Type Theory and Coq

Herman Geuvers

Principal Types and Type Checking

Overview of todays lecture

- Simple Type Theory à la Curry
  (versus Simple Type Theory à la Church)
- Principal Types algorithm
- Type checking dependent type theory: λP

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 Γ.
Derivation rules of $\lambda\!\!\rightarrow$ (à la Church):

$$\frac{x{:}\sigma \in \Gamma}{\Gamma \vdash x : \sigma} \qquad \frac{\Gamma \vdash M : \sigma{\rightarrow}\tau \;\; \Gamma \vdash N : \sigma}{\Gamma \vdash MN : \tau} \qquad \frac{\Gamma, x{:}\sigma \vdash P : \tau}{\Gamma \vdash \lambda x{:}\sigma.P : \sigma{\rightarrow}\tau}$$

$\Gamma \vdash_{\lambda\rightarrow} M : \sigma$ 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 σ
2. type as a proposition, term as its proof:
   M is a proof of the proposition σ

- 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 σ from the assumptions $\tau_1, \tau_2, \ldots, \tau_n$.

Untyped λ-calculus

Untyped λ-calculus

$$\Lambda ::= \mathsf{Var} \mid (\Lambda\,\Lambda) \mid (\lambda \mathsf{Var}.\Lambda)$$

Examples:
- $\mathbf{K} := \lambda x\, y.x$
- $\mathbf{S} := \lambda x\, y\, z.x\, z(y\, z)$
- $\omega := \lambda x.x\, x$
- $\Omega := \omega\, \omega$

$$\Omega \longrightarrow_\beta \Omega$$

Untyped λ-calculus

Untyped λ-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

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

Yes, because we have a fixed point combinator:
- $\mathbf{Y} := \lambda f.(\lambda x.f(x\,x))(\lambda x.f(x\,x))$

Property:

$$Y\,f =_\beta f(Y\,f)$$

Solving recursive equations using fixed point comb.

Property of Y: we have $Y\,f =_\beta f\,(Y\,f)$ for all f.

Is there a term M such that

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

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 σ to M (or detects that the term is not typable, and thus malformed)

STT à la Church and à la Curry

$\lambda\!\rightarrow$ (à la Church):

$$\frac{x{:}\sigma \in \Gamma}{\Gamma \vdash x : \sigma} \qquad \frac{\Gamma \vdash M : \sigma{\rightarrow}\tau \;\; \Gamma \vdash N : \sigma}{\Gamma \vdash MN : \tau} \qquad \frac{\Gamma, x{:}\sigma \vdash P : \tau}{\Gamma \vdash \lambda x{:}\sigma.P : \sigma{\rightarrow}\tau}$$

$\lambda\!\rightarrow$ (à la Curry):

$$\frac{x{:}\sigma \in \Gamma}{\Gamma \vdash x : \sigma} \qquad \frac{\Gamma \vdash M : \sigma{\rightarrow}\tau \;\; \Gamma \vdash N : \sigma}{\Gamma \vdash MN : \tau} \qquad \frac{\Gamma, x{:}\sigma \vdash P : \tau}{\Gamma \vdash \lambda x.P : \sigma{\rightarrow}\tau}$$

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

  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.

$$\begin{aligned} |x| &:= x \\ |M\,N| &:= |M|\,|N| \\ |\lambda x : \sigma.M| &:= \lambda x.|M| \end{aligned}$$

So, e.g.

$$|\lambda x : \alpha.\lambda y : (\beta{\to}\alpha){\to}\alpha.y(\lambda z : \beta.x)| = \lambda x.\lambda y.y(\lambda z.x)$$

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.

$$\begin{aligned} |x| &:= x \\ |M\,N| &:= |M|\,|N| \\ |\lambda x:\sigma.M| &:= \lambda x.|M| \end{aligned}$$

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.

$$\frac{x{:}\sigma \in \Gamma}{\Gamma \vdash x:\sigma} \qquad \frac{\Gamma \vdash M:\sigma{\to}\tau \;\; \Gamma \vdash N:\sigma}{\Gamma \vdash MN:\tau} \qquad \frac{\Gamma, x{:}\sigma \vdash P:\tau}{\Gamma \vdash \lambda x.P:\sigma{\to}\tau}$$

Example of computing a principal type

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

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: $\beta = \alpha{\to}\delta$ $\qquad\qquad$ $\beta = (\gamma{\to}\delta){\to}\varepsilon$
4. Find a most general unifier (a substitution) for the type vars that solves the equations: $\alpha := \gamma{\to}\varepsilon,\ \ \beta := (\gamma{\to}\varepsilon){\to}\varepsilon,\ \ \delta := \varepsilon$
5. The principal type of $\lambda x.\lambda y.y(\lambda z.yx)$ is now

$$(\gamma{\to}\varepsilon){\to}((\gamma{\to}\varepsilon){\to}\varepsilon){\to}\varepsilon$$

Example of computing a principal type

$$\lambda x.\lambda y.y\,(\lambda z.y\,x)$$

Examples of computing a principal type

- $\lambda x.x(\lambda y.x\,(\lambda z.z)$
- $\lambda x.x(\lambda y.y\,(\lambda z.x)$

Exercise

Compute principal types for

- $\mathbf{S} := \lambda x.\lambda y.\lambda z.x\,z(y\,z)$
- $M := \lambda x.\lambda y.x(y(\lambda z.x\,z\,z))(y(\lambda z.x\,z\,z))$.

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 σ and τ is a substitution that "makes σ and τ equal", i.e. an S such that $S(\sigma) = S(\tau)$.
  We also say that "S solves the equation $\sigma = \tau$".
- A most general unifier (or mgu) of the types σ and τ is the "simplest substitution" that solves $\sigma = \tau$, 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 equations $\sigma_1 = \tau_1, \ldots, \sigma_n = \tau_n$ in stead of one equation $\sigma = \tau$.

Computing a most general unifier

There is an algorithm U that, when given $\sigma_1 = \tau_1, \ldots, \sigma_n = \tau_n$ outputs

- A most general unifier of $\sigma_1 = \tau_1, \ldots, \sigma_n = \tau_n$, if this set of equations can be solved. (All $\sigma_i = \tau_i$ can be unified.)
- "Fail" if $\sigma_1 = \tau_1, \ldots, \sigma_n = \tau_n$, cannot be solved.

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

Principal type: Definition

Definition σ is a principal type for the closed untyped λ-term M if

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

A principal type is unique up to renaming of type variables.
Both $\alpha \to \alpha$ and $\beta \to \beta$ are principal type of $\lambda x.x$.

Principal Types Theorem

Theorem There is an algorithm PT that, when given a closed untyped λ-term M, outputs

- A principal type σ 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 λ-terms: There is an algorithm PP that, when given an untyped λ-term M, outputs

- A principal pair (Γ, σ) of M if M is typable in STT à la Curry,
- "Fail" if M is not typable in STT à la Curry.

Definition (Γ, σ) is a principal pair for M if

- $\Gamma \vdash M : \sigma$
- 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	TCP
$M : ?$	Type Synthesis Problem	TSP
$? : \sigma$	Type Inhabitation Problem (by a closed term)	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 τ, 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.

Type checking dependent type systems à la Church

In type systems à la Church there is type information in the λ-abstraction, so in the simple typed case, type checking is easy.

In more involved type systems with dependent types (like the Coq type system), type checking becomes more complicated because we have computation (β-reduction) in types.

We illustrate type checking for dependent type theory by the case for λP.

Rules for λP: axiom, application, abstraction, product

$$\frac{}{\vdash * : \Box}$$

$$\frac{\Gamma \vdash M : \Pi x : A.\, B \qquad \Gamma \vdash N : A}{\Gamma \vdash MN : B[x := N]}$$

$$\frac{\Gamma, x : A \vdash M : B \qquad \Gamma \vdash \Pi x : A.\, B : s}{\Gamma \vdash \lambda x : A.\, M : \Pi x : A.\, B}$$

$$\frac{\Gamma \vdash A : * \qquad \Gamma, x : A \vdash B : s}{\Gamma \vdash \Pi x : A.\, B : s}$$

Rules for λP: weakening, variable, conversion

$$\frac{\Gamma \vdash A : B \qquad \Gamma \vdash C : s}{\Gamma, x : C \vdash A : B}$$

$$\frac{\Gamma \vdash A : s}{\Gamma, x : A \vdash x : A}$$

$$\frac{\Gamma \vdash A : B \qquad \Gamma \vdash B' : s}{\Gamma \vdash A : B'} \quad \text{with } B =_\beta B'$$

Properties of λP

- Uniqueness of types
  If $\Gamma \vdash M : \sigma$ and $\Gamma \vdash M : \tau$, then $\sigma =_\beta \tau$.
- Subject Reduction
  If $\Gamma \vdash M : \sigma$ and $M \longrightarrow_\beta N$, then $\Gamma \vdash N : \sigma$.
- Strong Normalization
  If $\Gamma \vdash M : \sigma$, then all β-reductions from M terminate.

Proof of Strong Normalization is by defining a reduction preserving map from λP to $\lambda\!\!\rightarrow$.

Decidability Questions

$\Gamma \vdash M : \sigma ?$	TCP
$\Gamma \vdash M : ?$	TSP
$\Gamma \vdash ? : \sigma$	TIP

For λP:

- TIP is undecidable
  (Equivalent to provability in minimal predicate logic.)
- TCP/TSP: simultaneously with Context checking

Type Checking algorithm for λP

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

- $\mathrm{Ok}(-)$ takes a context and returns 'true' or 'false'
- $\mathrm{Type\_}(-)$ takes a context and a term and returns a term or 'false'.

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

$$\mathrm{Type}_\Gamma(M) = A \implies \Gamma \vdash M : A$$

for all Γ and M.

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

$$\Gamma \vdash M : A \implies \mathrm{Type}_\Gamma(M) =_\beta A$$

for all Γ, M and A.

$$
\begin{aligned}
\mathrm{Ok}(<>) &= \text{‘true’} \\
\mathrm{Ok}(\Gamma, x{:}A) &= \mathrm{Type}_\Gamma(A) \in \{*, \Box\}, \\
\mathrm{Type}_\Gamma(x) &= \text{if } \mathrm{Ok}(\Gamma) \text{ and } x{:}A \in \Gamma \text{ then } A \text{ else ‘false’}, \\
\mathrm{Type}_\Gamma(*) &= \text{if } \mathrm{Ok}(\Gamma) \text{ then } \Box \text{ else ‘false’}, \\
\mathrm{Type}_\Gamma(MN) &= \text{if } \mathrm{Type}_\Gamma(M) = C \text{ and } \mathrm{Type}_\Gamma(N) = D \\
&\quad \text{then} \quad \text{if } C \twoheadrightarrow_\beta \Pi x{:}A.B \text{ and } A =_\beta D \\
&\qquad\qquad \text{then } B[x := N] \text{ else ‘false’} \\
&\quad \text{else} \quad \text{‘false’},
\end{aligned}
$$

$$
\begin{aligned}
\mathrm{Type}_\Gamma(\lambda x{:}A.M) \quad &= \quad \text{if } \mathrm{Type}_{\Gamma,x:A}(M) = B \\
&\qquad \text{then} \qquad \text{if } \mathrm{Type}_\Gamma(\Pi x{:}A.B) \in \{*, \Box\} \\
&\qquad\qquad\qquad \text{then } \Pi x{:}A.B \text{ else `false'} \\
&\qquad \text{else `false'}, \\
\mathrm{Type}_\Gamma(\Pi x{:}A.B) \quad &= \quad \text{if } \mathrm{Type}_\Gamma(A) = * \text{ and } \mathrm{Type}_{\Gamma,x:A}(B) = s \\
&\qquad\qquad \text{then } s \text{ else `false'}
\end{aligned}
$$

Soundness and Completeness

Soundness

$$\mathrm{Type}_\Gamma(M) = A \quad \Longrightarrow \quad \Gamma \vdash M : A$$

Completeness

$$\Gamma \vdash M : A \quad \Longrightarrow \quad \mathrm{Type}_\Gamma(M) =_\beta A$$

As a consequence:

$$\mathrm{Type}_\Gamma(M) = \text{‘false’} \quad \Longrightarrow \quad M \text{ is not typable in } \Gamma$$

NB 1. Completeness only makes sense if types are uniqueness upto $=_\beta$ (Otherwise: let $\mathrm{Type\_}(-)$ generate a set of possible types)
NB 2. Completeness only implies that Type terminates on all well-typed terms. We want that Type terminates on all pseudo terms.

Termination

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

$$\mathrm{Type}_\Gamma(\lambda x{:}A.M) \quad = \quad \text{if } \mathrm{Type}_{\Gamma,x:A}(M) = B$$
$$\text{then} \qquad \text{if } \mathrm{Type}_\Gamma(\Pi x{:}A.B) \in \{*, \Box\}$$
$$\text{then } \Pi x{:}A.B \text{ else 'false'}$$
$$\text{else 'false'},$$

! Recursive call is not on a smaller term!
Replace the side condition

$$\text{if } \mathrm{Type}_\Gamma(\Pi x{:}A.B) \in \{*, \Box\}$$

by

$$\text{if } \mathrm{Type}_\Gamma(A) \in \{*\}$$

Termination

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

$$\begin{array}{rcl} \mathrm{Type}_\Gamma(MN) & = & \text{if } \mathrm{Type}_\Gamma(M) = C \text{ and } \mathrm{Type}_\Gamma(N) = D \\ & & \quad \text{then} \quad \text{if } C \twoheadrightarrow_\beta \Pi x{:}A.B \text{ and } A =_\beta D \\ & & \qquad\qquad \text{then } B[x := N] \text{ else `false'} \\ & & \quad \text{else} \quad \text{`false'}, \end{array}$$

! Need to decide β-reduction and β-equality!
For this case, termination follows from

- the soundness Type and
- the decidability of equality on well-typed terms,
  which again follows from
  - Strong Normalization and Confluence (to be discussed in the coming lectures).