Computational Logic

Functional Programming

Christian Sternagel Harald Zankl Evgeny Zuenko

Department of Computer Science
University of Innsbruck

WS 2017/2018

Lecture 10

Topics

abstract data types, algebraic data types, binary search trees, combinator parsing, efficiency, encoding data types as lambda-terms, evaluation strategies, formal verification, first steps, guarded recursion, Haskell introduction, higher-order functions, historical overview, implementing a type checker, induction, infinite data structures, input and output, lambda-calculus, lazy evaluation, list comprehensions, lists, modules, pattern matching, polymorphism, property-based testing, reasoning about functional programs, recursive functions, sets, strings, tail recursion, trees, tupling, type checking, type inference, types, types and type classes, unification, user-defined types

Topics

abstract data types, algebraic data types, binary search trees, combinator parsing, efficiency, encoding data types as lambda-terms, evaluation strategies, formal verification, first steps, guarded recursion, Haskell introduction, higher-order functions, historical overview, implementing a type checker, induction, infinite data structures, input and output, lambda-calculus, lazy evaluation, list comprehensions, lists, modules, pattern matching, polymorphism, property-based testing, reasoning about functional programs, recursive functions, sets, strings, tail recursion, trees, tupling, type checking, type inference, types, types and type classes, unification, user-defined types

Overview

- 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

e	$::=$	$x \mid e\ e \mid \lambda x.\, e$	λ-terms
	$\mid$	c	constant (for primitives)
	$\mid$	**let** $x = e$ **in** e	let-binding
	$\mid$	**if** e **then** e **else** e	conditional branching

Problem – Type Checking

input: expression e and type τ

output: YES (e has type τ) or NO

The Language of Expressions – Core FP

e	$::=$	$x \mid e\ e \mid \lambda x.\, e$	λ-terms
	$\mid$	c	constant (for primitives)
	$\mid$	**let** $x = e$ **in** e	let-binding
	$\mid$	**if** e **then** e **else** e	conditional branching

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 a (typing) environment (mapping variables to types)

What is Type Checking?

Given a (typing) environment (mapping variables to types), a core FP expression

What is Type Checking?

Given a (typing) environment (mapping variables to types), a core FP expression, and a type

What is Type Checking?

Given a (typing) environment (mapping variables to types), 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, ...

Types

- type variables, α, α_0, α_1, ...
- the (right-associative) function space constructor $\rightarrow$

Types

- type variables, α, α_0, α_1, ...
- the (right-associative) function space constructor $\rightarrow$
- type constructors C, C_1, ... (like List, for the type of lists)

Types

- type variables, α, α_0, α_1, ...
- the (right-associative) function space constructor $\rightarrow$
- type constructors C, C_1, ... (like List, for the type of lists)
- types $\tau ::= \alpha \mid \tau \rightarrow \tau \mid C(\tau, \ldots, \tau)$

Types

- type variables, α, α_0, α_1, ...
- the (right-associative) function space constructor $\rightarrow$
- type constructors C, C_1, ... (like List, for the type of lists)
- types $\tau ::= \alpha \mid \tau \rightarrow \tau \mid C(\tau, \ldots, \tau)$
- special case – base types: Int, Bool (instead of Int(), Bool())

Types

- type variables, α, α_0, α_1, ...
- the (right-associative) function space constructor $\rightarrow$
- type constructors C, C_1, ... (like List, for the type of lists)
- types $\tau ::= \alpha \mid \tau \rightarrow \tau \mid C(\tau, \ldots, \tau)$
- special case – base types: Int, Bool (instead of Int(), Bool())

Example Types

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

Typing Environments

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

Typing Environments

- set of pairs E, mapping variables and constants to types
- instead of $(e, \tau) \in E$, 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 constants to types
- instead of $(e, \tau) \in E$, 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
  $P = \{\mathsf{True} :: \mathsf{Bool}, + :: \mathsf{Int} \to \mathsf{Int} \to \mathsf{Int}, \mathsf{Nil} :: \mathsf{List}(\alpha), \ldots\}$
- $P \vdash \mathsf{True} :: \mathsf{Bool}$ – "using the primitive environment, it can be shown that True is of type Bool"

Type Substitutions

- mapping σ from type variables to types

Type Substitutions

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

$$\alpha\sigma = \sigma(\alpha)$$
$$(\tau_1 \to \tau_2)\sigma = \tau_1\sigma \to \tau_2\sigma$$
$$C(\tau_1, \ldots, \tau_n)\sigma = C(\tau_1\sigma, \ldots, \tau_n\sigma)$$

Type Substitutions

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

$$\alpha\sigma = \sigma(\alpha)$$
$$(\tau_1 \to \tau_2)\sigma = \tau_1\sigma \to \tau_2\sigma$$
$$C(\tau_1, \ldots, \tau_n)\sigma = C(\tau_1\sigma, \ldots, \tau_n\sigma)$$

- composition $\sigma_1\sigma_2 = (\lambda x.\, \sigma_1(x)\sigma_2)$ ("first apply σ_1 and then σ_2")

Type Substitutions

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

$$\alpha\sigma = \sigma(\alpha)$$
$$(\tau_1 \to \tau_2)\sigma = \tau_1\sigma \to \tau_2\sigma$$
$$C(\tau_1, \ldots, \tau_n)\sigma = C(\tau_1\sigma, \ldots, \tau_n\sigma)$$

- composition $\sigma_1\sigma_2 = (\lambda x.\, \sigma_1(x)\sigma_2)$ ("first apply σ_1 and then σ_2")

Examples

- $\sigma_1 = \{\alpha_1 \mapsto \mathsf{List}(\alpha_2), \alpha_2 \mapsto \mathsf{Bool}\}$
- $\sigma_2 = \{\alpha_2 \mapsto \mathsf{Int}, \alpha_3 \mapsto \mathsf{Int}\}$
- $\sigma_1\sigma_2 = \{\alpha_1 \mapsto \mathsf{List}(\mathsf{Int}), \alpha_2 \mapsto \mathsf{Bool}, \alpha_3 \mapsto \mathsf{Int}\}$

Type Checking as Natural Deduction Rules

$$\frac{e :: \tau \in E}{e :: \tau\sigma}\ \text{(ins)}$$

$$\frac{e_1 :: \tau_2 \to \tau_1 \qquad e_2 :: \tau_2}{e_1\ e_2 :: \tau_1}\ \text{(app)}$$

$$\frac{\boxed{\begin{array}{c} x :: \tau_1 \\ \vdots \\ e :: \tau_2 \end{array}}}{\lambda x.\, e :: \tau_1 \to \tau_2}\ \text{(abs)}$$

$$\frac{e_1 :: \tau_1 \qquad \boxed{\begin{array}{c} x :: \tau_1 \\ \vdots \\ e_2 :: \tau_2 \end{array}}}{\textbf{let}\ x = e_1\ \textbf{in}\ e_2 :: \tau_2}\ \text{(let)}$$

$$\frac{e :: \tau}{e :: \tau}\ \text{(copy)}$$

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

Examples

- environment $E = \{\mathsf{True} :: \mathsf{Bool}, + :: \mathsf{Int} \to \mathsf{Int} \to \mathsf{Int}\}$

Examples

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

Examples

- 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

Examples

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

Examples

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

Unification

Problem – Unification

input: equation $\tau_1 \approx \tau_2$

output: most general unifier σ such that $\tau_1\sigma = \tau_2\sigma$ or FAILURE

Problem – Unification

pair of types

input: equation $\tau_1 \approx \tau_2$

output: most general unifier σ such that $\tau_1\sigma = \tau_2\sigma$ or FAILURE

Problem – Unification

input: equation $\tau_1 \approx \tau_2$

output: most general unifier σ such that $\tau_1\sigma = \tau_2\sigma$ or FAILURE

a type substitution

Problem – Unification

input: equation $\tau_1 \approx \tau_2$

output: most general unifier σ such that $\tau_1\sigma = \tau_2\sigma$ or FAILURE

syntactic equality

Problem – Unification

input: equation $\tau_1 \approx \tau_2$

output: most general unifier σ such that $\tau_1\sigma = \tau_2\sigma$ or FAILURE

Notions

Problem – Unification

input: equation $\tau_1 \approx \tau_2$

output: most general unifier σ such that $\tau_1\sigma = \tau_2\sigma$ or FAILURE

Notions

- equation $\tau \approx \tau'$ is satisfiable iff exists σ such that $\tau\sigma = \tau'\sigma$

Problem – Unification

input: equation $\tau_1 \approx \tau_2$

output: most general unifier σ such that $\tau_1\sigma = \tau_2\sigma$ or FAILURE

Notions

- equation $\tau \approx \tau'$ is satisfiable iff exists σ such that $\tau\sigma = \tau'\sigma$
- σ is called solution of $\tau \approx \tau'$

Problem – Unification

input: equation $\tau_1 \approx \tau_2$

output: most general unifier σ such that $\tau_1\sigma = \tau_2\sigma$ or FAILURE

Notions

- equation $\tau \approx \tau'$ is satisfiable iff exists σ such that $\tau\sigma = \tau'\sigma$
- 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: most general unifier σ such that $\tau_1\sigma = \tau_2\sigma$ or FAILURE

Notions

- equation $\tau \approx \tau'$ is satisfiable iff exists σ such that $\tau\sigma = \tau'\sigma$
- 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

Problem – Unification

input: equation $\tau_1 \approx \tau_2$

output: most general unifier σ such that $\tau_1\sigma = \tau_2\sigma$ or FAILURE

Notions

- equation $\tau \approx \tau'$ is satisfiable iff exists σ such that $\tau\sigma = \tau'\sigma$
- 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: most general unifier σ such that $\tau_1\sigma = \tau_2\sigma$ or FAILURE

Notions

- equation $\tau \approx \tau'$ is satisfiable iff exists σ such that $\tau\sigma = \tau'\sigma$
- 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
- set of type variables of a type

$$\mathcal{V}(\alpha) = \{\alpha\}$$
$$\mathcal{V}(\tau_1 \to \tau_2) = \mathcal{V}(\tau_1) \cup \mathcal{V}(\tau_2)$$
$$\mathcal{V}(C(\tau_1, \ldots, \tau_n)) = \mathcal{V}(\tau_1) \cup \ldots \cup \mathcal{V}(\tau_n)$$

Unification Rules

(d) decomposition

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

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

Unification Rules

(d) decomposition

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

$$\frac{E_1; \tau_1 \to \tau_2 \approx \tau'_1 \to \tau'_2; E_2}{E_1; \tau_1 \approx \tau'_1; \tau_2 \approx \tau'_2; E_2} \ (\mathsf{d}_2)$$

(t) removal of trivial equations

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

Unification Rules

(d) decomposition

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

$$\frac{E_1; \tau_1 \to \tau_2 \approx \tau'_1 \to \tau'_2; E_2}{E_1; \tau_1 \approx \tau'_1; \tau_2 \approx \tau'_2; E_2} \ (\mathsf{d}_2)$$

(t) removal of trivial equations

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

(v) variable elimination

$$\frac{E_1; \alpha \approx \tau; E_2 \quad \alpha \notin \mathcal{V}(\tau)}{(E_1; E_2)\{\alpha \mapsto \tau\}} \ (\mathsf{v}_1) \qquad \frac{E_1; \tau \approx \alpha; E_2 \quad \alpha \notin \mathcal{V}(\tau)}{(E_1; E_2)\{\alpha \mapsto \tau\}} \ (\mathsf{v}_2)$$

Applying Unification Rules

- notation: $E \Rightarrow_{\sigma}^{(r)} E'$

Applying Unification Rules

- notation: $E \Rightarrow_{\sigma}^{(r)} E'$
- meaning: apply rule r to derive E' from E using substitution σ

Applying Unification Rules

- notation: $E \Rightarrow_{\sigma}^{(r)} E'$
- meaning: apply rule r to derive E' from E using substitution σ
- (if r is not variable elimination, σ is empty substitution)

Applying Unification Rules

- notation: $E \Rightarrow_{\sigma}^{(r)} E'$
- meaning: apply rule r to derive E' from E using substitution σ
- (if r is not variable elimination, σ is empty substitution)
- resulting substitution of sequence

$$E_1 \Rightarrow_{\sigma_1}^{(r_1)} E_2 \Rightarrow_{\sigma_2}^{(r_2)} E_3 \Rightarrow_{\sigma_3}^{(r_3)} \cdots \Rightarrow_{\sigma_{n-1}}^{(r_{n-1})} E_n$$

is composition

$$\sigma_1 \sigma_2 \cdots \sigma_{n-1}$$

Applying Unification Rules

- notation: $E \Rightarrow_{\sigma}^{(r)} E'$
- meaning: apply rule r to derive E' from E using substitution σ
- (if r is not variable elimination, σ is empty substitution)
- resulting substitution of sequence

$$E_1 \Rightarrow_{\sigma_1}^{(r_1)} E_2 \Rightarrow_{\sigma_2}^{(r_2)} E_3 \Rightarrow_{\sigma_3}^{(r_3)} \cdots \Rightarrow_{\sigma_{n-1}}^{(r_{n-1})} E_n$$

is composition

$$\sigma_1 \sigma_2 \cdots \sigma_{n-1}$$

Example

$$\mathsf{List}(\mathsf{Bool}) \approx \mathsf{List}(\alpha)$$

Applying Unification Rules

- notation: $E \Rightarrow_{\sigma}^{(r)} E'$
- meaning: apply rule r to derive E' from E using substitution σ
- (if r is not variable elimination, σ is empty substitution)
- resulting substitution of sequence

$$E_1 \Rightarrow_{\sigma_1}^{(r_1)} E_2 \Rightarrow_{\sigma_2}^{(r_2)} E_3 \Rightarrow_{\sigma_3}^{(r_3)} \cdots \Rightarrow_{\sigma_{n-1}}^{(r_{n-1})} E_n$$

is composition

$$\sigma_1 \sigma_2 \cdots \sigma_{n-1}$$

Example

$$\mathsf{List}(\mathsf{Bool}) \approx \mathsf{List}(\alpha) \quad \Rightarrow_{\{\}}^{(\mathsf{d}_1)}$$

Applying Unification Rules

- notation: $E \Rightarrow_{\sigma}^{(r)} E'$
- meaning: apply rule r to derive E' from E using substitution σ
- (if r is not variable elimination, σ is empty substitution)
- resulting substitution of sequence

$$E_1 \Rightarrow_{\sigma_1}^{(r_1)} E_2 \Rightarrow_{\sigma_2}^{(r_2)} E_3 \Rightarrow_{\sigma_3}^{(r_3)} \cdots \Rightarrow_{\sigma_{n-1}}^{(r_{n-1})} E_n$$

is composition

$$\sigma_1 \sigma_2 \cdots \sigma_{n-1}$$

Example

$$\mathsf{List}(\mathsf{Bool}) \approx \mathsf{List}(\alpha) \quad \Rightarrow_{\{\}}^{(\mathsf{d}_1)} \quad \mathsf{Bool} \approx \alpha$$

Applying Unification Rules

- notation: $E \Rightarrow_{\sigma}^{(r)} E'$
- meaning: apply rule r to derive E' from E using substitution σ
- (if r is not variable elimination, σ is empty substitution)
- resulting substitution of sequence

$$E_1 \Rightarrow_{\sigma_1}^{(r_1)} E_2 \Rightarrow_{\sigma_2}^{(r_2)} E_3 \Rightarrow_{\sigma_3}^{(r_3)} \cdots \Rightarrow_{\sigma_{n-1}}^{(r_{n-1})} E_n$$

is composition

$$\sigma_1 \sigma_2 \cdots \sigma_{n-1}$$

Example

$$\mathsf{List}(\mathsf{Bool}) \approx \mathsf{List}(\alpha) \Rightarrow_{\{\}}^{(\mathsf{d}_1)} \mathsf{Bool} \approx \alpha \Rightarrow^{(\mathsf{v}_2)}$$

Applying Unification Rules

- notation: $E \Rightarrow_{\sigma}^{(r)} E'$
- meaning: apply rule r to derive E' from E using substitution σ
- (if r is not variable elimination, σ is empty substitution)
- resulting substitution of sequence

$$E_1 \Rightarrow_{\sigma_1}^{(r_1)} E_2 \Rightarrow_{\sigma_2}^{(r_2)} E_3 \Rightarrow_{\sigma_3}^{(r_3)} \cdots \Rightarrow_{\sigma_{n-1}}^{(r_{n-1})} E_n$$

is composition

$$\sigma_1 \sigma_2 \cdots \sigma_{n-1}$$

Example

$$\mathsf{List}(\mathsf{Bool}) \approx \mathsf{List}(\alpha) \Rightarrow_{\{\}}^{(\mathsf{d}_1)} \mathsf{Bool} \approx \alpha \Rightarrow_{\{\alpha \mapsto \mathsf{Bool}\}}^{(\mathsf{v}_2)}$$

Applying Unification Rules

- notation: $E \Rightarrow_{\sigma}^{(r)} E'$
- meaning: apply rule r to derive E' from E using substitution σ
- (if r is not variable elimination, σ is empty substitution)
- resulting substitution of sequence

$$E_1 \Rightarrow_{\sigma_1}^{(r_1)} E_2 \Rightarrow_{\sigma_2}^{(r_2)} E_3 \Rightarrow_{\sigma_3}^{(r_3)} \cdots \Rightarrow_{\sigma_{n-1}}^{(r_{n-1})} E_n$$

is composition

$$\sigma_1 \sigma_2 \cdots \sigma_{n-1}$$

Example

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

Type Inference

What is Type Inference?

Given an environment (assigning types to primitives)

What is Type Inference?

Given an environment (assigning types to primitives), a core FP expression

What is Type Inference?

Given an environment (assigning types to primitives), a core FP expression, and an initial type

What is Type Inference?

Given an environment (assigning types to primitives), a core FP expression, and an initial type, try to infer a solution (that is, type substitution) such that applying it to the initial type yields the most general compatible 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

- $E = \{0 :: \mathsf{Int}\}$

Type Inference Problems

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

Example

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

Type Inference Problems

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

Example

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

Type Inference Problems

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

Example

- $E = \{0 :: \mathsf{Int}\}$
- $E \rhd \textbf{let}\ id = \lambda x.\, x\ \textbf{in}\ id\ 0 :: \alpha_0$
- $\sigma = \{\alpha_0 \mapsto \mathsf{Int}\}$
1	$x :: \mathsf{Int}$	assumption
2	$\lambda x.\, x :: \mathsf{Int} \to \mathsf{Int}$	abs 1
3	$id :: \mathsf{Int} \to \mathsf{Int}$	assumption
4	$0 :: \mathsf{Int}$	ins E
5	$id\ 0 :: \mathsf{Int}$	app 3, 4
6	$\textbf{let}\ id = \lambda x.\, x\ \textbf{in}\ id\ 0 :: \mathsf{Int}$	let 2, 3–5

Typing Constraint Rules

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

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

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

Typing Constraint Rules

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

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

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

where α in (app) and (let), and α_1 and α_2 in (abs) are fresh (do not occur in premises of rule)

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 $E \rhd e :: \alpha_0$ (for some 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 some 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 some 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 unification rules is applicable before reaching $\square$) then Not Typable, otherwise, obtain solution σ

Recipe – Type Inference

- to find most general type of e under E
- first take $E \rhd e :: \alpha_0$ (for some 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 unification rules is applicable before reaching $\square$) then Not Typable, otherwise, obtain solution σ
- finally, $\alpha_0\sigma$ is most general type of e

Exercise

- find most general type of **let** $id = \lambda x.\, x$ **in** $id\ 1$ with respect to P

Exercise

- find most general type of **let** $id = \lambda x.\, x$ **in** $id\ 1$ with respect to P
- start from $P \rhd$ **let** $id = \lambda x.\, x$ **in** $id\ 1 :: \alpha_0$

Exercise

- find most general type of **let** $id = \lambda x.\, x$ **in** $id\ 1$ with respect to P
- start from $P \rhd$ **let** $id = \lambda x.\, x$ **in** $id\ 1 :: \alpha_0$

$$\frac{P \rhd \textbf{let } id = \lambda x.\, x \textbf{ in } id\ 1 :: \alpha_0}{\frac{P \rhd \lambda x.\, x :: \alpha_1;}{P, id :: \alpha_1 \rhd id\ 1 :: \alpha_0}}\ \Rightarrow^{(\text{let})}$$

$$\frac{\alpha_1 \approx \alpha_2 \to \alpha_3;\ \frac{P, x :: \alpha_2 \rhd x :: \alpha_3;}{P, id :: \alpha_1 \rhd id\ 1 :: \alpha_0}}{\alpha_1 \approx \alpha_2 \to \alpha_3;\ \alpha_2 \approx \alpha_3;\ \frac{P, id :: \alpha_1 \rhd id\ 1 :: \alpha_0}{\alpha_1 \approx \alpha_2 \to \alpha_3;\ \alpha_2 \approx \alpha_3;\ \frac{P, id :: \alpha_1 \rhd id :: \alpha_4 \to \alpha_0;}{P, id :: \alpha_1 \rhd 1 :: \alpha_4}}}\ \Rightarrow^{(\text{abs})}\ \Rightarrow^{(\text{con})}\ \Rightarrow^{(\text{app})}$$

$$\alpha_1 \approx \alpha_2 \to \alpha_3;\ \alpha_2 \approx \alpha_3;\ \alpha_1 \approx \alpha_4 \to \alpha_0;\ \mathsf{Int} \approx \alpha_4\ \Rightarrow^{(\text{con})^2}$$

Exercise

- find most general type of **let** $id = \lambda x.\, x$ **in** $id\ 1$ with respect to P
- start from $P \rhd$ **let** $id = \lambda x.\, x$ **in** $id\ 1 :: \alpha_0$

$$
\begin{array}{ll}
 & \underline{P \rhd \textbf{let}\ id = \lambda x.\, x\ \textbf{in}\ id\ 1 :: \alpha_0} \\
\Rightarrow^{(\mathsf{let})} & \underline{P \rhd \lambda x.\, x :: \alpha_1;} \\
 & P, id :: \alpha_1 \rhd id\ 1 :: \alpha_0 \\
\Rightarrow^{(\mathsf{abs})} & \alpha_1 \approx \alpha_2 \to \alpha_3; \\
 & \underline{P, x :: \alpha_2 \rhd x :: \alpha_3;} \\
 & P, id :: \alpha_1 \rhd id\ 1 :: \alpha_0 \\
\Rightarrow^{(\mathsf{con})} & \alpha_1 \approx \alpha_2 \to \alpha_3; \alpha_2 \approx \alpha_3; \\
 & \underline{P, id :: \alpha_1 \rhd id\ 1 :: \alpha_0} \\
\Rightarrow^{(\mathsf{app})} & \alpha_1 \approx \alpha_2 \to \alpha_3; \alpha_2 \approx \alpha_3; \\
 & \underline{P, id :: \alpha_1 \rhd id :: \alpha_4 \to \alpha_0;} \\
 & \underline{P, id :: \alpha_1 \rhd 1 :: \alpha_4} \\
\Rightarrow^{(\mathsf{con})^2} & \alpha_1 \approx \alpha_2 \to \alpha_3; \alpha_2 \approx \alpha_3; \\
 & \alpha_1 \approx \alpha_4 \to \alpha_0; \underline{\mathsf{Int} \approx \alpha_4}
\end{array}
$$

$$
\begin{array}{l}
\alpha_1 \approx \alpha_2 \to \alpha_3; \underline{\alpha_2 \approx \alpha_3;} \\
\alpha_1 \approx \alpha_4 \to \alpha_0; \underline{\mathsf{Int} \approx \alpha_4} \\
\Rightarrow^{(\mathsf{v})^2}_{\{\alpha_2 \mapsto \alpha_3, \alpha_4 \mapsto \mathsf{Int}\}} \\
\underline{\alpha_1 \approx \alpha_3 \to \alpha_3}; \alpha_1 \approx \mathsf{Int} \to \alpha_0 \\
\Rightarrow^{(\mathsf{v})}_{\{\alpha_1 \mapsto \alpha_3 \to \alpha_3\}} \\
\underline{\alpha_3 \to \alpha_3 \approx \mathsf{Int} \to \alpha_0} \\
\Rightarrow^{(\mathsf{d})} \\
\\
\underline{\alpha_3 \approx \mathsf{Int}}; \underline{\alpha_3 \approx \alpha_0} \\
\Rightarrow^{(\mathsf{v})^2}_{\{\alpha_0 \mapsto \mathsf{Int}, \alpha_3 \mapsto \mathsf{Int}\}} \\
\square
\end{array}
$$

Exercise

- find most general type of **let** $id = \lambda x.\, x$ **in** $id\ 1$ with respect to P
- start from $P \rhd$ **let** $id = \lambda x.\, x$ **in** $id\ 1 :: \alpha_0$

$$
\begin{array}{ll}
 & \underline{P \rhd \textbf{let } id = \lambda x.\, x \textbf{ in } id\ 1 :: \alpha_0} \\
\Rightarrow^{(\mathsf{let})} & \underline{P \rhd \lambda x.\, x :: \alpha_1;} \\
 & P, id :: \alpha_1 \rhd id\ 1 :: \alpha_0 \\
\Rightarrow^{(\mathsf{abs})} & \alpha_1 \approx \alpha_2 \to \alpha_3; \\
 & \underline{P, x :: \alpha_2 \rhd x :: \alpha_3;} \\
 & P, id :: \alpha_1 \rhd id\ 1 :: \alpha_0 \\
\Rightarrow^{(\mathsf{con})} & \alpha_1 \approx \alpha_2 \to \alpha_3; \alpha_2 \approx \alpha_3; \\
 & \underline{P, id :: \alpha_1 \rhd id\ 1 :: \alpha_0} \\
\Rightarrow^{(\mathsf{app})} & \alpha_1 \approx \alpha_2 \to \alpha_3; \alpha_2 \approx \alpha_3; \\
 & \underline{P, id :: \alpha_1 \rhd id :: \alpha_4 \to \alpha_0;} \\
 & \underline{P, id :: \alpha_1 \rhd 1 :: \alpha_4} \\
\Rightarrow^{(\mathsf{con})^2} & \alpha_1 \approx \alpha_2 \to \alpha_3; \alpha_2 \approx \alpha_3; \\
 & \alpha_1 \approx \alpha_4 \to \alpha_0; \mathsf{Int} \approx \alpha_4
\end{array}
$$

$$
\begin{array}{l}
\alpha_1 \approx \alpha_2 \to \alpha_3; \underline{\alpha_2 \approx \alpha_3;} \\
\alpha_1 \approx \alpha_4 \to \alpha_0; \underline{\mathsf{Int} \approx \alpha_4} \\
\Rightarrow^{(\mathsf{v})^2}_{\{\alpha_2 \mapsto \alpha_3, \alpha_4 \mapsto \mathsf{Int}\}} \\
\underline{\alpha_1 \approx \alpha_3 \to \alpha_3}; \alpha_1 \approx \mathsf{Int} \to \alpha_0 \\
\Rightarrow^{(\mathsf{v})}_{\{\alpha_1 \mapsto \alpha_3 \to \alpha_3\}} \\
\underline{\alpha_3 \to \alpha_3 \approx \mathsf{Int} \to \alpha_0} \\
\Rightarrow^{(\mathsf{d})} \\
\underline{\alpha_3 \approx \mathsf{Int}; \alpha_3 \approx \alpha_0} \\
\Rightarrow^{(\mathsf{v})^2}_{\{\alpha_0 \mapsto \mathsf{Int}, \alpha_3 \mapsto \mathsf{Int}\}} \\
\square
\end{array}
$$

mgu: $\{\alpha_0 \mapsto \mathsf{Int}, \alpha_1 \mapsto \mathsf{Int} \to \mathsf{Int}, \alpha_2 \mapsto \mathsf{Int}, \alpha_3 \mapsto \mathsf{Int}, \alpha_4 \mapsto \mathsf{Int}\}$

Exercises (for January 19th)

1. Read the lecture notes about type checking and type inference.
2. Check that **if** True **then** $x + 1$ **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 **let** $suc = \lambda x.\, x + 1$ **in let** $d = \lambda x.\, suc\ (suc\ x)$ **in** $d\ 2$ under P.