Types in Programming Languages (4)

Christian Urban

http://www4.in.tum.de/lehre/vorlesungen/types/WS0607/

Type-Schemes

- In addition to types of the form:

$$\begin{array}{rcll} T & ::= & X & \text{type variables} \\ & | & T \to T & \text{function types} \end{array}$$

- we introduced type-schemes:

$$S \quad ::= \quad \forall A.T$$

Where A ranges over a finite set of type-variables. When $A = \{X_1, \ldots, X_n\}$ we write $\forall A.T$ as

$$\forall\{X_1, \ldots, X_n\}.T$$

$\forall\{\}.T$ is possible; $\forall A.\forall B.T$ is not. Note that type-schemes are not types!

Typing Problem

- Given a valid Γ and an e, can we find a T such that

$$\Gamma \vdash e : T$$

holds?

- Completeness: For all (Γ, e) with e typable in the context Γ, the algorithm should produce a T.
- Soundness For all (Γ, e) with e untypable in the context Γ, the algorithm should fail.

Example of an untypable term: $\boldsymbol{\lambda} x.(x\ x)$

$$\varnothing \vdash \boldsymbol{\lambda} x.(x\ x) : ??$$

MiniML Type-System

- Variables

$$\frac{\mathsf{valid}\ \Gamma \quad (x : S) \in \Gamma \quad S \succ T}{\Gamma \vdash x : T}$$

- Applications

$$\frac{\Gamma \vdash e_1 : T_1 \to T_2 \quad \Gamma \vdash e_2 : T_1}{\Gamma \vdash e_1\ e_2 : T_2}$$

- Lambdas

$$\frac{x : \forall\{\}.T_1, \Gamma \vdash e : T_2 \quad x \notin \mathsf{dom}\,\Gamma}{\Gamma \vdash \lambda x.e : T_1 \to T_2}$$

- Lets $A = \mathsf{tv}(T_1) - \mathsf{ftv}(\mathsf{codom}\ \Gamma)$

$$\frac{\Gamma \vdash e_1 : T_1 \quad x : \forall A.T_1, \Gamma \vdash e_2 : T_2 \quad x \notin \mathsf{dom}\,\Gamma}{\Gamma \vdash \mathsf{let}\, x = e_1\, \mathsf{in}\, e_2 : T_2}$$

MiniML Type-System

- Variables

$$\frac{\mathsf{valid}\ \Gamma \quad (x : S) \in \Gamma \quad S \succ T}{\Gamma \vdash x : T}$$

- Applications

$$\frac{\Gamma \vdash e_1 : T_1 \to T_2 \quad \Gamma \vdash e_2 : T_1}{\Gamma \vdash e_1\ e_2 : T_2}$$

- Lambdas

$$\frac{x : \forall\{\}.T_1, \Gamma \vdash e : T_2 \quad x \not\in \mathsf{dom}\ \Gamma}{\Gamma \vdash \lambda x.e : T_1 \to T_2}$$

- Lets $A = \mathsf{tv}(T_1) - \mathsf{ftv}(\mathsf{codom}\ \Gamma)$

$$\frac{\Gamma \vdash e_1 : T_1 \quad x : \forall A.T_1, \Gamma \vdash e_2 : T_2 \quad x \not\in \mathsf{dom}\ \Gamma}{\Gamma \vdash \mathsf{let}\ x = e_1\ \mathsf{in}\ e_2 : T_2}$$

Unification

- TVar-TVar

  $$\{X =^? X, \ldots\} \stackrel{\varepsilon}{\Longrightarrow} \{\ldots\}$$

- Fun-Fun

  $$\{T_1 \to T_2 =^? U_1 \to U_2, \ldots\} \stackrel{\varepsilon}{\Longrightarrow} \{T_1 =^? U_1, T_2 =^? U_2, \ldots\}$$

- TVar-Ty

  $$\{X =^? T, \ldots\} \stackrel{[X:=T]}{\Longrightarrow} \{\ldots\}[X := T]$$

  Ty-TVar

  $$\{T =^? X, \ldots\} \stackrel{[X:=T]}{\Longrightarrow} \{\ldots\}[X := T]$$

  both transformation only if $X \not\in \mathrm{tv}(T)$

- transform until you reach $\varnothing$; if stuck, no unifier

Unifier

- If $\emptyset$ is reached, you have a sequence:

$$P_1 \overset{\theta_1}{\Longrightarrow} P_2 \overset{\theta_2}{\Longrightarrow} \ldots \overset{\theta_n}{\Longrightarrow} \emptyset$$

- The (most general) unifier θ for the problem P_1 is then

$$\theta = \theta_n \circ \ldots \circ \theta_2 \circ \theta_1$$

- Substitution composition $\sigma_1 \circ \sigma_2$ is defined:

$$\underbrace{[X_1 := T_1, \ldots, X_n := T_n]}_{\sigma_1} \circ \underbrace{[Y_1 := U_1, \ldots, Y_m := U_m]}_{\sigma_2}$$

gives $[\ldots, X_i := T_i, \ldots, Y_1 := \sigma_1(U_1), \ldots, Y_m := \sigma_1(U_m)]$
where all $X_j := \_$ are deleted which are in $\{Y_1, \ldots, Y_m\}$.

The Unification Algorithm

- In every sequence:

$$P_1 \stackrel{\theta_1}{\Longrightarrow} P_2 \stackrel{\theta_2}{\Longrightarrow} \ldots \stackrel{\theta_n}{\Longrightarrow} \varnothing$$

the lexicographic ordered measure (n_1, n_2) goes down (n_1 is the number of variables in a problem; n_2 is sum of the sizes of terms in a problem).

The Unification Algorithm

- In every sequence:

$$P_1 \xrightarrow{\theta_1} P_2 \xrightarrow{\theta_2} \ldots \xrightarrow{\theta_n} \varnothing$$

the lexicographic ordered measure (n_1, n_2) goes down (n_1 is the number of variables in a problem; n_2 is sum of the sizes of terms in a problem).

TVar-TVar:

$$\{X =^? X, \ldots\} \xrightarrow{\varepsilon} \{\ldots\}$$

The Unification Algorithm

- In every sequence:

$$P_1 \stackrel{\theta_1}{\Longrightarrow} P_2 \stackrel{\theta_2}{\Longrightarrow} \ldots \stackrel{\theta_n}{\Longrightarrow} \varnothing$$

  the lexicographic ordered measure (n_1, n_2) goes down (n_1 is the number of variables in a problem; n_2 is sum of the sizes of terms in a problem).

Fun-Fun:

$$\{T_1 \rightarrow T_2 =^? U_1 \rightarrow U_2, \ldots\} \stackrel{\varepsilon}{\Longrightarrow} \{T_1 =^? U_1, T_2 =^? U_2, \ldots\}$$

The Unification Algorithm

- In every sequence:

$$P_1 \overset{\theta_1}{\Longrightarrow} P_2 \overset{\theta_2}{\Longrightarrow} \ldots \overset{\theta_n}{\Longrightarrow} \varnothing$$

the lexicographic ordered measure (n_1, n_2) goes down (n_1 is the number of variables in a problem; n_2 is sum of the sizes of terms in a problem).

TVar-Ty:

$$\{X =^? T, \ldots\} \overset{[X:=T]}{\Longrightarrow} \{\ldots\}[X := T]$$

provided $X \not\in \mathsf{tv}(T)$

The Unification Algorithm

- In every sequence:

$$P_1 \xRightarrow{\theta_1} P_2 \xRightarrow{\theta_2} \ldots \xRightarrow{\theta_n} \varnothing$$

  the lexicographic ordered measure (n_1, n_2) goes down (n_1 is the number of variables in a problem; n_2 is sum of the sizes of terms in a problem).

- Therefore the unification algorithm will always terminate (either produces the empty set or gets stuck).

Soundness and Completeness

- Given a unification problem, let $U(P)$ be the set of all the solutions of P (set of some substitutions).
- For a transformation

$$P \overset{\theta}{\Longrightarrow} P'$$

we can show:

  - if $\theta' \in U(P)$ then $\theta' \in U(P')$
  - if $\theta' \in U(P')$ then $\theta' \circ \theta \in U(P)$.

Soundness and Completeness

- Given a unification problem, let $U(P)$ be the set of all the solutions of P (set of some substitutions).
- For a transformation

$$P \stackrel{\theta}{\Longrightarrow} P'$$

we can show:

  - if $\theta' \in U(P)$ then $\theta' \in U(P')$
  - if $\theta' \in U(P')$ then $\theta' \circ \theta \in U(P)$.

Soundness

- Since $\varepsilon \in U(\emptyset)$, we have $\theta_n \circ \ldots \theta_2 \circ \theta_1 \in U(P_1)$

$$P_1 \stackrel{\theta_1}{\Longrightarrow} P_2 \stackrel{\theta_2}{\Longrightarrow} \ldots \stackrel{\theta_n}{\Longrightarrow} \emptyset$$

Soundness and Completeness

- Given a unification problem, let $U(P)$ be the set of all the solutions of P (set of some substitutions).
- For a transformation

$$P \xrightarrow{\theta} P'$$

we can show:

  - if $\theta' \in U(P)$ then $\theta' \in U(P')$
  - if $\theta' \in U(P')$ then $\theta' \circ \theta \in U(P)$.

Completeness

- For a stuck problem $U(P_{stuck}) = \varnothing$, therefore $U(P_1) = \varnothing$.

$$P_1 \xrightarrow{\theta_1} P_2 \xrightarrow{\theta_2} \dots \xrightarrow{\theta_n} P_{stuck}$$

Typing Algorithm

- Input: an expression e and a (valid) context Γ
- Output: FAIL or a substitution θ and type T

  If a θ and T, then we know

$$\theta(\Gamma) \vdash e : T$$

Algorithm W

The original algorithm of Damas and Milner:

- Variables:

$$W(\Gamma, x) = (\varepsilon, T[X_1 := Y_1, \ldots, X_n := Y_n])$$

  where $(x : \forall\{X_1, \ldots, X_n\}.T) \in \Gamma$ and the Y_i are distinct and fresh (w.r.t. Γ and T).

- Lambdas: calculate first

$$W((x : \forall\{\}.Y, \Gamma), e) = (\theta, T_1)$$

  where Y is fresh (w.r.t. Γ). Then return

$$W(\Gamma, \lambda x.e) = (\theta, \theta(Y) \to T_1)$$

Algorithm W

$$\frac{\text{valid } \Gamma \quad (x : S) \in \Gamma \quad S \succ T}{\Gamma \vdash x : T}$$

The original algorithm of Damas and Milner:

- Variables:

$$W(\Gamma, x) = (\varepsilon, T[X_1 := Y_1, \ldots, X_n := Y_n])$$

where $(x : \forall\{X_1, \ldots, X_n\}.T) \in \Gamma$ and the Y_i are distinct and fresh (w.r.t. Γ and T).

- Lambdas: calculate first

$$W((x : \forall\{\}.Y, \Gamma), e) = (\theta, T_1)$$

where Y is fresh (w.r.t. Γ). Then return

$$W(\Gamma, \lambda x.e) = (\theta, \theta(Y) \to T_1)$$

Algorithm W

$$\frac{x \notin \mathsf{dom}\,\Gamma \quad x : \forall\{\}.T_1, \Gamma \vdash e : T_2}{\Gamma \vdash \lambda x.e : T_1 \to T_2}$$

The original algorithm of Damas and

- Variables:

$$W(\Gamma, x) = (\varepsilon, T[X_1 := Y_1, \ldots, X_n := Y_n])$$

where $(x : \forall\{X_1, \ldots, X_n\}.T) \in \Gamma$ and the Y_i are distinct and fresh (w.r.t. Γ and T).

- Lambdas: calculate first

$$W((x : \forall\{\}.Y, \Gamma), e) = (\theta, T_1)$$

where Y is fresh (w.r.t. Γ). Then return

$$W(\Gamma, \lambda x.e) = (\theta, \theta(Y) \to T_1)$$

Algorithm W

$$\frac{\Gamma \vdash e_1 : T_1 \to T_2 \qquad \Gamma \vdash e_2 : T_1}{\Gamma \vdash e_1\ e_2 : T_2}$$

- Applications: Calculate first

$$W(\Gamma, e_1) = (\theta_1, T)$$

then

$$W(\theta_1(\Gamma), e_2) = (\theta_2, T_1)$$

and then

$$\text{unify}\ \{\theta_2(T) =^? T_1 \to Y\} = \theta_3$$

where Y is fresh (w.r.t. Γ). Finally return

$$W(\Gamma, e_1\ e_2) = (\theta_3 \circ \theta_2 \circ \theta_1, \theta_3(Y))$$

Algorithm W

$$\frac{x \notin \mathsf{dom}\,\Gamma \quad \Gamma \vdash e_1 : T_1 \quad x : \forall A.T_1, \Gamma \vdash e_2 : T_2}{\Gamma \vdash \mathsf{let}\, x = e_1 \,\mathsf{in}\, e_2 : T_2}$$

- Lets: Calculate first

$$W(\Gamma, e_1) = (\theta_1, T_1)$$

then

$$A = \mathsf{tv}(T_1) - \mathsf{ftv}(\theta_1(\Gamma))$$

and then

$$W((x : \forall A.T_1, \theta_1(\Gamma)), e_2) = (\theta_2, T_2)$$

Finally return

$$W(\Gamma, \mathsf{let}\, x = e_1 \,\mathsf{in}\, e_2) = (\theta_2 \circ \theta_1, T_2)$$

Example

- $\Gamma \vdash \mathsf{let}\ f = \lambda x.x\ \mathsf{in}\ g\ (f\ a) :$?

  We expect $X_2 \to X_3$.

- $\Gamma = \left\{ \begin{array}{l} g : \forall\{\}.X_1 \to X_2 \to X_3, \\ a : \forall\{\}.X_1, \end{array} \right\}$

- $\Gamma' = \left\{ \begin{array}{l} x : \forall\{\}.Y \\ g : \forall\{\}.X_1 \to X_2 \to X_3, \\ a : \forall\{\}.X_1, \end{array} \right\}$

- $\Gamma'' = \left\{ \begin{array}{l} f : \forall\{Y\}.Y \to Y \\ g : \forall\{\}.X_1 \to X_2 \to X_3, \\ a : \forall\{\}.X_1, \end{array} \right\}$

Principal Type-Scheme

- Input: an expression e and a (valid) context Γ
- Output: FAIL or a substitution θ and type T
- "Real" Output: FAIL or a type-scheme $S = \forall \quad .T$ where

$$= \mathsf{tv}(T) - \mathsf{ftv}(\theta(\Gamma))$$

Remember the Homework?

- Type into your favourite functional language:

$$
\begin{array}{l}
\text{let } pair = \lambda x.\lambda y.\lambda z.\; z\, x\, y \text{ in} \\
\quad \text{let } x_1 = \lambda y. pair\; y\; y \text{ in} \\
\quad\quad \text{let } x_2 = \lambda y. x_1\,(x_1\, y) \text{ in} \\
\quad\quad\quad \text{let } x_3 = \lambda y. x_2\,(x_2\, y) \text{ in} \\
\quad\quad\quad\quad \text{let } x_4 = \lambda y. x_3\,(x_3\, y) \text{ in} \\
\quad\quad\quad\quad\quad \text{let } x_5 = \lambda y. x_4\,(x_4\, y) \text{ in} \\
\quad\quad\quad\quad\quad\quad x_5\,(\lambda y. y)
\end{array}
$$

and let it check what its prinicpal type-scheme is.

Remember the Homework?

- Type into your favourite functional language:

$$\mathsf{let}\ pair = \lambda x.\lambda y.\lambda z.\ z\ x\ y\ \mathsf{in}$$

Although typing is decidable, it is known to be exponential-time complete, and the type can be exponentially larger than the expression.

BUT this problem does not arise naturally in practice and the typing algorithm is **not** a bottle-neck in an ML-compiler.

and let it check what its prinicpal type-scheme is.

Story So Far

- We sould now start to show soundness and completeness of W (rather tricky).
- Instead, we will look at extensions of the language and interaction with the typing system — there are a few surprises.

  Remember type-systems should provide safety (prevent all forbidden errors which includes untrapped errors).

Polymorphic References

- Let's assume we have memory...

$$
\begin{array}{rcll}
T & ::= & \ldots & \\
 & | & \mathsf{unit} & \text{unit type} \\
 & | & T\ \mathsf{ref} & \text{type of references} \\
 & & & \\
e & ::= & \ldots & \\
 & | & () & \text{unit value} \\
 & | & \mathsf{ref}\ e & \text{creation of a reference} \\
 & | & !e & \text{de-referencing} \\
 & | & e := e & \text{assignment}
\end{array}
$$

MiniML Type-System

- Units

$$\frac{\mathsf{valid}\ \Gamma}{\Gamma \vdash ()\ :\ \mathsf{unit}}$$

- References

$$\frac{\Gamma \vdash e : T}{\Gamma \vdash \mathsf{ref}\ e : T\ \mathsf{ref}}$$

- De-References

$$\frac{\Gamma \vdash e : T\ \mathsf{ref}}{\Gamma \vdash !e : T}$$

- Assignments

$$\frac{\Gamma \vdash e_1 : T\ \mathsf{ref} \qquad \Gamma \vdash e_2 : T}{\Gamma \vdash e_1 := e_2 : \mathsf{unit}}$$

Interesting Example

- Consider the expression:

$$\begin{array}{l} \text{let } r = \text{ref } \lambda x.x \text{ in} \\ \quad \text{let } u = (r := \lambda y.(\text{ref } !y)) \text{ in} \\ \quad\ !r\ () \end{array}$$

- this expression has type unit since
  - $\emptyset \vdash \text{ref } \lambda x.x : (Y \rightarrow Y) \text{ ref}$
  - $\{r : \forall\{Y\}.(Y \rightarrow Y) \text{ ref}\}$ $\vdash r := \lambda y.(\text{ref } !y) : \text{unit}$
  - $\{r : \forall\{Y\}.(Y \rightarrow Y) \text{ ref}\} \vdash !r\ () : \text{unit}$

Interesting Example

- Consider the expression:

$$\begin{array}{l} \mathsf{let}\ r = \mathsf{ref}\ \lambda x.x\ \mathsf{in} \\ \quad \mathsf{let}\ u = (r := \lambda y.(\mathsf{ref}\ !y))\ \mathsf{in} \\ \quad !r\ () \end{array}$$

- this expression has type unit since
  - $\varnothing \vdash \mathsf{ref}\ \lambda x.x : (Y \to Y)\ \mathsf{ref}$
  - $\{r : \forall\{Y\}.(Y \to Y)\ \mathsf{ref}\}$
  - $\{r :$

So the typing-system does not complain!!

During Run-Time

$$\begin{array}{l} \text{let } r = \text{ref } \lambda x.x \text{ in} \\ \quad \text{let } u = (r := \lambda y.(\text{ref } !y)) \text{ in} \\ \quad\quad !r\ () \end{array}$$

During Run-Time

$$\text{let } r = \text{ref } \lambda x.x \text{ in}$$
$$\quad \text{let } u = (r := \lambda y.(\text{ref } !y)) \text{ in}$$
$$\quad\quad !r\ ()$$

Strore:

$$\boxed{\lambda x.x}$$

During Run-Time

let r = ref $\lambda x.x$ in
  let u = (r := $\lambda y.$(ref $!y$)) in
    $!r$ ()

Strore:

$r \;=\; \boxed{\lambda x.x}$

During Run-Time

$$\text{let } u = (r := \lambda y.(\text{ref } !y)) \text{ in}$$
$$!r\ ()$$

Strore:

$r = \boxed{\lambda x.x}$

During Run-Time

$$\mathsf{let}\ u = (r := \lambda y.(\mathsf{ref}\ !y))\ \mathsf{in}$$
$$!r\ ()$$

Strore:

$r = \boxed{\lambda x.x}$

During Run-Time

$$\text{let } u = (r := \lambda y.(\text{ref } !y)) \text{ in}$$
$$!r\ ()$$

Strore:

$$r \;=\; \boxed{\lambda y.(\text{ref } !y)}$$

During Run-Time

$$\text{let } u = (r := \lambda y.(\text{ref } !y)) \text{ in}$$
$$!r\ ()$$

Strore:

$$r = \boxed{\lambda y.(\text{ref } !y)}$$
$$u = \text{unit}$$

During Run-Time

$$!r\ ()$$

Strore:

$$r \;=\; \boxed{\lambda y.(\mathsf{ref}\ !y)}$$

$$u \;=\; \mathsf{unit}$$

During Run-Time

$$!r\ ()$$

Strore:

$$r = \boxed{\lambda y.(\mathsf{ref}\ !y)}$$

$$u = \mathsf{unit}$$

During Run-Time

$$!r\ ()$$

Strore:

$$r = \boxed{\lambda y.(\mathsf{ref}\ !y)}$$

$$u = \mathsf{unit}$$

During Run-Time

$$\lambda y.(\mathsf{ref}\ !y)\ ()$$

Strore:

$$r \;=\; \boxed{\lambda y.(\mathsf{ref}\ !y)}$$

$$u \;=\; \mathsf{unit}$$

During Run-Time

(ref !())

Strore:

$r \;=\; \boxed{\lambda y.(\mathsf{ref}\ !y)}$

$u \;=\; \mathsf{unit}$

During Run-Time

(ref !())

Strore:

$r = \boxed{\lambda y.(\mathsf{ref}\ !y)}$

$u = \mathsf{unit}$

During Run-Time

$$(\mathsf{ref}\ !()) \Rightarrow$$ ↯ ☠ 👎 †

Strore:

$$r \;=\; \boxed{\lambda y.(\mathsf{ref}\ !y)}$$

$$u \;=\; \mathsf{unit}$$

Restoring Safety I

- The rule for Lets gets restricted

$$\frac{\Gamma \vdash e_1 : T_1 \quad x : \forall A.T_1, \Gamma \vdash e_2 : T_2 \quad x \notin \mathsf{dom}\,\Gamma}{\Gamma \vdash \mathsf{let}\, x = e_1 \,\mathsf{in}\, e_2 : T_2}$$

where

$$A = \begin{cases} \{\} & \text{if } e_1 \text{ is not a value} \\ \mathsf{tv}(T_1) - \mathsf{ftv}(\Gamma) & \text{if } e_1 \text{ is a value} \end{cases}$$

- Values

$$V ::= x \mid \lambda x.e \mid ()$$

Restoring Safety II

- With the restricted rule some programs (not involving refs) that were typable beforehand, are not typable any longer.
  Is this a problem?

- Well, Wright checked 1995 approximately 400,000 lines of SML code and found that in practice the restriction does not cause any trouble. After that, the question how to solve the problem with type-safety was settled.

Can We Be Sure?

- Can we be sure to have safety with the restricted system?
- Well, the answer lies in a formal proof:
  - We have to define a transition relation for configurations $\langle e, s \rangle$:

$$\langle e, s \rangle \longrightarrow \langle e', s' \rangle$$

  - Show that typing is preserved under transitions.
  - Show that well-typed expressions cannot get stuck.

Next Week

- Next week we have a look at a version of the Damas-Milner typing algorithm, which provides better error-messages when a program is not typable.
- Also, many modern typing algorithms are formulated as a constraint solving system (we take a look at them). This technique generalises relatively easily to other type systems.

Possible Question

- Given the typing judgements we have defined for Mini-ML, show that if $\varnothing \vdash e : T$ is derivable, them e must be closed.

- Hint: Show by rule induction that for all derivable typing judgements, $\Gamma \vdash e : T$, we have $\mathsf{fv}(e) \subseteq \mathsf{dom}\,\Gamma$.

More Next Week

- Slides at the end of

  `http://www4.in.tum.de/lehre/vorlesungen/types/WS0607/`

  There is also an appraisal form where you can complain anonymously.

- You can say whether the lecture was too easy, too quiet, too hard to follow, too chaotic and so on. You can also comment on things I should repeat.