Typed Lambda Calculus

Motivations

- Partial specification of programs
- Avoid meaningless programs $(1 + \mathit{true})$
- Avoid memory violation
- Avoid programs with undefined semantics

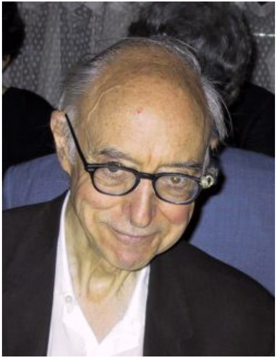
Contents

Simply Typed Lambda Calculus

Curry-Howard Isomorphism

Logical system	$\Leftrightarrow$	**Language**
Propositions	$\Leftrightarrow$	Types
Proofs	$\Leftrightarrow$	Programs
Proof normalisation	$\Leftrightarrow$	Program Evaluation

Curry'58, Howard'68



Adding (simply) types to λ-calculus

Grammar for types:

$$\begin{array}{lll} A, B ::= & \mathsf{b} & \text{(base types)} \quad | \\ & A \to B & \text{(functional types)} \end{array}$$

Example :

$$int \to bool \qquad bool \to (bool \to int) \qquad (bool \to bool) \to int$$

Remark

- $\to$ is right-associative, *e.g.* $A_1 \to A_2 \to A_3$ abbreviates $A_1 \to (A_2 \to A_3)$.
- Every type A can be written as $A_1 \to \ldots \to A_n \to \mathsf{b}$, where $A_1, \ldots, A_n$ $(n \geq 0)$ are arbitrary types and b is a base type.
- The standard order between types is given by
  $A < A \to B$ and $B < A \to B$.
  Thus base types are minimal with respect this order.
- In a typed framework substitutions are always well-typed, i.e, $t\{x\backslash u\}$ means that x and u have the same type.

Typing Environment

- A typing environment Γ is a finite function from variables to types, usually written $x_1 : A_1, \ldots, x_n : A_n$.
- Thus for example, $x : A, y : B$ and $y : B, x : A$ are two different notations for the same typing environment.
- The domain of $\Gamma = x_1 : A_1, \ldots, x_n : A_n$, written $\mathtt{dom}(\Gamma)$, is the set $\{x_1, \ldots, x_n\}$.
- We write $\Gamma, x : A$ for the typing environment extending Γ with the pair $x : A$. It is only defined iff $x \notin \mathtt{dom}(\Gamma)$.

Typed Lambda Calculus in Church-Style

$$\frac{}{\Gamma, x : A \vdash x : A}\ (ax)$$

$$\frac{\Gamma, x : A \vdash t : B}{\Gamma \vdash \lambda x : A.t : A \to B}\ (\to i) \qquad \frac{\Gamma \vdash t : A \to B \qquad \Gamma \vdash u : A}{\Gamma \vdash t\, u : B}\ (\to e)$$

We can also add constants: each constant c has an associated type $\mathtt{TC}(\mathsf{c}) : A$.

$$\frac{}{\Gamma \vdash \mathsf{c} : \mathtt{TC}(\mathsf{c})}$$

We can also add the let constructor:

$$\frac{\Gamma \vdash t : A \qquad \Gamma, x : A \vdash u : B}{\Gamma \vdash \mathtt{let}\ x : A = t\ \mathtt{in}\ u : B}$$

We denote by $\Gamma \vdash_C t : A$ the derivability/typing relation. We say that t is typable in Church-Style iff there is Γ and A such that $\Gamma \vdash_C t : A$.

Examples

Example of typable term in an empty environment:

$$\frac{\dfrac{y : A \to A \vdash y : A \to A}{\vdash \lambda y : A \to A.y : (A \to A) \to (A \to A)} \qquad \dfrac{x : A \vdash x : A}{\vdash \lambda x : A.x : A \to A}}{\vdash (\lambda y : A \to A.y)(\lambda x : A.x) : A \to A}$$

Example of typable term in a non-empty environment:

$$\frac{\mathtt{c_i} : \mathit{int} \to \mathit{int} \vdash \mathtt{c_i} : \mathit{int} \to \mathit{int} \qquad \dfrac{\mathtt{c_i} : \mathit{int} \to \mathit{int} \vdash \mathtt{c_i} : \mathit{int} \to \mathit{int} \qquad \mathtt{c_i} : \mathit{int} \to \mathit{int} \vdash 3 : \mathit{int}}{\mathtt{c_i}\,\mathit{int} \to \mathit{int} \vdash \mathtt{c_i}\ 3 : \mathit{int}}}{\mathtt{c_i} : \mathit{int} \to \mathit{int} \vdash \mathtt{c_i}\ (\mathtt{c_i}\ 3) : \mathit{int}}$$

Example of non-typable term: $\lambda x.xx$.

[Unicity] : If $\Gamma \vdash_C t : A$ and $\Gamma \vdash_C t : B$, then $A \equiv B$.

Proof.

By induction on t. □

[Weakening and Strengthening] : Let $\Gamma = \{x : B \mid x \in \mathtt{fv}(t)\}$ and $\Gamma \subseteq \Delta_1 \subseteq \Delta_2$. Then $\Delta_1 \vdash_C t : A$ iff $\Delta_2 \vdash_C t : A$.

Proof.

By induction on t. □

[Subject Reduction] If $\Gamma \vdash_C t : A$ and $t \to_\beta t'$, then $\Gamma \vdash_C t' : A$.

Proof.

By induction on $\Gamma \vdash_C t : A$ (blackboard). □

Typing Algorithm

$$
\begin{array}{lll}
\mathtt{Type}(\Gamma, \mathtt{c}) & = \mathtt{TC}(\mathtt{c}) & \\
\mathtt{Type}(\Gamma, x) & = A & \text{if } x : A \in \Gamma \\
\mathtt{Type}(\Gamma, \lambda x{:}\, A.t) & = A \to B & \text{if } \mathtt{Type}((\Gamma, x : A), t) = B \\
\mathtt{Type}(\Gamma, t\, u) & = B & \text{if } \mathtt{Type}(\Gamma, t) = A \to B \text{ and } \mathtt{Type}(\Gamma, u) = A \\
\mathtt{Type}(\Gamma, \mathtt{let}\ x{:}\, A = t\ \mathtt{in}\ u) & = B & \text{if } \mathtt{Type}(\Gamma, t) = A \text{ and } \mathtt{Type}((\Gamma, x : A), u) = B \\
\mathtt{Type}(\Gamma, t) & = \mathit{error} & \text{otherwise}
\end{array}
$$

Properties of the Typing Algorithm

[Termination]
For every term t and every environment Γ, the call $\texttt{Type}(\Gamma, t)$ terminates.

[Soundness]
If $\texttt{Type}(\Gamma, t) = A$, then $\Gamma \vdash_C t : A$.

[Completeness]
If $\Gamma \vdash_C t : A$, then $\texttt{Type}(\Gamma, t) = A$.

Said differently:
If $\texttt{Type}(\Gamma, t) = erreur$, then t is not typable in Γ.

Typed Lambda Calculus in Curry-Style

$$\frac{}{\Gamma, x : A \vdash x : A}\,(ax)$$

$$\frac{\Gamma, x : A \vdash t : B}{\Gamma \vdash \lambda x.t : A \to B}\,(\to i) \qquad \frac{\Gamma \vdash t : A \to B \qquad \Gamma \vdash u : A}{\Gamma \vdash t\, u : B}\,(\to e)$$

We can also add constants: each constant $\mathtt{c}$ has an associated type $\mathtt{TC}(\mathtt{c}) : A$.

$$\frac{}{\Gamma \vdash \mathtt{c} : \mathtt{TC}(\mathtt{c})}$$

We can also add lets:

$$\frac{\Gamma \vdash t : A \qquad \Gamma, x : A \vdash u : B}{\Gamma \vdash \mathtt{let}\ x = t\ \mathtt{in}\ u : B}$$

We denote by $\Gamma \vdash_\lambda t : A$ the derivability/typing relation. We say that t is typable in Curry-Style iff there is Γ and A such that $\Gamma \vdash_\lambda t : A$.

Unicity does not hold anymore:

$$\vdash_\lambda \lambda x.x : int \to int \qquad \vdash_\lambda \lambda x.x : bool \to bool$$

The identity function behaves in the same way for int and $bool$:

Polymorphism

Difficulties for a Typing Algorithm

$$\begin{array}{ll} \texttt{Type}(\Gamma, \lambda x.t) = A \to B & \textit{if} \text{ there exists } A \text{ s.t. } \texttt{Type}((\Gamma, x : A), T) = B \\ \texttt{Type}(\Gamma, \texttt{let } x = t \texttt{ in } u) = B & \textit{if} \text{ there exists } A \text{ s.t. } \texttt{Type}(\Gamma, t) = A \text{ and} \\ & \qquad \texttt{Type}((\Gamma, x : A), u) = B \end{array}$$

Monomorphic Type Inference

The type inference problem for a term t: $\exists\Gamma\ \exists A$ such that $\Gamma \vdash_\lambda t : A$?

General Technical Tools:

- We consider a table of types for constants, called TC. E.g. $TC(3) = int$ and $TC(\mathtt{c_i}) = int \to int$.
- Let t be a term. For each sub-term u of t we introduce a type variable α_u.
- We associate to every term t a set of equations $SE(t) = \{\alpha_1 \doteq u_1, \ldots, \alpha_m \doteq u_m\}$.
- The solution to the type inference problem for the term t will be given by the **most general unifier** (mgu) of the corresponding set of equations.

t	$SE(t)$
x	$\{\alpha_t \doteq \alpha_x\}$
$\mathtt{c}$	$\{\alpha_t \doteq TC(\mathtt{c})\}$
$u\,v$	$\{\alpha_u \doteq \alpha_v \to \alpha_t\} \cup SE(u) \cup SE(v)$
$\lambda x.u$	$\{\alpha_t \doteq \alpha_x \to \alpha_u\} \cup SE(u)$
$\mathtt{let}\ x = u\ \mathtt{in}\ v$	$\{\alpha_t \doteq \alpha_v; \alpha_x \doteq \alpha_u\} \cup SE(u) \cup SE(v)$

Examples

- $t_0 := \lambda f.\lambda g.f\ g$
  $SE(t_0) = \{\alpha_{t_0} \doteq \alpha_f \rightarrow \alpha_{\lambda g.fg}, \alpha_{\lambda g.fg} \doteq \alpha_g \rightarrow \alpha_{fg}, \alpha_f \doteq \alpha_g \rightarrow \alpha_{fg}, \alpha_f \doteq \alpha_f, \alpha_g \doteq \alpha_g\}$
  - The mgu of $SE(t_0)$ is the substitution σ_{t_0} such that $\sigma_{t_0}(\alpha_{t_0}) = (\alpha_g \rightarrow \alpha_{fg}) \rightarrow (\alpha_g \rightarrow \alpha_{fg})$.
  - t_0 is typable with instances of $\sigma_{t_0}(\alpha_{t_0})$ which are types of the form $(A \rightarrow B) \rightarrow (A \rightarrow B)$, for any arbitrary types A and B.
- $t_1 := \texttt{let}\ f = \lambda x.x\ \texttt{in}\ f(f\ z)$
  $SE(t_1) = \{\alpha_{t_1} \doteq \alpha_{f(fz)}, \alpha_f \doteq \alpha_{\lambda x.x}, \alpha_{\lambda x.x} \doteq \alpha_x \rightarrow \alpha_x, \alpha_x \doteq \alpha_x, \alpha_f \doteq \alpha_{fz} \rightarrow \alpha_{f(fz)}, \alpha_f \doteq \alpha_f, \alpha_f \doteq \alpha_z \rightarrow \alpha_{fz}, \alpha_z \doteq \alpha_z\}$
  - The mgu of $SE(t_1)$ is the substitution σ_{t_1} such that $\sigma_{t_1}(\alpha_{t_1}) = \alpha_{f(fz)}$.
  - t_1 is typable with instances of $\sigma_{t_1}(\alpha_{t_1})$ which are arbitrary types A.

Properties of the Type Inference Algorithm

Theorem

(Soundness) *If σ is a solution of $SE(t)$, then t is (Curry) typable. Moreover, $\Delta \vdash_\lambda t : \sigma'(\alpha_t)$, where $\Delta = \{x : \sigma'(\alpha_x) \mid x \in \mathtt{fv}(t)\}$ and σ' is an instance of σ.*

Theorem

(Completeness) *If t is (Curry) typable, then $SE(t)$ admits a solution.*

Theorem

(Principality) *If t is (Curry) typable, i.e. $\Delta \vdash_\lambda t : A$, then A is an instance of the principal type (i.e. $A = \sigma'(\sigma(\alpha_t))$), where σ is the mgu of system $SE(t)$ and σ' is a substitution.*

Polymorphic Lambda Calculus

Motivations

- General behaviour of an operator, i.e. without looking at the particular nature (or type) of its parameters.
- More clear and concise programs.
- Reutilisation of operators.

Different Forms of Polymorphism

- Ad-hoc polymorphisme (overloaded constructors), originally described by Strachey.
- Subtyping polymorphism, introduced by Wegner and Cardelli: Si $\Gamma \vdash t : A \to B$, $\Gamma \vdash u : A'$ et $A' \leq A$ alors $tu : B$.
- Parametric polymorphism, introduced by Reynolds and Girard.

Motivation :
Many indentity functions:
$Id_{int} : int \to int$, $Id_{bool} : bool \to bool$, $Id_{int \to int} : (int \to int) \to (int \to int), \ldots$
Many functions to add an element to a list:
$Aj_{int} : int \to list(int) \to list(int)$, $Aj_{bool} : bool \to list(bool) \to list(bool), \ldots$
Idea :
$Id : \forall \alpha . \alpha \to \alpha$
and thus:
$Id_{int} = Id[int]$
$Id_{bool} = Id[bool]$
$Id_{int \to int} = Id[int \to int]$

Remark The key idea is the instantiation relation between the general type α, and the particular used types $int, bool, int \to int$.

Girard-Reynolds Polymorphism

Types : $A ::= \mathsf{b} \mid \alpha \mid A \rightarrow A \mid \forall\alpha.A$

Notation : $\forall\alpha.\forall\beta.A = \forall(\alpha,\beta).A$.

Expressions :

$$\begin{array}{rl} t ::= & x \mid \mathsf{c} \mid t\,t \mid \\ & \lambda x : A.t \mid \texttt{let}\ x : A = t\ \texttt{in}\ t \mid \\ & t[A] \mid \Lambda\alpha t \end{array}$$

Reduction Rules :

$$\begin{array}{lcl} (\lambda x : A.t)\,u & \rightarrow & t\{x\backslash u\} \\ \texttt{let}\ x : A = u\ \texttt{in}\ t & \rightarrow & t\{x\backslash u\} \\ (\Lambda\alpha t)[A] & \rightarrow & t\{\alpha\backslash A\} \end{array}$$

Reduction Rules :

Let $Id = \Lambda\alpha\lambda x : \alpha.x$

$$\begin{array}{lcl} (Id[int]) & \rightarrow & \lambda x : int.x \\ (Id[int])\,3 & \rightarrow & (\lambda x : int.x)\,3 \rightarrow\ 3 \end{array}$$

Type Free Variables

$$\begin{array}{lcl} \mathtt{tfv}(\mathrm{b}) & = & \emptyset \\ \mathtt{tfv}(\alpha) & = & \{\alpha\} \\ \mathtt{tfv}(\forall\alpha.A) & = & \mathtt{tfv}(A) \setminus \{\alpha\} \\ \mathtt{tfv}(A \to B) & = & \mathtt{tfv}(A) \cup \mathtt{tfv}(B) \end{array}$$

A type A is closed iff $\mathtt{tfv}(A) = \emptyset$.

Example : $\mathtt{tfv}(\beta \to (\forall\alpha.\alpha \to \gamma)) = \{\beta, \gamma\}$ and $\forall\alpha.\forall\beta.(\alpha \to \beta)$ is closed.

The same rules used for monomorphic Church-style plus:

$$\frac{\Gamma \vdash t : A \qquad \alpha \notin \mathtt{tfv}(\Gamma)}{\Gamma \vdash \Lambda\alpha t : \forall\alpha.A} \qquad \frac{\Gamma \vdash t : \forall\alpha.A}{\Gamma \vdash t[B] : A\{\alpha\backslash B\}}$$

where $\mathtt{tfv}(\Gamma) = \bigcup_{x\in\Gamma} \mathtt{tfv}(\Gamma(x))$ is the set of type free variables of Γ.

We denote by $\Gamma \vdash_G t : A$ the derivability/typing relation.

Example :

$$\cfrac{\cfrac{\cfrac{\cfrac{x : \alpha \vdash x : \alpha}{\vdash \lambda x : \alpha.x : \alpha \to \alpha}}{\vdash \Lambda\alpha\lambda x : \alpha.x : \forall\alpha.(\alpha \to \alpha)}}{\vdash (\Lambda\alpha\lambda x : \alpha.x)[int] : int \to int} \qquad \vdash 3 : int}{\vdash ((\Lambda\alpha\lambda x : \alpha.x)[int])\, 3 : int}$$

Theorem

(Schubert) *The inference problem in Girard system is undecidable.*

Idea to become decidable: to restrict the grammar of types.

ML Polymorphism

Type matrix : $M ::= \mathsf{b} \mid \alpha \mid M \to M$
Type : $S ::= \forall\alpha_1.\ldots.\forall\alpha_n.M$
A type matrix is a particular case of type.
Type for constants :

$$\begin{array}{ll}
TC(+): & int \to int \to int \\
TC(\mathtt{fst}): & \forall\alpha\forall\beta.(\alpha \to \beta) \to \alpha \\
TC(\mathtt{snd}): & \forall\alpha\forall\beta.(\alpha \to \beta) \to \beta \\
TC(\mathtt{ifthenelse}): & \forall\alpha.(bool \to \alpha \to \alpha) \to \alpha \\
TC(\mathtt{fix}): & \forall\alpha.(\alpha \to \alpha) \to \alpha
\end{array}$$

Type Instance

Definition

A type A is an instance of type $\forall\alpha_1.\dots\forall\alpha_n.B$, written $A \leq \forall\alpha_1.\dots\forall\alpha_n.B$ iff there exist $C_1,\dots,C_n$ s.t. $A = B\{\alpha_1,\dots,\alpha_n \backslash C_1,\dots,C_n\}$.
In particular, for $n = 0$, we have $A \leq B$ iff $A \equiv B$.

Example :

$$
\begin{array}{lcl}
int \to int & \leq & \forall\alpha.\alpha \to \alpha \\
bool \to int & \leq & \forall\alpha.\forall\beta.\alpha \to \beta \\
bool \to int & \not\leq & \forall\alpha.\alpha \to \alpha \\
\forall\beta.int \to \beta & \leq & \forall\alpha.\forall\beta.\alpha \to \beta \\
\forall\beta.(\beta \to \beta) \to \forall\beta.(\beta \to \beta) & \not\leq & \forall\alpha.\alpha \to \alpha
\end{array}
$$

Type Generalization

Definition

The Gen operator is given by $\texttt{Gen}(A, \Gamma) = \forall\alpha_1 \ldots . \forall\alpha_n.A$, where each variable α_i is free in A but not in Γ, i.e. forall $i = 1 \ldots n$, we have $\alpha_i \in \texttt{tfv}(A) \setminus \texttt{tfv}(\Gamma)$.

Example : Let $A = \alpha \to \beta$ and $\Gamma = x : \beta, y : \forall\alpha.\alpha$. Then $\texttt{Gen}(A, \Gamma) = \forall\alpha.\alpha \to \beta$.

Polymorphic Lambda Calculus in Curry-Style

$$\frac{A \leq \Gamma(x)}{\Gamma \vdash x : A} \qquad \frac{A \leq TC(\mathtt{c})}{\Gamma \vdash \mathtt{c} : A}$$

$$\frac{\Gamma \vdash t : A \to B \qquad \Gamma \vdash u : A}{\Gamma \vdash t\,u : B} \qquad \frac{\Gamma, x : A \vdash t : B}{\Gamma \vdash \lambda x.t : A \to B}$$

$$\frac{\Gamma \vdash u : A \qquad \Gamma, x : \mathtt{Gen}(A, \Gamma) \vdash t : B}{\Gamma \vdash \mathtt{let}\ x = u\ \mathtt{in}\ t : B}$$

We denote by $\Gamma \vdash_{ML} t : A$ the derivability/typing relation.

Example

Let $A = \forall\alpha.\alpha \to \alpha$

$$
\dfrac{\dfrac{y : \alpha \vdash y : \alpha}{\vdash \lambda y.y : \alpha \to \alpha} \qquad \dfrac{\dfrac{int \to int \leq A}{f : A \vdash f : int \to int} \quad f : A \vdash 1 : int}{f : \forall\alpha.\alpha \to \alpha \vdash f\,1 : int}}{\vdash \texttt{let}\ f = \lambda y.y\ \texttt{in}\ f\,1 : int}
$$

Properties

[Substitution] :
If $\Gamma, x : \forall\alpha_1. \ldots . \forall\alpha_n.B \vdash_{ML} t : A$ and $\Gamma \vdash_{ML} u : B$, then $\Gamma \vdash_{ML} t\{x \backslash u\} : A$.

[Subject Reduction] :
If $\Gamma \vdash_{ML} t : A$ and $t \rightarrow t'$, then $\Gamma \vdash_{ML} t' : A$.

Let $t \equiv \mathtt{let}\ x = \lambda y.y\ \mathtt{in}\ x$. Let consider the set of equations:

$$\{\alpha_t \doteq \alpha_x, \alpha_x \doteq \mathtt{Gen}(\alpha_{\lambda y.y}, \emptyset), \alpha_{\lambda y.y} \doteq \alpha_y \rightarrow \alpha_y\}$$

If we treat the second equation we obtain $\alpha_x \equiv \forall \alpha_{\lambda y.y}.\alpha_{\lambda y.y}$, which is **incorrect** since α_x should be a functional type (an arrow).

Towards a Type Inference Algorithm: Notations

Let $\Gamma = x_1 : A_1, \ldots, x_n : A_n$ be an environment and let σ be a type substitution. then $\sigma(\Gamma) = x_1 : \sigma(A_1), \ldots, x_n : \sigma(A_n)$.

We note $inst(\forall\alpha_1 . \ldots \forall\alpha_n . A)$ the type $A\{\alpha_1, \ldots, \alpha_n \backslash \beta_1, \ldots, \beta_n\}$, where $\beta_1, \ldots, \beta_n$ are fresh variables.

Damas-Milner-Tofte Algorithm

Input : an environment Γ and a term t s.t. $\mathtt{fv}(t) \subseteq \Gamma$.
Output : a type A and a substitution σ s.t. $\sigma(\Gamma) \vdash_{ML} t : A$ (id is the empty substitution).

$$\begin{array}{lcl}
W((\Delta, x : A), x) & = & (inst(A), id) \\
W(\Delta, \mathtt{c}) & = & (inst(TC(\mathtt{c})), id) \\
W(\Delta, \lambda x.u) & = & (\rho_B(\alpha_x) \to B, \rho_B) \\
\end{array}$$

where $W((\Delta, x : \alpha_x), u) = (B, \rho_B)$ and α_x is a fresh variable

$$W(\Delta, u\, v) = (\mu(\alpha), \mu \circ \rho_C \circ \rho_B)$$

where $W(\Delta, u) = (B, \rho_B), W(\rho_B(\Delta), v) = (C, \rho_C)$,
α is a fresh variable and $\mu = mgu\{\rho_C(B) \doteq C \to \alpha\}$

$$W(\Delta, \mathtt{let}\ x = u\ \mathtt{in}\ v) = (C, \rho_C \circ \rho_B)$$

where $W(\Delta, u) = (B, \rho_B)$ and $W((\rho_B(\Delta), x : \mathtt{Gen}(B, \rho_B(\Delta))), v) = (C, \rho_C)$

Damas-Milner-Tofte Algorithm in (a more) Sequential Style

- Case $W((\Delta, x : A), x)$: return $(inst(A), id)$
- Case $W(\Delta, \mathtt{c})$: return $(inst(TC(\mathtt{c})), id)$
- Case $W(\Delta, \lambda x.u)$:
  1. Make a recursive call $W((\Delta, x : \alpha_x), u)$, where α_x is a fresh variable.
  2. Let (B, ρ_B) be the result of the previous recursive call.
  3. Return $(\rho_B(\alpha_x) \to B, \rho_B)$.
- Case $W(\Delta, u\, v)$:
  1. Make a first recursive call $W(\Delta, u)$.
  2. Let (B, ρ_B) be the result of the first recursive call.
  3. Make a second recursive call $W(\rho_B(\Delta), v)$.
  4. Let (C, ρ_C) be the result of the second recursive call.
  5. Compute the mgu μ of the equation $\{\rho_C(B) \doteq C \to \alpha\}$.
  6. Return $(\mu(\alpha), \mu \circ \rho_C \circ \rho_B)$.
- Case $W(\Delta, \mathtt{let}\ x = u\ \mathtt{in}\ v)$:
  1. Make a first recursive call $W(\Delta, u)$.
  2. Let (B, ρ_B) be the result of the first recursive call.
  3. Make a second recursive call $W((\rho_B(\Delta), x : \mathtt{Gen}(B, \rho_B(\Delta))), v)$.
  4. Let (C, ρ_C) be the result of the second recursive call.
  5. Return $(C, \rho_C \circ \rho_B)$.

Premier exemple

$W(\emptyset, \lambda x. * 4x) = (int \to int, \{\alpha/int \to int, \beta/int, \alpha_x/int\})$
$W(x : \alpha_x, *4x) = (int, \{\alpha/int \to int, \beta/int, \alpha_x/int\})$, computes $mgu\{int \to int \doteq \alpha_x \to \beta\}$
$W(x : \alpha_x, *4) = (int \to int, \{\alpha/int \to int\})$, computes $mgu\{int \to int \to int \doteq int \to \alpha\}$
$W(x : \alpha_x, *) = (int \to int \to int, id)$
$W(x : \alpha_x, 4) = (int, id)$
$W(x : \alpha_x, x) = (\alpha_x, id)$

Then $\emptyset \vdash_{ML} \lambda x. * 4x : int \to int$.

Second Example

$W(\emptyset, \texttt{let}\ f = \lambda x.x\ \texttt{in}\ f\,2) = (int, \{\beta/int, \gamma/int\})$
$W(\emptyset, \lambda x.x) = (\alpha_x \to \alpha_x, id)$
$W(x : \alpha_x, x) = (\alpha_x, id)$
$W(f : \forall \alpha_x.\alpha_x \to \alpha_x, f\,2) = (int, \{\beta/int, \gamma/int\})$, where $mgu\{\beta \to \beta \doteq int \to \gamma\} = \{\beta/int, \gamma/int\}$
$W(f : \forall \alpha_x.\alpha_x \to \alpha_x, f) = (\beta \to \beta, id)$
$W(f : \forall \alpha_x.\alpha_x \to \alpha_x, 2) = (int, id)$

Then $\emptyset \vdash_{ML} \texttt{let}\ f = \lambda x.x\ \texttt{in}\ f\,2 : int$.

Properties of the Type Inference Algorithm

Theorem

(Soundness) *If* $W(\Delta, t) = (A, \sigma)$*, then* $\sigma(\Delta) \vdash_{ML} t : A$*.*

Theorem

(Completeness) *If* $\tau(\Delta) \vdash_{ML} t : B$*, then* $W(\Delta, t) = (A, \sigma)$*, where* B *is an instance of* A *and* τ *is an instance of* σ*.*

Corollary : $\emptyset \vdash_{ML} t : A$ iff $W(\emptyset, t) = (B, \sigma)$ and A is an of B.