Type theory and Coq

Herman Geuvers

Lecture Principal types and Type Checking

Overview of todays lecture

- Recap of Simple Type Theory a la Church
- Simple Type Theory a la Curry (versus a la Church)
 A programmers view on type theory
- 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 : σ_1, x_2 : σ_2, \ldots, x_n : σ_n

is a context, usually denoted by Γ . Derivation rules of $\lambda \rightarrow$ (à la Church):

| $x:\sigma\in\Gamma$ | $\Gamma \vdash M : \sigma \rightarrow \tau \ \Gamma \vdash N : \sigma$ | $\Gamma, x: \sigma \vdash P : \tau$ |
|------------------------------------------------|------------------------------------------------------------------------|----------------------------------------------------------------|
| $\overline{\Gamma \vdash \mathbf{x} : \sigma}$ | $\Gamma \vdash MN : \tau$ | $\Gamma \vdash \lambda x: \sigma. P : \sigma \rightarrow \tau$ |

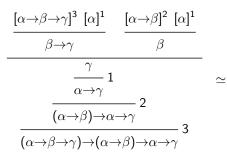
 $\Gamma \vdash_{\lambda \to} M : \sigma$ if there is a derivation using these rules with conclusion $\Gamma \vdash M : \sigma$

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 $\tau_1, \tau_2, \ldots, \tau_n$.

Recap: Example



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

Why do we want types?

- Types give a (partial) specification
- ► Typed terms can't go wrong (Milner) Subject Reduction property: If M : A and M → B N, then N : A.
- 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 not)

Simple Type Theory $(\lambda \rightarrow)$ à la Church and à la Curry

 $\lambda
ightarrow$ (à la Church):

| $x:\sigma\in\Gamma$ | $\Gamma \vdash M : \sigma \rightarrow \tau \ \Gamma \vdash N : \sigma$ | $\Gamma, x: \sigma \vdash P : \tau$ |
|-------------------------------------|------------------------------------------------------------------------|----------------------------------------------------------------------------|
| $\Gamma \vdash \mathbf{x} : \sigma$ | $\Gamma \vdash MN : \tau$ | $\overline{\Gamma \vdash \lambda x : \sigma. P} : \sigma \rightarrow \tau$ |

 $\lambda \rightarrow$ (à la Curry):

| $x:\sigma\in\Gamma$ | $\Gamma \vdash M : \sigma {\rightarrow} \tau \ \Gamma \vdash N : \sigma$ | $\Gamma, x: \sigma \vdash P : \tau$ |
|-------------------------------------|--------------------------------------------------------------------------|-------------------------------------------------------|
| $\Gamma \vdash \mathbf{x} : \sigma$ | $\Gamma \vdash MN : \tau$ | $\Gamma \vdash \lambda x.P : \sigma \rightarrow \tau$ |

Typed Terms versus Type Assignment:

With typed terms also called typing à la Church, we have terms with type information in the λ-abstraction

 $\lambda x : \alpha . x : \alpha \rightarrow \alpha$

As a consequence:

- Terms have unique types,
- The type is directly computed from the type info in the variables.
- With typed assignment also called typing à la Curry, we assign types to untyped λ-terms

$$\lambda x.x: \alpha \rightarrow \alpha$$

As a consequence:

- Terms do not have unique types,
- A principal type can be computed using unification.

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

with $\alpha \rightarrow ((\beta \rightarrow \alpha) \rightarrow \gamma) \rightarrow \gamma$ being the principal type

Connection between Church and Curry typed $\lambda \rightarrow$

Definition The erasure map |-| from $\lambda \rightarrow a$ la Church to $\lambda \rightarrow a$ 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.

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

Theorem If $M : \sigma$ in $\lambda \rightarrow \dot{a}$ la Church, then $|M| : \sigma$ in $\lambda \rightarrow \dot{a}$ la Curry. Theorem If $P : \sigma$ in $\lambda \rightarrow \dot{a}$ la Curry, then there is an M such that $|M| \equiv P$ and $M : \sigma$ in $\lambda \rightarrow \dot{a}$ la Church.

Example of computing a principal type

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

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

2. Assign type vars to all applicative subterms: y x and $y(\lambda z.y x)$: $\lambda x^{\alpha} . \lambda y^{\beta} . \underbrace{y^{\beta}(\lambda z^{\gamma}, y^{\beta} x^{\alpha})}_{\delta}$

- 3. Generate equations between types, necessary for the term to be typable: $\beta = \alpha \rightarrow \delta$ $\beta = (\gamma \rightarrow \delta) \rightarrow \varepsilon$
- 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 $\lambda x.\lambda y.y(\lambda z.yx)$ is now

$$(\gamma \rightarrow \varepsilon) \rightarrow ((\gamma \rightarrow \varepsilon) \rightarrow \varepsilon) \rightarrow \varepsilon$$

Example of computing a principal type (ctd)

 $\lambda x.\lambda y.x y x$

Which of these terms is typable?

$$M_1 := \lambda x. x (\lambda y. y x)$$

$$M_2 := \lambda x \cdot \lambda y \cdot x (x y)$$

•
$$M_3 := \lambda x \cdot \lambda y \cdot x (\lambda z \cdot y \cdot x)$$

Poll:

A M_1 is not typable, M_2 and M_3 are typable.

- B M_2 is not typable, M_1 and M_3 are typable.
- C M_3 is not typable, M_1 and M_2 are typable.

Principal Types: Definitions

- ► A type substitution (or just substitution) is a map S from type variables to types with a finite domain and such that variables that occur in the range of S are not in the domain of S.
- We write S as $[\alpha_1 := \sigma_1, \ldots, \alpha_n := \sigma_n]$ with $\alpha_i \notin \sigma_j \ (\forall i, j)$.
- We can compose substitutions: S; T. We write τ S for substitution S applied to τ. (So we have τ (S; T) = (τ S)T.)
- A unifier of the types σ and τ is a substitution that "makes σ = τ hold, i.e. an S such that σ S = τ S
- A most general unifier (or mgu) of the types σ and τ is the "simplest substitution" that makes σ = τ hold, i.e. an S such that

 $\blacktriangleright \ \sigma S = \tau S$

• for all substitutions T such that $\sigma T = \tau T$ there is a substitution R such that T = S; R.

All these notions generalize to lists of equations $\langle \sigma_1 = \tau_1, \dots, \sigma_n = \tau_n \rangle$ instead of a single equation $\sigma = \tau$.

Computability of most general unifiers

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

- A most general unifier of ⟨σ₁ = τ₁,..., σ_n = τ_n⟩ if these types can be unified.
- "Fail" if $\langle \sigma_1 = \tau_1, \dots, \sigma_n = \tau_n \rangle$ can't be unified.

$$\blacktriangleright U(\langle \alpha = \alpha, \ldots, \sigma_n = \tau_n \rangle) := U(\langle \sigma_2 = \tau_2, \ldots, \sigma_n = \tau_n \rangle).$$

•
$$U(\langle \alpha = \tau_1, \ldots, \sigma_n = \tau_n \rangle) :=$$
 "reject" if $\alpha \in FV(\tau_1)$, $\tau_1 \neq \alpha$.

$$U(\langle \sigma_1 = \alpha, \ldots, \sigma_n = \tau_n \rangle) := U(\langle \alpha = \sigma_1, \ldots, \sigma_n = \tau_n \rangle)$$

- $U(\langle \alpha = \tau_1, \dots, \sigma_n = \tau_n \rangle) := [\alpha := V(\tau_1), V]$, if $\alpha \notin FV(\tau_1)$, where V abbreviates $U(\langle \sigma_2[\alpha := \tau_1] = \tau_2[\alpha := \tau_1], \dots, \sigma_n[\alpha := \tau_1] = \tau_n[\alpha := \tau_1] \rangle).$
- $U(\langle \mu \to \nu = \rho \to \xi, \dots, \sigma_n = \tau_n \rangle) := U(\langle \mu = \rho, \nu = \xi, \dots, \sigma_n = \tau_n \rangle)$

Principal type

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

- $M: \sigma \text{ in } \lambda \rightarrow a$ la Curry
- for all types τ , if $M : \tau$, then $\tau = \sigma S$ for some substitution S.

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

- A principal type σ such that $M : \sigma$ in $\lambda \rightarrow a$ la Curry.
- "Fail" if *M* is not typable in $\lambda \rightarrow$ à la Curry.

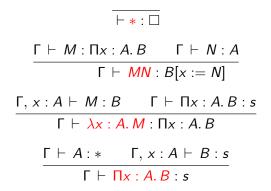
Typical problems one would like to have an algorithm for

| <i>M</i> : <i>σ</i> ? | Type Checking Problem | ТСР |
|-----------------------|----------------------------------------------|-----|
| 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 : σ, one has to solve N :? (and if this gives answer τ, solve M : τ→σ).
- For Curry systems, TCP and TSP soon become undecidable beyond λ→.
- ► TIP is undecidable for most extensions of *λ*→, as it corresponds to provability in some logic.

Rules for λP : axiom, application, abstraction, product



Rules for λP : weakening, variable, conversion

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

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

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 SN is by defining a reduction preserving map from λP to $\lambda {\rightarrow}.$

Decidability Questions

| $\Gamma \vdash M : \sigma$? | ТСР |
|------------------------------|-----|
| $\Gamma \vdash M : ?$ | ТSР |
| $\Gamma \vdash ?: \sigma$ | ΤIΡ |

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 Ok(-) and $Type_{-}(-)$ simultaneously:

- Ok(-) takes a context and returns 'true' or 'false'
- Type_(-) takes a context and a term and returns a term or 'false'.

Definition. The type synthesis algorithm $Type_{-}(-)$ is sound if

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

for all Γ and M.

Definition. The type synthesis algorithm $Type_{-}(-)$ is complete if

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

for all Γ , M and A.

$$Ok(<>) = 'true'$$

$$\operatorname{Ok}(\Gamma, x: A) = \operatorname{Type}_{\Gamma}(A) \in \{*, \Box\},\$$

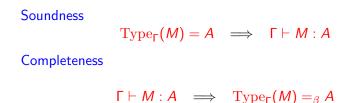
 $\operatorname{Type}_{\Gamma}(x) = \operatorname{if} \operatorname{Ok}(\Gamma) \text{ and } x: A \in \Gamma \text{ then } A \text{ else 'false'},$

$$\operatorname{Type}_{\Gamma}(*) = \text{ if } \operatorname{Ok}(\Gamma) \text{ then } \Box \text{ else 'false'},$$

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

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

Soundness and Completeness



As a consequence:

$\operatorname{Type}_{\Gamma}(M) = \text{`false'} \implies M \text{ is not typable in } \Gamma$

NB 1. Completeness only makes sense if types are unique upto $=_{\beta}$ (Otherwise: let 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 Type₋(-) to terminate on all inputs. Interesting cases: λ -abstraction and application:

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

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

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

by

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

Termination

We want Type₋(-) to terminate on all inputs. Interesting cases: λ -abstraction and application:

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

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