

# Proving with Computer Assistance

Herman Geuvers

Lecture The Curry variant of Simple Type Theory: principal type  
algorithm  
(Or: a functional programmers' view on type theory)

## Overview of todays lecture

- ▶ Simple Type Theory a la Church and a la Curry
- ▶ Principal Types algorithm: A **programmers view** on type theory
- ▶ Properties of Simple Type Theory.

## Recap: Simple type theory a la Church.

Formulation with **contexts** to declare the free variables:

$$x_1 : \sigma_1, x_2 : \sigma_2, \dots, x_n : \sigma_n$$

is a **context**, usually denoted by  $\Gamma$ .

**Derivation rules** of  $\lambda \rightarrow$  (à la Church):

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

$\Gamma \vdash_{\lambda \rightarrow} M : \sigma$  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

►  $x_1 : \tau_1, x_2 : \tau_2, \dots, x_n : \tau_n \vdash M : \sigma$  can be read as  
 $M$  is a **proof** of  $\sigma$  from the **assumptions**  $\tau_1, \tau_2, \dots, \tau_n$ .

## Example

Find a term  $M$  of type  $((A \rightarrow B) \rightarrow A) \rightarrow (A \rightarrow A \rightarrow B) \rightarrow A$ , and give a **typing derivation** that shows this typing.

# Why do programmers want types?

- ▶ Types give a (partial) specification
- ▶ Typed terms can't go wrong (Milner)  
**Subject Reduction property:** If  $M : A$  and  $M \rightarrow_{\beta} N$ , 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  $\sigma$  to  $M$  (or not)

# Simple Type Theory à la Church and à la Curry

$\lambda \rightarrow$  (à la Church):

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

$\lambda \rightarrow$  (à la Curry):

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

## Typed Terms versus Type Assignment:

- ▶ With **typed terms** also called **typing à la Church**, we have **terms with type information** in the  $\lambda$ -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  $\lambda$ -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

- ▶  $\alpha \rightarrow ((\beta \rightarrow \alpha) \rightarrow \alpha) \rightarrow \alpha$
- ▶  $(\alpha \rightarrow \alpha) \rightarrow ((\beta \rightarrow \alpha \rightarrow \alpha) \rightarrow \gamma) \rightarrow \gamma$
- ▶ ...

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

## Example derivation

$\lambda x.\lambda y.y(\lambda z.x)$  can be assigned the type  
 $(\alpha \rightarrow \alpha) \rightarrow ((\beta \rightarrow \alpha \rightarrow \alpha) \rightarrow \gamma) \rightarrow \gamma$  in  $\lambda \rightarrow$  a la Curry.

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

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

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

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  $\Gamma \vdash M : \sigma$  in  $\lambda \rightarrow$  à la Church, then  $\Gamma \vdash |M| : \sigma$  in  $\lambda \rightarrow$  à la Curry.

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

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

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

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

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

Proof: by induction on derivations.

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

## 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  $y(\lambda z. y x)$ :

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

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 (II)

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

## Steps in computing the most general unifier

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

## Which of these terms is typable?

- ▶  $M_1 := \lambda x.x (\lambda y.y x)$
- ▶  $M_2 := \lambda x.\lambda y.x (x y)$
- ▶  $M_3 := \lambda x.\lambda y.x (\lambda z.y x)$

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** such that variables that occur in the **range** of  $S$  are **not in the domain** of  $S$ .
- ▶ A substitution  $S$  is written as  $[\alpha_1 := \sigma_1, \dots, \alpha_n := \sigma_n]$  where
  - ▶ all  $\alpha_i$  are different
  - ▶  $\alpha_i \notin \sigma_j$  (for all  $i, j$ ).
- ▶ We write  $\tau S$  for  $\tau$  with substitution  $S$  applied to it.
- ▶ We can **compose** substitutions:  $S; T$ . (So we have  $\tau(S; T) = (\tau S)T$ .)
- ▶ A **unifier** of the types  $\sigma$  and  $\tau$  is a substitution that **solves**  $\sigma = \tau$ , i.e. an  $S$  such that  $\sigma S = \tau S$
- ▶ A **most general unifier** (**mgu**) of the types  $\sigma$  and  $\tau$  is the “simplest substitution” that solves  $\sigma = \tau$ , i.e. an  $S$  such that
  - ▶  $\sigma S = \tau S$
  - ▶ for all substitutions  $T$  such that  $\sigma T = \tau T$  there is a substitution  $R$  such that  $T = S; R$ .

## Principal Types: solving a list of equations

All notions generalize to lists of equations

$$\mathcal{E} = \langle \sigma_1 = \tau_1, \dots, \sigma_n = \tau_n \rangle$$

instead of a single equation  $\sigma = \tau$ .

- ▶ A **unifier** of  $\mathcal{E}$  is a substitution  $S$  such that  $\sigma_i S = \tau_i S$  for all  $i$ .
- ▶ A **most general unifier (mgu)** for  $\mathcal{E}$  is an  $S$  such that
  - ▶  $\sigma_i S = \tau_i S$  for all  $i$
  - ▶ for all substitutions  $T$  such that  $\sigma_i T = \tau_i T$  for all  $i$ , there is a substitution  $R$  such that  $T = S; R$ .

# Computability of most general unifiers

**THEOREM** There is an algorithm  $U$  that, given a list of equations  $\mathcal{E} = \langle \sigma_1 = \tau_1, \dots, \sigma_n = \tau_n \rangle$  outputs

- ▶ A **most general unifier** of  $\mathcal{E}$  if these equations can be solved.
- ▶ “**Fail**” if  $\mathcal{E}$  can't be solved.

**PROOF**

- ▶  $U(\langle \alpha = \alpha, \dots, \sigma_n = \tau_n \rangle) := U(\langle \sigma_2 = \tau_2, \dots, \sigma_n = \tau_n \rangle)$ .
- ▶  $U(\langle \alpha = \tau_1, \dots, \sigma_n = \tau_n \rangle) :=$  “Fail” if  $\alpha \in \text{FV}(\tau_1)$ ,  $\tau_1 \neq \alpha$ .
- ▶  $U(\langle \sigma_1 = \alpha, \dots, \sigma_n = \tau_n \rangle) := U(\langle \alpha = \sigma_1, \dots, \sigma_n = \tau_n \rangle)$
- ▶  $U(\langle \alpha = \tau_1, \dots, \sigma_n = \tau_n \rangle) := [\alpha := \mathbf{V}(\tau_1), \mathbf{V}]$ , if  $\alpha \notin \text{FV}(\tau_1)$ , where  $\mathbf{V}$  abbreviates  $U(\langle \sigma_2[\alpha := \tau_1] = \tau_2[\alpha := \tau_1], \dots, \sigma_n[\alpha := \tau_1] = \tau_n[\alpha := \tau_1] \rangle)$ .
- ▶  $U(\langle \mu \rightarrow \nu = \rho \rightarrow \xi, \dots, \sigma_n = \tau_n \rangle) := U(\langle \mu = \rho, \nu = \xi, \dots, \sigma_n = \tau_n \rangle)$

## Principal type

DEFINITION  $\sigma$  is a **principal type** for the untyped closed  $\lambda$ -term  $M$  if

- ▶  $\vdash M : \sigma$  in  $\lambda \rightarrow$  à la Curry
- ▶ for all types  $\tau$ , if  $\vdash M : \tau$ , then  $\tau = \sigma T$  for some substitution  $T$ .

# Principal Type Theorem

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

- ▶ A **principal type**  $\sigma$  for  $M$  if  $M$  is typable in  $\lambda \rightarrow$  à la Curry.
- ▶ “Fail” if  $M$  is not typable in  $\lambda \rightarrow$  à la Curry.

**PROOF** In the algorithm we

- ▶ first label the bound variables and all applicative sub-terms with type variables, and we give the candidate type  $\tau$ ,
- ▶ then we generate the equations that need to hold for the term to be typable,
- ▶ then we compute the mgu of this set of equations and we obtain the substitution  $S$  or “Fail”,
- ▶ then we have as output the principal type  $\tau S$  or “Fail”.

The proof that this output indeed correctly computes the principal type can be found in the literature.

## Principal Types for open terms

The definitions and the theory for principal types immediately extends open terms. Consider for example the term

$$M := y (\lambda z. y x)$$

We look for a **principal pair**  $(x : \sigma_1, y : \sigma_2 \ ; \ \sigma)$  satisfying

- ▶  $x : \sigma_1, y : \sigma_2 \vdash M : \sigma$ ,
- ▶ for all  $\tau_1, \tau_2, \tau$  satisfying  $x : \tau_1, y : \tau_2 \vdash M : \tau$  there is a substitution  $T$  such that  $\tau_1 = \sigma_1 T$ ,  $\tau_2 = \sigma_2 T$  and  $\tau = \sigma T$ .

We do this by closing  $M$  under  $\lambda$ -abstractions obtaining

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

for which we have the principal type

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

And so the principal pair for  $M$  is

$$(x : \gamma \rightarrow \varepsilon, y : (\gamma \rightarrow \varepsilon) \rightarrow \varepsilon \ ; \ \varepsilon).$$

## Typical problems one would like to have an algorithm for

$\vdash M : \sigma?$	Type Checking Problem	TCP
$\vdash M : ?$	Type Synthesis Problem	TSP
$\vdash ? : \sigma$	Type Inhabitation Problem	TIP

For  $\lambda \rightarrow$ , all these problems are **decidable**, both for the **Curry** style and for the **Church** style presentation (also if we ask them in a context  $\Gamma$ ).

- ▶ TCP and TSP are (as usual) 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.