

# 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

# Overview

- Type Checking
- Unification
- Type Inference

## Problem – Type Checking

input: expression  $e$  and type  $\tau$

output: YES ( $e$  has type  $\tau$ ) or NO

## The Language of Expressions – Core FP

$e ::=$	$x \mid e e \mid \lambda x. e$	$\lambda$ -terms
	$c$	constant (for primitives)
	<b>let</b> $x = e$ <b>in</b> $e$	let-binding
	<b>if</b> $e$ <b>then</b> $e$ <b>else</b> $e$	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), a core FP *expression*, and a *type*, check whether the expression is of the given type with respect to the environment.

## Types

- **type variables**,  $\alpha, \alpha_0, \alpha_1, \dots$
- the (right-associative) **function space constructor**  $\rightarrow$
- **type constructors**  $C, C_1, \dots$  (like List, for the type of lists)
- **types**  $\tau ::= \alpha \mid \tau \rightarrow \tau \mid C(\tau, \dots, \tau)$
- 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 Judgments

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

## Examples

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

## Type Substitutions

- **mapping**  $\sigma$  from type variables to types
- **apply substitution** to type

$$\alpha\sigma = \sigma(\alpha)$$

$$(\tau_1 \rightarrow \tau_2)\sigma = \tau_1\sigma \rightarrow \tau_2\sigma$$

$$C(\tau_1, \dots, \tau_n)\sigma = C(\tau_1\sigma, \dots, \tau_n\sigma)$$

- **composition**  $\sigma_1\sigma_2 = (\lambda x. \sigma_1(x)\sigma_2)$  (“first apply  $\sigma_1$  and then  $\sigma_2$ ”)

## Examples

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



## Type Checking as Natural Deduction Rules

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

$$\frac{e_1 :: \tau_2 \rightarrow \tau_1 \quad e_2 :: \tau_2}{e_1 e_2 :: \tau_1} \text{ (app)}$$

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

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

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

$$\frac{e_1 :: \mathbf{Bool} \quad e_2 :: \tau \quad e_3 :: \tau}{\mathbf{if } e_1 \mathbf{ then } e_2 \mathbf{ else } e_3 :: \tau} \text{ (ite)}$$

## Examples

- environment  $E = \{\text{True} :: \text{Bool}, + :: \text{Int} \rightarrow \text{Int} \rightarrow \text{Int}\}$

- prove judgment  $E \vdash (\lambda x. x) \text{ True} :: \text{Bool}$

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

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

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

**Problem – Unification**

pair of types

input: equation  $\tau_1 \approx \tau_2$ 

syntactic equality

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

a type substitution

- equation  $\tau \approx \tau'$  is **satisfiable** iff exists  $\sigma$  such that  $\tau\sigma = \tau'\sigma$
- $\sigma$  is called **solution** of  $\tau \approx \tau'$
- **unification problem** is finite sequence of equations

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

- $\square$  denotes empty sequence
- **unification** – solving given unification problem
- **set of type variables** of a type

$$\mathcal{V}(\alpha) = \{\alpha\}$$

$$\mathcal{V}(\tau_1 \rightarrow \tau_2) = \mathcal{V}(\tau_1) \cup \mathcal{V}(\tau_2)$$

$$\mathcal{V}(C(\tau_1, \dots, \tau_n)) = \mathcal{V}(\tau_1) \cup \dots \cup \mathcal{V}(\tau_n)$$

# Unification Rules

(d) **decomposition**

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

$$\frac{E_1; \tau_1 \rightarrow \tau_2 \approx \tau'_1 \rightarrow \tau'_2; E_2}{E_1; \tau_1 \approx \tau'_1; \tau_2 \approx \tau'_2; E_2} \quad (\text{d}_2)$$

(t) removal of **trivial** equations

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

(v) **variable** elimination

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

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

## Applying Unification Rules

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

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

is composition

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

### Example

$$\begin{array}{l} \text{List}(\text{Bool}) \approx \text{List}(\alpha) \Rightarrow_{\{\}}^{(d_1)} \text{Bool} \approx \alpha \\ \Rightarrow_{\{\alpha \mapsto \text{Bool}\}}^{(v_2)} \square \end{array}$$

## 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

- $E \triangleright e :: \tau$
- read: “try to infer most general substitution  $\sigma$  such that  $E \vdash e :: \tau\sigma$ ”

### Example

- $E = \{0 :: \text{Int}\}$
- $E \triangleright \mathbf{let\ } id = \lambda x. x \mathbf{\ in\ } id\ 0 :: \alpha_0$
- $\sigma = \{\alpha_0 \mapsto \text{Int}\}$

• 1	$x :: \text{Int}$	assumption
2	$\lambda x. x :: \text{Int} \rightarrow \text{Int}$	abs 1
3	$id :: \text{Int} \rightarrow \text{Int}$	assumption
4	$0 :: \text{Int}$	ins $E$
5	$id\ 0 :: \text{Int}$	app 3, 4
6	$\mathbf{let\ } id = \lambda x. x \mathbf{\ in\ } id\ 0 :: \text{Int}$	let 2, 3–5

## Typing Constraint Rules

$$\frac{E, e :: \tau_0 \triangleright e :: \tau_1}{\tau_0 \approx \tau_1} \text{ (con)}$$

$$\frac{E \triangleright e_1 e_2 :: \tau}{E \triangleright e_1 :: \alpha \rightarrow \tau; E \triangleright e_2 :: \alpha} \text{ (app)}$$

$$\frac{E \triangleright \lambda x. e :: \tau}{\tau \approx \alpha_1 \rightarrow \alpha_2; E, x :: \alpha_1 \triangleright e :: \alpha_2} \text{ (abs)}$$

$$\frac{E \triangleright \mathbf{let} x = e_1 \mathbf{in} e_2 :: \tau}{E \triangleright e_1 :: \alpha; E, x :: \alpha \triangleright e_2 :: \tau} \text{ (let)}$$

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

where  $\alpha$  in (app) and (let), and  $\alpha_1$  and  $\alpha_2$  in (abs) are **fresh** (do not occur in premises of rule)



## Recipe – Type Inference

- to find most general type of  $e$  under  $E$
- first take  $E \triangleright e :: \alpha_0$  (for some fresh type variable  $\alpha_0$ )
- 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  $\sigma$
- 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  $P$
- start from  $P \triangleright \mathbf{let}\ id = \lambda x. x\ \mathbf{in}\ id\ 1 :: \alpha_0$

$$\begin{array}{l}
 \Rightarrow^{(\text{let})} \frac{P \triangleright \mathbf{let}\ id = \lambda x. x\ \mathbf{in}\ id\ 1 :: \alpha_0}{P \triangleright \lambda x. x :: \alpha_1;} \quad \alpha_1 \approx \alpha_2 \rightarrow \alpha_3; \underline{\alpha_2 \approx \alpha_3}; \\
 \frac{P, id :: \alpha_1 \triangleright id\ 1 :: \alpha_0}{\alpha_1 \approx \alpha_4 \rightarrow \alpha_0; \underline{\text{Int} \approx \alpha_4}} \\
 \Rightarrow^{(\text{abs})} \frac{\alpha_1 \approx \alpha_2 \rightarrow \alpha_3;}{P, x :: \alpha_2 \triangleright x :: \alpha_3;} \quad \Rightarrow^{(\text{v})^2} \\
 \frac{P, id :: \alpha_1 \triangleright id\ 1 :: \alpha_0}{\alpha_1 \approx \alpha_3 \rightarrow \alpha_3; \alpha_1 \approx \text{Int} \rightarrow \alpha_0} \\
 \Rightarrow^{(\text{con})} \frac{\alpha_1 \approx \alpha_2 \rightarrow \alpha_3; \alpha_2 \approx \alpha_3;}{P, id :: \alpha_1 \triangleright id\ 1 :: \alpha_0} \quad \Rightarrow^{(\text{v})} \\
 \frac{\alpha_3 \rightarrow \alpha_3 \approx \text{Int} \rightarrow \alpha_0}{\alpha_3 \approx \text{Int}; \underline{\alpha_3 \approx \alpha_0}} \\
 \Rightarrow^{(\text{app})} \frac{\alpha_1 \approx \alpha_2 \rightarrow \alpha_3; \alpha_2 \approx \alpha_3;}{P, id :: \alpha_1 \triangleright id :: \alpha_4 \rightarrow \alpha_0;} \quad \Rightarrow^{(\text{v})^2} \\
 \frac{P, id :: \alpha_1 \triangleright 1 :: \alpha_4}{\alpha_3 \approx \text{Int}; \underline{\alpha_3 \approx \alpha_0}} \quad \Rightarrow^{(\text{v})^2} \\
 \Rightarrow^{(\text{con})^2} \frac{\alpha_1 \approx \alpha_2 \rightarrow \alpha_3; \alpha_2 \approx \alpha_3;}{\alpha_1 \approx \alpha_4 \rightarrow \alpha_0; \underline{\text{Int} \approx \alpha_4}} \quad \square \\
 \text{mgu: } \{ \alpha_0 \mapsto \text{Int}, \alpha_1 \mapsto \text{Int} \rightarrow \text{Int}, \\
 \alpha_2 \mapsto \text{Int}, \alpha_3 \mapsto \text{Int}, \alpha_4 \mapsto \text{Int} \}
 \end{array}$$

## 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  $P \cup \{x :: \text{Int}\}$ .
3. Give a proof of  $\emptyset \vdash \lambda xy. x :: \alpha_0 \rightarrow \alpha_1 \rightarrow \alpha_0$ .
4. Solve the unification problem  $\text{Pair}(\text{Bool}, \alpha_0) \approx \text{Pair}(\alpha_1, \text{Int})$ .
5. Show that the unification problem  $\text{Pair}(\text{Bool}, \alpha_0) \approx \text{Pair}(\alpha_0, \text{Int})$  does not have a solution.
6. Infer the most general type of **let**  $\text{suc} = \lambda x. x + 1$  **in let**  $d = \lambda x. \text{suc} (\text{suc } x)$  **in**  $d\ 2$  under  $P$ .