Introduction à la compilation

Polytech'Paris-Saclay– 4ème année –

Inference de Types dans le λ-calculus

Burkhart Wolff

Plan of this Course: „λ-calculus"

Plan of this Course: „λ-calculus“

Plan of this Course: „λ-calculus“

- The λ-calculus and its (basic) Hindley-Milner type system

Plan of this Course: „λ-calculus“

- The λ-calculus and its (basic) Hindley-Milner type system
- Properties

Plan of this Course: „λ-calculus“

- The λ-calculus and its (basic) Hindley-Milner type system
- Properties
- Encoding Languages in the typed λ-calculus

Plan of this Course: „λ-calculus"

- The λ-calculus and its (basic) Hindley-Milner type system
- Properties
- Encoding Languages in the typed λ-calculus
- Syntax-directed Type-Inference

Foundations:

Typed λ-Terms

Background: The λ-calculus

Background: The λ-calculus

- Developed in the 30ies by Alonzo Church (and his students Kleene and Rosser)

Background: The λ-calculus

- Developed in the 30ies by Alonzo Church (and his students Kleene and Rosser)
- … to develop a representation

Background: The λ-calculus

- Developed in the 30ies by Alonzo Church (and his students Kleene and Rosser)
- … to develop a representation of Whitehead‘s and Russel‘s „Principia Mathematica“

Background: The λ-calculus

- Developed in the 30ies by Alonzo Church (and his students Kleene and Rosser)
- … to develop a representation of Whitehead‘s and Russel‘s „Principia Mathematica“
- … was early on detected as Turing-complete and actually a “functional computation model“ (Turing)

Foundations:

Typed λ-Terms

The typed λ-calculus

The typed λ-calculus

Motivation:

The typed λ-calculus

Motivation:

- a term - language for representing maths
  (with quantifiers, integrals, limits and stuff -
  thus: variables with binding.)
  in a logic [seminal paper by Church in 1940]

The typed λ-calculus

Motivation:

- a term - language for representing maths
  (with quantifiers, integrals, limits and stuff -
  thus: variables with binding.)
  in a logic [seminal paper by Church in 1940]
- ... in other words: a minimal generic AST
  for expressions and formulas

The typed λ-calculus

Motivation:

- a term - language for representing maths (with quantifiers, integrals, limits and stuff - thus: variables with binding.) in a logic [seminal paper by Church in 1940]
- … in other words: a minimal generic AST for expressions and formulas
- no divergence admissible

The typed λ-calculus

Motivation:

- a term - language for representing maths (with quantifiers, integrals, limits and stuff - thus: variables with binding.) in a logic [seminal paper by Church in 1940]
- … in other words: a minimal generic AST for expressions and formulas
- no divergence admissible
- equality on terms decidable by normalisation

The typed λ-calculus

Motivation:

- a term - language for representing maths
  (with quantifiers, integrals, limits and stuff -
  thus: variables with binding.)
  in a logic [seminal paper by Church in 1940]
- ... in other words: a minimal generic AST
  for expressions and formulas
- no divergence admissible
- equality on terms decidable by normalisation
- turned out to be easy to implement.

The typed λ-calculus

The typed λ-calculus

Idea:

The typed λ-calculus

Idea:

- we use an applied λ-calculus Programming Language Representation

The typed λ-calculus

Idea:

- we use an applied λ-calculus Programming Language Representation
- we introduce the syntactic category of types

The typed λ-calculus

Idea:

- we use an applied λ-calculus Programming Language Representation
- we introduce the syntactic category of types
- we require all „legal" terms to be typed, i.e. an association of a term to a type according to typing rules must be possible.

The typed λ-calculus

Idea:

- we use an applied λ-calculus Programming Language Representation
- we introduce the syntactic category of types
- we require all „legal" terms to be typed, i.e. an association of a term to a type according to typing rules must be possible.
- typed terms were defined inductively.

The typed λ-calculus

The typed λ-calculus

Applied λ-terms T are built inductively over:

- V, a set of "variable symbols"
- C, a set of "constant symbols"
- λV. T, a term construction called "λ–abstraction" ,
- T T , a term construction called "application"

The typed λ-calculus

The typed λ-calculus

Types (1):

The typed λ-calculus

Types (1):

- We assume a set of type constructors χ with symbols like bool, nat, int, _list, _set, _⇒_, ...

The typed λ-calculus

Types (1):

- We assume a set of type constructors χ with symbols like bool, nat, int, _list, _set, _⇒_, ...
- We assume a set of type variables TV for α,β,γ…

The typed λ-calculus

Types (1):

- We assume a set of type constructors χ with symbols like bool, nat, int, _list, _set, _⇒_, ...
- We assume a set of type variables TV for $\alpha,\beta,\gamma\ldots$
- The set of types τ is inductively defined:

$$\tau ::= \mathrm{TV} \mid \chi(\tau_1,\ldots,\tau_n)$$

The typed λ-calculus

The typed λ-calculus

Types (2):

The typed λ-calculus

Types (2):

- We assume a set of type constructors χ with symbols like bool, nat, int, _list, _set, _⇒_, ...

The typed λ-calculus

Types (2):

- We assume a set of type constructors χ with symbols like bool, nat, int, _list, _set, _⇒_, ...
- For type constructors (and constant symbols),

The typed λ-calculus

Types (2):

- We assume a set of type constructors χ with symbols like bool, nat, int, _list, _set, _⇒_, ...
- For type constructors (and constant symbols), we will allow infix/circumfix notation:

  we will write:

The typed λ-calculus

Types (2):

- We assume a set of type constructors χ with symbols like bool, nat, int, _list, _set, _⇒_, ...
- For type constructors (and constant symbols), we will allow infix/circumfix notation:

  we will write:

nat	for	nat()
bool	for	bool()

The typed λ-calculus

Types (2):

- We assume a set of type constructors χ with symbols like bool, nat, int, _list, _set, _⇒_, ...
- For type constructors (and constant symbols), we will allow infix/circumfix notation:

  we will write:

nat	for	nat()
bool	for	bool()
nat list	for	(list_)(nat)

The typed λ-calculus

Types (2):

- We assume a set of type constructors χ with symbols like bool, nat, int, _list, _set, _⇒_, ...
- For type constructors (and constant symbols), we will allow infix/circumfix notation:

  we will write:

nat	for	nat()
bool	for	bool()
nat list	for	(list_)(nat)
bool ⇒ nat	for	(_⇒_)(bool, nat)

The typed λ-calculus

The typed λ-calculus

Types (3):

The typed λ-calculus

Types (3):

- We assume constant environment which assigns each constant symbol a type:

$$\Sigma :: C \mapsto \tau$$

The typed λ-calculus

Types (3):

- We assume constant environment which assigns each constant symbol a type:

  $$\Sigma :: C \mapsto \tau$$

- We assume a variable-environment which assigns to each variable symbol a type:

  $$\Gamma :: V \mapsto \tau$$

(we write $\Gamma = \{a \mapsto \tau_1, b \mapsto \tau_2, c \mapsto \tau_3 \ldots\}$)

The typed λ-calculus

The typed λ-calculus

Type instances :

The typed λ-calculus

Type instances :

- We define a function type_instance as follows:

The typed λ-calculus

Type instances :

- We define a function type_instance as follows:

$$\theta_{\{\alpha_1 \mapsto \tau_1, \ldots, \alpha_n \mapsto \tau_n\}}(\alpha_i) = \tau_i$$

The typed λ-calculus

Type instances :

- We define a function type_instance as follows:

$$\theta_{\{\alpha_1 \mapsto \tau_1, \ldots, \alpha_n \mapsto \tau_n\}}(\alpha_i) = \tau_i$$

$$\theta_{\{\alpha_1 \mapsto \tau_1, \ldots, \alpha_n \mapsto \tau_n\}}(\chi(\tau_1, \ldots, \tau_n)) =$$

The typed λ-calculus

Type instances :

- We define a function type_instance as follows:

$$\theta_{\{\alpha_1 \mapsto \tau_1, \ldots, \alpha_n \mapsto \tau_n\}}(\alpha_i) = \tau_i$$

$$\theta_{\{\alpha_1 \mapsto \tau_1, \ldots, \alpha_n \mapsto \tau_n\}}(\chi(\tau_1, \ldots, \tau_n)) =$$

$$\chi(\theta_{\{\alpha_1 \mapsto \tau_1, \ldots, \alpha_n \mapsto \tau_n\}} \tau_1, \ldots, \theta_{\{\alpha_1 \mapsto \tau_1, \ldots, \alpha_n \mapsto \tau_n\}} \tau_n)$$

The typed λ-calculus

Type instances :

- We define a function type_instance as follows:

$$\theta_{\{\alpha_1 \mapsto \tau_1, \ldots, \alpha_n \mapsto \tau_n\}}(\alpha_i) = \tau_i$$

$$\theta_{\{\alpha_1 \mapsto \tau_1, \ldots, \alpha_n \mapsto \tau_n\}}(\chi(\tau_1, \ldots, \tau_n)) =$$

$$\chi(\theta_{\{\alpha_1 \mapsto \tau_1, \ldots, \alpha_n \mapsto \tau_n\}} \tau_1, \ldots, \theta_{\{\alpha_1 \mapsto \tau_1, \ldots, \alpha_n \mapsto \tau_n\}} \tau_n)$$

- Example:

The typed λ-calculus

Type instances :

- We define a function type_instance as follows:

$$\theta_{\{\alpha_1 \mapsto \tau_1, \ldots, \alpha_n \mapsto \tau_n\}}(\alpha_i) = \tau_i$$

$$\theta_{\{\alpha_1 \mapsto \tau_1, \ldots, \alpha_n \mapsto \tau_n\}}(\chi(\tau_1, \ldots, \tau_n)) =$$

$$\chi(\theta_{\{\alpha_1 \mapsto \tau_1, \ldots, \alpha_n \mapsto \tau_n\}} \tau_1, \ldots, \theta_{\{\alpha_1 \mapsto \tau_1, \ldots, \alpha_n \mapsto \tau_n\}} \tau_n)$$

- Example:

$$\theta_{\{\alpha \mapsto int\}}$$

The typed λ-calculus

Type instances :

- We define a function type_instance as follows:

$$\theta_{\{\alpha_1 \mapsto \tau_1, \ldots, \alpha_n \mapsto \tau_n\}}(\alpha_i) = \tau_i$$

$$\theta_{\{\alpha_1 \mapsto \tau_1, \ldots, \alpha_n \mapsto \tau_n\}}(\chi(\tau_1, \ldots, \tau_n)) =$$

$$\chi(\theta_{\{\alpha_1 \mapsto \tau_1, \ldots, \alpha_n \mapsto \tau_n\}} \tau_1, \ldots, \theta_{\{\alpha_1 \mapsto \tau_1, \ldots, \alpha_n \mapsto \tau_n\}} \tau_n)$$

- Example:

$$\theta_{\{\alpha \mapsto int\}}(\alpha \Rightarrow \alpha \Rightarrow \mathrm{bool})$$

The typed λ-calculus

Type instances :

- We define a function type_instance as follows:

$$\theta_{\{\alpha_1 \mapsto \tau_1, \ldots, \alpha_n \mapsto \tau_n\}}(\alpha_i) = \tau_i$$

$$\theta_{\{\alpha_1 \mapsto \tau_1, \ldots, \alpha_n \mapsto \tau_n\}}(\chi(\tau_1,\ldots,\tau_n)) =$$

$$\chi(\theta_{\{\alpha_1 \mapsto \tau_1, \ldots, \alpha_n \mapsto \tau_n\}}\tau_1,\ldots,\theta_{\{\alpha_1 \mapsto \tau_1, \ldots, \alpha_n \mapsto \tau_n\}}\tau_n)$$

- Example:

$$\theta_{\{\alpha \mapsto int\}}(\alpha \Rightarrow \alpha \Rightarrow bool) = int \Rightarrow int \Rightarrow bool$$

The typed λ-calculus

The typed λ-calculus

Types (4):

The typed λ-calculus

Types (4):

- A type judgement stating that a term t has type τ in environments Σ and Γ:

$$\Sigma, \Gamma \vdash \ t :: \tau$$

The typed λ-calculus

Types (4):

- A type judgement stating that a term t has type τ in environments Σ and Γ:

$$\Sigma, \Gamma \vdash t :: \tau$$

- ... and a set of inductive type inference rules establishing type judgements.

The typed λ-calculus

Types (4):

- A type judgement stating that a term t has type τ in environments Σ and Γ:

$$\Sigma, \Gamma \vdash t :: \tau$$

- … and a set of inductive type inference rules establishing type judgements.

The typed λ-calculus

The typed λ-calculus

- Type Inferences:

The typed λ-calculus

- Type Inferences:

$$\frac{}{\Sigma, \Gamma \vdash c_i :: \theta\ (\Sigma\ c_i)}$$

The typed λ-calculus

- Type Inferences:

$$\overline{\Sigma, \Gamma \vdash c_i :: \theta\ (\Sigma\ c_i)}$$

$$\overline{\Sigma, \Gamma \vdash x_i :: \Gamma\ x_i}$$

The typed λ-calculus

- Type Inferences:

$$\frac{}{\Sigma, \Gamma \vdash c_i :: \theta\ (\Sigma\ c_i)} \qquad \frac{}{\Sigma, \Gamma \vdash x_i :: \Gamma\ x_i}$$

$$\frac{\Sigma, \Gamma \vdash E :: \tau \Rightarrow \tau' \quad \Sigma, \Gamma \vdash E' :: \tau}{\Sigma, \Gamma \vdash E\ E' :: \tau'}$$

The typed λ-calculus

- Type Inferences:

$$\frac{}{\Sigma, \Gamma \vdash c_i :: \theta\ (\Sigma\ c_i)} \qquad \frac{}{\Sigma, \Gamma \vdash x_i :: \Gamma\ x_i}$$

$$\frac{\Sigma, \Gamma \vdash E :: \tau \Rightarrow \tau' \quad \Sigma, \Gamma \vdash E' :: \tau}{\Sigma, \Gamma \vdash E\ E' :: \tau'}$$

$$\frac{\Sigma, \{x_i \mapsto \tau\} \uplus \Gamma \vdash E :: \tau'}{\Sigma, \Gamma \vdash \lambda x_i.E :: \tau \Rightarrow \tau'}$$

The typed λ-calculus

The typed λ-calculus

- Note that constant symbols where treated slightly different than variable symbols:

The typed λ-calculus

- Note that constant symbols where treated slightly different than variable symbols:
  - constant symbols may be instantiated (the type variables may be substituted via θ)

The typed λ-calculus

- Note that constant symbols where treated slightly different than variable symbols:
  - constant symbols may be instantiated (the type variables may be substituted via θ)
  - a constant symbol may therefore have different types in a term.

Typed λ-calculus

Typed λ-calculus

- We assume

Typed λ-calculus

- We assume

$$\Sigma = \{\ 0 \mapsto nat,\ 1 \mapsto nat,\ 2 \mapsto nat,\ 3 \mapsto nat,$$
$$Suc\ \_ \mapsto nat \Rightarrow nat,\ \ \_+\_ \ \mapsto\ nat \Rightarrow nat \Rightarrow nat,$$
$$\_=\_ \mapsto \alpha \Rightarrow \alpha \Rightarrow bool,\ True \mapsto bool,$$
$$False \mapsto bool\}$$

Typed λ-calculus

Typed λ-calculus

- Example:

  does λx. x + 3 have a type, and which one ?

Typed λ-calculus

- Example:

  does λx. x + 3 have a type, and which one ?

$$\Sigma, \{\} \vdash \lambda x.x + 3 :: nat \Rightarrow nat$$

Typed λ-calculus

- Example:

  does λx. x + 3 have a type, and which one ?

$$\frac{\Sigma, \{x \mapsto nat\} \uplus \{\} \vdash x + 3 :: nat}{\Sigma, \{\} \vdash \lambda x. x + 3 :: nat \Rightarrow nat}$$

Typed λ-calculus

- Example:

  does λx. x + 3 have a type, and which one ?

$$\cfrac{\cfrac{\cfrac{}{\Sigma, \{x \mapsto nat\} \vdash (\_ + \_)(x) :: nat \Rightarrow nat} \qquad \cfrac{}{\Sigma, \{x \mapsto nat\} \vdash 3 :: nat}}{\Sigma, \{x \mapsto nat\} \uplus \{\} \vdash x + 3 :: nat}}{\Sigma, \{\} \vdash \lambda x . x + 3 :: nat \Rightarrow nat}$$

Typed λ-calculus

- Example:

  does λx. x + 3 have a type, and which one ?

$$\cfrac{\cfrac{\cfrac{}{\Sigma, \{x \mapsto nat\} \vdash (\_ + \_) :: nat \Rightarrow nat \Rightarrow nat} \quad \cfrac{}{\Sigma, \{x \mapsto nat\} \vdash x :: nat}}{\Sigma, \{x \mapsto nat\} \vdash (\_ + \_)(x) :: nat \Rightarrow nat} \quad \cfrac{}{\Sigma, \{x \mapsto nat\} \vdash 3 :: nat}}{\cfrac{\Sigma, \{x \mapsto nat\} \uplus \{\} \vdash x + 3 :: nat}{\Sigma, \{\} \vdash \lambda x.x + 3 :: nat \Rightarrow nat}}$$

Revisions: Typed λ-calculus

Revisions: Typed λ-calculus

- Examples:

  Are there variable instances $\rho = \{\alpha_1 \mapsto \tau_1, \ldots, \alpha_n \mapsto \tau_n\}$ such that the following terms are typable in Σ:

  (note the infix notation: we write 0 + x for "_+_" 0 x)

Revisions: Typed λ-calculus

- Examples:

  Are there variable instances $\rho = \{\alpha_1 \mapsto \tau_1, \ldots, \alpha_n \mapsto \tau_n\}$ such that the following terms are typable in Σ:
  (note the infix notation: we write 0 + x for "_+_" 0 x)

  - (_+_ 0) = (Suc x)

Revisions: Typed λ-calculus

- Examples:

Are there variable instances $\rho = \{\alpha_1 \mapsto \tau_1, \ldots, \alpha_n \mapsto \tau_n\}$ such that the following terms are typable in Σ:

(note the infix notation: we write 0 + x for "_+_" 0 x)

  - (_+_ 0) = (Suc x)
  - ((x + y) = (y + x)) = False

Revisions: Typed λ-calculus

- Examples:

Are there variable instances $\rho = \{\alpha_1 \mapsto \tau_1, \ldots, \alpha_n \mapsto \tau_n\}$ such that the following terms are typable in Σ:
(note the infix notation: we write 0 + x for "_+_" 0 x)

  - (_+_ 0) = (Suc x)
  - ((x + y) = (y + x)) = False
  - f(_+_ 0) = (λc. g c) x

Revisions: Typed λ-calculus

- Examples:

  Are there variable instances $\rho = \{\alpha_1 \mapsto \tau_1, \ldots, \alpha_n \mapsto \tau_n\}$ such that the following terms are typable in Σ:
  (note the infix notation: we write 0 + x for "_+_" 0 x)

  - (_+_ 0) = (Suc x)
  - ((x + y) = (y + x)) = False
  - f(_+_ 0) = (λc. g c) x
  - _+_ z (_+_ (Suc 0)) = (0 + f False)

Revisions: Typed λ-calculus

- Examples:

  Are there variable instances $\rho = \{\alpha_1 \mapsto \tau_1, \ldots, \alpha_n \mapsto \tau_n\}$ such that the following terms are typable in Σ:
  (note the infix notation: we write 0 + x for "_+_" 0 x)

  - (_+_ 0) = (Suc x)
  - ((x + y) = (y + x)) = False
  - f(_+_ 0) = (λc. g c) x
  - _+_ z (_+_ (Suc 0)) = (0 + f False)
  - a + b = (True = c)

Application:

Encoding a Simple Logic in typed λ-Terms

HOL in Typed λ-calculus

HOL in Typed λ-calculus

- We assume for Higher-Order Logic:

$\Sigma_{HOL} = \Sigma \uplus$

$\{$ True $\mapsto$ bool, False $\mapsto$ bool,

$\_\wedge\_ \mapsto \text{bool}\Rightarrow\text{bool}\Rightarrow\text{bool},$

$\_\vee\_ \mapsto \text{bool}\Rightarrow\text{bool}\Rightarrow\text{bool},$

$\_ \longrightarrow \_ \mapsto \text{bool} \Rightarrow \text{bool} \Rightarrow \text{bool},$

$\neg\ \_ \mapsto \text{bool} \Rightarrow \text{bool},$

$\_ = \_ \mapsto \alpha\Rightarrow\alpha\Rightarrow \text{bool},$

HOL in Typed λ-calculus

- We assume for Higher-Order Logic:

$\Sigma_{HOL} = \Sigma \uplus$

{ True ↦ bool, False ↦ bool,

_∧_ ↦ bool⇒bool⇒bool,

_∨_ ↦ bool⇒bool⇒bool,

_ ⟶ _ ↦ bool ⇒ bool ⇒ bool,

¬ _ ↦ bool ⇒ bool,

_ = _ ↦ α⇒α⇒ bool,

∀_._ ↦ (α⇒bool)⇒ bool,

∃_._ ↦ (α⇒bool)⇒ bool }

Type Inference

The typed λ-calculus

The typed λ-calculus

Theoretical Properties (without proof)

The typed λ-calculus

Theoretical Properties (without proof)

- The type inference problem is decidable, i.e. for

$$\Sigma, ? \vdash t :: ??$$

there is an algorithm that finds solutions for ? and ?? if existing.

The typed λ-calculus

Theoretical Properties (without proof)

- The type inference problem is decidable, i.e. for

$$\Sigma, ? \vdash t :: ??$$

there is an algorithm that finds solutions for ? and ?? if existing.

The typed λ-calculus

Theoretical Properties (without proof)

- The type inference problem is decidable, i.e. for

$$\Sigma, ? \vdash t :: ??$$

there is an algorithm that finds solutions for ? and ?? if existing.

The typed λ-calculus

Theoretical Properties (without proof)

- The type inference problem is decidable, i.e. for

$$\Sigma, ? \vdash t :: ??$$

there is an algorithm that finds solutions for $?$ and $??$ if existing.

The typed λ-calculus

Theoretical Properties (without proof)

- The type inference problem is decidable, i.e. for

$$\Sigma, ? \vdash t :: ??$$

there is an algorithm that finds solutions for $?$ and $??$ if existing.

Type-Inference

Type-Inference

- We assume the operation

  mgu : $\tau \times \tau \Rightarrow (TV \times \tau)$ set

Type-Inference

- We assume the operation

  $\text{mgu} : \tau \times \tau \Rightarrow (TV \times \tau)\ \text{set}$

- where:

  $\text{mgu}(\tau, \tau') = \{\alpha_1 \mapsto \tau_1, \ldots, \alpha_n \mapsto \tau_n\}$ (= ρ)

  if there exists a ρ s.t.

  $$\theta_\rho \tau = \theta_\rho \tau'$$

  $\text{mgu}(t_1, t_2) = \text{error}$ otherwise

Type-Inference

Type-Inference

- We assume the operation

  $$\text{mgu} : \tau \times \tau \Rightarrow (TV \times \tau) \text{ set}$$

Type-Inference

- We assume the operation

  $$\text{mgu} : \tau \times \tau \Rightarrow (TV \times \tau) \text{ set}$$

- Examples:

Type-Inference

- We assume the operation

  $\text{mgu} : \tau \times \tau \Rightarrow (\text{TV} \times \tau)\ \text{set}$

- Examples:

  $\text{mgu}(\alpha_1 \Rightarrow \text{int} \Rightarrow \alpha_1,\ \alpha_1 \Rightarrow \alpha_2 \Rightarrow \alpha_2) = ?$

  $\text{mgu}(\alpha_1 \Rightarrow \text{int} \Rightarrow \alpha_1,\ \alpha_1 \Rightarrow \text{bool} \Rightarrow \alpha_2) = ?$

  $\text{mgu}(\alpha_1 \Rightarrow \text{int} \Rightarrow \alpha_1,\ \alpha_1 \Rightarrow \alpha_2) = ?$

  $\text{mgu}(\alpha_2 \Rightarrow \text{int} \Rightarrow \alpha_2,\ \alpha_1 \Rightarrow \alpha_2) = ?$

Type-Inference

- We assume the operation

  $\text{mgu} : \tau \times \tau \Rightarrow (\text{TV} \times \tau)\ \text{set}$

- Examples:

  $\text{mgu}(\alpha_1 \Rightarrow \text{int} \Rightarrow \alpha_1,\ \alpha_1 \Rightarrow \alpha_2 \Rightarrow \alpha_2) = ?$

  $\text{mgu}(\alpha_1 \Rightarrow \text{int} \Rightarrow \alpha_1,\ \alpha_1 \Rightarrow \text{bool} \Rightarrow \alpha_2) = ?$

  $\text{mgu}(\alpha_1 \Rightarrow \text{int} \Rightarrow \alpha_1,\ \alpha_1 \Rightarrow \alpha_2) = ?$

  $\text{mgu}(\alpha_2 \Rightarrow \text{int} \Rightarrow \alpha_2,\ \alpha_1 \Rightarrow \alpha_2) = ?$

Type-Inference

Type-Inference

- We assume the operation

  `shift:nat` $\Rightarrow (TV \times \tau)$ `set` (renommage index)

  where:

  `shift n` $= \{\alpha_0 \mapsto \alpha_n, \ldots, \alpha_k \mapsto \alpha_{n+k}\}$

Type-Inference

- We assume the operation

  $\texttt{shift:nat} \Rightarrow (TV \times \tau)\ \texttt{set}$ (renommage index)

  where:

  $\texttt{shift}\ n = \{\alpha_0 \mapsto \alpha_n, \ldots, \alpha_k \mapsto \alpha_{n+k}\}$

- We assume an operation

  $\texttt{vars}\ \alpha_i = \{\alpha_i\}$

  $\texttt{vars}\ (\chi(\tau_1, \ldots, \tau_n)) = \bigcup_{i \in \{1..n\}} \texttt{vars}\ \tau_i$

Type-Inference

- We assume the operation

  $\texttt{shift:nat} \Rightarrow (TV \times \tau)\ \texttt{set}$ (renommage index)

  where:

  $\texttt{shift}\ n = \{\alpha_0 \mapsto \alpha_n, \ldots, \alpha_k \mapsto \alpha_{n+k}\}$

- We assume an operation

  $\texttt{vars}\ \alpha_i = \{\alpha_i\}$

  $\texttt{vars}\ (\chi(\tau_1,\ldots,\tau_n)) = \bigcup_{i\in\{1..n\}} \texttt{vars}\ \tau_i$

- We assume that all type variables have an index; `max_index` computes the maximal index out of a set of type vars

Type-Inference

Type-Inference

- The AST of our lambda-calculus:

```
term = C string
     |  V string
     |  lam string term
     |  app term term
```

Type-Inference

Type-Inference

- We will describe the type inference as syntax-directed attribution of terms (alternative: as a deductive rule set)

Type-Inference

- We will describe the type inference as syntax-directed attribution of terms (alternative: as a deductive rule set)
- Inherited attributes:

  $\text{maxindex}_{in} : \text{nat}$

  $\Sigma : C \mapsto \tau$

  $\Gamma : V \mapsto \tau$

Type-Inference

- We will describe the type inference as syntax-directed attribution of terms (alternative: as a deductive rule set)
- Inherited attributes:

  $\text{maxindex}_{in} : \text{nat}$

  $\Sigma : C \mapsto \tau$

  $\Gamma : V \mapsto \tau$

- Synthesised attributes:

  $\text{maxindex}_{out} : \text{nat}$

  $\text{type} : \tau$

Type-Inference

Type-Inference

- Syntax-directed Def of the 'Application'

Type-Inference

- Syntax-directed Def of the 'Application'
- t_0 = app t_1 t_2 :

  $\text{maxindex}_{in}(t_1) = \text{maxindex}_{in}(t_0)$ $\text{maxindex}_{in}(t_2) = \text{maxindex}_{out}(t_1)$

  $\text{maxindex}_{out}(t_0) = \text{maxindex}_{out}(t_2)$

  $\Sigma(t_1) = \Sigma(t_2) = \Sigma(t_0)$ $\Gamma(t_1) = \Gamma(t_2) = \Gamma(t_0)$

  $\text{type}(t_0)$ = case $\text{type}(t_1)$ of

  $\tau_1 \Rightarrow \tau_2$: case $\text{mgu}(\tau_1, \text{type}(t_2))$ of

  {} : error

  | ρ : $\theta_\rho\ \tau_2$

  α : $\alpha \Rightarrow \text{type}(t_2)$

  _ : error

Type-Inference

Type-Inference

- Attribution of the ‘Abstraction’

Type-Inference

- Attribution of the 'Abstraction'
- $t_0 = \text{lam } x\ t_1$:

  $\text{maxindex}_{in}(t_1) = \text{maxindex}_{in}(t_0)+1$

  $\text{maxindex}_{out}(t_0) = \text{maxindex}_{out}(t_1)$

  $\Sigma(t_1) = \Sigma(t_0)$

  $\Gamma(t_1) = \Gamma(t_0) \uplus \{x \mapsto \alpha_m\}$ where $m = \text{maxindex}_{in}(t_0)+1$

  $\text{type}(t_0) =$ '$\alpha_m \Rightarrow \text{type}(t_1)$'

Type-Inference

- Attribution of the 'Abstraction'
- t_0 = lam x t_1 :

  $maxindex_{in}(t_1) = maxindex_{in}(t_0)+1$

  $maxindex_{out}(t_0) = maxindex_{out}(t_1)$

  $\Sigma(t_1) = \Sigma(t_0)$

  $\Gamma(t_1) = \Gamma(t_0) \uplus \{x \mapsto \alpha_m\}$ where $m = maxindex_{in}(t_0)+1$

  $type(t_0) =$ '$\alpha_m \Rightarrow type(t_1)$'

- Recall: '$\alpha_m \Rightarrow type(t_1)$' is a notation for $\_\Rightarrow\_(\alpha_m, t_1)$

Type-Inference

Type-Inference

- Attribution of the ‘free variable’

Type-Inference

- Attribution of the 'free variable'
- $t_0 = V\ x$:

  $maxindex_{out}(t_0) = maxindex_{in}(t_0)$

  $type(t_0) =$ case $lookup(\Gamma(t_0), x)$ of

  $(x \mapsto \alpha_m)$: α_m

  | _ : error

Type-Inference

Type-Inference

- Attribution of the 'constant symbols'

Type-Inference

- Attribution of the ‘constant symbols’
- $t_0 = C\ x$:

  $maxindex_{out}(t_0) = maxindex_{in}(t_0) + m = k$

  $(m, type(t_0)) = $ case lookup$(\Sigma(t_0), x)$ of

  $(x \mapsto \tau)$: (`max_index` (`vars` τ), `shift` $k\ \tau$)

  | _ : error

Type-Inference

Type-Inference

- Examples:

Type-Inference

- Examples:

  - `λf.(λx.f(x x))(λx.f(x x))` typable ?

Type-Inference

- Examples:

  - λf.(λx.f(x x))(λx.f(x x)) typable ?

  - Let $\Sigma = \{I \mapsto \alpha_0 \Rightarrow \alpha_0\}$.

    I I I typable ?

Type-Inference

- Examples:

  - λf.(λx.f(x x))(λx.f(x x)) typable ?

  - Let $\Sigma = \{I \mapsto \alpha_0 \Rightarrow \alpha_0\}$.

    I I I typable ?

Conclusion

Conclusion

- Instead of using a "taylor-made" AST for a given programming language, one can use the typed λ-calculus instead

Conclusion

- Instead of using a "taylor-made" AST
  for a given programming language,
  one can use the typed λ-calculus instead
- Technique is called:
  Higher-order abstract syntax

Conclusion

- Instead of using a "taylor-made" AST for a given programming language, one can use the typed λ-calculus instead
- Technique is called: Higher-order abstract syntax
- Advantage: proper handling of binding and substitution, reduction is terminating, type-inference is possible and practically viable.

Conclusion

- Instead of using a "taylor-made" AST for a given programming language, one can use the typed λ-calculus instead
- Technique is called: Higher-order abstract syntax
- Advantage: proper handling of binding and substitution, reduction is terminating, type-inference is possible and practically viable.