Functional Programming

WS 2010/11

Christian Sternagel (VO)
Friedrich Neurauter (PS) Ulrich Kastlunger (PS)

Computational Logic
Institute of Computer Science
University of Innsbruck

December 15, 2010

Today's Topics

- Type Checking
- Unification
- Type Inference

Type Checking

Problem - Type Checking

input:	expression e and type τ
output:	YES (e has type τ) or NO

Problem - Type Checking

input:	expression e and type τ
output:	YES (e has type τ) or NO

The Language of Expressions - Core FP

$$
\begin{array}{rcll}
e & \overset{\text{def}}{=} & x \mid e\ e \mid \lambda x.\, e & \lambda\text{-calculus} \\
 & \mid & c & \text{constant (for primitives)} \\
 & \mid & \textbf{let}\ x = e\ \textbf{in}\ e & \text{let binding} \\
 & \mid & \textbf{if}\ e\ \textbf{then}\ e\ \textbf{else}\ e & \text{conditional}
\end{array}
$$

Problem - Type Checking

input:	expression e and type τ
output:	YES (e has type τ) or NO

The Language of Expressions - Core FP

$$
\begin{array}{rcll}
e & \overset{\text{def}}{=} & x \mid e\ e \mid \lambda x.\, e & \lambda\text{-calculus} \\
 & \mid & c & \text{constant (for primitives)} \\
 & \mid & \textbf{let}\ x = e\ \textbf{in}\ e & \text{let binding} \\
 & \mid & \textbf{if}\ e\ \textbf{then}\ e\ \textbf{else}\ e & \text{conditional}
\end{array}
$$

Primitives

- used for predefined “functions” and “constants”
- **Boolean:** True, False, $<$, $>$, . . .
- **arithmetic:** $\times$, $+$, $\div$, $-$, 0, 1, . . .
- **tuples:** Pair, fst, snd
- **lists:** Nil, Cons, head, tail

What is Type Checking?

Given some environment (assigning types to primitives) together with a core FP expression and a type, check whether the expression is of the given type with respect to the environment.

Types

- type variables, α, α_0, α_1, $\ldots$

Types

- type variables, α, α_0, α_1, . . .
- the function type constructor $\longrightarrow$

Types

- type variables, α, α_0, α_1, ...
- the function type constructor $\longrightarrow$
- type constructors C, C_1, ... (like: List)

Types

- type variables, α, α_0, α_1, ...
- the function type constructor $\rightarrow$
- type constructors C, C_1, ... (like: List)
- types $\tau \overset{\text{def}}{=} \alpha \mid \tau \rightarrow \tau \mid C(\tau, \ldots, \tau)$

Types

- type variables, α, α_0, α_1, ...
- the function type constructor $\rightarrow$
- type constructors C, C_1, ... (like: List)
- types $\tau \overset{\text{def}}{=} \alpha \mid \tau \rightarrow \tau \mid C(\tau, \ldots, \tau)$
- special case: base types: Int, Bool (instead of Int(), Bool())

Types

- type variables, α, α_0, α_1, ...
- the function type constructor $\rightarrow$
- type constructors C, C_1, ... (like: List)
- types $\tau \overset{\text{def}}{=} \alpha \mid \tau \rightarrow \tau \mid C(\tau, \ldots, \tau)$
- special case: base types: Int, Bool (instead of Int(), Bool())

Example Types

- List(Bool) - list of Booleans
- $\text{Pair}(\text{Int}, \text{Int})$ - pairs of integers
- $\text{Int} \rightarrow \text{Int} \rightarrow \text{Bool}$ - functions from two integers to Boolean

Typing Environments

- set of pairs E, mapping variables and primitives to types
- instead of $(e, \tau) \in E$, we write $e :: \tau \in E$

Typing Environments

- set of pairs E, mapping variables and primitives to types
- instead of $(e, \tau) \in E$, we write $e :: \tau \in E$

Typing Judgments

- $E \vdash e :: \tau$
- read: "it can be proved that e is of type τ under E"

Typing Environments

- set of pairs E, mapping variables and primitives to types
- instead of $(e, \tau) \in E$, we write $e :: \tau \in E$

Typing Judgments

- $E \vdash e :: \tau$
- read: “it can be proved that e is of type τ under E”

Examples

- primitive environment
  $P = \{+ :: \mathsf{Int} \to \mathsf{Int} \to \mathsf{Int}, \mathsf{Nil} :: \mathsf{List}(\alpha), \mathsf{True} :: \mathsf{Bool}, \ldots\}$
- $P \vdash \mathsf{True} :: \mathsf{Bool}$ - “using the primitive environment, it can be shown that True is of type Bool”

Type Substitutions

- mapping σ from type variables to types
- apply substitution to type

$$\tau\sigma \overset{\text{def}}{=} \begin{cases} \sigma(\alpha) & \text{if } \tau = \alpha \\ \tau_1\sigma \rightarrow \tau_2\sigma & \text{if } \tau = \tau_1 \rightarrow \tau_2 \\ C(\tau_1\sigma, \ldots, \tau_n\sigma) & \text{if } \tau = C(\tau_1, \ldots, \tau_n) \end{cases}$$

- composition $\sigma_1\sigma_2 \overset{\text{def}}{=} \sigma_2 \circ \sigma_1$ (where $\circ$ is function composition)

Type Substitutions

- mapping σ from type variables to types
- apply substitution to type

$$\tau\sigma \overset{\text{def}}{=} \begin{cases} \sigma(\alpha) & \text{if } \tau = \alpha \\ \tau_1\sigma \rightarrow \tau_2\sigma & \text{if } \tau = \tau_1 \rightarrow \tau_2 \\ C(\tau_1\sigma, \ldots, \tau_n\sigma) & \text{if } \tau = C(\tau_1, \ldots, \tau_n) \end{cases}$$

- composition $\sigma_1\sigma_2 \overset{\text{def}}{=} \sigma_2 \circ \sigma_1$ (where $\circ$ is function composition)

Examples

- $\sigma_1 = \{\alpha_1/\mathsf{List}(\alpha_2), \alpha_2/\mathsf{Bool}\}$
- $\sigma_2 = \{\alpha_2/\mathsf{Int}\}$
- $\sigma_1\sigma_2 = \sigma_2 \circ \sigma_1 = \{\alpha_1/\mathsf{List}(\mathsf{Int}), \alpha_2/\mathsf{Bool}\}$

Type Checking as Natural Deduction Rules

$$\frac{e :: \tau \in E}{e :: \tau\sigma}\ (\textsf{ins}) \qquad \frac{e_1 :: \tau_2 \to \tau_1 \qquad e_2 :: \tau_2}{e_1\ e_2 :: \tau_1}\ (\textsf{app})$$

$$\frac{\boxed{\begin{array}{c} x :: \tau_1 \\ \vdots \\ e :: \tau_2 \end{array}}}{\lambda x.\, e :: \tau_1 \to \tau_2}\ (\textsf{abs}) \qquad \frac{e_1 :: \tau_1 \qquad \boxed{\begin{array}{c} x :: \tau_1 \\ \vdots \\ e_2 :: \tau_2 \end{array}}}{\textbf{let}\ x = e_1\ \textbf{in}\ e_2 :: \tau_2}\ (\textsf{let})$$

$$\frac{e_1 :: \mathsf{Bool} \qquad e_2 :: \tau \qquad e_3 :: \tau}{\textbf{if}\ e_1\ \textbf{then}\ e_2\ \textbf{else}\ e_3 :: \tau}\ (\textsf{ite})$$

Example

- environment $E = \{\mathsf{True} :: \mathsf{Bool}, + :: \mathsf{Int} \to \mathsf{Int} \to \mathsf{Int}\}$
- prove judgment $E \vdash (\lambda x.\, x)\ \mathsf{True} :: \mathsf{Bool}$

1	$\mathsf{True} :: \mathsf{Bool}$	ins E
2	$x :: \mathsf{Bool}$	assumption
3	$\lambda x.\, x :: \mathsf{Bool} \to \mathsf{Bool}$	abs 2
4	$(\lambda x.\, x)\ \mathsf{True} :: \mathsf{Bool}$	app 3, 1

Example

- environment $E = \{\text{True} :: \text{Bool}, + :: \text{Int} \to \text{Int} \to \text{Int}\}$
- prove judgment $E \vdash (\lambda x.\, x)\ \text{True} :: \text{Bool}$

1	$\text{True} :: \text{Bool}$	ins E
2	$x :: \text{Bool}$	assumption
3	$\lambda x.\, x :: \text{Bool} \to \text{Bool}$	abs 2
4	$(\lambda x.\, x)\ \text{True} :: \text{Bool}$	app 3, 1

Example

- prove $E \vdash \lambda x.\, x + x :: \text{Int} \to \text{Int}$

1	$x :: \text{Int}$	assumption
2	$+ :: \text{Int} \to \text{Int} \to \text{Int}$	ins E
3	$(+)\ x :: \text{Int} \to \text{Int}$	app 2, 1
4	$x + x :: \text{Int}$	app 3, 1
5	$\lambda x.\, x + x :: \text{Int} \to \text{Int}$	abs 1–4

Unification

Problem - Unification

input:	equation $\tau_1 \approx \tau_2$
output:	substitution (σ s.t. $\tau_1\sigma = \tau_2\sigma$) or FAILURE

Problem - Unification

a pair of types

input: equation $\tau_1 \approx \tau_2$

output: substitution $(\sigma$ s.t. $\tau_1\sigma = \tau_2\sigma)$ or FAILURE

Problem - Unification

input: equation $\tau_1 \approx \tau_2$

output: substitution (σ s.t. $\tau_1\sigma = \tau_2\sigma$) or FAILURE

syntactic equality

Problem - Unification

input:	equation $\tau_1 \approx \tau_2$
output:	substitution $(\sigma$ s.t. $\tau_1\sigma = \tau_2\sigma\,)$ or FAILURE

Notions

Problem - Unification

input: equation $\tau_1 \approx \tau_2$

output: substitution (σ s.t. $\tau_1\sigma = \tau_2\sigma$) or FAILURE

Notions

- equation $\tau \approx \tau'$ is satisfiable iff exists σ s.t., $\tau\sigma = \tau'\sigma$

Problem - Unification

input: equation $\tau_1 \approx \tau_2$

output: substitution $(\sigma$ s.t. $\tau_1\sigma = \tau_2\sigma)$ or FAILURE

Notions

- equation $\tau \approx \tau'$ is satisfiable iff exists σ s.t., $\tau\sigma = \tau'\sigma$
- σ is called solution of $\tau \approx \tau'$

Problem - Unification

input: equation $\tau_1 \approx \tau_2$

output: substitution (σ s.t. $\tau_1\sigma = \tau_2\sigma$) or FAILURE

Notions

- equation $\tau \approx \tau'$ is satisfiable iff exists σ s.t., $\tau\sigma = \tau'\sigma$
- σ is called solution of $\tau \approx \tau'$
- unification problem is finite sequence of equations

$$\tau_1 \approx \tau'_1; \ldots; \tau_n \approx \tau'_n$$

Problem - Unification

input: equation $\tau_1 \approx \tau_2$

output: substitution $(\sigma$ s.t. $\tau_1\sigma = \tau_2\sigma)$ or FAILURE

Notions

- equation $\tau \approx \tau'$ is satisfiable iff exists σ s.t., $\tau\sigma = \tau'\sigma$
- σ is called solution of $\tau \approx \tau'$
- unification problem is finite sequence of equations

$$\tau_1 \approx \tau'_1; \ldots; \tau_n \approx \tau'_n$$

- $\Box$ denotes empty sequence

Problem - Unification

input: equation $\tau_1 \approx \tau_2$
output: substitution (σ s.t. $\tau_1\sigma = \tau_2\sigma$) or FAILURE

Notions

- equation $\tau \approx \tau'$ is satisfiable iff exists σ s.t., $\tau\sigma = \tau'\sigma$
- σ is called solution of $\tau \approx \tau'$
- unification problem is finite sequence of equations

$$\tau_1 \approx \tau_1'; \ldots; \tau_n \approx \tau_n'$$

- $\square$ denotes empty sequence
- unification - solving given unification problem

Problem - Unification

input: equation $\tau_1 \approx \tau_2$

output: substitution (σ s.t. $\tau_1\sigma = \tau_2\sigma$) or FAILURE

Notions

- equation $\tau \approx \tau'$ is satisfiable iff exists σ s.t., $\tau\sigma = \tau'\sigma$
- σ is called solution of $\tau \approx \tau'$
- unification problem is finite sequence of equations

$$\tau_1 \approx \tau_1'; \ldots ; \tau_n \approx \tau_n'$$

- $\square$ denotes empty sequence
- unification - solving given unification problem
- type variables

$$\mathcal{TV}\mathsf{ar}(\tau) \overset{\text{def}}{=} \begin{cases} \{\alpha\} & \text{if } \tau = \alpha \\ \mathcal{TV}\mathsf{ar}(\tau_1) \cup \mathcal{TV}\mathsf{ar}(\tau_2) & \text{if } \tau = \tau_1 \to \tau_2 \\ \bigcup_{1 \le i \le n} \mathcal{TV}\mathsf{ar}(\tau_i) & \text{if } \tau = C(\tau_1, \ldots, \tau_n) \end{cases}$$

Unification Rules

$$\frac{E_1; C(\tau_1, \ldots, \tau_n) \approx C(\tau_1', \ldots, \tau_n'); E_2}{E_1; \tau_1 \approx \tau_1'; \ldots; \tau_n \approx \tau_n'; E_2}\ (\mathsf{d}_1)$$

$$\frac{E_1; \tau_1 \rightarrow \tau_2 \approx \tau_1' \rightarrow \tau_2'; E_2}{E_1; \tau_1 \approx \tau_1'; \tau_2 \approx \tau_2'; E_2}\ (\mathsf{d}_2)$$

$$\frac{E_1; \alpha \approx \tau; E_2 \quad \alpha \notin \mathcal{TV}\mathsf{ar}(\tau)}{(E_1; E_2)\{\alpha/\tau\}}\ (\mathsf{v}_1)$$

$$\frac{E_1; \tau \approx \alpha; E_2 \quad \alpha \notin \mathcal{TV}\mathsf{ar}(\tau)}{(E_1; E_2)\{\alpha/\tau\}}\ (\mathsf{v}_2)$$

$$\frac{E_1; \tau \approx \tau; E_2}{E_1; E_2}\ (\mathsf{t})$$

Example

$$\begin{aligned} \mathsf{List}(\mathsf{Bool}) \approx \mathsf{List}(\alpha) & \Rightarrow^{(\mathsf{d}_1)}_{\{\}} && \mathsf{Bool} \approx \alpha \\ & \Rightarrow^{(\mathsf{v}_2)}_{\{\alpha/\mathsf{Bool}\}} && \square \end{aligned}$$

Type Inference

What is Type Inference?

Given some environment together with a core FP expression and a type, infer a solution (i.e., type substitution)—if possible—such that applying the substitution to the initial type yields the most general type of the initial expression.

Type Inference Problems

- $E \rhd e :: \tau$
- read: "try to infer most general substitution σ such that $E \vdash e :: \tau\sigma$"

Type Inference Problems

- $E \rhd e :: \tau$
- read: "try to infer most general substitution σ such that $E \vdash e :: \tau\sigma$"

Example

- $E = \{0 :: \mathsf{Int}\}$
- $E \rhd \textbf{let } \mathit{id} = \lambda x.\, x \textbf{ in } \mathit{id}\ 0 :: \alpha_0$
- $\sigma = \{\alpha_0/\mathsf{Int}\}$

1	$x :: \mathsf{Int}$	assumption
2	$\lambda x.\, x :: \mathsf{Int} \to \mathsf{Int}$	abs 1
3	$\mathit{id} :: \mathsf{Int} \to \mathsf{Int}$	assumption
4	$0 :: \mathsf{Int}$	ins E
5	$\mathit{id}\ 0 :: \mathsf{Int}$	app 3, 4
6	$\textbf{let } \mathit{id} = \lambda x.\, x \textbf{ in } \mathit{id}\ 0 :: \mathsf{Int}$	let 2, 3–5

Typing Constraint Rules

$$\frac{E, e :: \tau_0 \rhd e :: \tau_1}{\tau_0 \approx \tau_1}\,(\mathsf{con}) \qquad \frac{E \rhd e_1\ e_2 :: \tau}{E \rhd e_1 :: \alpha \rightarrow \tau; E \rhd e_2 :: \alpha}\,(\mathsf{app})$$

$$\frac{E \rhd \lambda x.\, e :: \tau}{E, x :: \alpha_1 \rhd e :: \alpha_2; \tau \approx \alpha_1 \rightarrow \alpha_2}\,(\mathsf{abs}) \qquad \frac{E \rhd \textbf{let}\ x = e_1\ \textbf{in}\ e_2 :: \tau}{E \rhd e_1 :: \alpha; E, x :: \alpha \rhd e_2 :: \tau}\,(\mathsf{let})$$

$$\frac{E \rhd \textbf{if}\ e_1\ \textbf{then}\ e_2\ \textbf{else}\ e_3 :: \tau}{E \rhd e_1 :: \mathsf{Bool}; E \rhd e_2 :: \tau; E \rhd e_3 :: \tau}\,(\mathsf{ite})$$

Recipe - Type Inference

- to find most general type of e under E

Recipe - Type Inference

- to find most general type of e under E
- first, take $E \triangleright e :: \alpha_0$ (for fresh type variable α_0)

Recipe - Type Inference

- to find most general type of e under E
- first, take $E \rhd e :: \alpha_0$ (for fresh type variable α_0)
- then, use typing constraint rules to generate unification problem u (if at any point no rule applicable Not Typable)

Recipe - Type Inference

- to find most general type of e under E
- first, take $E \rhd e :: \alpha_0$ (for fresh type variable α_0)
- then, use typing constraint rules to generate unification problem u (if at any point no rule applicable Not Typable)
- if u has no solution (none of the rules is applicable before reaching $\square$) then Not Typable, otherwise, solve u obtaining solution σ

Recipe - Type Inference

- to find most general type of e under E
- first, take $E \triangleright e :: \alpha_0$ (for fresh type variable α_0)
- then, use typing constraint rules to generate unification problem u (if at any point no rule applicable Not Typable)
- if u has no solution (none of the rules is applicable before reaching $\Box$) then Not Typable, otherwise, solve u obtaining solution σ
- finally, $\alpha_0\sigma$ is the most general type of e

Exercise

find most general type of **let** $id = \lambda x.\, x$ **in** id 0 w.r.t. P

Exercises (for January 14th)

1. Read the lecture notes about type checking and type inference.
2. Check that $\textbf{if}\ \mathsf{True}\ \textbf{then}\ x + 1\ \textbf{else}\ x - 1$ is of type Int under $P \cup \{x :: \mathsf{Int}\}$.
3. Give a proof of $\varnothing \vdash \lambda xy.\, x :: \alpha_0 \to \alpha_1 \to \alpha_0$.
4. Solve the unification problem $\mathsf{Pair}(\mathsf{Bool}, \alpha_0) \approx \mathsf{Pair}(\alpha_1, \mathsf{Int})$.
5. Show that the unification problem $\mathsf{Pair}(\mathsf{Bool}, \alpha_0) \approx \mathsf{Pair}(\alpha_0, \mathsf{Int})$ does not have a solution.
6. Infer the most general type of $\textbf{let}\ \mathit{suc} = \lambda x.\, x + 1\ \textbf{in}\ \textbf{let}\ d = \lambda x.\, \mathit{suc}\ (\mathit{suc}\ x)\ \textbf{in}\ d\ 2$ under P.