Type reconstruction

constraint-based, unification and a little interpreter

Massimo Nocentini
massimo.nocentini@gmail.com

Prof. Battistina Venneri

Università degli Studi di Firenze

Firenze, May 11, 2013

Road map

1. Introduction
   - Type variables and substitutions
   - Parametric polymorphism and type inference
   - Definition of solution for (Γ, t)
2. Constrait-based typing
   - Constraint set and relations
   - Constraint typing relation
   - Definition of solution for $(\Gamma, t, S, \mathcal{C})$
   - Soundness and Completeness of Constraint typing
3. Unification
   - Algorithm
   - Properties
   - Definition of principal solution for $(\Gamma, t, S, \mathcal{C})$
4. Interpreter
   - Lambda calculus grammar
   - Term substitution issues
   - Nameless representation of terms

Road map

1. Introduction
   - Type variables and substitutions
   - Parametric polymorphism and type inference
   - Definition of solution for (Γ, t)
2. Constrait-based typing
   - Constraint set and relations
   - Constraint typing relation
   - Definition of solution for $(\Gamma, t, S, \mathcal{C})$
   - Soundness and Completeness of Constraint typing
3. Unification
   - Algorithm
   - Properties
   - Definition of principal solution for $(\Gamma, t, S, \mathcal{C})$
4. Interpreter
   - Lambda calculus grammar
   - Term substitution issues
   - Nameless representation of terms

Road map

1. Introduction
   - Type variables and substitutions
   - Parametric polymorphism and type inference
   - Definition of solution for (Γ, t)
2. Constrait-based typing
   - Constraint set and relations
   - Constraint typing relation
   - Definition of solution for $(\Gamma, t, S, \mathcal{C})$
   - Soundness and Completeness of Constraint typing
3. Unification
   - Algorithm
   - Properties
   - Definition of principal solution for $(\Gamma, t, S, \mathcal{C})$
4. Interpreter
   - Lambda calculus grammar
   - Term substitution issues
   - Nameless representation of terms

Road map

1. Introduction
   - Type variables and substitutions
   - Parametric polymorphism and type inference
   - Definition of solution for (Γ, t)
2. Constrait-based typing
   - Constraint set and relations
   - Constraint typing relation
   - Definition of solution for $(\Gamma, t, S, \mathcal{C})$
   - Soundness and Completeness of Constraint typing
3. Unification
   - Algorithm
   - Properties
   - Definition of principal solution for $(\Gamma, t, S, \mathcal{C})$
4. Interpreter
   - Lambda calculus grammar
   - Term substitution issues
   - Nameless representation of terms

Syntax review

Terms

$$\begin{array}{rlr} t ::= & x & \textit{variable} \\ & \lambda x : T.t & \textit{abstraction} \\ & t\ t & \textit{application} \end{array}$$

Values

$$v ::= \quad \lambda x : T.t \quad \textit{abstraction value}$$

Types

$$T ::= \quad T \to T \quad \textit{function type}$$

Contexts

$$\begin{array}{rlr} \Gamma ::= & \emptyset & \textit{empty context} \\ & \Gamma, x : T & \textit{variable binding} \end{array}$$

Evaluation rules review

Evaluation rules

$$\frac{t_1 \rightarrow t_1'}{t_1\ t_2 \rightarrow t_1'\ t_2} \qquad (E\text{-}APP1)$$

$$\frac{t_2 \rightarrow t_2'}{v_1\ t_2 \rightarrow v_1\ t_2'} \qquad (E\text{-}APP2)$$

$$(\lambda x : T_{11}.t_{12})v_2 \rightarrow [x \mapsto v_2]t_{12} \qquad (E\text{-}APPABS)$$

Typing rules review and Inversion lemma

Typing rules

$$\frac{x : T \in \Gamma}{\Gamma \vdash x : T} \quad (T\text{-}VAR)$$

$$\frac{\Gamma, x : T_1 \vdash t_2 : T_2}{\Gamma \vdash \lambda x : T_1 . t_2 : T_1 \to T_2} \quad (T\text{-}ABS)$$

$$\frac{\Gamma \vdash t_1 : T_{11} \to T_{12} \quad \Gamma \vdash t_2 : T_{11}}{\Gamma \vdash t_1\ t_2 : T_{12}} \quad (T\text{-}APP)$$

Inversion Lemma

- if $\Gamma \vdash x : S_1$ then $x : S_1 \in \Gamma$
- if $\Gamma \vdash \lambda x : S_1 . t_2 : S_3$ then $S_3 = S_1 \to S_2$ for some S_2 such that $\Gamma, x : S_1 \vdash t_2 : S_2$
- if $\Gamma \vdash t_1\ t_2 : S_2$ then there exists S_1 such that $\Gamma \vdash t_1 : S_1 \to S_2$ and $\Gamma \vdash t_2 : S_1$

Contents

Introduction

Up to now we've worked with terms which all depends on *explicit* type annotation, that is every λ- abstraction have to declare the *concrete* type for the variable it introduce.

It should be interesting to be "lazy" (in the fisical sense, not because we like to be some instance of a *lazy* structures ☺) and say:

> "I don't care to declare a *concrete* type for my variable now,
> suspend the decision and call this type X, I'll specify it later"

Suppose we're deeply "lazy", why not apply the previous though to *all* variables we introduce in our program? When are our terms well typed if no concrete type is declared at all?

In this lecture we'll see a method which allow us to be "deeply lazy" while been able to type our meaningful terms and discard divergent ones (`lambda x.(x x)`, remember?)

Introduction

Up to now we've worked with terms which all depends on *explicit* type annotation, that is every λ- abstraction have to declare the *concrete* type for the variable it introduce.

It should be interesting to be "lazy" (in the fisical sense, not because we like to be some instance of a *lazy* structures ☺) and say:

> "I don't care to declare a *concrete* type for my variable now,
> suspend the decision and call this type X, I'll specify it later"

Suppose we're deeply "lazy", why not apply the previous though to *all* variables we introduce in our program? When are our terms well typed if no concrete type is declared at all?

In this lecture we'll see a method which allow us to be "deeply lazy" while been able to type our meaningful terms and discard divergent ones (`lambda x.(x x)`, remember?)

Introduction

Up to now we've worked with terms which all depends on *explicit* type annotation, that is every λ- abstraction have to declare the *concrete* type for the variable it introduce.

It should be interesting to be "lazy" (in the fisical sense, not because we like to be some instance of a *lazy* structures ☺) and say:

> "I don't care to declare a *concrete* type for my variable now,
> suspend the decision and call this type X, I'll specify it later"

Suppose we're deeply "lazy", why not apply the previous though to *all* variables we introduce in our program? When are our terms well typed if no concrete type is declared at all?

In this lecture we'll see a method which allow us to be "deeply lazy" while been able to type our meaningful terms and discard divergent ones (`lambda x.(x x)`, remember?)

Introduction

Up to now we've worked with terms which all depends on *explicit* type annotation, that is every λ- abstraction have to declare the *concrete* type for the variable it introduce.

It should be interesting to be "lazy" (in the fisical sense, not because we like to be some instance of a *lazy* structures ☺) and say:

> "I don't care to declare a *concrete* type for my variable now,
> suspend the decision and call this type X, I'll specify it later"

Suppose we're deeply "lazy", why not apply the previous though to *all* variables we introduce in our program? When are our terms well typed if no concrete type is declared at all?

In this lecture we'll see a method which allow us to be "deeply lazy" while been able to type our meaningful terms and discard divergent ones (`lambda x.(x x)`, remember?)

Augment type set with type variables

In order to leave unspecified the type for a variable we've to augment our language with a new type category, which we'll call *type variables*, written as $\mathcal{A}$:

Types - augmented with type variables

$$\begin{array}{lll} T ::= & T \to T & \textit{function type} \\ & \mathcal{A} & \textit{type variable} \\ \mathcal{A} ::= & \{A, B, \ldots, X_{i \in \mathbb{N}}, \ldots\} & \textit{type variables} \end{array}$$

Let $X \in \mathcal{A}$:

- there's no typing rule that uses the category $\mathcal{A}$
- X can represent a *basic* type (ie. `Bool`, `Nat`, ...) or another unspecified type
- X, being a type, can be used by the defined typing rules
- $X \neq Y, \forall Y \in \mathcal{A} \setminus \{X\}$, in other words, $\mathcal{A}$ is infinite and types represented by different type variables are different too

Example

$$\Gamma \vdash \lambda x : X.x : X \to X$$
$$\Gamma \vdash \lambda x : A.x : A \to A$$
$$\Gamma \vdash \lambda s : Z \to Z.\lambda z : Z.(s\,(s\,z)) : (Z \to Z) \to Z \to Z$$

Augment type set with type variables

In order to leave unspecified the type for a variable we've to augment our language with a new type category, which we'll call *type variables*, written as $\mathcal{A}$:

Types - augmented with type variables

$$\begin{array}{lll} T ::= & T \to T & \textit{function type} \\ & \mathcal{A} & \textit{type variable} \\ \mathcal{A} ::= & \{A, B, \ldots, X_{i \in \mathbb{N}}, \ldots\} & \textit{type variables} \end{array}$$

Let $X \in \mathcal{A}$:

- there's no typing rule that uses the category $\mathcal{A}$
- X can represent a *basic* type (ie. `Bool`, `Nat`, ...) or another unspecified type
- X, being a type, can be used by the defined typing rules
- $X \neq Y, \forall Y \in \mathcal{A} \setminus \{X\}$, in other words, $\mathcal{A}$ is infinite and types represented by different type variables are different too

Example

$$\Gamma \vdash \lambda x : X.x : X \to X$$
$$\Gamma \vdash \lambda x : A.x : A \to A$$
$$\Gamma \vdash \lambda s : Z \to Z.\lambda z : Z.(s\,(s\,z)) : (Z \to Z) \to Z \to Z$$

Augment type set with type variables

In order to leave unspecified the type for a variable we've to augment our language with a new type category, which we'll call *type variables*, written as $\mathcal{A}$:

Types - augmented with type variables

$$\begin{array}{lll} T ::= & T \to T & \textit{function type} \\ & \mathcal{A} & \textit{type variable} \\ \mathcal{A} ::= & \{A, B, \ldots, X_{i \in \mathbb{N}}, \ldots\} & \textit{type variables} \end{array}$$

Let $X \in \mathcal{A}$:

- there's no typing rule that uses the category $\mathcal{A}$
- X can represent a *basic* type (ie. `Bool`, `Nat`, ...) or another unspecified type
- X, being a type, can be used by the defined typing rules
- $X \neq Y, \forall Y \in \mathcal{A} \setminus \{X\}$, in other words, $\mathcal{A}$ is infinite and types represented by different type variables are different too

Example

$$\Gamma \vdash \lambda x : X.x : X \to X$$
$$\Gamma \vdash \lambda x : A.x : A \to A$$
$$\Gamma \vdash \lambda s : Z \to Z.\lambda z : Z.(s\,(s\,z)) : (Z \to Z) \to Z \to Z$$

Substitutions

"I'd like that type variable X in my term t to stands for `Nat` type.
Is it possible to do a *substitution*?"

Definition of substitution

A *type substitution* σ is a mapping $\sigma : \mathcal{A} \to T$

A *substitution application* σS_1 of a type substitution σ to a type S_1 is defined inductively on the structure of S_1:

$$X \in \mathcal{A} \wedge \exists T_1 : (X \mapsto T_1) \in \sigma \to \sigma X = T_1$$
$$X \in \mathcal{A} \wedge \forall T_1 : (X \mapsto T_1) \notin \sigma \to \sigma X = X$$
$$T_1 \text{ is a concrete type } \to \sigma T_1 = T_1$$
$$T_1, T_2 \in T \to \sigma(T_1 \to T_2) = \sigma T_1 \to \sigma T_2$$

For the following it is useful to introduce two combinations (let $\mathcal{C}$ is a set of contexts):

- $\Gamma = \{x_1 : T_1, \ldots, x_n : T_n\} \in \mathcal{C} \to \sigma\Gamma = (x_1 : \sigma T_1, \ldots, x_n : \sigma T_n) \in \mathcal{C}$
- let σ, γ be two type substitutions, their composition $\sigma \circ \gamma$ is defined as follows:

$$\sigma \circ \rho = \{X \mapsto \sigma T_1 | \forall T_1 : (X \mapsto T_1) \in \rho\} \cup$$
$$\{X \mapsto T_1 | \forall T_1 : ((X \mapsto T_1) \in \sigma \wedge (X \mapsto T_1) \notin \rho)\}$$

Substitutions

"I'd like that type variable X in my term t to stands for `Nat` type.
Is it possible to do a *substitution*?"

Definition of substitution

A *type substitution* σ is a mapping $\sigma : \mathcal{A} \to T$

A *substitution application* σS_1 of a type substitution σ to a type S_1 is defined inductively on the structure of S_1:

$$X \in \mathcal{A} \wedge \exists T_1 : (X \mapsto T_1) \in \sigma \to \sigma X = T_1$$
$$X \in \mathcal{A} \wedge \forall T_1 : (X \mapsto T_1) \notin \sigma \to \sigma X = X$$
$$T_1 \text{ is a concrete type } \to \sigma T_1 = T_1$$
$$T_1, T_2 \in T \to \sigma(T_1 \to T_2) = \sigma T_1 \to \sigma T_2$$

For the following it is useful to introduce two combinations (let $\mathcal{C}$ is a set of contexts):

- $\Gamma = \{x_1 : T_1, \ldots, x_n : T_n\} \in \mathcal{C} \to \sigma\Gamma = (x_1 : \sigma T_1, \ldots, x_n : \sigma T_n) \in \mathcal{C}$
- let σ, γ be two type substitutions, their composition $\sigma \circ \gamma$ is defined as follows:

$$\sigma \circ \rho = \{X \mapsto \sigma T_1 | \forall T_1 : (X \mapsto T_1) \in \rho\} \cup$$
$$\{X \mapsto T_1 | \forall T_1 : ((X \mapsto T_1) \in \sigma \wedge (X \mapsto T_1) \notin \rho)\}$$

Substitutions

"I'd like that type variable X in my term t to stands for `Nat` type.
Is it possible to do a *substitution*?"

Definition of substitution

A *type substitution* σ is a mapping $\sigma : \mathcal{A} \to T$

A *substitution application* σS_1 of a type substitution σ to a type S_1 is defined inductively on the structure of S_1:

$$X \in \mathcal{A} \wedge \exists T_1 : (X \mapsto T_1) \in \sigma \to \sigma X = T_1$$
$$X \in \mathcal{A} \wedge \forall T_1 : (X \mapsto T_1) \notin \sigma \to \sigma X = X$$
$$T_1 \text{ is a concrete type } \to \sigma T_1 = T_1$$
$$T_1, T_2 \in T \to \sigma(T_1 \to T_2) = \sigma T_1 \to \sigma T_2$$

For the following it is useful to introduce two combinations (let $\mathcal{C}$ is a set of contexts):

- $\Gamma = \{x_1 : T_1, \ldots, x_n : T_n\} \in \mathcal{C} \to \sigma\Gamma = (x_1 : \sigma T_1, \ldots, x_n : \sigma T_n) \in \mathcal{C}$
- let σ, γ be two type substitutions, their composition $\sigma \circ \gamma$ is defined as follows:

$$\sigma \circ \rho = \{X \mapsto \sigma T_1 | \forall T_1 : (X \mapsto T_1) \in \rho\} \cup$$
$$\{X \mapsto T_1 | \forall T_1 : ((X \mapsto T_1) \in \sigma \wedge (X \mapsto T_1) \notin \rho)\}$$

Typing relation is closed under substitution application

Theorem

Let σ be a type substitution, Γ a context, $T_1 \in T$ and t a term. Then:

$$\Gamma \vdash t : T_1 \rightarrow \sigma\Gamma \vdash \sigma t : \sigma T_1$$

Proof. By structural induction on the shape of term t.

$t = x$ By theorem hp $\Gamma \vdash x : S_1$. By inversion lemma $x : S_1 \in \Gamma$. Get any substitution σ and apply it memberwise, $x : \sigma S_1 : \sigma\Gamma$ holds because if $\sigma S_1 = S_2$ for some S_2, then the same happens in $\sigma\Gamma$. By *T-VAR* rule follow $\sigma\Gamma \vdash x : \sigma S_1$ as required

$t = \lambda x : S_1 . t_2$ By theorem hp $\Gamma \vdash \lambda x : S_1 . t_2 : S_2$. By inversion lemma $S_2 = S_1 \rightarrow S_3$ for some S_3 such that $\Gamma, x : S_1 \vdash t_2 : S_3$. By induction hp $\sigma\Gamma, x : \sigma S_1 \vdash \sigma t_2 : \sigma S_3$ holds. By *T-ABS* rule follow $\sigma\Gamma \vdash \lambda x : \sigma S_1 . \sigma t_2 : \sigma S_1 \rightarrow \sigma S_3$ as required

$t = t_1\, t_2$ By theorem hp $\Gamma \vdash t_1\ t_2 : S_1$. By inversion lemma exists S_2 such that $\Gamma \vdash t_1 : S_2 \rightarrow S_1$ and $\Gamma \vdash t_2 : S_2$. By induction hp $\sigma\Gamma \vdash \sigma t_1 : \sigma S_2 \rightarrow \sigma S_1$ and $\sigma\Gamma \vdash \sigma t_2 : \sigma S_2$ hold. By *T-APP* rule follow $\sigma\Gamma \vdash (\sigma t_1)\,(\sigma t_2) : \sigma S_1$ as required

□

Contents

Parametric Polymorphism

Let Γ be a context and t be a term containing free variables. A first question arises:

- $\forall \sigma . \exists T_1 : \sigma\Gamma \vdash \sigma t : T_1$

  Do all type sustitution σ makes the term σt typeable under assumption $\sigma\Gamma$?

  If this is the case, why not write *any* concrete type using a type variable and use a substitution to obtain the original one?

Example

Write $\lambda s : Z \to Z . \lambda z : Z . (s\,(s\,z))$ instead of $\lambda s : Nat \to Nat . \lambda z : Nat . (s\,(s\,z))$

For example, if we work with concrete type *String* just use $\sigma = \{Z \mapsto String\}$

Holding type variables abstract during type checking is called *parametric polymorphism*: type variables are used to allow a term in which they appear usable in many concrete contexts

Parametric Polymorphism

Let Γ be a context and t be a term containing free variables. A first question arises:

- $\forall \sigma . \exists T_1 : \sigma\Gamma \vdash \sigma t : T_1$

  Do all type sustitution σ makes the term σt typeable under assumption $\sigma\Gamma$?

  If this is the case, why not write *any* concrete type using a type variable and use a substitution to obtain the original one?

Example

Write $\lambda s : Z \to Z . \lambda z : Z . (s\,(s\,z))$ instead of $\lambda s : Nat \to Nat . \lambda z : Nat . (s\,(s\,z))$

For example, if we work with concrete type *String* just use $\sigma = \{Z \mapsto String\}$

Holding type variables abstract during type checking is called *parametric polymorphism*: type variables are used to allow a term in which they appear usable in many concrete contexts

Parametric Polymorphism

Let Γ be a context and t be a term containing free variables. A first question arises:

- $\forall \sigma . \exists T_1 : \sigma\Gamma \vdash \sigma t : T_1$

  Do all type sustitution σ makes the term σt typeable under assumption $\sigma\Gamma$?

  If this is the case, why not write *any* concrete type using a type variable and use a substitution to obtain the original one?

Example

Write $\lambda s : Z \to Z . \lambda z : Z . (s\,(s\,z))$ instead of $\lambda s : Nat \to Nat . \lambda z : Nat . (s\,(s\,z))$

For example, if we work with concrete type *String* just use $\sigma = \{Z \mapsto String\}$

Holding type variables abstract during type checking is called *parametric polymorphism*: type variables are used to allow a term in which they appear usable in many concrete contexts

Type inference

Let Γ be a context and t be a term containing free variables. A second question arises:

- $\exists\sigma.\exists T_1 : \sigma\Gamma \vdash \sigma t : T_1$
  Is it always possible to find a type sustitution σ such that the term σt is typeable under assumption $\sigma\Gamma$?

  If this is the case, suppose $\nexists T_1.\Gamma \vdash t : T_1$, using type substitution σ we're able to give a type T_2 to σt, formally $\sigma\Gamma \vdash \sigma t : T_2$

Example

$\lambda s : S.\lambda z : Z.(s\,(s\,z))$ has no type, $\forall S, Z \in \mathcal{A}$

but it has type if we use $\sigma = \{S \mapsto Nat \to Nat, Z \mapsto Nat\}$ or $\sigma = \{S \mapsto Z \to Z\}$

Looking for valid "instantiations" of type variables is called *type inference*: the compiler fill in type information wherever the user don't specify them

Type inference

Let Γ be a context and t be a term containing free variables. A second question arises:

- $\exists\sigma.\exists T_1 : \sigma\Gamma \vdash \sigma t : T_1$

  Is it always possible to find a type sustitution σ such that the term σt is typeable under assumption $\sigma\Gamma$?

  If this is the case, suppose $\nexists T_1.\Gamma \vdash t : T_1$, using type substitution σ we're able to give a type T_2 to σt, formally $\sigma\Gamma \vdash \sigma t : T_2$

Example

$\lambda s : S.\lambda z : Z.(s\,(s\,z))$ has no type, $\forall S, Z \in \mathcal{A}$

but it has type if we use $\sigma = \{S \mapsto Nat \to Nat, Z \mapsto Nat\}$ or $\sigma = \{S \mapsto Z \to Z\}$

Looking for valid "instantiations" of type variables is called *type inference*: the compiler fill in type information wherever the user don't specify them

Type inference

Let Γ be a context and t be a term containing free variables. A second question arises:

- $\exists\sigma.\exists T_1 : \sigma\Gamma \vdash \sigma t : T_1$
  Is it always possible to find a type sustitution σ such that the term σt is typeable under assumption $\sigma\Gamma$?

  If this is the case, suppose $\nexists T_1.\Gamma \vdash t : T_1$, using type substitution σ we're able to give a type T_2 to σt, formally $\sigma\Gamma \vdash \sigma t : T_2$

Example

$\lambda s : S.\lambda z : Z.(s\,(s\,z))$ has no type, $\forall S, Z \in \mathcal{A}$

but it has type if we use $\sigma = \{S \mapsto Nat \to Nat, Z \mapsto Nat\}$ or $\sigma = \{S \mapsto Z \to Z\}$

Looking for valid "instantiations" of type variables is called *type inference*: the compiler fill in type information wherever the user don't specify them

Contents

Let Γ be a context and t be a term containing free variables.

Definition of *solution* for (Γ, t)

Let σ be a type substitution and $T_1 \in T$.
A *solution* for (Γ, t) is a pair (σ, T_1) such that $\sigma\Gamma \vdash \sigma t : T_1$

Example

Let $\Gamma = \{f : X, a : Y\}$ and $t = f\ a$ then:

$$([X \mapsto Nat \to Nat, Y \mapsto Nat], Nat) \quad ([X \mapsto Y \to Z], Z)$$

are both solutions for (Γ, t).

Let Γ be a context and t be a term containing free variables.

Definition of *solution* for (Γ, t)

Let σ be a type substitution and $T_1 \in T$.
A *solution* for (Γ, t) is a pair (σ, T_1) such that $\sigma\Gamma \vdash \sigma t : T_1$

Example

Let $\Gamma = \{f : X, a : Y\}$ and $t = f\ a$ then:

$$([X \mapsto Nat \to Nat, Y \mapsto Nat], Nat) \quad ([X \mapsto Y \to Z], Z)$$

are both solutions for (Γ, t).

Contents

Questions:

- During an execution of a type-checking algorithm, why not record constraints of the form $S_i = T_i$ instead to actually perform a type comparison?
- As we've seen in the last example, there exists countable solutions for a pair (Γ, t), are they related in some way?

Definition of *constraint set* and *unify* relation

A *constraint set* $\mathcal{C}$ is a set of equations $\{S_i \triangleq T_i\}_{i \in \mathbb{N}}$ such that $\mathcal{C} \subseteq T \times T$

A type substitution σ *unify* an equation $S \triangleq T$, written as $\sigma \bowtie S \triangleq T$, if $\sigma S = \sigma T$

Extending the unification relation to constraint sets, a type substitution σ *unify* a constraint set $\mathcal{C}$, written as $\sigma \bowtie \mathcal{C}$, if $\forall (S \triangleq T) \in \mathcal{C} : \sigma \bowtie (S \triangleq T)$

Answers:

- The modified type checking algorithm prove that a term t has type T_1 under assumptions Γ whenever there exists a type substitution σ such that $\sigma \bowtie \mathcal{C}$
- let $\mathcal{C}$ be a constraint set for a pair (Γ, t). Two solutions (σ_1, T_1) and (σ_2, T_2) are related if $\sigma_1 \bowtie \mathcal{C}$ and $\sigma_2 \bowtie \mathcal{C}$

Questions:

- During an execution of a type-checking algorithm, why not record constraints of the form $S_i = T_i$ instead to actually perform a type comparison?
- As we've seen in the last example, there exists countable solutions for a pair (Γ, t), are they related in some way?

Definition of *constraint set* and *unify* relation

A *constraint set* $\mathcal{C}$ is a set of equations $\{S_i \triangleq T_i\}_{i \in \mathbb{N}}$ such that $\mathcal{C} \subseteq T \times T$

A type substitution σ *unify* an equation $S \triangleq T$, written as $\sigma \bowtie S \triangleq T$, if $\sigma S = \sigma T$

Extending the unification relation to constraint sets, a type substitution σ *unify* a constraint set $\mathcal{C}$, written as $\sigma \bowtie \mathcal{C}$, if $\forall (S \triangleq T) \in \mathcal{C} : \sigma \bowtie (S \triangleq T)$

Answers:

- The modified type checking algorithm prove that a term t has type T_1 under assumptions Γ whenever there exists a type substitution σ such that $\sigma \bowtie \mathcal{C}$
- let $\mathcal{C}$ be a constraint set for a pair (Γ, t). Two solutions (σ_1, T_1) and (σ_2, T_2) are related if $\sigma_1 \bowtie \mathcal{C}$ and $\sigma_2 \bowtie \mathcal{C}$

Questions:

- During an execution of a type-checking algorithm, why not record constraints of the form $S_i = T_i$ instead to actually perform a type comparison?
- As we've seen in the last example, there exists countable solutions for a pair (Γ, t), are they related in some way?

Definition of *constraint set* and *unify* relation

A *constraint set* $\mathcal{C}$ is a set of equations $\{S_i \triangleq T_i\}_{i \in \mathbb{N}}$ such that $\mathcal{C} \subseteq T \times T$

A type substitution σ *unify* an equation $S \triangleq T$, written as $\sigma \bowtie S \triangleq T$, if $\sigma S = \sigma T$

Extending the unification relation to constraint sets, a type substitution σ *unify* a constraint set $\mathcal{C}$, written as $\sigma \bowtie \mathcal{C}$, if $\forall (S \triangleq T) \in \mathcal{C} : \sigma \bowtie (S \triangleq T)$

Answers:

- The modified type checking algorithm prove that a term t has type T_1 under assumptions Γ whenever there exists a type substitution σ such that $\sigma \bowtie \mathcal{C}$
- let $\mathcal{C}$ be a constraint set for a pair (Γ, t). Two solutions (σ_1, T_1) and (σ_2, T_2) are related if $\sigma_1 \bowtie \mathcal{C}$ and $\sigma_2 \bowtie \mathcal{C}$

Let's see an example of how we collect a constraint $S \triangleq T$

Example

Suppose to have an application term $t_1\, t_2$ with $\Gamma \vdash t_1 : T_1$ and $\Gamma \vdash t_2 : T_2$

Instead of checking:

- T_1 has the form $T_2 \to T_{12}$, for some T_{12}
- $t_1\, t_2$ has type *exactly* T_{12}

We suspend a decision for the type T_{12}, abstracting it with a *fresh* type variable X, creating the constraint $T_1 \triangleq T_2 \to X$ (hence $t_1\, t_2$ has type X from now on!)

The following questions drive what follow:

- given a pair (Γ, t) there always exist a constraint set?
- suppose a constraint set exists, is it unique?
- assume the existence is enough, how is its construction defined for all cases?
- how can we use the constraint set to build a solution for (Γ, t)?

Let's see an example of how we collect a constraint $S \triangleq T$

Example

Suppose to have an application term $t_1\, t_2$ with $\Gamma \vdash t_1 : T_1$ and $\Gamma \vdash t_2 : T_2$

Instead of checking:

- T_1 has the form $T_2 \to T_{12}$, for some T_{12}
- $t_1\, t_2$ has type *exactly* T_{12}

We suspend a decision for the type T_{12}, abstracting it with a *fresh* type variable X, creating the constraint $T_1 \triangleq T_2 \to X$ (hence $t_1\, t_2$ has type X from now on!)

The following questions drive what follow:

- given a pair (Γ, t) there always exist a constraint set?
- suppose a constraint set exists, is it unique?
- assume the existence is enough, how is its construction defined for all cases?
- how can we use the constraint set to build a solution for (Γ, t)?

Contents

Definition of *constraint typing relation* $\Gamma \vdash t : T_1 \mid_{\mathcal{X}} \mathcal{C}$

The *constraint typing relation* where a term t has type T_1 under assumptions Γ whenever exists a type substitution σ such that $\sigma \bowtie \mathcal{C}$, written as $\Gamma \vdash t : T_1 \mid_{\mathcal{X}} \mathcal{C}$, is defined inductively as follow:

$$\frac{x : T \in \Gamma}{\Gamma \vdash x : T \mid_{\emptyset} \emptyset} \quad (\textit{CT-VAR})$$

$$\frac{\Gamma, x : T_1 \vdash t_2 : T_2 \mid_{\mathcal{X}} \mathcal{C}}{\Gamma \vdash \lambda x : T_1 . t_2 : T_1 \to T_2 \mid_{\mathcal{X}} \mathcal{C}} \quad (\textit{CT-ABS})$$

$$\frac{\begin{array}{rl} \text{let} & X \text{ be fresh variable} \\ \Gamma \vdash & t_1 : T_1 \mid_{\mathcal{X}_1} \mathcal{C}_1 \\ \Gamma \vdash & t_2 : T_2 \mid_{\mathcal{X}_2} \mathcal{C}_2 \\ \mathcal{C}' = & \mathcal{C}_1 \cup \mathcal{C}_2 \cup \{T_1 \triangleq T_2 \to X\} \end{array}}{\Gamma \vdash t_1\ t_2 : X \mid_{\mathcal{X}_1 \cup \mathcal{X}_2 \cup \{X\}} \mathcal{C}'} \quad (\textit{CT-APP})$$

The set $\mathcal{X}$ is used to track type variables introduced by applications of rule *CT-APP*

Extended example

$$\frac{}{\Gamma \vdash \lambda x : X.\lambda y : Y.\lambda z : Z.((x\ z)\ (y\ z)) : S_1 \mid_{\mathcal{X}} \mathcal{C}}$$

Extended example

$$\frac{\dfrac{}{\Gamma, x : X \vdash \lambda y : Y.\lambda z : Z.((x\ z)\ (y\ z)) : S_2 \mid_{\mathcal{X}} \mathcal{C}}}{\Gamma \vdash \lambda x : X.\lambda y : Y.\lambda z : Z.((x\ z)\ (y\ z)) : X \to S_2 \mid_{\mathcal{X}} \mathcal{C}}$$

Extended example

$$\dfrac{\dfrac{\dfrac{}{\Gamma, x : X, y : Y \vdash \lambda z : Z.((x\ z)\ (y\ z)) : S_3 \mid_{\mathcal{X}} \mathcal{C}}}{\Gamma, x : X \vdash \lambda y : Y.\lambda z : Z.((x\ z)\ (y\ z)) : Y \to S_3 \mid_{\mathcal{X}} \mathcal{C}}}{\Gamma \vdash \lambda x : X.\lambda y : Y.\lambda z : Z.((x\ z)\ (y\ z)) : X \to Y \to S_3 \mid_{\mathcal{X}} \mathcal{C}}$$

Extended example

$$\dfrac{\dfrac{\dfrac{\dfrac{}{\Gamma, x : X, y : Y, z : Z \vdash (x\ z)\ (y\ z) : S_4 \mid_{\mathcal{X}} \mathcal{C}}}{\Gamma, x : X, y : Y \vdash \lambda z : Z.((x\ z)\ (y\ z)) : Z \to S_4 \mid_{\mathcal{X}} \mathcal{C}}}{\Gamma, x : X \vdash \lambda y : Y.\lambda z : Z.((x\ z)\ (y\ z)) : Y \to Z \to S_4 \mid_{\mathcal{X}} \mathcal{C}}}{\Gamma \vdash \lambda x : X.\lambda y : Y.\lambda z : Z.((x\ z)\ (y\ z)) : X \to Y \to Z \to S_4 \mid_{\mathcal{X}} \mathcal{C}}$$

Extended example

$$\dfrac{\dfrac{\dfrac{\dfrac{}{\Gamma, x : X, z : Z \vdash x\ z : S_1 \mid_{\mathcal{X}_1} \mathcal{C}_1} \qquad \dfrac{}{\Gamma, y : Y, z : Z \vdash y\ z : S_2 \mid_{\mathcal{X}_2} \mathcal{C}_2}}{\Gamma, x : X, y : Y, z : Z \vdash (x\ z)\ (y\ z) : A \mid_{\mathcal{X}_1 \cup \mathcal{X}_2 \cup \{A\}} \mathcal{C}_1 \cup \mathcal{C}_2 \cup \{S_1 \triangleq S_2 \to A\}}}{\Gamma, x : X, y : Y \vdash \lambda z : Z.((x\ z)\ (y\ z)) : Z \to A \mid_{\mathcal{X}_1 \cup \mathcal{X}_2 \cup \{A\}} \mathcal{C}_1 \cup \mathcal{C}_2 \cup \{S_1 \triangleq S_2 \to A\}}}{\Gamma, x : X \vdash \lambda y : Y.\lambda z : Z.((x\ z)\ (y\ z)) : Y \to Z \to A \mid_{\mathcal{X}_1 \cup \mathcal{X}_2 \cup \{A\}} \mathcal{C}_1 \cup \mathcal{C}_2 \cup \{S_1 \triangleq S_2 \to A\}}}{\Gamma \vdash \lambda x : X.\lambda y : Y.\lambda z : Z.((x\ z)\ (y\ z)) : X \to Y \to Z \to A \mid_{\mathcal{X}_1 \cup \mathcal{X}_2 \cup \{A\}} \mathcal{C}_1 \cup \mathcal{C}_2 \cup \{S_1 \triangleq S_2 \to A\}}$$

Extended example

$$\dfrac{\dfrac{\dfrac{\dfrac{\dfrac{}{\Gamma, x : X \vdash x : S_3 \mid_{\mathcal{X}_3} \mathcal{C}_3} \quad \dfrac{}{\Gamma, z : Z \vdash z : S_4 \mid_{\mathcal{X}_4} \mathcal{C}_4}}{\Gamma, x : X, z : Z \vdash x\, z : B \mid_{\mathcal{X}_3 \cup \mathcal{X}_4 \cup \{B\}} \mathcal{C}_3 \cup \mathcal{C}_4 \cup \{S_3 \triangleq S_4 \to B\}} \quad \dfrac{}{\Gamma, y : Y, z : Z \vdash y\, z : S_2 \mid_{\mathcal{X}_2} \mathcal{C}_2}}{\Gamma, x : X, y : Y, z : Z \vdash (x\, z)\,(y\, z) : A \mid_{\mathcal{X}_3 \cup \mathcal{X}_4 \cup \mathcal{X}_2 \cup \{A,B\}} \mathcal{C}_3 \cup \mathcal{C}_4 \cup \mathcal{C}_2 \cup \{B \triangleq S_2 \to A, S_3 \triangleq S_4 \to B\}}}{\Gamma, x : X, y : Y \vdash \lambda z : Z.((x\, z)\,(y\, z)) : Z \to A \mid_{\mathcal{X}_3 \cup \mathcal{X}_4 \cup \mathcal{X}_2 \cup \{A,B\}} \mathcal{C}_3 \cup \mathcal{C}_4 \cup \mathcal{C}_2 \cup \{B \triangleq S_2 \to A, S_3 \triangleq S_4 \to B\}}}{\Gamma, x : X \vdash \lambda y : Y.\lambda z : Z.((x\, z)\,(y\, z)) : Y \to Z \to A \mid_{\mathcal{X}_3 \cup \mathcal{X}_4 \cup \mathcal{X}_2 \cup \{A,B\}} \mathcal{C}_3 \cup \mathcal{C}_4 \cup \mathcal{C}_2 \cup \{B \triangleq S_2 \to A, S_3 \triangleq S_4 \to}}{\Gamma \vdash \lambda x : X.\lambda y : Y.\lambda z : Z.((x\, z)\,(y\, z)) : X \to Y \to Z \to A \mid_{\mathcal{X}_3 \cup \mathcal{X}_4 \cup \mathcal{X}_2 \cup \{A,B\}} \mathcal{C}_3 \cup \mathcal{C}_4 \cup \mathcal{C}_2 \cup \{B \triangleq S_2 \to A, S_3 \triangleq S_4}$$

Extended example

$$\dfrac{\dfrac{\dfrac{\dfrac{\dfrac{}{x : X \in \Gamma, x : X}}{\Gamma, x : X \vdash x : X \mid_{\emptyset} \emptyset} \quad \dfrac{}{\Gamma, z : Z \vdash z : S_4 \mid_{\mathcal{X}_4} \mathcal{C}_4}}{\Gamma, x : X, z : Z \vdash x\, z : B \mid_{\mathcal{X}_4 \cup \{B\}} \mathcal{C}_4 \cup \{X \triangleq S_4 \to B\}} \quad \dfrac{}{\Gamma, y : Y, z : Z \vdash y\, z : S_2 \mid_{\mathcal{X}_2} \mathcal{C}_2}}{\Gamma, x : X, y : Y, z : Z \vdash (x\, z)\,(y\, z) : A \mid_{\mathcal{X}_4 \cup \mathcal{X}_2 \cup \{A,B\}} \mathcal{C}_4 \cup \mathcal{C}_2 \cup \{B \triangleq S_2 \to A, X \triangleq S_4 \to B\}}}{\Gamma, x : X, y : Y \vdash \lambda z : Z.((x\, z)\,(y\, z)) : Z \to A \mid_{\mathcal{X}_4 \cup \mathcal{X}_2 \cup \{A,B\}} \mathcal{C}_4 \cup \mathcal{C}_2 \cup \{B \triangleq S_2 \to A, X \triangleq S_4 \to B\}}}{\Gamma, x : X \vdash \lambda y : Y.\lambda z : Z.((x\, z)\,(y\, z)) : Y \to Z \to A \mid_{\mathcal{X}_4 \cup \mathcal{X}_2 \cup \{A,B\}} \mathcal{C}_4 \cup \mathcal{C}_2 \cup \{B \triangleq S_2 \to A, X \triangleq S_4 \to B\}}}{\Gamma \vdash \lambda x : X.\lambda y : Y.\lambda z : Z.((x\, z)\,(y\, z)) : X \to Y \to Z \to A \mid_{\mathcal{X}_4 \cup \mathcal{X}_2 \cup \{A,B\}} \mathcal{C}_4 \cup \mathcal{C}_2 \cup \{B \triangleq S_2 \to A, X \triangleq S_4 \to B\}}$$

Extended example

$$\dfrac{\dfrac{\dfrac{\dfrac{}{x : X \in \Gamma, x : X}}{\Gamma, x : X \vdash x : X \mid_{\emptyset} \emptyset} \quad \dfrac{\dfrac{}{z : Z \in \Gamma, z : Z}}{\Gamma, z : Z \vdash z : Z \mid_{\emptyset} \emptyset} \quad \dfrac{}{\Gamma, y : Y, z : Z \vdash y\, z : S_2 \mid_{\mathcal{X}_2} \mathcal{C}_2}}{\Gamma, x : X, z : Z \vdash x\, z : B \mid_{\{B\}} \{X \triangleq Z \to B\}}}{\Gamma, x : X, y : Y, z : Z \vdash (x\, z)\,(y\, z) : A \mid_{\mathcal{X}_2 \cup \{A,B\}} \mathcal{C}_2 \cup \{B \triangleq S_2 \to A, X \triangleq Z \to B\}}}{\Gamma, x : X, y : Y \vdash \lambda z : Z.((x\, z)\,(y\, z)) : Z \to A \mid_{\mathcal{X}_2 \cup \{A,B\}} \mathcal{C}_2 \cup \{B \triangleq S_2 \to A, X \triangleq Z \to B\}}}{\Gamma, x : X \vdash \lambda y : Y.\lambda z : Z.((x\, z)\,(y\, z)) : Y \to Z \to A \mid_{\mathcal{X}_2 \cup \{A,B\}} \mathcal{C}_2 \cup \{B \triangleq S_2 \to A, X \triangleq Z \to B\}}}{\Gamma \vdash \lambda x : X.\lambda y : Y.\lambda z : Z.((x\, z)\,(y\, z)) : X \to Y \to Z \to A \mid_{\mathcal{X}_2 \cup \{A,B\}} \mathcal{C}_2 \cup \{B \triangleq S_2 \to A, X \triangleq Z \to B\}}$$

Extended example

$$\dfrac{\dfrac{\dfrac{\dfrac{\dfrac{}{\Gamma, x : X, z : Z \vdash x\, z : B \mid_{\{B\}} \{X \triangleq Z \to B\}} \qquad \dfrac{}{\Gamma, y : Y \vdash y : S_5 \mid_{\mathcal{X}_5} \mathcal{C}_5} \qquad \dfrac{}{\Gamma, z : Z \vdash z : S_6 \mid_{\mathcal{X}_6} \mathcal{C}_6}}{\Gamma, y : Y, z : Z \vdash y\, z : C \mid_{\mathcal{X}_5 \cup \mathcal{X}_6 \cup \{C\}} \mathcal{C}_5 \cup \mathcal{C}_6 \cup \{S_5 \triangleq S_6 \to}}}{\Gamma, x : X, y : Y, z : Z \vdash (x\, z)\,(y\, z) : A \mid_{\mathcal{X}_5 \cup \mathcal{X}_6 \cup \{A,B,C\}} \mathcal{C}_5 \cup \mathcal{C}_6 \cup \{B \triangleq C \to A, X \triangleq Z \to B, S_5 \triangleq S_6 \to C\}}}{\Gamma, x : X, y : Y \vdash \lambda z : Z.((x\, z)\,(y\, z)) : Z \to A \mid_{\mathcal{X}_5 \cup \mathcal{X}_6 \cup \{A,B,C\}} \mathcal{C}_5 \cup \mathcal{C}_6 \cup \{B \triangleq C \to A, X \triangleq Z \to B, S_5 \triangleq S_6 \to}}}{\Gamma, x : X \vdash \lambda y : Y.\lambda z : Z.((x\, z)\,(y\, z)) : Y \to Z \to A \mid_{\mathcal{X}_5 \cup \mathcal{X}_6 \cup \{A,B,C\}} \mathcal{C}_5 \cup \mathcal{C}_6 \cup \{B \triangleq C \to A, X \triangleq Z \to B, S_5 \triangleq S}}}{\Gamma \vdash \lambda x : X.\lambda y : Y.\lambda z : Z.((x\, z)\,(y\, z)) : X \to Y \to Z \to A \mid_{\mathcal{X}_5 \cup \mathcal{X}_6 \cup \{A,B,C\}} \mathcal{C}_5 \cup \mathcal{C}_6 \cup \{B \triangleq C \to A, X \triangleq Z \to B, S_5 \triangleq}}$$

Extended example

$$\dfrac{\dfrac{\dfrac{\dfrac{}{\Gamma, x : X, z : Z \vdash x\, z : B \,|_{\{B\}}\, \{X \triangleq Z \to B\}} \qquad \dfrac{\dfrac{}{y : Y \in \Gamma, y : Y}}{\Gamma, y : Y \vdash y : Y \,|_{\emptyset}\, \emptyset} \qquad \dfrac{}{\Gamma, z : Z \vdash z : S_6 \,|_{\mathcal{X}_6}\, \mathcal{C}_6}}{\Gamma, y : Y, z : Z \vdash y\, z : C \,|_{\mathcal{X}_6 \cup \{C\}}\, \mathcal{C}_6 \cup \{Y \triangleq S_6 \to C\}}}{\Gamma, x : X, y : Y, z : Z \vdash (x\, z)\, (y\, z) : A \,|_{\mathcal{X}_6 \cup \{A,B,C\}}\, \mathcal{C}_6 \cup \{B \triangleq C \to A, X \triangleq Z \to B, Y \triangleq S_6 \to C\}}}{\Gamma, x : X, y : Y \vdash \lambda z : Z.((x\, z)\, (y\, z)) : Z \to A \,|_{\mathcal{X}_6 \cup \{A,B,C\}}\, \mathcal{C}_6 \cup \{B \triangleq C \to A, X \triangleq Z \to B, Y \triangleq S_6 \to C\}}}{\Gamma, x : X \vdash \lambda y : Y.\lambda z : Z.((x\, z)\, (y\, z)) : Y \to Z \to A \,|_{\mathcal{X}_6 \cup \{A,B,C\}}\, \mathcal{C}_6 \cup \{B \triangleq C \to A, X \triangleq Z \to B, Y \triangleq S_6 \to C\}}}{\Gamma \vdash \lambda x : X.\lambda y : Y.\lambda z : Z.((x\, z)\, (y\, z)) : X \to Y \to Z \to A \,|_{\mathcal{X}_6 \cup \{A,B,C\}}\, \mathcal{C}_6 \cup \{B \triangleq C \to A, X \triangleq Z \to B, Y \triangleq S_6 \to C$$

Extended example

$$\dfrac{\dfrac{\dfrac{\dfrac{}{\Gamma, x : X, z : Z \vdash x\, z : B \mid_{\{B\}} \{X \triangleq Z \to B\}} \quad \dfrac{\dfrac{}{y : Y \in \Gamma, y : Y}}{\Gamma, y : Y \vdash y : Y \mid_{\emptyset} \emptyset} \quad \dfrac{\dfrac{}{z : Z \in \Gamma, z : Z}}{\Gamma, z : Z \vdash z : Z \mid_{\emptyset} \emptyset}}{\Gamma, y : Y, z : Z \vdash y\, z : C \mid_{\{C\}} \{Y \triangleq Z \to C\}}}{\Gamma, x : X, y : Y, z : Z \vdash (x\, z)\,(y\, z) : A \mid_{\{A,B,C\}} \{B \triangleq C \to A, X \triangleq Z \to B, Y \triangleq Z \to C\}}}{\Gamma, x : X, y : Y \vdash \lambda z : Z.((x\, z)\,(y\, z)) : Z \to A \mid_{\{A,B,C\}} \{B \triangleq C \to A, X \triangleq Z \to B, Y \triangleq Z \to C\}}}{\Gamma, x : X \vdash \lambda y : Y.\lambda z : Z.((x\, z)\,(y\, z)) : Y \to Z \to A \mid_{\{A,B,C\}} \{B \triangleq C \to A, X \triangleq Z \to B, Y \triangleq Z \to C\}}}{\Gamma \vdash \lambda x : X.\lambda y : Y.\lambda z : Z.((x\, z)\,(y\, z)) : X \to Y \to Z \to A \mid_{\{A,B,C\}} \{B \triangleq C \to A, X \triangleq Z \to B, Y \triangleq Z \to C\}}$$

Extended example

$$\frac{}{\Gamma, x : X, z : Z \vdash x\, z : B \mid_{\{B\}} \{X \triangleq Z \to B\}} \qquad \frac{}{\Gamma, y : Y, z : Z \vdash y\, z : C \mid_{\{C\}} \{Y \triangleq Z \to C\}}$$

$$\frac{}{\Gamma, x : X, y : Y, z : Z \vdash (x\, z)\,(y\, z) : A \mid_{\{A,B,C\}} \{B \triangleq C \to A, X \triangleq Z \to B, Y \triangleq Z \to C\}}$$

$$\frac{}{\Gamma, x : X, y : Y \vdash \lambda z : Z.((x\, z)\,(y\, z)) : Z \to A \mid_{\{A,B,C\}} \{B \triangleq C \to A, X \triangleq Z \to B, Y \triangleq Z \to C\}}$$

$$\frac{}{\Gamma, x : X \vdash \lambda y : Y.\lambda z : Z.((x\, z)\,(y\, z)) : Y \to Z \to A \mid_{\{A,B,C\}} \{B \triangleq C \to A, X \triangleq Z \to B, Y \triangleq Z \to C\}}$$

$$\frac{}{\Gamma \vdash \lambda x : X.\lambda y : Y.\lambda z : Z.((x\, z)\,(y\, z)) : X \to Y \to Z \to A \mid_{\{A,B,C\}} \{B \triangleq C \to A, X \triangleq Z \to B, Y \triangleq Z \to C\}}$$

Conclusion

We conclude that the term $\lambda x : X.\lambda y : Y.\lambda z : Z.((x\, z)\,(y\, z))$ has *abstract type* $X \to Y \to Z \to A$ if it is possible to find a type substitution σ such that $\sigma \bowtie \{B \triangleq C \to A, X \triangleq Z \to B, Y \triangleq Z \to C\}$

Extended example

$$\frac{}{\Gamma \vdash \lambda x : X.(x\,x) : S_1 \mid_{\mathcal{X}} \mathcal{C}}$$

Extended example

$$\frac{\overline{\Gamma, x : X \vdash x\, x : S_2 \mid_{\mathcal{X}} \mathcal{C}}}{\Gamma \vdash \lambda x : X.(x\, x) : X \to S_2 \mid_{\mathcal{X}} \mathcal{C}}$$

Extended example

$$\dfrac{\dfrac{\overline{\Gamma, x : X \vdash x : S_1 \,|_{\mathcal{X}_1}\, \mathcal{C}_1} \qquad \overline{\Gamma, x : X \vdash x : S_2 \,|_{\mathcal{X}_2}\, \mathcal{C}_2}}{\Gamma, x : X \vdash x\, x : A \,|_{\mathcal{X}_1 \cup \mathcal{X}_2 \cup \{A\}}\, \mathcal{C}_1 \cup \mathcal{C}_2 \cup \{S_1 \triangleq S_2 \to A\}}}{\Gamma \vdash \lambda x : X.(x\, x) : X \to A \,|_{\mathcal{X}_1 \cup \mathcal{X}_2 \cup \{A\}}\, \mathcal{C}_1 \cup \mathcal{C}_2 \cup \{S_1 \triangleq S_2 \to A\}}$$

Extended example

$$\dfrac{\dfrac{\dfrac{}{x : X \in \Gamma, x : X}}{\Gamma, x : X \vdash x : X \mid_{\emptyset} \emptyset} \qquad \dfrac{}{\Gamma, x : X \vdash x : S_2 \mid_{\mathcal{X}_2} \mathcal{C}_2}}{\dfrac{\Gamma, x : X \vdash x\, x : A \mid_{\mathcal{X}_2 \cup \{A\}} \mathcal{C}_2 \cup \{X \triangleq S_2 \to A\}}{\Gamma \vdash \lambda x : X.(x\, x) : X \to A \mid_{\mathcal{X}_2 \cup \{A\}} \mathcal{C}_2 \cup \{X \triangleq S_2 \to A\}}}$$

Extended example

$$\dfrac{\dfrac{\dfrac{}{x : X \in \Gamma, x : X}}{\Gamma, x : X \vdash x : X \mid_{\emptyset} \emptyset} \qquad \dfrac{\dfrac{}{x : X \in \Gamma, x : X}}{\Gamma, x : X \vdash x : X \mid_{\emptyset} \emptyset}}{\dfrac{\Gamma, x : X \vdash x\, x : A \mid_{\{A\}} \{X \triangleq X \to A\}}{\Gamma \vdash \lambda x : X.(x\, x) : X \to A \mid_{\{A\}} \{X \triangleq X \to A\}}}$$

Conclusion

We conclude that the term $\lambda x : X.(x\, x))$ has *abstract type* $X \to A$ if it is possible to find a type substitution σ such that $\sigma \bowtie \{X \triangleq X \to A\}$

Contents

It is helpful make the following observations:

- when a type variable X is choosen by a final rule which has some premises, then X is different from all other type variables introduced by premises' subderivations
- for any given pair (Γ, t) the rules provide a procedure to build sets $\mathcal{X}, \mathcal{C}$ and find type T_1 such that $\Gamma \vdash t : T_1 \mid_{\mathcal{X}} \mathcal{C}$
- if we consider the relation $\Gamma \vdash t : T_1 \mid_{\mathcal{X}} \mathcal{C}$ modulo the choice of fresh variables, then the constraint set $\mathcal{C}$ and type T_1 are *uniquely* determined
- in order to find solutions for (Γ, t) we use the given rules to:
  1. build the constraint set $\mathcal{C}$, that must be satisfied in order for t to have a type
  2. determine a type S possibly containing type variables (which are subjects under constraints in $\mathcal{C}$), which caracterizes the types of t in terms of these variables
  3. find a type sustitution σ such that $\sigma \bowtie \mathcal{C}$: for each such σ the type σS is a possible type for t, hence $(\sigma, \sigma S)$ is a solution for (Γ, t)
  4. if no type sustitution $\sigma \bowtie \mathcal{C}$ exists then there is no way to instantiate type variables in t in order for t to have a type.

Let Γ be a context and t be a term containing free variables.

Definition of *solution* for $(\Gamma, t, S, \mathcal{C})$

Let σ be a type substitution, $T_1 \in T$ and suppose $\Gamma \vdash t : S \mid_{\mathcal{X}} \mathcal{C}$
A *solution* for $(\Gamma, t, S, \mathcal{C})$ is a pair (σ, T_1) such that $\sigma \bowtie \mathcal{C} \wedge \sigma S = T_1$

Example

Let $t = \lambda x : X \to Y.x\,0$ then:

$$S = (X \to Y) \to Z$$
$$\mathcal{C} = \{Nat \to Z \triangleq X \to Y\}$$
$$\sigma = \{X \mapsto Nat, Z \mapsto Bool, Y \mapsto Bool\}$$

hence $(\sigma, (Nat \to Bool) \to Bool)$ is a solution for $(\Gamma, t, S, \mathcal{C})$.

Let Γ be a context and t be a term containing free variables.

Definition of *solution* for $(\Gamma, t, S, \mathcal{C})$

Let σ be a type substitution, $T_1 \in T$ and suppose $\Gamma \vdash t : S \mid_{\mathcal{X}} \mathcal{C}$
A *solution* for $(\Gamma, t, S, \mathcal{C})$ is a pair (σ, T_1) such that $\sigma \bowtie \mathcal{C} \wedge \sigma S = T_1$

Example

Let $t = \lambda x : X \to Y.x\,0$ then:

$$S = (X \to Y) \to Z$$
$$\mathcal{C} = \{Nat \to Z \triangleq X \to Y\}$$
$$\sigma = \{X \mapsto Nat, Z \mapsto Bool, Y \mapsto Bool\}$$

hence $(\sigma, (Nat \to Bool) \to Bool)$ is a solution for $(\Gamma, t, S, \mathcal{C})$.

Contents

Let Γ be a context and t be a term containing free variables

We have two different ways of instantiating type variables appearing in t to produce a typeable term. In the next definitions σ is a type sustitution and $T_1 \in T$ as usual:

Declarative approach

$\Omega = \{\omega = (\sigma, T_1) : \omega \text{ is solution of } (\Gamma, t)\}$

Algorithmic approach

$\Theta = \{\theta = (\sigma, T_1) : \theta \text{ is solution of } (\Gamma, t, S, \mathcal{C})\}$

Theorem

$\Omega = \Theta$

Let Γ be a context and t be a term containing free variables

We have two different ways of instantiating type variables appearing in t to produce a typeable term. In the next definitions σ is a type sustitution and $T_1 \in T$ as usual:

Declarative approach

$\Omega = \{\omega = (\sigma, T_1) : \omega \text{ is solution of } (\Gamma, t)\}$

Algorithmic approach

$\Theta = \{\theta = (\sigma, T_1) : \theta \text{ is solution of } (\Gamma, t, S, \mathcal{C})\}$

Theorem

$\Omega = \Theta$

Proof. Soundness of Constraint typing: $\Omega \supseteq \Theta$.

By hypothesis, let $\Gamma \vdash t : S \mid_{\mathcal{X}} \mathcal{C}$ ($\mathcal{X}$ is useless in this proof) and (σ, T_1) a solution for $(\Gamma, t, S, \mathcal{C})$. Proceed by induction on the last constraint type relation rule applied:

CT-VAR $\frac{x:S \in \Gamma}{\Gamma \vdash x:S \mid \emptyset}$ is the applied rule. By typing rule *T-VAR* follows $x : S \in \Gamma \rightarrow \Gamma \vdash x : S$. Since typing is preserved under type substitution, $\sigma\Gamma \vdash \sigma x : \sigma S$. But (σ, T_1) is solution for $(\Gamma, x, S, \emptyset)$ by theorem hp, hence $\sigma S = T_1$ getting $\sigma\Gamma \vdash \sigma x : T_1$, so (σ, T_1) is solution for (Γ, x) which is like to say $(\sigma, T_1) \in \Omega$ as required.

CT-ABS $\frac{\Gamma, x:S_1 \vdash t_2:S_2 \mid \mathcal{C}}{\Gamma \vdash x:S_1 \,.\, t_2:S_1 \rightarrow S_2 \mid \mathcal{C}}$ is the applied rule. By premises, let $(\sigma, \sigma S_2)$ be solution for $(x : S_1 \cup \Gamma, t_2, S_2, \mathcal{C})$ so we apply induction hp, having $(\sigma, \sigma S_2) \in \Omega$, in other words $(\sigma, \sigma S_2)$ is solution for $(x : S_1 \cup \Gamma, t_2)$, so we write $\sigma\Gamma, x : \sigma S_1 \vdash \sigma t_2 : \sigma S_2$ which is the premise of *T-ABS* rule, hence $\sigma\Gamma \vdash \lambda x : \sigma S_1 . \sigma t_2 : \sigma S_1 \rightarrow \sigma S_2$. But (σ, T_1) is solution for $(\Gamma, \lambda x : S_1 . t_2, S_1 \rightarrow S_2, \mathcal{C})$ by theorem hp, hence $\sigma \bowtie \mathcal{C} \wedge \sigma(S_1 \rightarrow S_2) = T_1$, so we can rewrite $\sigma\Gamma \vdash \lambda x : \sigma S_1 . \sigma t_2 : T_1$, so (σ, T_1) is solution for $(\Gamma, \lambda x : S_1 . t_2)$ which is like to say $(\sigma, T_1) \in \Omega$ as required.

CT-APP $\frac{\begin{array}{ll} \Gamma \vdash t_1 : S_1 \mid \mathcal{C}_1 & \Gamma \vdash t_2 : S_2 \mid \mathcal{C}_2 \\ \mathcal{C}' = \mathcal{C}_1 \cup \mathcal{C}_2 \cup & \{S_1 \triangleq S_2 \rightarrow X\} \end{array}}{\Gamma \vdash t_1\ t_2 : X \mid_{\mathcal{X}_1 \cup \mathcal{X}_2 \cup \{X\}} \mathcal{C}'}$ is the applied rule. By induction hp, we have both $(\sigma, \sigma S_1)$ is solution for (Γ, t_1) both $(\sigma, \sigma S_2)$ is solution for (Γ, t_2), in other words $\sigma\Gamma \vdash \sigma t_1 : \sigma S_1$ and $\sigma\Gamma \vdash \sigma t_2 : \sigma S_2$. By theorem hp, (σ, T_1) is solution for $(\Gamma, t_1\, t_2, X, \mathcal{C}')$, hence $\sigma \bowtie \mathcal{C}'$, in particular $\sigma \bowtie S_1 \triangleq S_2 \rightarrow X$, which does mean $\sigma S_1 = \sigma S_2 \rightarrow \sigma X$. Substituting σS_1 in the previous $\sigma\Gamma \vdash \sigma t_1 : \sigma S_1$ we have the two premises to apply *T-APP* rule, from $\sigma\Gamma \vdash \sigma t_1 : \sigma S_2 \rightarrow \sigma X$ and $\sigma\Gamma \vdash \sigma t_2 : \sigma S_2$ follows that $\sigma\Gamma \vdash (\sigma t_1)(\sigma t_2) : \sigma X$. Again, (σ, T_1) is solution for $(\Gamma, t_1\, t_2, X, \mathcal{C}')$, hence $\sigma X = T_1$, we have $\sigma\Gamma \vdash (\sigma t_1)(\sigma t_2) : T_1$, so (σ, T_1) is solution for $(\Gamma, t_1\ t_2)$ which is like to say $(\sigma, T_1) \in \Omega$ as required.

□

Proof. Completeness of Constraint typing: $\Omega \subseteq \Theta$.

By hypothesis, let $\Gamma \vdash t : S \mid_{\mathcal{X}} \mathcal{C}$ and (σ, T_1) a solution for (Γ, t) such that $dom(\sigma) \cap \mathcal{X} = \emptyset$. We want to show that exists a solution (ρ, T_1) for $(\Gamma, t, S, \mathcal{C})$ such that $\rho \setminus \mathcal{X} = \sigma$. Proceed by induction on the last constraint type relation rule applied:

CT-VAR $\frac{x:S \in \Gamma}{\Gamma \vdash x:S \mid \emptyset}$ is the applied rule. By theorem hp, (σ, T_1) is solution for (Γ, x), that is $\sigma\Gamma \vdash x : T_1$ ($\sigma x = x$ since x is a var name). By inversion lemma holds $x : T_1 \in \sigma\Gamma$. If it was possible to apply the rule, then $x : S \in \Gamma$, hence $x : \sigma S \in \sigma\Gamma$. By pattern matching follows that $T_1 = \sigma S$, hence (σ, T_1) is solution for $(\Gamma, x, S, \emptyset)$ which is like to say $(\sigma, T_1) \in \Theta$ as required.

CT-ABS $\frac{\Gamma, x:S_1 \vdash t_2:S_2 \mid_{\mathcal{X}} \mathcal{C}}{\Gamma \vdash x:S_1 . t_2 : S_1 \to S_2 \mid_{\mathcal{X}} \mathcal{C}}$ is the applied rule. By theorem hp, (σ, T_1) is solution for $(\Gamma, \lambda x : S_1 . t_2)$, that is $\sigma\Gamma \vdash \lambda x : \sigma S_1 . \sigma t_2 : T_1$. By inversion lemma holds $T_1 = \sigma S_1 \to S_3$ for some S_3 such that $\sigma\Gamma, x : \sigma S_1 \vdash \sigma t_2 : S_3$, so (σ, S_3) is solution for $(\Gamma \cup x : S_1, t_2)$. This allow us to apply induction hp: let (ρ, S_3) be a solution for $(\Gamma \cup x : S_1, t_2, S_2, \mathcal{C})$ such that $\rho \setminus \mathcal{X} = \sigma$. Observe that $\mathcal{X} \cap FV(S_1) = \emptyset$ by def of constraint rules, hence $\rho S_1 = \sigma S_1$. So $\rho(S_1 \to S_2) = \rho S_1 \to \rho S_2 = \sigma S_1 \to \rho S_2 = \sigma S_1 \to S_3 = T_1$. This imply that (ρ, T_1) is solution for $(\Gamma, \lambda x : S_1 . t_2, S_1 \to S_2, \mathcal{C})$ which is like to say $(\rho, T_1) \in \Theta$ as required.

□

Proof. Completeness of Constraint typing: $\Omega \subseteq \Theta$.

By hypothesis, let $\Gamma \vdash t : S \mid_{\mathcal{X}} \mathcal{C}$ and (σ, T_1) a solution for (Γ, t) such that $dom(\sigma) \cap \mathcal{X} = \emptyset$. We want to show that exists a solution (ρ, T_1) for $(\Gamma, t, S, \mathcal{C})$ such that $\rho \setminus \mathcal{X} = \sigma$. Proceed by induction on the last constraint type relation rule applied:

$$\text{CT-APP}\ \frac{\begin{array}{ll}\Gamma \vdash t_1 : S_1 \mid \mathcal{C}_1 & \Gamma \vdash t_2 : S_2 \mid \mathcal{C}_2 \\ \mathcal{C}' = \mathcal{C}_1 \cup \mathcal{C}_2 \cup & \{S_1 \triangleq S_2 \to X\}\end{array}}{\Gamma \vdash t_1\ t_2 : X \mid_{\mathcal{X}_1 \cup \mathcal{X}_2 \cup \{X\}} \mathcal{C}'}$$

is the applied rule. By theorem hp, (σ, T_1) is solution for $(\Gamma, t_1\ t_2)$, that is $\sigma\Gamma \vdash \sigma t_1\ \sigma t_2 : T_1$. By inversion lemma there exists S_3 such that $\sigma\Gamma \vdash \sigma t_1 : S_3 \to T_1$ and $\sigma\Gamma \vdash \sigma t_2 : S_3$, so $(\sigma, S_3 \to T_1)$ is solution for (Γ, t_1) and (σ, S_3) is solution for (Γ, t_2). This allow us to apply induction hp: let $(\rho_1, S_3 \to T_1)$ be solution for $(\Gamma, t_1, S_1, \mathcal{C}_1)$ and (ρ_2, S_3) be solution for $(\Gamma, t_2, S_2, \mathcal{C}_2)$, such that $\rho_1 \setminus \mathcal{X}_1 = \rho_2 \setminus \mathcal{X}_2 = \sigma$. We construct a new substitution ρ such that:

$$\rho = \begin{cases} Y \mapsto U & \text{if} \quad Y \notin \mathcal{X} \wedge Y \mapsto U \in \sigma \\ Y_1 \mapsto U_1 & \text{if} \quad Y_1 \in \mathcal{X}_1 \wedge Y_1 \mapsto U_1 \in \rho_1 \\ Y_2 \mapsto U_2 & \text{if} \quad Y_2 \in \mathcal{X}_2 \wedge Y_2 \mapsto U_2 \in \rho_2 \\ X \mapsto T_1 & \end{cases}$$

Now we have to show that (ρ, T_1) is solution for $(\Gamma, t_1\ t_2, X, \mathcal{C}_1 \cup \mathcal{C}_2 \cup \{S_1 \triangleq S_2 \to X\})$ and $\rho \setminus (\mathcal{X}_1 \cup \mathcal{X}_2 \cup \{X\}) = \sigma$:

- by def of ρ follows $\rho X = T_1$. By induction hp $\mathcal{X}_1, \mathcal{X}_2$ do not overlap and *rho* does what ρ_1, ρ_2 do on variables belong to either $\mathcal{X}_1$ or $\mathcal{X}_2$, hence $\rho \bowtie \mathcal{C}_1$ and $\rho \bowtie \mathcal{C}_2$. To show that $\rho \bowtie S_1 \triangleq S_2 \to X$ observe that $\rho S_1 = \rho_1 S_1$ and $\rho S_2 = \rho_2 S_2$ since free variable choosen in a derivation cannot appear in other subderivations, so $\rho S_1 = \rho_1 S_1 = S_3 \to T_1$, but $\rho_2 S_2 = S_3$ by induction hp, hence $S_3 \to T_1 = \rho_2 S_2 \to T_1 = \rho S_2 \to \rho X = \rho(S_2 \to X)$ as required. This prove that (ρ, T_1) is solution
- $\rho \setminus (\mathcal{X}_1 \cup \mathcal{X}_2 \cup \{X\}) = \sigma$ by first rule in def of ρ

So (ρ, T_1) is solution for $(\Gamma, t_1\ t_2, X, \mathcal{C}_1 \cup \mathcal{C}_2 \cup \{S_1 \triangleq S_2 \to X\})$ which is like to say $(\rho, T_1) \in \Theta$ as required.

□

Contents

Unification algorithm

By pattern matching on the structure of constraint set $\mathcal{C}$ given as argument:

$$
\begin{aligned}
\mathit{unify}(\emptyset) &= [\,] \\
\mathit{unify}(\{T_1 \triangleq T_2\} \cup \mathcal{C}') &= \mathit{unify}(\mathcal{C}') \quad \text{if } T_1 = T_2 \\
\mathit{unify}(\{X \triangleq T_1\} \cup \mathcal{C}') &= \mathit{unify}([X \mapsto T_1]\mathcal{C}') \circ [X \mapsto T_1] \quad \text{if } X \notin FV(T_1) \\
\mathit{unify}(\{T_1 \triangleq X\} \cup \mathcal{C}') &= \mathit{unify}([X \mapsto T_1]\mathcal{C}') \circ [X \mapsto T_1] \quad \text{if } X \notin FV(T_1) \\
\mathit{unify}(\{T_1 \to T_2 \triangleq S_1 \to S_2\} \cup \mathcal{C}') &= \mathit{unify}(\mathcal{C}' \cup \{T_1 \triangleq S_1, T_2 \triangleq S_2\}) \\
\mathit{unify}(\_) &= \text{raise failure}
\end{aligned}
$$

For the properties' discussion are useful the following concepts:

- A type substitution σ is *less specific* (or *more general*) than a type substitution ρ, written as $\sigma \sqsubseteq \rho$, if $\rho = \gamma \circ \sigma$, for some type substitution γ
- A *principal unifier* for a constraint set $\mathcal{C}$ is a type substitution σ such that $\sigma \bowtie \mathcal{C}$ and $\forall \rho \bowtie \mathcal{C} : \sigma \sqsubseteq \rho$

Example

Let $\rho = \{S \mapsto Nat \to Nat, Z \mapsto Nat\}$ and $\sigma = \{S \mapsto Z \to Z\}$. We have $\sigma \sqsubseteq \rho$ because $\exists \gamma = \{Z \mapsto Nat\}$ such that $\rho = \gamma \circ \sigma$

Unification algorithm

By pattern matching on the structure of constraint set $\mathcal{C}$ given as argument:

$$
\begin{aligned}
\mathit{unify}(\emptyset) &= [\,] \\
\mathit{unify}(\{T_1 \triangleq T_2\} \cup \mathcal{C}') &= \mathit{unify}(\mathcal{C}') \quad \text{if } T_1 = T_2 \\
\mathit{unify}(\{X \triangleq T_1\} \cup \mathcal{C}') &= \mathit{unify}([X \mapsto T_1]\mathcal{C}') \circ [X \mapsto T_1] \quad \text{if } X \notin FV(T_1) \\
\mathit{unify}(\{T_1 \triangleq X\} \cup \mathcal{C}') &= \mathit{unify}([X \mapsto T_1]\mathcal{C}') \circ [X \mapsto T_1] \quad \text{if } X \notin FV(T_1) \\
\mathit{unify}(\{T_1 \to T_2 \triangleq S_1 \to S_2\} \cup \mathcal{C}') &= \mathit{unify}(\mathcal{C}' \cup \{T_1 \triangleq S_1, T_2 \triangleq S_2\}) \\
\mathit{unify}(\_) &= \text{raise failure}
\end{aligned}
$$

For the properties' discussion are useful the following concepts:

- A type substitution σ is *less specific* (or *more general*) than a type substitution ρ, written as $\sigma \sqsubseteq \rho$, if $\rho = \gamma \circ \sigma$, for some type substitution γ
- A *principal unifier* for a constraint set $\mathcal{C}$ is a type substitution σ such that $\sigma \bowtie \mathcal{C}$ and $\forall \rho \bowtie \mathcal{C} : \sigma \sqsubseteq \rho$

Example

Let $\rho = \{S \mapsto \mathit{Nat} \to \mathit{Nat}, Z \mapsto \mathit{Nat}\}$ and $\sigma = \{S \mapsto Z \to Z\}$. We have $\sigma \sqsubseteq \rho$ because $\exists \gamma = \{Z \mapsto \mathit{Nat}\}$ such that $\rho = \gamma \circ \sigma$

We return to the extended example: we built the constraint set $\mathcal{C} = \{B \triangleq C \to A, X \triangleq Z \to B, Y \triangleq Z \to C\}$, now we apply the unification in order to build a type substitution σ (hoping such that $\sigma \bowtie \mathcal{C}$)

$$\mathit{unify}(\{B \triangleq C \to A, X \triangleq Z \to B, Y \triangleq Z \to C\})$$

We return to the extended example: we built the constraint set $\mathcal{C} = \{B \triangleq C \to A, X \triangleq Z \to B, Y \triangleq Z \to C\}$, now we apply the unification in order to build a type substitution σ (hoping such that $\sigma \bowtie \mathcal{C}$)

$$\mathit{unify}(\{B \triangleq C \to A\} \cup \{X \triangleq Z \to B, Y \triangleq Z \to C\})$$

We return to the extended example: we built the constraint set $\mathcal{C} = \{B \triangleq C \to A, X \triangleq Z \to B, Y \triangleq Z \to C\}$, now we apply the unification in order to build a type substitution σ (hoping such that $\sigma \bowtie \mathcal{C}$)

$$\mathit{unify}([B \mapsto C \to A]\{X \triangleq Z \to B, Y \triangleq Z \to C\}) \circ [B \mapsto C \to A]$$

We return to the extended example: we built the constraint set $\mathcal{C} = \{B \triangleq C \to A, X \triangleq Z \to B, Y \triangleq Z \to C\}$, now we apply the unification in order to build a type substitution σ (hoping such that $\sigma \bowtie \mathcal{C}$)

$$\mathit{unify}(\{X \triangleq Z \to C \to A, Y \triangleq Z \to C\}) \circ [B \mapsto C \to A]$$

We return to the extended example: we built the constraint set $\mathcal{C} = \{B \triangleq C \to A, X \triangleq Z \to B, Y \triangleq Z \to C\}$, now we apply the unification in order to build a type substitution σ (hoping such that $\sigma \bowtie \mathcal{C}$)

$$\mathit{unify}(\{X \triangleq Z \to C \to A\} \cup \{Y \triangleq Z \to C\}) \circ [B \mapsto C \to A]$$

We return to the extended example: we built the constraint set $\mathcal{C} = \{B \triangleq C \to A, X \triangleq Z \to B, Y \triangleq Z \to C\}$, now we apply the unification in order to build a type substitution σ (hoping such that $\sigma \bowtie \mathcal{C}$)

$$\mathit{unify}([X \mapsto Z \to C \to A]\{Y \triangleq Z \to C\}) \circ [X \mapsto Z \to C \to A] \circ [B \mapsto C \to A]$$

We return to the extended example: we built the constraint set $\mathcal{C} = \{B \triangleq C \to A, X \triangleq Z \to B, Y \triangleq Z \to C\}$, now we apply the unification in order to build a type substitution σ (hoping such that $\sigma \bowtie \mathcal{C}$)

$$\mathit{unify}(\{Y \triangleq Z \to C\}) \circ [X \mapsto Z \to C \to A] \circ [B \mapsto C \to A]$$

We return to the extended example: we built the constraint set $\mathcal{C} = \{B \triangleq C \to A, X \triangleq Z \to B, Y \triangleq Z \to C\}$, now we apply the unification in order to build a type substitution σ (hoping such that $\sigma \bowtie \mathcal{C}$)

$$\mathit{unify}(\emptyset) \circ [Y \mapsto Z \to C] \circ [X \mapsto Z \to C \to A] \circ [B \mapsto C \to A]$$

We return to the extended example: we built the constraint set $\mathcal{C} = \{B \triangleq C \to A, X \triangleq Z \to B, Y \triangleq Z \to C\}$, now we apply the unification in order to build a type substitution σ (hoping such that $\sigma \bowtie \mathcal{C}$)

$$[] \circ [Y \mapsto Z \to C] \circ [X \mapsto Z \to C \to A] \circ [B \mapsto C \to A]$$

Contents

Theorem

unify($\mathcal{C}$) *halts, either by failing or by returning a type substitution* σ

Call *degree* of a constraint set $\mathcal{C}$ a pair (m, n) where m is the number of distinct type variables and n is the number of distinct types in $\mathcal{C}$

Proof. By complete induction on the degree of $\mathcal{C}$.

Base $(0, 0) \equiv \mathcal{C} = \emptyset$, *unify*$(\emptyset) = [\,]$ applies, which returns the empty substitution
Induction HP assume the theorem holds $\forall (s, t) : (s, t) < (m, n)$
Induction Step proceed by pattern matching on the structure of $\mathcal{C}$:

- *unify*$(\{T_1 \triangleq T_2\} \cup \mathcal{C}')$ if $T_1 = T_2$ does what *unify*$(\mathcal{C}')$ does, but $\mathcal{C}'$ has a type less, so has a smaller degree than $\{T_1 \triangleq T_2\} \cup \mathcal{C}'$, hence induction hp applies
- *unify*$(\{X \triangleq T_1\} \cup \mathcal{C}')$ compose what *unify*$([X \mapsto T_1]\mathcal{C}')$ returns with $[X \mapsto T_1]$, but $[X \mapsto T_1]\mathcal{C}'$ has a variable less, so has a smaller degree than $\{X \triangleq T_1\} \cup \mathcal{C}'$, hence induction hp applies and composition always halt
- *unify*$(\{T_1 \to T_2 \triangleq S_1 \to S_2\} \cup \mathcal{C}')$ does what *unify*$(\mathcal{C}' \cup \{T_1 \triangleq S_1, T_2 \triangleq S_2\})$ does, but $\mathcal{C}' \cup \{T_1 \triangleq S_1, T_2 \triangleq S_2\}$ has two arrow types less, so has a smaller degree than $\{T_1 \to T_2 \triangleq S_1 \to S_2\} \cup \mathcal{C}'$, hence induction hp applies
- if $\mathcal{C}$ has a different structure from the above ones, *unify*$(\mathcal{C})$ fails to return a type substitution

All structures of $\mathcal{C}$ have been covered, hence *unify*$(\mathcal{C})$ either produces a type substitution σ or fails to do so, this complete the induction. □

Theorem

$unify(\mathcal{C}) = \sigma \rightarrow \sigma \bowtie \mathcal{C}$

Proof. By structural induction on $\mathcal{C}$.

Base $\mathcal{C} = \emptyset$, $unify(\emptyset) = [\,]$ applies, $\forall \sigma : \sigma \bowtie \emptyset$ (no equation needs to be unified)

Induction HP assume the theorem holds for a generic $\mathcal{C}'$ and prove for a new set $\{T_1 \triangleq T_2\} \cup \mathcal{C}'$ where $T_1, T_2 \in T$ (possibly containing type variables)

Induction Step proceed by pattern matching on the structure of the new set $\{T_1 \triangleq T_2\} \cup \mathcal{C}'$:

- $unify(\{T_1 \triangleq T_2\} \cup \mathcal{C}')$ if $T_1 = T_2$ does what $unify(\mathcal{C}')$ does: by induction hp $unify(\mathcal{C}') = \sigma \rightarrow \sigma \bowtie \mathcal{C}'$, but the added equation is already an identity, hence $\sigma \bowtie \{T_1 \triangleq T_2\} \cup \mathcal{C}'$ if $T_1 = T_2$
- Let $X \in \mathcal{A}$: $unify(\{X \triangleq T_1\} \cup \mathcal{C}')$ compose what $unify([X \mapsto T_1]\mathcal{C}')$ returns with $[X \mapsto T_1]$: by induction hp, $unify([X \mapsto T_1]\mathcal{C}') = \sigma \rightarrow \sigma \bowtie [X \mapsto T_1]\mathcal{C}'$, hence the composition ρ, defined as $\sigma \circ [X \mapsto T_1]$, is such that $\rho \bowtie \{X \triangleq T_1\} \cup \mathcal{C}'$
- $unify(\{T_1 \rightarrow T_2 \triangleq S_1 \rightarrow S_2\} \cup \mathcal{C}')$ does what $unify(\mathcal{C}' \cup \{T_1 \triangleq S_1, T_2 \triangleq S_2\})$ does: by induction hp $unify(\mathcal{C}' \cup \{T_1 \triangleq S_1, T_2 \triangleq S_2\}) = \sigma \rightarrow \sigma \bowtie \mathcal{C}' \cup \{T_1 \triangleq S_1, T_2 \triangleq S_2\}$. To unify $T_1 \rightarrow T_2 \triangleq S_1 \rightarrow S_2$ use def of substitution application: $\sigma(T_1 \rightarrow T_2) = \sigma T_1 \rightarrow \sigma T_2$ and $\sigma(S_1 \rightarrow S_2) = \sigma S_1 \rightarrow \sigma S_2$. Since $\sigma \bowtie \mathcal{C}' \cup \{T_1 \triangleq S_1, T_2 \triangleq S_2\}$ the arrow types $\sigma T_1 \rightarrow \sigma T_2$ and $\sigma S_1 \rightarrow \sigma S_2$ are really the same, hence $\sigma \bowtie \{T_1 \rightarrow T_2 \triangleq S_1 \rightarrow S_2\} \cup \mathcal{C}'$
- if $\{T_1 \triangleq T_2\} \cup \mathcal{C}'$ has a different structure from the above ones, $unify(\{T_1 \triangleq T_2\} \cup \mathcal{C}')$ fails to return a type substitution, hence the theorem is vacaously true by “ex falso quod libet”

All structures of $\mathcal{C}$ have been covered, hence $unify(\mathcal{C}) = \sigma \rightarrow \sigma \bowtie \mathcal{C}$, this complete the induction. □

Theorem

$\rho \bowtie \mathcal{C} \rightarrow \mathit{unify}(\mathcal{C}) = \sigma \wedge \sigma \sqsubseteq \rho$

Proof. By structural induction on $\mathcal{C}$.

Base $\mathcal{C} = \emptyset$, Get any ρ such that $\rho \bowtie \emptyset$: $\mathit{unify}(\emptyset) = [\,]$ applies, since $\forall \gamma : \gamma = \gamma \circ [\,]$, $[\,] \sqsubseteq \rho$ holds as required
Induction HP assume the theorem holds for a generic $\mathcal{C}'$ and prove for a new set $\{T_1 \triangleq T_2\} \cup \mathcal{C}'$ where $T_1, T_2 \in T$ (possibly containing type variables)
Induction Step proceed by pattern matching on the structure of the new set $\{T_1 \triangleq T_2\} \cup \mathcal{C}'$:

- $\mathit{unify}(\{T_1 \triangleq T_2\} \cup \mathcal{C}')$ if $T_1 = T_2$ does what $\mathit{unify}(\mathcal{C}')$ does: let $\rho \bowtie \{T_1 \triangleq T_2\} \cup \mathcal{C}'$, hence $\rho \bowtie \mathcal{C}'$. By induction hp $\rho \bowtie \mathcal{C}' \rightarrow \mathit{unify}(\mathcal{C}') = \sigma \wedge \sigma \sqsubseteq \rho$, but the added equation is already an identity, hence $\sigma \bowtie \{T_1 \triangleq T_2\} \cup \mathcal{C}'$ if $T_1 = T_2$ and $\sigma \sqsubseteq \rho$
- Let $X \in \mathcal{A}$: $\mathit{unify}(\{X \triangleq T_1\} \cup \mathcal{C}')$ compose what $\mathit{unify}([X \mapsto T_1]\mathcal{C}')$ returns with $[X \mapsto T_1]$: let $\rho \bowtie \{X \triangleq T_1\} \cup \mathcal{C}'$, since $\rho X = \rho T_1$ and $\rho \bowtie \mathcal{C}'$, follows that $\rho \bowtie [X \mapsto T_1]\mathcal{C}'$. By induction hp, $\mathit{unify}([X \mapsto T_1]\mathcal{C}') = \sigma \wedge \sigma \sqsubseteq \rho$ and by def of $\sqsubseteq$, holds $\rho = \gamma \circ \sigma$ for a γ. By rule application, $\mathit{unify}(\{X \triangleq T_1\} \cup \mathcal{C}') = \sigma \circ [X \mapsto T_1]$, so remain to prove $\sigma \circ [X \mapsto T_1] \sqsubseteq \rho$, by def of $\sqsubseteq$, $\rho = \gamma \circ (\sigma \circ [X \mapsto T_1])$ for a γ. Consider any $Y \in \mathcal{A}$: if $Y \neq X$ then $(\gamma \circ (\sigma \circ [X \mapsto T_1]))Y = (\gamma \circ \sigma)Y$ which holds by induction hp. Otherwise, if $Y = X$ then $(\gamma \circ (\sigma \circ [X \mapsto T_1]))Y = (\gamma \circ \sigma)T_1$ which holds because $\rho \bowtie X \triangleq T_1$
- observe that $\rho \bowtie \{T_1 \rightarrow T_2 \triangleq S_1 \rightarrow S_2\} \cup \mathcal{C}')$ if and only if $\rho \bowtie \{T_1 \triangleq S_1, T_2 \triangleq S_2\} \cup \mathcal{C}'$: hence we apply the argument recursively on $\{T_1 \triangleq S_1, T_2 \triangleq S_2\} \cup \mathcal{C}'$
- if $\{T_1 \triangleq T_2\} \cup \mathcal{C}'$ has a different structure from the above ones, it can only left two cases such that $\nexists \rho : \rho \bowtie \mathcal{C}$: T_1 is a concrete type and T_2 is an arrow type (really "incompossible"), or let $X \in \mathcal{A}$ in $T_1 = X$ and X appears in T_2 (by absurd: assume $\rho X = \rho T_2$, but ρT_2 is strictly larger that ρX by choice of T_2).

□

Contents

Let Γ be a context and t be a term containing free variables.

Definition of *principal solution* for $(\Gamma, t, S, \mathcal{C})$

A *principal solution* for $(\Gamma, t, S, \mathcal{C})$ is a solution (σ, T_1) such that for any other solution (ρ, T_2) for $(\Gamma, t, S, \mathcal{C})$ we have $\sigma \sqsubseteq \rho$.

Theorem

if $(\Gamma, t, S, \mathcal{C})$ has a solution, then it has a principal one too. The unification algorithm can be used to decide if $(\Gamma, t, S, \mathcal{C})$ has solutions and if it is the case, it compute the principal one.

Let Γ be a context and t be a term containing free variables.

Definition of *principal solution* for $(\Gamma, t, S, \mathcal{C})$

A *principal solution* for $(\Gamma, t, S, \mathcal{C})$ is a solution (σ, T_1) such that for any other solution (ρ, T_2) for $(\Gamma, t, S, \mathcal{C})$ we have $\sigma \sqsubseteq \rho$.

Theorem

if $(\Gamma, t, S, \mathcal{C})$ has a solution, then it has a principal one too. The unification algorithm can be used to decide if $(\Gamma, t, S, \mathcal{C})$ has solutions and if it is the case, it compute the principal one.

We finish our extended example to show a principal solution. Using unification algorithm we found the type substitution:

$$\sigma = [] \circ [Y \mapsto Z \to C] \circ [X \mapsto Z \to C \to A] \circ [B \mapsto C \to A]$$

And, using the deduction for constraint typing relation, we found that:

$$\lambda x : X.\lambda y : Y.\lambda z : Z.((x\ z)\ (y\ z)) : X \to Y \to Z \to A$$

In order to find a principal solution we apply σ to the abstract type above to have a *concrete* type for the term (possibly containing free variables, heart of parametric polymorphism, remember?):

$$\sigma(X \to Y \to Z \to A)$$

We finish our extended example to show a principal solution. Using unification algorithm we found the type substitution:

$$\sigma = [] \circ [Y \mapsto Z \to C] \circ [X \mapsto Z \to C \to A] \circ [B \mapsto C \to A]$$

And, using the deduction for constraint typing relation, we found that:

$$\lambda x : X.\lambda y : Y.\lambda z : Z.((x\ z)\ (y\ z)) : X \to Y \to Z \to A$$

In order to find a principal solution we apply σ to the abstract type above to have a *concrete* type for the term (possibly containing free variables, heart of parametric polymorphism, remember?):

$$\sigma X \to \sigma(Y \to Z \to A)$$

We finish our extended example to show a principal solution. Using unification algorithm we found the type substitution:

$$\sigma = [] \circ [Y \mapsto Z \to C] \circ [X \mapsto Z \to C \to A] \circ [B \mapsto C \to A]$$

And, using the deduction for constraint typing relation, we found that:

$$\lambda x : X.\lambda y : Y.\lambda z : Z.((x\ z)\ (y\ z)) : X \to Y \to Z \to A$$

In order to find a principal solution we apply σ to the abstract type above to have a *concrete* type for the term (possibly containing free variables, heart of parametric polymorphism, remember?):

$$\sigma X \to \sigma Y \to \sigma(Z \to A)$$

We finish our extended example to show a principal solution. Using unification algorithm we found the type substitution:

$$\sigma = [] \circ [Y \mapsto Z \to C] \circ [X \mapsto Z \to C \to A] \circ [B \mapsto C \to A]$$

And, using the deduction for constraint typing relation, we found that:

$$\lambda x : X.\lambda y : Y.\lambda z : Z.((x\ z)\ (y\ z)) : X \to Y \to Z \to A$$

In order to find a principal solution we apply σ to the abstract type above to have a *concrete* type for the term (possibly containing free variables, heart of parametric polymorphism, remember?):

$$\sigma X \to \sigma Y \to \sigma Z \to \sigma A$$

We finish our extended example to show a principal solution. Using unification algorithm we found the type substitution:

$$\sigma = [] \circ [Y \mapsto Z \to C] \circ [X \mapsto Z \to C \to A] \circ [B \mapsto C \to A]$$

And, using the deduction for constraint typing relation, we found that:

$$\lambda x : X.\lambda y : Y.\lambda z : Z.((x\ z)\ (y\ z)) : X \to Y \to Z \to A$$

In order to find a principal solution we apply σ to the abstract type above to have a *concrete* type for the term (possibly containing free variables, heart of parametric polymorphism, remember?):

$$(Z \to C \to A) \to \sigma Y \to \sigma Z \to \sigma A$$

We finish our extended example to show a principal solution. Using unification algorithm we found the type substitution:

$$\sigma = [] \circ [Y \mapsto Z \to C] \circ [X \mapsto Z \to C \to A] \circ [B \mapsto C \to A]$$

And, using the deduction for constraint typing relation, we found that:

$$\lambda x : X.\lambda y : Y.\lambda z : Z.((x\ z)\ (y\ z)) : X \to Y \to Z \to A$$

In order to find a principal solution we apply σ to the abstract type above to have a *concrete* type for the term (possibly containing free variables, heart of parametric polymorphism, remember?):

$$(Z \to C \to A) \to (Z \to C) \to \sigma Z \to \sigma A$$

We finish our extended example to show a principal solution. Using unification algorithm we found the type substitution:

$$\sigma = [] \circ [Y \mapsto Z \to C] \circ [X \mapsto Z \to C \to A] \circ [B \mapsto C \to A]$$

And, using the deduction for constraint typing relation, we found that:

$$\lambda x : X.\lambda y : Y.\lambda z : Z.((x\ z)\ (y\ z)) : X \to Y \to Z \to A$$

In order to find a principal solution we apply σ to the abstract type above to have a *concrete* type for the term (possibly containing free variables, heart of parametric polymorphism, remember?):

$$(Z \to C \to A) \to (Z \to C) \to Z \to \sigma A$$

We finish our extended example to show a principal solution. Using unification algorithm we found the type substitution:

$$\sigma = [] \circ [Y \mapsto Z \to C] \circ [X \mapsto Z \to C \to A] \circ [B \mapsto C \to A]$$

And, using the deduction for constraint typing relation, we found that:

$$\lambda x : X.\lambda y : Y.\lambda z : Z.((x\ z)\ (y\ z)) : X \to Y \to Z \to A$$

In order to find a principal solution we apply σ to the abstract type above to have a *concrete* type for the term (possibly containing free variables, heart of parametric polymorphism, remember?):

$$(Z \to C \to A) \to (Z \to C) \to Z \to A$$

Hence $(\sigma, (Z \to C \to A) \to (Z \to C) \to Z \to A), \forall A, C, Z \in T$, is the principal solution for

$$(\emptyset, \lambda x : X.\lambda y : Y.\lambda z : Z.((x\ z)\ (y\ z)), X \to Y \to Z \to A, \{B \triangleq C \to A, X \triangleq Z \to B, Y \triangleq Z \to C\})$$

Contents

Grammar

Definizione

The set T of terms representing the lambda calculus language is the smallest set generated by the following rules:

$$\begin{aligned}
\text{Term} &\triangleq \text{AppTerm} \mid \text{“lambda” ID “:” Type “.” ATerm} \\
\text{AppTerm} &\triangleq \text{ATerm ATerm} \\
\text{ATerm} &\triangleq \text{“(” Term “)” } \mid \text{ ID} \\
\text{Type} &\triangleq \text{ArrowType} \\
\text{AType} &\triangleq \text{“(” Type “)” } \mid \text{ “Bool” } \mid \text{ “Nat” } \mid \text{ ID} \\
\text{ArrowType} &\triangleq \text{AType}(\rightarrow \text{ArrowType} \mid \text{AType})
\end{aligned}$$

where:

- Σ is our alphabet;
- ID $\in \Sigma \setminus \{$“lambda”, “:”, “.”, “(”, “)”, “Bool”, “Nat”$\}$.

Some observations

We make the following observations:

- we design the grammar to be processed by *ANTLR*, a $LL(k)$ top-down parser;
- the grammar seen during lectures has left recursion on $T - APP$ rule (good for LR bottom-up parser);
- our grammar maybe "naive" in the sense that allow to write some strange terms (ie: `x (lambda y:Y. x)`) but for our scope is enough;
- it allows to write some "obscure" terms (ie: `lambda x:X.(x x)`) which we desire to prove not safe.

Contents

Substitution definition - first attempt

Consider the abstraction application rule: $(\lambda x : T_{11}.t_{12})v_2 \rightarrow [x \mapsto v_2]t_{12}$

Definition of $[x \mapsto s]t$

The substitution of a term s for a variable named x in a term t, written as $[x \mapsto s]t$, is defined as follows:

$$
\begin{aligned}
[x \mapsto s]x &= s \\
[x \mapsto s]y &= y \quad \text{if } x \neq y \\
[x \mapsto s](\lambda y.t_1) &= \lambda y.[x \mapsto s]t_1 \\
[x \mapsto s](t_1\ t_2) &= ([x \mapsto s]t_1)\,([x \mapsto s]t_2)
\end{aligned}
$$

The names of bound variables do not matter

$$
\begin{aligned}
[x \mapsto (\lambda z.z\,w)](\lambda y.x) &= \lambda y.\lambda z.z\,w \quad \text{☺} \\
[x \mapsto y](\lambda x.x) &= \lambda x.y \quad \text{☹}
\end{aligned}
$$

We've not distinguished between *free* occurrences of x in a term t, which should get replaced during substitution, and *bound* ones, which shouldn't.

Substitution definition - first attempt

Consider the abstraction application rule: $(\lambda x : T_{11}.t_{12})v_2 \rightarrow [x \mapsto v_2]t_{12}$

Definition of $[x \mapsto s]t$

The substitution of a term s for a variable named x in a term t, written as $[x \mapsto s]t$, is defined as follows:

$$
\begin{aligned}
[x \mapsto s]x &= s \\
[x \mapsto s]y &= y \quad \text{if } x \neq y \\
[x \mapsto s](\lambda y.t_1) &= \lambda y.[x \mapsto s]t_1 \\
[x \mapsto s](t_1\ t_2) &= ([x \mapsto s]t_1)\,([x \mapsto s]t_2)
\end{aligned}
$$

The names of bound variables do not matter

$$
\begin{aligned}
[x \mapsto (\lambda z.z\ w)](\lambda y.x) &= \lambda y.\lambda z.z\ w \quad \text{☺} \\
[x \mapsto y](\lambda x.x) &= \lambda x.y \quad \text{☹}
\end{aligned}
$$

We've not distinguished between *free* occurrences of x in a term t, which should get replaced during substitution, and *bound* ones, which shouldn't.

Substitution definition - first attempt

Consider the abstraction application rule: $(\lambda x : T_{11}.t_{12})v_2 \rightarrow [x \mapsto v_2]t_{12}$

Definition of $[x \mapsto s]t$

The substitution of a term s for a variable named x in a term t, written as $[x \mapsto s]t$, is defined as follows:

$$
\begin{aligned}
[x \mapsto s]x &= s \\
[x \mapsto s]y &= y \quad \text{if } x \neq y \\
[x \mapsto s](\lambda y.t_1) &= \lambda y.[x \mapsto s]t_1 \\
[x \mapsto s](t_1\ t_2) &= ([x \mapsto s]t_1)\,([x \mapsto s]t_2)
\end{aligned}
$$

The names of bound variables do not matter

$$
\begin{aligned}
[x \mapsto (\lambda z.z\ w)](\lambda y.x) &= \lambda y.\lambda z.z\ w \quad \text{☺} \\
[x \mapsto y](\lambda x.x) &= \lambda x.y \quad \text{☹}
\end{aligned}
$$

We've not distinguished between *free* occurrences of x in a term t, which should get replaced during substitution, and *bound* ones, which shouldn't.

Substitution definition - first attempt

Consider the abstraction application rule: $(\lambda x : T_{11}.t_{12})v_2 \rightarrow [x \mapsto v_2]t_{12}$

Definition of $[x \mapsto s]t$

The substitution of a term s for a variable named x in a term t, written as $[x \mapsto s]t$, is defined as follows:

$$
\begin{aligned}
[x \mapsto s]x &= s \\
[x \mapsto s]y &= y \quad \text{if } x \neq y \\
[x \mapsto s](\lambda y.t_1) &= \lambda y.[x \mapsto s]t_1 \\
[x \mapsto s](t_1\ t_2) &= ([x \mapsto s]t_1)\,([x \mapsto s]t_2)
\end{aligned}
$$

The names of bound variables do not matter

$$
\begin{aligned}
[x \mapsto (\lambda z.z\ w)](\lambda y.x) &= \lambda y.\lambda z.z\ w \quad \text{☺} \\
[x \mapsto y](\lambda x.x) &= \lambda x.y \quad \text{☹}
\end{aligned}
$$

We've not distinguished between *free* occurrences of x in a term t, which should get replaced during substitution, and *bound* ones, which shouldn't.

Substitution definition - second attempt

Consider the abstraction application rule: $(\lambda x : T_{11}.t_{12})v_2 \rightarrow [x \mapsto v_2]t_{12}$

Definition of $[x \mapsto s]t$

The substitution of a term s for a variable named x in a term t, written as $[x \mapsto s]t$, is defined as follows:

$$
\begin{aligned}
[x \mapsto s]x &= s \\
[x \mapsto s]y &= y \quad \text{if } x \neq y \\
[x \mapsto s](\lambda y.t_1) &= \lambda y.t_1 \quad \text{if } x = y \\
[x \mapsto s](\lambda y.t_1) &= \lambda y.[x \mapsto s]t_1 \quad \text{if } x \neq y \\
[x \mapsto s](t_1\ t_2) &= ([x \mapsto s]t_1)\,([x \mapsto s]t_2)
\end{aligned}
$$

Variable capture

$$
\begin{aligned}
[x \mapsto y](\lambda x.x) &= \lambda x.x \quad \text{☺} \\
[x \mapsto z](\lambda z.x) &= \lambda z.z \quad \text{☹}
\end{aligned}
$$

To avoid it, we need to make sure that the bound variable names of t are kept distinct from the free variables of s.

Substitution definition - second attempt

Consider the abstraction application rule: $(\lambda x : T_{11}.t_{12})v_2 \rightarrow [x \mapsto v_2]t_{12}$

Definition of $[x \mapsto s]t$

The substitution of a term s for a variable named x in a term t, written as $[x \mapsto s]t$, is defined as follows:

$$
\begin{aligned}
[x \mapsto s]x &= s \\
[x \mapsto s]y &= y \quad \text{if } x \neq y \\
[x \mapsto s](\lambda y.t_1) &= \lambda y.t_1 \quad \text{if } x = y \\
[x \mapsto s](\lambda y.t_1) &= \lambda y.[x \mapsto s]t_1 \quad \text{if } x \neq y \\
[x \mapsto s](t_1\ t_2) &= ([x \mapsto s]t_1)\,([x \mapsto s]t_2)
\end{aligned}
$$

Variable capture

$$
\begin{aligned}
[x \mapsto y](\lambda x.x) &= \lambda x.x \quad \text{☺} \\
[x \mapsto z](\lambda z.x) &= \lambda z.z \quad \text{☹}
\end{aligned}
$$

To avoid it, we need to make sure that the bound variable names of t are kept distinct from the free variables of s.

Substitution definition - second attempt

Consider the abstraction application rule: $(\lambda x : T_{11}.t_{12})v_2 \rightarrow [x \mapsto v_2]t_{12}$

Definition of $[x \mapsto s]t$

The substitution of a term s for a variable named x in a term t, written as $[x \mapsto s]t$, is defined as follows:

$$
\begin{aligned}
[x \mapsto s]x &= s \\
[x \mapsto s]y &= y \quad \text{if } x \neq y \\
[x \mapsto s](\lambda y.t_1) &= \lambda y.t_1 \quad \text{if } x = y \\
[x \mapsto s](\lambda y.t_1) &= \lambda y.[x \mapsto s]t_1 \quad \text{if } x \neq y \\
[x \mapsto s](t_1\ t_2) &= ([x \mapsto s]t_1)\,([x \mapsto s]t_2)
\end{aligned}
$$

Variable capture

$$
\begin{aligned}
[x \mapsto y](\lambda x.x) &= \lambda x.x \quad \text{☺} \\
[x \mapsto z](\lambda z.x) &= \lambda z.z \quad \text{☹}
\end{aligned}
$$

To avoid it, we need to make sure that the bound variable names of t are kept distinct from the free variables of s.

Substitution definition - second attempt

Consider the abstraction application rule: $(\lambda x : T_{11}.t_{12})v_2 \rightarrow [x \mapsto v_2]t_{12}$

Definition of $[x \mapsto s]t$

The substitution of a term s for a variable named x in a term t, written as $[x \mapsto s]t$, is defined as follows:

$$
\begin{aligned}
[x \mapsto s]x &= s \\
[x \mapsto s]y &= y \quad \text{if } x \neq y \\
[x \mapsto s](\lambda y.t_1) &= \lambda y.t_1 \quad \text{if } x = y \\
[x \mapsto s](\lambda y.t_1) &= \lambda y.[x \mapsto s]t_1 \quad \text{if } x \neq y \\
[x \mapsto s](t_1\ t_2) &= ([x \mapsto s]t_1)\,([x \mapsto s]t_2)
\end{aligned}
$$

Variable capture

$$
\begin{aligned}
[x \mapsto y](\lambda x.x) &= \lambda x.x \quad \text{☺} \\
[x \mapsto z](\lambda z.x) &= \lambda z.z \quad \text{☹}
\end{aligned}
$$

To avoid it, we need to make sure that the bound variable names of t are kept distinct from the free variables of s.

Substitution definition - third attempt

Consider the abstraction application rule: $(\lambda x : T_{11}.t_{12})v_2 \rightarrow [x \mapsto v_2]t_{12}$

Definition of $[x \mapsto s]t$

The substitution of a term s for a variable named x in a term t, written as $[x \mapsto s]t$, is defined as follows:

$$
\begin{aligned}
[x \mapsto s]x &= s \\
[x \mapsto s]y &= y \quad \text{if } x \neq y \\
[x \mapsto s](\lambda y.t_1) &= \lambda y.t_1 \quad \text{if } x = y \\
[x \mapsto s](\lambda y.t_1) &= \lambda y.[x \mapsto s]t_1 \quad \text{if } x \neq y \wedge y \notin FV(s) \\
[x \mapsto s](t_1\ t_2) &= ([x \mapsto s]t_1)\,([x \mapsto s]t_2)
\end{aligned}
$$

$[x \mapsto s]t$ isn't a total function

$$
\begin{aligned}
[x \mapsto z](\lambda z.x) &\neq \lambda z.z \quad \text{☺} \\
[x \mapsto z](\lambda z.x) &\quad \text{yields no term at all!} \quad \text{☹}
\end{aligned}
$$

A common fix is to work with terms *up to renaming of bound variables*.

Substitution definition - third attempt

Consider the abstraction application rule: $(\lambda x : T_{11}.t_{12})v_2 \rightarrow [x \mapsto v_2]t_{12}$

Definition of $[x \mapsto s]t$

The substitution of a term s for a variable named x in a term t, written as $[x \mapsto s]t$, is defined as follows:

$$
\begin{aligned}
[x \mapsto s]x &= s \\
[x \mapsto s]y &= y \quad \text{if } x \neq y \\
[x \mapsto s](\lambda y.t_1) &= \lambda y.t_1 \quad \text{if } x = y \\
[x \mapsto s](\lambda y.t_1) &= \lambda y.[x \mapsto s]t_1 \quad \text{if } x \neq y \wedge y \notin FV(s) \\
[x \mapsto s](t_1\ t_2) &= ([x \mapsto s]t_1)\,([x \mapsto s]t_2)
\end{aligned}
$$

$[x \mapsto s]t$ isn't a total function

$[x \mapsto z](\lambda z.x) \neq \lambda z.z$ ☺

$[x \mapsto z](\lambda z.x)$ yields no term at all! ☹

A common fix is to work with terms *up to renaming of bound variables*.

Substitution definition - third attempt

Consider the abstraction application rule: $(\lambda x : T_{11}.t_{12})v_2 \rightarrow [x \mapsto v_2]t_{12}$

Definition of $[x \mapsto s]t$

The substitution of a term s for a variable named x in a term t, written as $[x \mapsto s]t$, is defined as follows:

$$
\begin{aligned}
[x \mapsto s]x &= s \\
[x \mapsto s]y &= y \quad \text{if } x \neq y \\
[x \mapsto s](\lambda y.t_1) &= \lambda y.t_1 \quad \text{if } x = y \\
[x \mapsto s](\lambda y.t_1) &= \lambda y.[x \mapsto s]t_1 \quad \text{if } x \neq y \wedge y \notin FV(s) \\
[x \mapsto s](t_1\ t_2) &= ([x \mapsto s]t_1)\,([x \mapsto s]t_2)
\end{aligned}
$$

$[x \mapsto s]t$ isn't a total function

$$
\begin{aligned}
&[x \mapsto z](\lambda z.x) \neq \lambda z.z \quad \text{☺} \\
&[x \mapsto z](\lambda z.x) \quad \text{yields no term at all!} \quad \text{☹}
\end{aligned}
$$

A common fix is to work with terms *up to renaming of bound variables*.

Substitution definition - third attempt

Consider the abstraction application rule: $(\lambda x : T_{11}.t_{12})v_2 \rightarrow [x \mapsto v_2]t_{12}$

Definition of $[x \mapsto s]t$

The substitution of a term s for a variable named x in a term t, written as $[x \mapsto s]t$, is defined as follows:

$$
\begin{aligned}
[x \mapsto s]x &= s \\
[x \mapsto s]y &= y \quad \text{if } x \neq y \\
[x \mapsto s](\lambda y.t_1) &= \lambda y.t_1 \quad \text{if } x = y \\
[x \mapsto s](\lambda y.t_1) &= \lambda y.[x \mapsto s]t_1 \quad \text{if } x \neq y \wedge y \notin FV(s) \\
[x \mapsto s](t_1\ t_2) &= ([x \mapsto s]t_1)\,([x \mapsto s]t_2)
\end{aligned}
$$

$[x \mapsto s]t$ isn't a total function

$$
\begin{aligned}
&[x \mapsto z](\lambda z.x) \neq \lambda z.z \quad \text{☺} \\
&[x \mapsto z](\lambda z.x) \quad \text{yields no term at all!} \quad \text{☹}
\end{aligned}
$$

A common fix is to work with terms *up to renaming of bound variables*.

Substitution definition - final attempt

Consider the abstraction application rule: $(\lambda x : T_{11}.t_{12})v_2 \rightarrow [x \mapsto v_2]t_{12}$

Observation

Terms that differ only in the names of bound variables are interchangeable in all contexts.

Any λ-bound variable can be changed to another name (changing the body consistently too) at any point where this is convenient.

Example

$[x \mapsto y\ z](\lambda y.x\ y)$ renames abstraction to $(\lambda w.x\ w)$

$[x \mapsto y\ z](\lambda w.x\ w) = \lambda w.y\ z\ w$ ☺

This observation makes $[x \mapsto s]t$ a total function: whenever we have to apply it to arguments for which it is undefined, we can rename the λ abstraction bound variable in order to satisfy side conditions.

Substitution definition - final attempt

Consider the abstraction application rule: $(\lambda x : T_{11}.t_{12})v_2 \rightarrow [x \mapsto v_2]t_{12}$

Observation

Terms that differ only in the names of bound variables are interchangeable in all contexts.

Any λ-bound variable can be changed to another name (changing the body consistently too) at any point where this is convenient.

Example

$[x \mapsto y\ z](\lambda y.x\ y)$ renames abstraction to $(\lambda w.x\ w)$

$[x \mapsto y\ z](\lambda w.x\ w) = \lambda w.y\ z\ w$ ☺

This observation makes $[x \mapsto s]t$ a total function: whenever we have to apply it to arguments for which it is undefined, we can rename the λ abstraction bound variable in order to satisfy side conditions.

Substitution definition - final attempt

Consider the abstraction application rule: $(\lambda x : T_{11}.t_{12})v_2 \rightarrow [x \mapsto v_2]t_{12}$

Observation

Terms that differ only in the names of bound variables are interchangeable in all contexts.

Any λ-bound variable can be changed to another name (changing the body consistently too) at any point where this is convenient.

Example

$[x \mapsto y\ z](\lambda y.x\ y)$ renames abstraction to $(\lambda w.x\ w)$

$[x \mapsto y\ z](\lambda w.x\ w) = \lambda w.y\ z\ w$ ☺

This observation makes $[x \mapsto s]t$ a total function: whenever we have to apply it to arguments for which it is undefined, we can rename the λ abstraction bound variable in order to satisfy side conditions.

Substitution definition - final attempt

Consider the abstraction application rule: $(\lambda x : T_{11}.t_{12})v_2 \rightarrow [x \mapsto v_2]t_{12}$

Definition of $[x \mapsto s]t$

The substitution of a term s for a variable named x in a term t, written as $[x \mapsto s]t$, is defined as follows:

$$
\begin{aligned}
[x \mapsto s]x &= s \\
[x \mapsto s]y &= y \quad \text{if } x \neq y \\
[x \mapsto s](\lambda y.t_1) &= \lambda y.[x \mapsto s]t_1 \quad \text{if } x \neq y \wedge (y \notin FV(s) \text{ by renaming}) \\
[x \mapsto s](t_1\ t_2) &= ([x \mapsto s]t_1)\,([x \mapsto s]t_2)
\end{aligned}
$$

Contents

De Bruijn's idea

Up to now we've worked with terms *up to renaming of bound variables*. This does mean that an abstraction term is a *schema* term indeed, which could represent *infinite* terms.

$$\lambda x : T.t \text{ is a term} \leftrightarrow \forall x \notin FV(t)$$

- it is good for theoretical reasoning ☺
- do not supply a single representation of terms ☹

De Bruijn's idea

Variable occurrences *point* to their binders, rather than referring to them by name.

Using this idea we abstract from *variable names*, replacing names by natural numbers. Let $k \in \mathbb{N}$ then k represents *the variable bound by the k-th enclosing λ, counting from the most nested abstraction*.

Example

$$\lambda x.x \quad \text{transforms to} \quad \lambda.0$$
$$\lambda s.\lambda z.s\,(s\,z) \quad \text{transforms to} \quad \lambda.\lambda.1\,(1\,0)$$

De Bruijn's idea

Up to now we've worked with terms *up to renaming of bound variables*. This does mean that an abstraction term is a *schema* term indeed, which could represent *infinite* terms.

$$\lambda x : T.t \text{ is a term} \leftrightarrow \forall x \notin FV(t)$$

- it is good for theoretical reasoning ☺
- do not supply a single representation of terms ☹

De Bruijn's idea

Variable occurrences *point* to their binders, rather than referring to them by name.

Using this idea we abstract from *variable names*, replacing names by natural numbers. Let $k \in \mathbb{N}$ then k represents *the variable bound by the k-th enclosing* λ*, counting from the most nested abstraction*.

Example

$$\lambda x.x \quad \text{transforms to} \quad \lambda.0$$
$$\lambda s.\lambda z.s\,(s\,z) \quad \text{transforms to} \quad \lambda.\lambda.1\,(1\,0)$$

De Bruijn's idea

Up to now we've worked with terms *up to renaming of bound variables*. This does mean that an abstraction term is a *schema* term indeed, which could represent *infinite* terms.

$$\lambda x : T.t \text{ is a term} \leftrightarrow \forall x \notin FV(t)$$

- it is good for theoretical reasoning ☺
- do not supply a single representation of terms ☹

De Bruijn's idea

Variable occurrences *point* to their binders, rather than referring to them by name.

Using this idea we abstract from *variable names*, replacing names by natural numbers. Let $k \in \mathbb{N}$ then k represents *the variable bound by the k-th enclosing* λ*, counting from the most nested abstraction*.

Example

$$\begin{array}{rcl} \lambda x.x & \text{transforms to} & \lambda.0 \\ \lambda s.\lambda z.s\,(s\,z) & \text{transforms to} & \lambda.\lambda.1\,(1\,0) \end{array}$$

New definition of terms

The new definition of *nameless* terms is similar to the one given before, the only difference is that we need to track of how many free variables can appear in each term.

Definition

The set of terms is defined as the smallest family of sets $\{\mathcal{T}_0, \mathcal{T}_1, \ldots\}$ such that:

$$k \in \{0, \ldots, n-1\} \rightarrow k \in \mathcal{T}_n$$
$$t_1 \in \mathcal{T}_{n+1} \rightarrow \lambda.t \in \mathcal{T}_n$$
$$t_1 \in \mathcal{T}_n \wedge t_2 \in \mathcal{T}_n \rightarrow t_1\ t_2 \in \mathcal{T}_n$$

For all $j \in \mathcal{T}_n$ we say that j is a n-term and in j appear *at most n* different free variables (pay attention: this is not a constraint on the number of *occurrences* of the free variables, that is $\lambda.\lambda.(3\,(3\,(1\,(2\,(0\,3)))))\,((3\,1)\,3) \in \mathcal{T}_4$ where there are 6 free variables).

- two ordinary terms t_1, t_2 are equivalent modulo renaming of bound $\leftrightarrow$ t_1, t_2 have the same nameless representation ☺
- how to deal with terms containing free variables, ie $\lambda x.y\ x$?
  We know how far the binder of x is , but we doesn't for the binder of y...

Naming context

A naming context is an assignment of natural numbers to variables names, more formally:

$$\Gamma \subseteq \mathcal{V} \times \mathbb{N}$$

where $\mathcal{V}$ is the set of variables.

Example

Let $\Gamma = \{x \mapsto 4, y \mapsto 3, z \mapsto 2, a \mapsto 1, b \mapsto 0\}$ be a naming context. We have:

$$\begin{array}{rcll} x\,(y\,z) & \text{transforms to} & 4\,(3\,2) & \\ \lambda w.y\ w & \text{transforms to} & \lambda.4\,0 & (4?!) \\ \lambda w.\lambda a.x & \text{transforms to} & \lambda.\lambda.6 & (6?!) \end{array}$$

- two ordinary terms t_1, t_2 are equivalent modulo renaming of bound $\leftrightarrow$ t_1, t_2 have the same nameless representation ☺
- how to deal with terms containing free variables, ie $\lambda x.y\ x$?
  We know how far the binder of x is , but we doesn't for the binder of y...

Naming context

A naming context is an assignment of natural numbers to variables names, more formally:

$$\Gamma \subseteq \mathcal{V} \times \mathbb{N}$$

where $\mathcal{V}$ is the set of variables.

Example

Let $\Gamma = \{x \mapsto 4, y \mapsto 3, z \mapsto 2, a \mapsto 1, b \mapsto 0\}$ be a naming context. We have:

$$\begin{array}{rll} x\,(y\,z) & \text{transforms to} & 4\,(3\,2) \\ \lambda w.y\ w & \text{transforms to} & \lambda.4\,0 \quad (4?!) \\ \lambda w.\lambda a.x & \text{transforms to} & \lambda.\lambda.6 \quad (6?!) \end{array}$$

- two ordinary terms t_1, t_2 are equivalent modulo renaming of bound $\leftrightarrow$ t_1, t_2 have the same nameless representation ☺
- how to deal with terms containing free variables, ie $\lambda x.y\ x$?
  We know how far the binder of x is , but we doesn't for the binder of y...

Naming context

A naming context is an assignment of natural numbers to variables names, more formally:

$$\Gamma \subseteq \mathcal{V} \times \mathbb{N}$$

where $\mathcal{V}$ is the set of variables.

Example

Let $\Gamma = \{x \mapsto 4, y \mapsto 3, z \mapsto 2, a \mapsto 1, b \mapsto 0\}$ be a naming context. We have:

$$\begin{array}{rlll} x\,(y\,z) & \text{transforms to} & 4\,(3\,2) & \\ \lambda w.y\ w & \text{transforms to} & \lambda.4\,0 & (4?!) \\ \lambda w.\lambda a.x & \text{transforms to} & \lambda.\lambda.6 & (6?!) \end{array}$$

Shifting and substitution

Consider *again* the rule: $(\lambda x.t_{12})v_2 \rightarrow [x \mapsto v_2]t_{12}$ where $t_{12} = \lambda y.t, x \neq y$.

When we can apply the substitution $[x \mapsto s]$ to an abstraction $\lambda.t$ the context in which the substitution is happening introduces a λ, in other word introduces a variable.

Shifting

In order to remain consistent within s, we have to increment the indices of free variables appearing in s by one, since we're introducing a λ binder between the free variables in s and their binders. We call this increment operation, or *shifting*, with $\uparrow^1(s)$.

For the same reason that the abstraction $\lambda.t$ introduces a variable, we've to modify the definition of $[x \mapsto s]$, paying attention to increment x by one too because x isn't the variable bound by the context $\lambda.t$.

Shifting and substitution

Consider *again* the rule: $(\lambda x.t_{12})v_2 \rightarrow [x \mapsto v_2]t_{12}$ where $t_{12} = \lambda y.t, x \neq y$.

When we can apply the substitution $[x \mapsto s]$ to an abstraction $\lambda.t$ the context in which the substitution is happening introduces a λ, in other word introduces a variable.

Shifting

In order to remain consistent within s, we have to increment the indices of free variables appearing in s by one, since we're introducing a λ binder between the free variables in s and their binders. We call this increment operation, or *shifting*, with $\uparrow^1(s)$.

For the same reason that the abstraction $\lambda.t$ introduces a variable, we've to modify the definition of $[x \mapsto s]$, paying attention to increment x by one too because x isn't the variable bound by the context $\lambda.t$.

Shifting and substitution

Consider *again* the rule: $(\lambda x.t_{12})v_2 \rightarrow [x \mapsto v_2]t_{12}$ where $t_{12} = \lambda y.t, x \neq y$.

When we can apply the substitution $[x \mapsto s]$ to an abstraction $\lambda.t$ the context in which the substitution is happening introduces a λ, in other word introduces a variable.

Shifting

In order to remain consistent within s, we have to increment the indices of free variables appearing in s by one, since we're introducing a λ binder between the free variables in s and their binders. We call this increment operation, or *shifting*, with $\uparrow^1(s)$.

For the same reason that the abstraction $\lambda.t$ introduces a variable, we've to modify the definition of $[x \mapsto s]$, paying attention to increment x by one too because x isn't the variable bound by the context $\lambda.t$.

Formal Substitution definition

The substitution of a term s for a variable number j in a term t, written as $[j \mapsto s]t$, is defined inductively as follows:

$$[j \mapsto s]k = s \quad \text{if } k = j$$
$$[j \mapsto s]k = k \quad \text{if } k \neq j$$
$$[j \mapsto s]\lambda.t_1 = \lambda.[j+1 \mapsto \uparrow^1 (s)]t_1$$
$$[j \mapsto s]t_1\ t_2 = ([j \mapsto s]t_1)\,([j \mapsto s]t_2)$$

Formal Shifting definition

The d-place shift of a term t above cutoff c, written as $\uparrow_c^d (t)$, is defined inductively as follows (where $\uparrow^d (t)$ stands for $\uparrow_0^d$):

$$\uparrow_c^d (k) = k \quad \text{if } k < c$$
$$\uparrow_c^d (k) = k + d \quad \text{if } k \geq c$$
$$\uparrow_c^d (\lambda.t_1) = \lambda.\ \uparrow_{c+1}^d (t_1)$$
$$\uparrow_c^d (t_1\ t_2) = \uparrow_c^d (t_1)\ \uparrow_c^d (t_2)$$

Formal Substitution definition

The substitution of a term s for a variable number j in a term t, written as $[j \mapsto s]t$, is defined inductively as follows:

$$[j \mapsto s]k = s \quad \text{if } k = j$$
$$[j \mapsto s]k = k \quad \text{if } k \neq j$$
$$[j \mapsto s]\lambda.t_1 = \lambda.[j+1 \mapsto \uparrow^1 (s)]t_1$$
$$[j \mapsto s]t_1\ t_2 = ([j \mapsto s]t_1)\,([j \mapsto s]t_2)$$

Formal Shifting definition

The d-place shift of a term t above cutoff c, written as $\uparrow_c^d (t)$, is defined inductively as follows (where $\uparrow^d (t)$ stands for $\uparrow_0^d$):

$$\uparrow_c^d (k) = k \quad \text{if } k < c$$
$$\uparrow_c^d (k) = k + d \quad \text{if } k \geq c$$
$$\uparrow_c^d (\lambda.t_1) = \lambda.\ \uparrow_{c+1}^d (t_1)$$
$$\uparrow_c^d (t_1\ t_2) = \uparrow_c^d (t_1)\ \uparrow_c^d (t_2)$$

Evaluation

Consider *for the last time* the rule: $(\lambda x.t_{12})v_2 \rightarrow [x \mapsto v_2]t_{12}$ where $t_{12} = \lambda y.t, x \neq y$.

Reducing an abstraction $\lambda x.t_{12}$ make disappear the bound variable x, so the resulting term $[x \mapsto v_2]t_{12}$ belong to a smaller context (respect the number of λ binders), hence we have to shift its free variables *backward* by one.

Similarly, in the resulting term we've to shift free variables in v_2 *forward* by one because the term t_{12} is defined in a larger context (is protected by a λ-abstraction).

Finally, we're interested to start substituting the inner most bound variable since this is our reduction strategy, so the initial index given to substitution function is 0.

Modified reduction rule

$$(\lambda.t_{12})\, v_2 \rightarrow \uparrow^{-1} ([0 \mapsto \uparrow^{1} (v_2)]t_{12})$$

Evaluation

Consider *for the last time* the rule: $(\lambda x.t_{12})v_2 \rightarrow [x \mapsto v_2]t_{12}$ where $t_{12} = \lambda y.t, x \neq y$.

Reducing an abstraction $\lambda x.t_{12}$ make disappear the bound variable x, so the resulting term $[x \mapsto v_2]t_{12}$ belong to a smaller context (respect the number of λ binders), hence we have to shift its free variables *backward* by one.

Similarly, in the resulting term we've to shift free variables in v_2 *forward* by one because the term t_{12} is defined in a larger context (is protected by a λ-abstraction).

Finally, we're interested to start substituting the inner most bound variable since this is our reduction strategy, so the initial index given to substitution function is 0.

Modified reduction rule

$$(\lambda.t_{12})\, v_2 \rightarrow \uparrow^{-1} ([0 \mapsto \uparrow^{1} (v_2)]t_{12})$$

Evaluation

Consider *for the last time* the rule: $(\lambda x.t_{12})v_2 \rightarrow [x \mapsto v_2]t_{12}$ where $t_{12} = \lambda y.t, x \neq y$.

Reducing an abstraction $\lambda x.t_{12}$ make disappear the bound variable x, so the resulting term $[x \mapsto v_2]t_{12}$ belong to a smaller context (respect the number of λ binders), hence we have to shift its free variables *backward* by one.

Similarly, in the resulting term we've to shift free variables in v_2 *forward* by one because the term t_{12} is defined in a larger context (is protected by a λ-abstraction).

Finally, we're interested to start substituting the inner most bound variable since this is our reduction strategy, so the initial index given to substitution function is 0.

Modified reduction rule

$$(\lambda.t_{12})\, v_2 \rightarrow \uparrow^{-1} ([0 \mapsto \uparrow^{1} (v_2)]t_{12})$$

Evaluation

Consider *for the last time* the rule: $(\lambda x.t_{12})v_2 \rightarrow [x \mapsto v_2]t_{12}$ where $t_{12} = \lambda y.t, x \neq y$.

Reducing an abstraction $\lambda x.t_{12}$ make disappear the bound variable x, so the resulting term $[x \mapsto v_2]t_{12}$ belong to a smaller context (respect the number of λ binders), hence we have to shift its free variables *backward* by one.

Similarly, in the resulting term we've to shift free variables in v_2 *forward* by one because the term t_{12} is defined in a larger context (is protected by a λ-abstraction).

Finally, we're interested to start substituting the inner most bound variable since this is our reduction strategy, so the initial index given to substitution function is 0.

Modified reduction rule

$$(\lambda.t_{12})\, v_2 \rightarrow \uparrow^{-1} ([0 \mapsto \uparrow^{1} (v_2)]t_{12})$$

Evaluation

Consider *for the last time* the rule: $(\lambda x.t_{12})v_2 \rightarrow [x \mapsto v_2]t_{12}$ where $t_{12} = \lambda y.t, x \neq y$.

Reducing an abstraction $\lambda x.t_{12}$ make disappear the bound variable x, so the resulting term $[x \mapsto v_2]t_{12}$ belong to a smaller context (respect the number of λ binders), hence we have to shift its free variables *backward* by one.

Similarly, in the resulting term we've to shift free variables in v_2 *forward* by one because the term t_{12} is defined in a larger context (is protected by a λ-abstraction).

Finally, we're interested to start substituting the inner most bound variable since this is our reduction strategy, so the initial index given to substitution function is 0.

Modified reduction rule

$$(\lambda.t_{12})\ v_2 \rightarrow \uparrow^{-1} ([0 \mapsto \uparrow^{1} (v_2)]t_{12})$$