Proving with Computer Assistance

Herman Geuvers

Lecture 3: Simple Type Theory a la Curry: assigning types to untyped terms, principal type algorithm
Overview of today's lecture

- Recap of Simple Type Theory a la Church
- Recap of Untyped lambda calculus
- Simple Type Theory a la Curry (versus a la Church)
- Principal Types algorithm
- Properties of Simple Type Theory.
Recap: Simple type theory a la Church.

Formulation with contexts to declare the free variables:

\[ x_1 : \sigma_1, x_2 : \sigma_2, \ldots, x_n : \sigma_n \]

is a context, usually denoted by \( \Gamma \).

Derivation rules of \( \lambda \) (à la Church):

\[
\begin{align*}
\frac{x : \sigma \in \Gamma}{\Gamma \vdash x : \sigma} & \quad \frac{\Gamma \vdash M : \sigma \rightarrow \tau \quad \Gamma \vdash N : \sigma}{\Gamma \vdash MN : \tau} & \quad \frac{\Gamma, x : \sigma \vdash P : \tau}{\Gamma \vdash \lambda x : \sigma. P : \sigma \rightarrow \tau}
\end{align*}
\]

\( \Gamma \vdash \lambda \rightarrow M : \sigma \) if there is a derivation using these rules with conclusion \( \Gamma \vdash M : \sigma \)
Recap: Formulas-as-Types (Curry, Howard)

There are two readings of a judgement $M : \sigma$

1. term as algorithm/program, type as specification:
   $M$ is a function of type $\sigma$

2. type as a proposition, term as its proof:
   $M$ is a proof of the proposition $\sigma$

• There is a one-to-one correspondence:

   typable terms in $\lambda \rightarrow \simeq$ derivations in minimal proposition logic

• $x_1 : \tau_1, x_2 : \tau_2, \ldots, x_n : \tau_n \vdash M : \sigma$ can be read as
  $M$ is a proof of $\sigma$ from the assumptions $\tau_1, \tau_2, \ldots, \tau_n$. 
Recap: Example

\[
\begin{array}{c}
[\alpha \to \beta \to \gamma]^3 [\alpha]^1 \\
\beta \to \gamma
\end{array} \quad \begin{array}{c}
[\alpha \to \beta]^2 [\alpha]^1 \\
\beta
\end{array}
\]

\[
\begin{array}{c}
\gamma \quad 1 \\
\alpha \to \gamma
\end{array} \quad \begin{array}{c}
2 \\
(\alpha \to \beta) \to \alpha \to \gamma
\end{array} \quad \begin{array}{c}
3 \\
(\alpha \to \beta \to \gamma) \to (\alpha \to \beta) \to \alpha \to \gamma
\end{array}
\]

\[
\lambda x : \alpha \to \beta \to \gamma. \lambda y : \alpha \to \beta. \lambda z : \alpha. x z (y z)
\]

\[
: (\alpha \to \beta \to \gamma) \to (\alpha \to \beta) \to \alpha \to \gamma
\]
Untyped $\lambda$-calculus

Untyped $\lambda$-calculus

\[ \Lambda ::= \text{Var} \mid (\Lambda \Lambda) \mid (\lambda \text{Var.}\Lambda) \]

Examples:
- $K := \lambda x \; y. \; x$
- $S := \lambda x \; y \; z. \; x \; z(y \; z)$
- $\omega := \lambda x. \; x \; x$
- $\Omega := \omega \; \omega$

\[ \Omega \rightarrow^\beta \Omega \]
Untyped $\lambda$-calculus

Untyped $\lambda$-calculus is Turing complete. Its power lies in the fact that you can solve recursive equations:

Is there a term $M$ such that

$$M \ x =_\beta x \ M \ x?$$

Is there a term $M$ such that

$$M \ x =_\beta \text{if} \ (\text{Zero} \ x) \ \text{then} \ 1 \ \text{else} \ \text{Mult} \ x \ (M \ (\text{Pred} \ x))?$$

Yes, because we have a fixed point combinator:

- $Y := \lambda f. (\lambda x. f (x \ x)) (\lambda x. f (x \ x))$

Property:

$$Y \ f =_\beta f (Y \ f)$$
Why do we want types?

- Types give a (partial) specification
- Typed terms can’t go wrong (Milner) Subject Reduction property
- Typed terms always terminate
- The type checking algorithm detects (simple) mistakes

But: The compiler should compute the type information for us! (Why would the programmer have to type all that?)

This is called a type assignment system, or also typing à la Curry:

For $M$ an untyped term, the type system assigns a type $\sigma$ to $M$ (or not)
STT à la Church and à la Curry

\( \lambda \rightarrow \) (à la Church):

\[
\begin{array}{c}
  x : \sigma \in \Gamma \\
  \Gamma \vdash M : \sigma \rightarrow \tau \\
  \Gamma \vdash N : \sigma \\
  \Gamma, x : \sigma \vdash P : \tau \\
  \Gamma \vdash x : \sigma \\
  \Gamma \vdash x \in \Gamma \\
  \Gamma \vdash MN : \tau \\
  \Gamma \vdash \lambda x : \sigma . P : \sigma \rightarrow \tau
\end{array}
\]

\( \lambda \rightarrow \) (à la Curry):

\[
\begin{array}{c}
  x : \sigma \in \Gamma \\
  \Gamma \vdash M : \sigma \rightarrow \tau \\
  \Gamma \vdash N : \sigma \\
  \Gamma, x : \sigma \vdash P : \tau \\
  \Gamma \vdash x : \sigma \\
  \Gamma \vdash x \in \Gamma \\
  \Gamma \vdash MN : \tau \\
  \Gamma \vdash \lambda x . P : \sigma \rightarrow \tau
\end{array}
\]
Examples

• Typed Terms:

\[ \lambda x : \alpha . \lambda y : (\beta \to \alpha) \to \alpha . y (\lambda z : \beta . x) \, \]

has only the type \( \alpha \to ((\beta \to \alpha) \to \alpha) \to \alpha \)

• Type Assignment:

\[ \lambda x . \lambda y . y (\lambda z . x) \, \]

can be assigned the types

- \( \alpha \to ((\beta \to \alpha) \to \alpha) \to \alpha \)

- \( (\alpha \to \alpha) \to ((\beta \to \alpha \to \alpha) \to \gamma) \to \gamma \)

- \( \ldots \)

with \( \alpha \to ((\beta \to \alpha) \to \gamma) \to \gamma \) being the principal type
Connection between Church and Curry typed STT

**Definition** The erasure map $| - |$ from STT à la Church to STT à la Curry is defined by erasing all type information.

$$
|x| := x \\
|M N| := |M| |N| \\
|\lambda x : \sigma. M| := \lambda x.|M|
$$

So, e.g.

$$|\lambda x : \alpha. \lambda y : (\beta \rightarrow \alpha) \rightarrow \alpha. y(\lambda z : \beta. x)| = \lambda x. \lambda y. y(\lambda z.x)$$

**Theorem** If $M : \sigma$ in STT à la Church, then $|M| : \sigma$ in STT à la Curry.

**Theorem** If $P : \sigma$ in STT à la Curry, then there is an $M$ such that $|M| \equiv P$ and $M : \sigma$ in STT à la Church.
Connection between Church and Curry typed STT

Definition The erasure map $| - |$ from STT à la Church to STT à la Curry is defined by erasing all type information.

$$
| x | := x \\
|M N| := | M | | N | \\
| \lambda x : \sigma. M | := \lambda x . | M |
$$

Theorem If $P : \sigma$ in STT à la Curry, then there is an $M$ such that $| M | \equiv P$ and $M : \sigma$ in STT à la Church.

Proof: by induction on derivations.

$$
\frac{
 x : \sigma \in \Gamma \\
\Gamma \vdash M : \sigma \rightarrow \tau \\
\Gamma \vdash N : \sigma
}{\Gamma \vdash x : \sigma}
\quad
\frac{
\Gamma \vdash M : \sigma \rightarrow \tau \\
\Gamma \vdash N : \sigma
}{\Gamma \vdash MN : \tau}
\quad
\frac{
\Gamma, x : \sigma \vdash P : \tau
}{\Gamma \vdash \lambda x : \sigma. P : \sigma \rightarrow \tau}
$$
Example of computing a principal type

$$\lambda x^\alpha. \lambda y^\beta. \underbrace{y^\beta (\lambda z^\gamma. y^\beta x^\alpha)}_\varepsilon$$

1. Assign type vars to all variables: \( x : \alpha, y : \beta, z : \gamma. \)

2. Assign type vars to all applicative subterms: \( y \ x : \delta, y(\lambda z.y \ x) : \varepsilon. \)

3. Generate equations between types, necessary for the term to be typable:
   \[
   \beta = \alpha \rightarrow \delta \quad \beta = (\gamma \rightarrow \delta) \rightarrow \varepsilon
   \]

4. Find a most general unifier (a substitution) for the type vars that solves the equations:
   \( \alpha := \gamma \rightarrow \delta, \ \beta := (\gamma \rightarrow \delta) \rightarrow \varepsilon, \ \delta := \varepsilon \)

5. The principal type of \( \lambda x. \lambda y.y(\lambda z.yx) \) is now
   \[
   (\gamma \rightarrow \varepsilon) \rightarrow ((\gamma \rightarrow \varepsilon) \rightarrow \varepsilon) \rightarrow \varepsilon
   \]
Exercise: Compute principal types for $S := \lambda x.\lambda y.\lambda z. x \, z \, y \, z$ and for $M := \lambda x.\lambda y. x \,(y \,(\lambda z. x \, z \, z) \, y \,(\lambda z. x \, z \, z))$. 
Principal Types: Definitions

- A type substitution (or just substitution) is a map $S$ from type variables to types. (Note: we can compose substitutions.)

- A unifier of the types $\sigma$ and $\tau$ is a substitution that “makes $\sigma$ and $\tau$ equal”, i.e. an $S$ such that $S(\sigma) = S(\tau)$

- A most general unifier (or mgu) of the types $\sigma$ and $\tau$ is the “simplest substitution” that makes $\sigma$ and $\tau$ equal, i.e. an $S$ such that
  - $S(\sigma) = S(\tau)$
  - for all substitutions $T$ such that $T(\sigma) = T(\tau)$ there is a substitution $R$ such that $T = R \circ S$.

All these notions generalize to lists of types $\sigma_1, \ldots, \sigma_n$ in stead of pairs $\sigma, \tau$. 
Computability of most general unifiers

There is an algorithm $U$ that, when given types $\sigma_1, \ldots, \sigma_n$ outputs

- A most general unifier of $\sigma_1, \ldots, \sigma_n$, if $\sigma_1, \ldots, \sigma_n$ can be unified.
- "Fail" if $\sigma_1, \ldots, \sigma_n$ can't be unified.

- $U(\langle \alpha = \alpha, \ldots, \sigma_n = \tau_n \rangle) := U(\langle \sigma_2 = \tau_2, \ldots, \sigma_n = \tau_n \rangle)$.
- $U(\langle \alpha = \tau_1, \ldots, \sigma_n = \tau_n \rangle) := \text{"reject" if } \alpha \in \text{FV}(\tau_1), \tau_1 \neq \alpha.$
- $U(\langle \sigma_1 = \alpha, \ldots, \sigma_n = \tau_n \rangle) := U(\langle \alpha = \sigma_1, \ldots, \sigma_n = \tau_n \rangle)$
- $U(\langle \alpha = \tau_1, \ldots, \sigma_n = \tau_n \rangle) := [\alpha := V(\tau_1), V], \text{ if } \alpha \notin \text{FV}(\tau_1)$, where $V$ abbreviates $U(\langle \sigma_2[\alpha := \tau_1] = \tau_2[\alpha := \tau_1], \ldots, \sigma_n[\alpha := \tau_1] = \tau_n[\alpha := \tau_1] \rangle)$.
- $U(\langle \mu \rightarrow \nu = \rho \rightarrow \xi, \ldots, \sigma_n = \tau_n \rangle) := U(\langle \mu = \rho, \nu = \xi, \ldots, \sigma_n = \tau_n \rangle)$
Principal type

Definition $\sigma$ is a principal type for the untyped $\lambda$-term $M$ if

- $M : \sigma$ in STT à la Curry
- for all types $\tau$, if $M : \tau$, then $\tau = S(\sigma)$ for some substitution $S$. 
Theorem: Principal Types

There is an algorithm PT that, when given an (untyped) $\lambda$-term $M$, outputs

- A principal type $\sigma$ such that $M : \sigma$ in STT à la Curry.

- "Fail" if $M$ is not typable in STT à la Curry.
Typical problems one would like to have an algorithm for

\[ M : \sigma? \quad \text{Type Checking Problem} \quad \text{TCP} \]
\[ M : ? \quad \text{Type Synthesis Problem} \quad \text{TSP} \]
\[ ? : \sigma \quad \text{Type Inhabitation Problem (by a closed term)} \quad \text{TIP} \]

For \( \lambda \rightarrow \), all these problems are decidable,
both for the Curry style and for the Church style presentation.

- TCP and TSP are (usually) equivalent: To solve \( MN : \sigma \), one has to solve \( N : ? \) (and if this gives answer \( \tau \), solve \( M : \tau \rightarrow \sigma \)).
- For Curry systems, TCP and TSP soon become undecidable beyond \( \lambda \rightarrow \).
• TIP is undecidable for most extensions of $\lambda \rightarrow$, as it corresponds to provability in some logic.