Type Reconstruction

Klaus Ostermann
Uni Marburg

Based on lecture notes by Andrew Myers

Type Reconstruction

- A.k.a. type inference
- Advantage of type annotations:
  - Type checker is simple
- Disadvantage:
  - Writing type annotations all the time can be tiresome, since many annotations are obvious

Places where we required type annotations

$$\frac{\Gamma, x{:}\tau \vdash e : \tau'}{\Gamma \vdash \lambda x{:}\tau.\, e : \tau \to \tau'}$$

$$\frac{\Gamma, y{:}\tau \to \tau', x{:}\tau \vdash e : \tau'}{\Gamma \vdash \mathbf{rec}\ y{:}\tau \to \tau'.\, \lambda x.\, e : \tau \to \tau'} \qquad \frac{\Gamma \vdash e : \tau \quad \Gamma, x{:}\tau \vdash e' : \tau'}{\Gamma \vdash \mathbf{let}\ x{:}\tau = e\ \mathbf{in}\ e' : \tau'}$$

Typing rules are easy to specify w/o type annotations

$$\frac{\Gamma, x{:}\tau \vdash e : \tau'}{\Gamma \vdash \lambda x.\, e : \tau \to \tau'} \qquad \frac{\Gamma, y{:}\tau \to \tau', x{:}\tau \vdash e : \tau'}{\Gamma \vdash \mathbf{rec}\ y.\ \lambda x.\, e : \tau \to \tau'}$$

$$\frac{\Gamma \vdash e : \tau \quad \Gamma, x{:}\tau \vdash e' : \tau'}{\Gamma \vdash \mathbf{let}\ x = e\ \mathbf{in}\ e' : \tau'}$$

- But how can they be implemented?
  - Problem: Need to guess τ

Infering types by hand…

```
let d = λz. z+z in
  (λf.λx.λy.
     if (f x y) then
       f (d x) y
     else
       f x (f x y)
```

Since + is an operation on ints, z: Int

Hence d: Int -> Int

Hence x : Int

Hence f : Int -> ??? -> ???

Since (f x y) is used as condition
f: Int -> ??? -> Bool

Hence (see else branch)
f: Int -> Bool -> Bool and y : Bool

Hence the type of the expression is
(Int -> Bool -> Bool) -> Int -> Bool -> Bool

Type equality constraints

$$\dfrac{\dfrac{\dfrac{f{:}T2,\ x{:}T5 \vdash f : \text{int} \to T6 \qquad f{:}T2,\ x{:}T5 \vdash 1 : \text{int}}{f{:}T2,\ x{:}T5 \vdash f\ 1 : T6}}{f{:}T2 \vdash \lambda x.\ f\ 1 : T1 \quad (=T5 \to T6)} \qquad \dfrac{}{y{:}T3 \vdash y : T4}\ (T3=T4)}{\vdash \lambda f.\ \lambda x.\ f\ 1 : T2 \to T1 \qquad \vdash (\lambda y.y) : T2 \quad (T2 = T3 \to T4)}$$

$$\vdash (\lambda f.\lambda x.\ (f\ 1))\ (\lambda y.y) : T1$$

$$T2 = T3 \to T4,\ T3 = T4,\ T1 = T5 \to T6,\ T2 = \text{int} \to T6$$

Type equality constraints

$$\frac{}{\Gamma, x{:}\tau \vdash x : \tau}$$

$$\frac{}{\Gamma \vdash b : B}$$

$$\frac{\Gamma \vdash e_0 : \tau_0 \quad \Gamma \vdash e_1 : \tau_1 \quad \tau_0 = \tau_1 \to T}{\Gamma \vdash e_0\ e_1 : T}$$

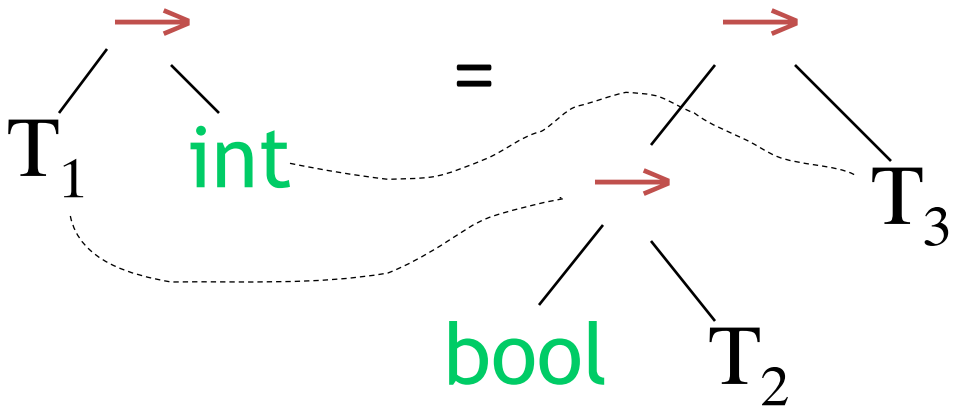
$$\frac{\Gamma, x{:}T \vdash e : \tau'}{\Gamma \vdash \lambda x.\, e : T \to \tau'}$$

$$\frac{\Gamma \vdash e_1 : \tau_1 \quad \Gamma, x{:}\tau_1 \vdash e_2 : \tau_2}{\Gamma \vdash \mathbf{let}\ x = e_1\ \mathbf{in}\ e_2}$$

$$\frac{\Gamma, x{:}T_1, y{:}T_1 \to T_2 \vdash e : \tau' \quad \tau' = T_2}{\Gamma \vdash \mathbf{rec}\ y.\ \lambda x.\, e : T_1 \to T_2}$$

Unification

- How to solve equations?
- Idea: given equation $\tau_1 = \tau_2$, *unify* type expressions to solve for variables in both
- Example: $T_1 \rightarrow int = (bool \rightarrow T_2) \rightarrow T_3$
- Result: *substitution* $T_1 := bool \rightarrow T_2$, T_3 int



Substitution and Unifier

- A substitution S maps type variables T to types τ
- We write S(τ) for the result of applying all substitutions in S on τ
- A substitution S is a unifier of τ and τ‘, if S(τ) = S(τ‘)
- To preserve maximal polymorphism we want the *weakest* unifier
- S_1 is weaker than S_2, if there is a non-trivial substitution S_3 such that $S_2 = S_3 \,.\, S_1$ whereby $S_3 \,.\, S_1\,(\tau) = S_3\,(S_1\,(\tau))$

Robinson's Algorithm

Often called "Occurs Check"

$$
\begin{aligned}
unify(\emptyset) &= \emptyset \\
unify(\{B = B\} \cup E) &= unify(E) \\
unify(\{B_1 = B_2\} \cup E) &= error \quad (\text{if } B_1 \neq B_2) \\
unify(\{T = T\} \cup E) &= unify(E) \\
unify(\{T = \tau\} \cup E) &= unify(\{\tau = T\} \cup E) = unify(E\{\tau/T\}) \circ \{T \mapsto \tau\} \quad (\text{if } T \text{ is not mentioned in } \tau) \\
unify(\{\tau_1 \to \tau_2 = \tau_1' \to \tau_2'\} \cup E) &= unify(\{\tau_1 = \tau_1', \tau_2 = \tau_2'\} \cup E) \\
unify(\{\tau = \tau'\} \cup E) &= error \quad (\text{if no previous rules match})
\end{aligned}
$$

This definition is well-founded, but this is not obvious.
Either the number of variables in the equations becomes smaller, or it stays equal.
In the latter case, the total size of the equations or the number of arrows becomes smaller.

Complexity of Unification

$$
\begin{array}{l}
\textbf{let b} = \textbf{true in} \\
\textbf{let } f_0 = \lambda \textbf{x. x+1 in} \\
\textbf{let } f_1 = \lambda \textbf{x. if b then } f_0 \textbf{ else } \lambda \textbf{y.x y in} \\
\textbf{let } f_2 = \lambda \textbf{x. if b then } f_1 \textbf{ else } \lambda \textbf{y.x y in} \\
\vdots \\
\textbf{let } f_n = \lambda \textbf{x. if b then } f_{n-1} \textbf{ else } \lambda \textbf{y.x y in} \\
\quad \textbf{0}
\end{array}
$$

If $f_n : \tau_n$, then $\tau_0 = \text{Int} \rightarrow \text{Int}$ and $\tau_{n+1} = \tau_n \rightarrow \tau_n$

Hence type inference can take exponential time

With a better representations of types (DAGs instead of trees) the complexity can be improved to approximately $O(n^2)$

Curry-Howard Isomorphism

- There is a deep connection between type systems and (intuitionstic/constructive) logic
- A proof of a proposition in constructive logic is a construction of an object that witnesses the proposition
- The Curry-Howard isomorphism says that proofs are the same as terms/programs

Constructive vs classical proofs

- Not every proof in classical logic is also valid in intuitionistic logic

Theorem There exist irrational numbers a and b such that a^b is rational.

Proof. Either $\sqrt{2}^{\sqrt{2}}$ is rational or not. If it is, take $a = b = \sqrt{2}$ and we are done. If it is not, take $a = \sqrt{2}^{\sqrt{2}}$ and $b = \sqrt{2}$; then $a^b = (\sqrt{2}^{\sqrt{2}})^{\sqrt{2}} = \sqrt{2}^2 = 2$, and again we are done. □

- Law of excluded middle is not valid in intuitionistic logic: It is not constructive!

Intuitionistic logic

- Syntax of formulas:

$$\phi \quad ::= \quad \top \mid \bot \mid P \mid \phi_1 \Rightarrow \phi_2 \mid \phi_1 \vee \phi_2 \mid \phi_1 \wedge \phi_2 \mid \neg\phi.$$

- With second-order quantification:

$$\phi \quad ::= \quad \cdots \mid \forall P.\phi.$$

Natural Deduction

- Calculus developed by Gentzen to define proof rules of a logic
- Operators (so-called connectives) typically have introduction and elimination rules
- We will see that the deduction rules in natural deduction style correspond exactly to the typing rules of System F with sums and products
  - Terms are a linear notation of proofs!

Proof- and Typing Rules Side-by-Side

	intuitionistic logic	$\lambda^{\rightarrow}$ *or System F type system*
(axiom)	$\Gamma, \phi \vdash \phi$	$\Gamma, x : \tau \vdash x : \tau$
($\rightarrow$-intro)	$\dfrac{\Gamma, \phi \vdash \psi}{\Gamma \vdash \phi \Rightarrow \psi}$	$\dfrac{\Gamma, x : \sigma \vdash e : \tau}{\Gamma \vdash (\lambda x : \sigma.\, e) : \sigma \rightarrow \tau}$
($\rightarrow$-elim)	$\dfrac{\Gamma \vdash \phi_1 \Rightarrow \phi_2 \quad \Gamma \vdash \phi_1}{\Gamma \vdash \phi_2}$	$\dfrac{\Gamma \vdash e_0 : \sigma \rightarrow \tau \quad \Gamma \vdash e_1 : \sigma}{\Gamma \vdash (e_0\ e_1) : \tau}$
($\wedge$-intro)	$\dfrac{\Gamma \vdash \phi \quad \Gamma \vdash \psi}{\Gamma \vdash \phi \wedge \psi}$	$\dfrac{\Gamma \vdash e_1 : \sigma \quad \Gamma \vdash e_2 : \tau}{\Gamma \vdash (e_1, e_2) : \sigma * \tau}$
($\wedge$-elim)	$\dfrac{\Gamma \vdash \phi \wedge \psi}{\Gamma \vdash \phi} \quad \dfrac{\Gamma \vdash \phi \wedge \psi}{\Gamma \vdash \psi}$	$\dfrac{\Gamma \vdash e : \sigma * \tau}{\Gamma \vdash \#1\, e : \sigma} \quad \dfrac{\Gamma \vdash e : \sigma * \tau}{\Gamma \vdash \#2\, e : \tau}$

Proof- and Typing Rules Side-by-Side

	intuitionistic logic	$\lambda^{\rightarrow}$ *or System F type system*
($\vee$-intro)	$\dfrac{\Gamma \vdash \phi}{\Gamma \vdash \phi \vee \psi} \quad \dfrac{\Gamma \vdash \psi}{\Gamma \vdash \phi \vee \psi}$	$\dfrac{\Gamma \vdash e : \sigma}{\Gamma \vdash \mathbf{inl}_{\sigma+\tau}\, e : \sigma + \tau} \quad \dfrac{\Gamma \vdash e : \tau}{\Gamma \vdash \mathbf{inr}_{\sigma+\tau}\, e : \sigma + \tau}$
($\vee$-elim)	$\dfrac{\Gamma \vdash \phi \vee \psi \quad \Gamma \vdash \phi \rightarrow \chi \quad \Gamma \vdash \psi \rightarrow \chi}{\Gamma \vdash \chi}$	$\dfrac{\Gamma \vdash e : \sigma + \tau \quad \Gamma \vdash e_1 : \sigma \rightarrow \rho \quad \Gamma \vdash e_2 : \tau \rightarrow \rho}{\Gamma \vdash \mathbf{case}\ e_0\ \mathbf{of}\ e_1 \mid e_2 : \rho}$
($\forall$-intro)	$\dfrac{\Gamma, P \vdash \phi}{\Gamma \vdash \forall P\,.\,\phi}$	$\dfrac{\Delta, \alpha; \Gamma \vdash e : \tau \quad \alpha \notin FV(\Gamma)}{\Delta; \Gamma \vdash (\Lambda\alpha.\, e) : \forall\alpha\,.\,\tau}$
($\forall$-elim)	$\dfrac{\Gamma \vdash \forall P\,.\,\phi}{\Gamma \vdash \phi\{\psi/P\}}$	$\dfrac{\Delta; \Gamma \vdash e : \forall\alpha\,.\,\tau \quad \Delta \vdash \sigma}{\Delta; \Gamma \vdash (e\ \sigma) : \tau\{\sigma/\alpha\}}$

The Curry-Howard Isomorphism

- A.k.a as “Propositions as Types”

type theory		*logic*	
τ	type	ϕ	proposition
τ	inhabited type	ϕ	theorem
e	well-typed program	π	proof
$\rightarrow$	function space	$\rightarrow$	implication
$*$	product	$\wedge$	conjunction
$+$	sum	$\vee$	disjunction
$\forall$	type quantifier	$\forall$	2nd order quantifier
B	inhabited type	$\top$	truth
void	uninhabited type	$\bot$	falsity

Logical Interpretation of Program Transformations

- Reduction = Proof Normalization
  - Existence of normal form can be formalized as *Cut Elimination Theorem* (Gentzen's "Hauptsatz")
  - Typically presented using sequent calculus rather than natural deduction
- CPS Transformation = Double Negation