Type Theory and Coq

Herman Geuvers

Principal Types and Type Checking
Overview of today's lecture

• Simple Type Theory à la Curry  
  (versus Simple Type Theory à la Church)

• Principal Types algorithm

• Type checking dependent type theory: \( \lambda P \)
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 \( \lambda \rightarrow \) (à la Church):

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

\( \Gamma \vdash_{\lambda \rightarrow} M : \sigma \) 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

\[ \alpha \to \beta \to \gamma \]

\[ \beta \to \gamma \]

\[ \gamma \]

\[ \alpha \to \gamma \]

\[ (\alpha \to \beta) \to \alpha \to \gamma \]

\[ (\alpha \to \beta \to \gamma) \to (\alpha \to \beta) \to \alpha \to \gamma \]

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

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

Untyped λ-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. It’s 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 to add types to λ-calculus?

- 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):

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

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

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

• **Typed Terms:**

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

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

• **Type Assignment:**

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

can be assigned the types

- \( \alpha \rightarrow ((\beta \rightarrow \alpha) \rightarrow \alpha) \rightarrow \alpha \)
- \( (\alpha \rightarrow \alpha) \rightarrow ((\beta \rightarrow \alpha \rightarrow \alpha) \rightarrow \gamma) \rightarrow \gamma \)
- \( \ldots \)

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

**Definition** The erasure map \( \mathcal{E} \) from STT à la Church to STT à la Curry is defined by erasing all type information.

\[
\begin{align*}
| x | & := x \\
| M N | & := | M | | N | \\
| \lambda x : \sigma. M | & := \lambda x. | M |
\end{align*}
\]

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.

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

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

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:
   \( \alpha := \gamma \rightarrow \varepsilon, \beta := (\gamma \rightarrow \varepsilon) \rightarrow \varepsilon, \delta := \varepsilon \)
5. The principal type of \( \lambda x.\lambda y.y(\lambda z.\ y x) \) is now
   \[ (\gamma \rightarrow \varepsilon) \rightarrow ((\gamma \rightarrow \varepsilon) \rightarrow \varepsilon) \rightarrow \varepsilon \]
Exercise

Compute principal types for

- \( S := \lambda x.\lambda y.\lambda z. x \ z \ (y \ z) \)

- \( M := \lambda x.\lambda y. x \ (y (\lambda z. x \ z \ z)) (y (\lambda z. x \ z \ z)). \)
• 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
  
  $- S(\sigma) = S(\tau)$

  $- \text{for all substitutions } T \text{ such that } T(\sigma) = T(\tau) \text{ there is a substitution } R \text{ such that } T = R \circ S.$

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

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

Definition $\sigma$ is a principal type for the closed 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 $S$.

A principal type is unique up to renaming of type variables.

Both $\alpha \rightarrow \alpha$ and $\beta \rightarrow \beta$ are principal type of $\lambda x.x$. 
Principal Types Theorem

**Theorem** There is an algorithm PT that, when given a closed untyped $\lambda$-term $M$, outputs

A principal type $\sigma$ of $M$  if $M$ is typable in STT à la Curry,

“Fail”                      if $M$ is not typable in STT à la Curry.

This can be extended to open untyped $\lambda$-terms: There is an algorithm PP that, when given an untyped $\lambda$-term $M$, outputs

A principal pair $(\Gamma, \sigma)$ of $M$  if $M$ is typable in STT à la Curry,

“Fail”                      if $M$ is not typable in STT à la Curry.

**Definition** $(\Gamma, \sigma)$ is a principal pair for $M$ if $\Gamma \vdash M : \sigma$ and for every typing $\Delta \vdash M : \tau$ there is a substitution $S$ such that $\tau = S(\sigma)$ and $\Delta = S(\Gamma)$. 

Typical problems one would like to have an algorithm for:

- $M : \sigma$? Type Checking Problem \hspace{2cm} TCP
- $M : ?$ Type Synthesis Problem \hspace{2cm} TSP
- $? : \sigma$ Type Inhabitation Problem (by a closed term) \hspace{2cm} 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.
Rules for $\lambda P$: axiom, application, abstraction, product

\[
\Gamma \vdash \star : \square
\]

\[
\begin{array}{c}
\Gamma \vdash M : \Pi x : A. B \\
\Gamma \vdash N : A
\end{array}
\quad
\begin{array}{c}
\Gamma \vdash MN : B[x := N]
\end{array}
\]

\[
\begin{array}{c}
\Gamma, x : A \vdash M : B \\
\Gamma \vdash \Pi x : A. B : s
\end{array}
\quad
\begin{array}{c}
\Gamma \vdash \lambda x : A. M : \Pi x : A. B
\end{array}
\]

\[
\begin{array}{c}
\Gamma \vdash A : \star \\
\Gamma, x : A \vdash B : s
\end{array}
\quad
\begin{array}{c}
\Gamma \vdash \Pi x : A. B : s
\end{array}
\]
Rules for $\lambda P$: weakening, variable, conversion

\[
\frac{\Gamma \vdash A : B \quad \Gamma \vdash C : s}{\Gamma \vdash C : s} \\
\quad \frac{\Gamma \vdash A : B}{\Gamma, x : C \vdash A : B}
\]

\[
\frac{\Gamma \vdash A : s}{\Gamma, x : A \vdash x : A}
\]

\[
\frac{\Gamma \vdash A : B \quad \Gamma \vdash B' : s}{\Gamma \vdash A : B'} \quad \text{with } B =_\beta B'
\]
Properties of $\lambda P$

- **Uniqueness of types**
  
  If $\Gamma \vdash M : \sigma$ and $\Gamma \vdash M : \tau$, then $\sigma \equiv_\beta \tau$.

- **Subject Reduction**

  If $\Gamma \vdash M : \sigma$ and $M \rightarrow_\beta N$, then $\Gamma \vdash N : \sigma$.

- **Strong Normalization**

  If $\Gamma \vdash M : \sigma$, then all $\beta$-reductions from $M$ terminate.

Proof of SN is by defining a reduction preserving map from $\lambda P$ to $\lambda \rightarrow$. 


Decidability Questions

\[ \Gamma \vdash M : \sigma? \quad \text{TCP} \]
\[ \Gamma \vdash M : ? \quad \text{TSP} \]
\[ \Gamma \vdash ? : \sigma \quad \text{TIP} \]

For \( \lambda P \):

- TIP is undecidable
  (Equivalent to provability in minimal predicate logic.)
- TCP/TSP: simultaneously with Context checking
Type Checking algorithm for \( \lambda P \)

Define algorithms \( \text{Ok}(\_\_) \) and \( \text{Type}_\_(-) \) simultaneously:

- \( \text{Ok}(\_\_) \) takes a context and returns ‘true’ or ‘false’
- \( \text{Type}_\_(-) \) takes a context and a term and returns a term or ‘false’.

Definition. The type synthesis algorithm \( \text{Type}_\_(-) \) is sound if

\[
\text{Type}_\Gamma(M) = A \Rightarrow \Gamma \vdash M : A
\]

for all \( \Gamma \) and \( M \).

Definition. The type synthesis algorithm \( \text{Type}_\_(-) \) is complete if

\[
\Gamma \vdash M : A \Rightarrow \text{Type}_\Gamma(M) =_\beta A
\]

for all \( \Gamma, M \) and \( A \).
\[
\text{Ok}(\text{<>}) = \text{‘true’}
\]

\[
\text{Ok}(\Gamma, x:A) = \text{Type}_\Gamma(A) \in \{\ast, \text{kind}\},
\]

\[
\text{Type}_\Gamma(x) = \text{if Ok}(\Gamma) \text{ and } x:A \in \Gamma \text{ then } A \text{ else ‘false’},
\]

\[
\text{Type}_\Gamma(\text{type}) = \text{if Ok}(\Gamma) \text{ then kind else ‘false’},
\]

\[
\text{Type}_\Gamma(MN) = \text{if Type}_\Gamma(M) = C \text{ and Type}_\Gamma(N) = D
\]
\[
\text{then if } C \rightarrow_\beta \Pi x:A.B \text{ and } A =_\beta D
\]
\[
\text{then } B[x := N] \text{ else ‘false’}
\]
\[
\text{else ‘false’},
\]
\[ \text{Type}_\Gamma(\lambda x:A.M) = \ \text{if Type}_{\Gamma,x:A}(M) = B \]

\[ \quad \text{then if Type}_\Gamma(\Pi x:A.B) \in \{\text{type, kind}\} \]

\[ \quad \text{then } \Pi x:A.B \text{ else 'false'} \]

\[ \quad \text{else 'false',} \]

\[ \text{Type}_\Gamma(\Pi x:A.B) = \ \text{if Type}_\Gamma(A) = \text{type and Type}_{\Gamma,x:A}(B) = s \]

\[ \quad \text{then } s \text{ else 'false'} \]
Soundness and Completeness

**Soundness**

\[ \text{Type}_\Gamma(M) = A \Rightarrow \Gamma \vdash M : A \]

**Completeness**

\[ \Gamma \vdash M : A \Rightarrow \text{Type}_\Gamma(M) =_\beta A \]

As a consequence:

\[ \text{Type}_\Gamma(M) = \text{‘false’} \Rightarrow M \text{ is not typable in } \Gamma \]

**NB 1.** Completeness only makes sense if types are **uniqueness up to \( =_\beta \)**

(Otherwise: let \( \text{Type}_-(-) \) generate a **set of possible types**)

**NB 2.** Completeness only implies that \( \text{Type} \) terminates on all **well-typed terms.** We want that \( \text{Type} \) terminates on **all pseudo terms.**
Termination

We want $\text{Type}_-(-)$ to terminate on all inputs.

Interesting cases: $\lambda$-abstraction and application:

$$\text{Type}_\Gamma(\lambda x:A.M) = \begin{cases} \text{if } \text{Type}_{\Gamma,x:A}(M) = B \\ \text{then } \begin{cases} \text{if } \text{Type}_\Gamma(\Pi x:A.B) \in \{\text{type, kind}\} \\ \text{then } \Pi x:A.B \text{ else 'false'} \\ \text{else 'false'}, \end{cases} \end{cases}$$

! Recursive call is not on a smaller term!

Replace the side condition

$$\text{if } \text{Type}_\Gamma(\Pi x:A.B) \in \{\text{type, kind}\}$$

by

$$\text{if } \text{Type}_\Gamma(A) \in \{\text{type}\}$$
Termination

We want $\text{Type}_\rightarrow(-)$ to terminate on all inputs.
Interesting cases: $\lambda$-abstraction and application:

$$\text{Type}_\Gamma(MN) = \begin{cases} \text{if } \text{Type}_\Gamma(M) = C \text{ and } \text{Type}_\Gamma(N) = D \\
\text{then } \text{if } C \rightarrow_\beta \Pi x:A.B \text{ and } A =_\beta D \\
\text{then } B[x := N] \text{ else 'false'} \\
\text{else 'false',} \end{cases}$$

! Need to decide $\beta$-reduction and $\beta$-equality!

For this case, termination follows from soundness of Type and the decidability of equality on well-typed terms (using SN and CR).