma, x : T \vdash t : T' \mid C}{\Gamma \vdash (\lambda x : T.\, t) : T \to T' \mid C} \qquad \text{(CT-ABS)}$$

$$\frac{\Gamma \vdash t_1 : T_1 \mid C_1 \qquad \Gamma \vdash t_2 : T_2 \mid C_2 \qquad X \text{ fresh}}{\Gamma \vdash t_1\, t_2 : X \mid C_1 \cup C_2 \cup \{T_1 = T_2 \to X\}} \qquad \text{(CT-APP)}$$

Compare with [Handout 07, page 3]. What is new is in red.

Exercise

[Pierce, Fig 22-1] includes additional constraint typing rules, besides the three above, namely, CT-ZERO, CT-SUCC, CT-PRED, CT-ISZERO, CT-TRUE, CT-FALSE and CT-IF. Write these rules again, but now in the style of this handout.

Generating typing constraints

Not in [Pierce]. Pseudo-code implementation in the next page is based on our adaptation of the rules in [Pierce, Fig 22-1], **restricted to CT-VAR, CT-ABS and CT-APP. Extension to other rules (CT-SUCC, CT-PRED, etc.) is left to you.**

The pseudo-code implementation defines a function `constraints` of the form:

$$\texttt{constraints} : \texttt{contexts} \times \texttt{terms} \to (\texttt{types} \times \texttt{constraint sets}) \cup \{\texttt{error}\}$$

The input of `constraints` is a pair consisting of a type context (or type environment) `G` together with a (fully type-decorated) term `t`. The output is either `error` or else a pair consisting of a type `T` together with a set of constraints `C`. If the output is `error`, it means it is impossible to find a set of constraints for a typing derivation which is legal according to the rules of page 7. If the output is $(\mathtt{T}, \mathtt{C})$, the type `T` is the type assgined to the input term `t` in the final judgement of a legal typing derivation.


```
constraints(G,t) =
  IF t is a variable x                          (* t is a variable *)
  THEN IF G(x) is defined THEN return (G(x),{}) ELSE ''error''
  ELSE
  IF t = t1 t2                                  (* t is an application *)
  THEN LET constraints(G,t1) = (T1,C1) and
           constraints(G,t2) = (T2,C2) and
           X = fresh type variable
       IN  return (X, C1 union C2 union {T1 = T2 -> X})
  ELSE
  IF t = lambda x:T. t'                         (* t is an abstraction *)
  THEN LET constraints((G - G(x),x:T),t') = (T',C)
       IN  return (T -> T',C)
  ELSE ``error''                                (* t is unrecognized *)
```

Typically, we run the algorithm to find out whether a non-type-decorated term, say $\mathtt{t}_0$, is typable. For this, we first insert a fresh type variable at every binding occurrence of a program variable in $\mathtt{t}_0$, to obtain a type-decorated term `t`. We also need a type context `G` to run the algorithm. We take this `G` to be $\mathtt{x}_1 : \mathtt{X}_1, \ldots, \mathtt{x}_n : \mathtt{X}_n$ where $\{\mathtt{x}_1, \ldots, \mathtt{x}_n\} = FV(\mathtt{t}_0)$ and $\mathtt{X}_1, \ldots, \mathtt{X}_n$ are fresh type variables.

Solving Constraints

Preliminary definitions to state Soundness and Completeness precisely.

Lemma (not in [Pierce])
Let Γ be a context and t a term such that $dom(\Gamma) \supseteq FV(t)$. Then there is a unique type T and a unique constraint set C such that $\Gamma \vdash t : T \mid C$ is derivable according to the constraint typing rules (page 7).

Solving Constraints

Preliminary definitions to state Soundness and Completeness precisely.

Lemma (not in [Pierce])
Let Γ be a context and t a term such that $dom(\Gamma) \supseteq FV(t)$. Then there is a unique type T and a unique constraint set C such that $\Gamma \vdash t : T \mid C$ is derivable according to the constraint typing rules (page 7).

Proof of the lemma is based on the implementation of the function $\texttt{constraints}(\Gamma, t)$ on page 8. The lemma justifies the following definition:

$$\texttt{constraints}(\Gamma, t) = \begin{cases} (T, C) & \text{if } dom(\Gamma) \supseteq FV(t)\text{, in which case} \\ & T \text{ is unique type and } C \text{ is unique constraint set} \\ & \text{such that } \Gamma \vdash t : T \mid C, \\ \text{error} & \text{if } dom(\Gamma) \not\supseteq FV(t). \end{cases}$$

Solving Constraints

Definition [Pierce, Definition 22.2.1]
Let Γ be a context and t a term such that $dom(\Gamma) \supseteq FV(t)$. A substitution σ is a solution for (Γ, t) if there is a type T such that $\sigma\Gamma \vdash \sigma t : T$ is derivable.

See [Pierce, Example 22.2.2].

Soundness and Completeness of Constraint Typing [Pierce, Section 22.3]

Assume throughout Γ is a context and t a term such that $dom(\Gamma) \supseteq FV(t)$.

Soundness [Pierce, Theorem 22.3.5]
If substitution σ is a solution of $\texttt{constraints}(\Gamma, t)$ then σ is a solution of (Γ, t).

Soundness and Completeness of Constraint Typing [Pierce, Section 22.3]

Assume throughout Γ is a context and t a term such that $dom(\Gamma) \supseteq FV(t)$.

Soundness [Pierce, Theorem 22.3.5]
If substitution σ is a solution of $\texttt{constraints}(\Gamma, t)$ then σ is a solution of (Γ, t).

Completeness (not stated precisely enough!) [Pierce, Theorem 22.3.7]
If substitution σ is a solution of (Γ, t) then σ is a solution of $\texttt{constraints}(\Gamma, t)$.

Soundness and Completeness of Constraint Typing [Pierce, Section 22.3]

Assume throughout Γ is a context and t a term such that $dom(\Gamma) \supseteq FV(t)$.

Soundness [Pierce, Theorem 22.3.5]
If substitution σ is a solution of $\texttt{constraints}(\Gamma, t)$ then σ is a solution of (Γ, t).

Completeness (not stated precisely enough!) [Pierce, Theorem 22.3.7]
If substitution σ is a solution of (Γ, t) then σ is a solution of $\texttt{constraints}(\Gamma, t)$.

Completeness (stated precisely and correctly!) [Pierce, Theorem 22.3.7]
If substitution σ is a solution of (Γ, t) $\boxed{\text{and } dom(\sigma) \subseteq FTV(\Gamma) \cup FTV(t)}$ then there is a substitution σ' which is a solution of $\texttt{constraints}(\Gamma, t)$ $\boxed{\text{with } \sigma' \supseteq \sigma}$.

$FTV(\Gamma)$ means "free type variables in Γ".

How to find solutions for constraint sets [Pierce, Section 22.4]

$\texttt{unify} : \text{constraint sets} \rightarrow (\text{substitutions} \cup \{\text{fail}\})$

```
unify(C) =
  IF C = ∅ THEN  [ ]  ELSE
  LET C = ({T = U} ∪ C')
  IN   IF T = U
       THEN  unify(C')  ELSE
       IF T = X AND X ∉ FTV(U)
       THEN  unify([X ↦ U]C') ∘ [X ↦ U]  ELSE
       IF U = X AND X ∉ FTV(T)
       THEN  unify([X ↦ T]C') ∘ [X ↦ T]  ELSE
       IF T = T₁ → T₂ AND U = U₁ → U₂
       THEN  unify(C' ∪ {T₁ = U₁, T₂ = U₂})  ELSE
       "fail"
```


Principal unifiers [Pierce, Section 22.4]

Definition [Pierce, Definition 22.4.1]
A substitution σ is more general than a substitution σ', written $\sigma \sqsubseteq \sigma'$, if there is a substitution σ'' such that $\sigma' = \sigma'' \circ \sigma$.

Principal unifiers [Pierce, Section 22.4]

Definition [Pierce, Definition 22.4.1]
A substitution σ is more general than a substitution σ', written $\sigma \sqsubseteq \sigma'$, if there is a substitution σ'' such that $\sigma' = \sigma'' \circ \sigma$.

Definition [Pierce, Definition 22.4.2]
A principal unifier for a constraint set C is a substitution σ such that:

1. σ is a solution of C,
2. for every solution σ' of C, it holds that $\sigma \sqsubseteq \sigma'$.

Principal unifiers [Pierce, Section 22.4]

Definition [Pierce, Definition 22.4.1]
A substitution σ is more general than a substitution σ', written $\sigma \sqsubseteq \sigma'$, if there is a substitution σ'' such that $\sigma' = \sigma'' \circ \sigma$.

Definition [Pierce, Definition 22.4.2]
A principal unifier for a constraint set C is a substitution σ such that:

1. σ is a solution of C,
2. for every solution σ' of C, it holds that $\sigma \sqsubseteq \sigma'$.

Theorem [Pierce, Theorem 22.4.5]
The algorithm `unify` always terminates, failing when given a non-unifiable constraint set C as input and otherwise returning a principal unifier.

Principal types [Pierce, Section 22.5]

Assume throughout Γ is a context and t a term such that $dom(\Gamma) \supseteq FV(t)$.

Definition [Pierce, Definition 22.5.1]
• Let T and C be the unique type and constraint such that $\Gamma \vdash t : T \mid C$ (by Lemma on page 9).
A substitution σ is a principal solution for (Γ, t) if σ is a principal unifier of C, in which case we call σT a principal type of t relative to Γ.

Principal types [Pierce, Section 22.5]

Assume throughout Γ is a context and t a term such that $dom(\Gamma) \supseteq FV(t)$.

Definition [Pierce, Definition 22.5.1]
• Let T and C be the unique type and constraint such that $\Gamma \vdash t : T \mid C$ (by Lemma on page 9).
A substitution σ is a principal solution for (Γ, t) if σ is a principal unifier of C, in which case we call σT a principal type of t relative to Γ.

Theorem [Pierce, Theorem 22.5.3]
If (Γ, t) has any solution, then it has a principal one.