



# Functional Programming

## Lecture 10

Cezary Kaliszyk   Jonas Schöpf  
Christian Sternagel   Vincent van Oostrom

Department of Computer Science

## 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, 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

CK,JS,CS,VvO (DCS @ UIBK)

lecture 10

2/23

Type Checking

## Overview

- Type Checking
- Unification and its Implementation
- Type Inference

## Problem – Type Checking

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

## Haskell Examples

```
(\x -> x) True :: Bool
```

```
double :: Int -> Int
double x = x + x
```

## 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

### 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 primitive environment, it can be shown that True is of type Bool”

## Types

- **type variables**  $\alpha, \alpha_0, \alpha_1, \dots$
- (right-associative) **function space constructor**  $\rightarrow$
- **type constructors**  $C, C_0, C_1, \dots$  (like List for 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
- $\text{Int} \rightarrow \text{Int} \rightarrow \text{Bool}$  – functions from two integers to Booleans

### A Type for Types

```
data Type = TVar Int           – for easy renaming of type variables
          | TCon String [Type] – type constructors and function type
          deriving Eq          – type equality can be checked
```

### Type Substitutions

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

$$\begin{aligned} \alpha\sigma &= \sigma(\alpha) \\ (\tau_1 \rightarrow \tau_2)\sigma &= \tau_1\sigma \rightarrow \tau_2\sigma \\ C(\tau_1, \dots, \tau_n)\sigma &= C(\tau_1\sigma, \dots, \tau_n\sigma) \end{aligned}$$

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

### Examples

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

domain

## Implementing Type Substitutions

```
type TSub = [(Int, Type)]
```

### “Extracting” Maybes with Default

- `Data.Maybe.fromMaybe :: a -> Maybe a -> a`
- satisfies equations `fromMaybe x Nothing = x`  
`fromMaybe x (Just y) = y`

### Applying Substitutions to Types

```
tsub :: TSub -> Type -> Type
tsub s (x@(TVar i)) = fromMaybe x $ lookup i s
tsub s (TCon g ts) = TCon g (map (tsub s) ts)
```

### Composition

```
tcomp :: TSub -> TSub -> TSub
s1 `tcomp` s2 =
  map (\(x, t) -> (x, tsub s2 t)) s1 ++
  filter ((`notElem` dom1) . fst) s2
  where dom1 = map fst s1
```

### Examples

- environment  $E = \{\text{True} :: \text{Bool}, + :: \text{Int} \rightarrow \text{Int} \rightarrow \text{Int}\}$
- recall  $(\lambda x \rightarrow x) \text{ True} :: \text{Bool}$
- prove judgment  $E \vdash (\lambda x. x) \text{ True} :: \text{Bool}$
- 1 

<code>True :: Bool</code>	<code>ins E</code>
<code>x :: Bool</code>	<code>assumption</code>
<code>λx. x :: Bool → Bool</code>	<code>abs 2</code>
<code>(λx. x) True :: Bool</code>	<code>app 3, 1</code>
- recall `double :: Int -> Int`  
`double x = x + x`
- prove  $E \vdash \lambda x. x + x :: \text{Int} \rightarrow \text{Int}$
- 1 

<code>x :: Int</code>	<code>assumption</code>
<code>+ :: Int → Int → Int</code>	<code>ins E</code>
<code>(+) x :: Int → Int</code>	<code>app 2, 1</code>
<code>x + x :: Int</code>	<code>app 3, 1</code>
<code>λx. x + x :: Int → Int</code>	<code>abs 1-4</code>

## Type Checking as Natural Deduction Rules

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

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

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

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

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

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

### Problem – Unification

input: equation  $\tau_1 \approx \tau_2$  (pair of types)

output: most general unifier (mgu)  $\sigma$  with  $\tau_1\sigma = \tau_2\sigma$  or FAILURE (syntactic equality)

### Notions

- equation  $\tau \approx \tau'$  is **satisfiable** iff exists  $\sigma$  such that  $\tau\sigma = \tau'\sigma$
- if  $\tau \approx \tau'$  satisfiable by  $\sigma$ , then  $\sigma$  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**

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

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

(t) removal of **trivial** equations

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

(v) **variable** elimination

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

## Unification.hs – Implementing Unification

### Interface

- input: unification problem `type UP = [(Type, Type)]`
- output: type substitution
- unification: `unify :: UP -> TSub`

### The Free Variables of a Type

```
tvars :: Type -> [Int]
tvars (TVar x) = [x]
tvars (TCon g ts) = foldr (union . tvars) [] ts
```

### Applying Type Substitutions to Unification Problems

```
tsubUP :: TSub -> UP -> UP
tsubUP s = map (\(l, r) -> (tsub s l, tsub s r))
```

## 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

$$E_1 \Rightarrow_{\sigma_1}^{(r_1)} E_2 \Rightarrow_{\sigma_2}^{(r_2)} E_3 \Rightarrow_{\sigma_3}^{(r_3)} \dots \Rightarrow_{\sigma_n}^{(r_n)} E_{n+1}$$

is composition

$$\sigma_1 \sigma_2 \dots \sigma_n$$

### Example

$$\text{List}(\text{Bool}) \approx \text{List}(\alpha) \xRightarrow{\{\}}^{(d_1)} \text{Bool} \approx \alpha$$

$$\xRightarrow{\{\alpha \mapsto \text{Bool}\}}^{(v_2)} \square$$

mgu:  $\{\}\{\alpha \mapsto \text{Bool}\} = \{\alpha \mapsto \text{Bool}\}$

## Unification

```
unify :: UP -> TSub
unify eqs = go [] eqs
  where
    go s [] = s
    go s (eq:eqs) =
      go (s `tcomp` s') (eqs' ++ tsubUP s' eqs)
      where
        (eqs', s') = step eq
    step :: (Type, Type) -> (UP, TSub)
    ...
```

### Removal of Trivial Equations

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

```
step (t, u) | t == u = ([], [])
```

### Variable Elimination

$$\frac{E_1; \alpha \approx \tau; E_2 \quad \alpha \notin \mathcal{V}(\tau)}{(E_1; E_2)\{\alpha \mapsto \tau\}} \quad (v_1) \qquad \frac{E_1; \tau \approx \alpha; E_2 \quad \alpha \notin \mathcal{V}(\tau)}{(E_1; E_2)\{\alpha \mapsto \tau\}} \quad (v_2)$$

```
step (TVar x, t) = singletonSub x t
step (t, TVar x) = singletonSub x t
where
  notUnif = error "not unifiable"
  singletonSub x t =
    if x `elem` tvars t then notUnif else ([], [(x, t)])
```

### Decomposition

$$\frac{E_1; C(\tau_1, \dots, \tau_n) \approx C(\tau'_1, \dots, \tau'_n); E_2}{E_1; \tau_1 \approx \tau'_1; \dots; \tau_n \approx \tau'_n; E_2} \quad (d_1) \qquad \frac{E_1; \tau_1 \rightarrow \tau_2 \approx \tau'_1 \rightarrow \tau'_2; E_2}{E_1; \tau_1 \approx \tau'_1; \tau_2 \approx \tau'_2; E_2} \quad (d_2)$$

```
step (TCon f ts, TCon g us) |
  f == g && length ts == length us = (zip ts us, [])
```

### Typing Constraint Rules

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

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

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

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

### Type Inference Problems

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

### Haskell Example

what is most general type of `let id = \x -> x in id 0`

### 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

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

## Homework (for January 25th)

- Read the [lecture notes about type checking and type inference](#).
- Use type checking to prove  $\{f :: \alpha \rightarrow \beta, y :: \alpha\} \vdash (\lambda x. f \ x) \ y :: \beta$ .
- Solve the unification problem  $\alpha_2 \approx \alpha_1 \rightarrow \alpha_0; \alpha_3 \approx \mathbf{Int} \rightarrow \alpha_2; \alpha_2 \approx \alpha_3$  and compute the resulting mgu, if possible. Check your result using [unify](#).
- Solve the unification problem  $\text{Pair}(\mathbf{Bool}, \alpha_0) \approx \text{Pair}(\alpha_1, \mathbf{Int})$ . Check your solution using [unify](#).
- Use type inference to obtain the most general type of the expression  $\text{map} \ (\lambda x. x)$  with respect to the environment  $E = \{\text{map} :: (\alpha \rightarrow \beta) \rightarrow \text{List}(\alpha) \rightarrow \text{List}(\beta)\}$ .
- Given the following types for environments and inference problems
 

```
type Env = [(String, Type)]
type IP = [(Env, Exp, Type)]
```

 implement a function `toUp :: IP -> UP` that transforms a given type inference problem into the corresponding unification problem according to our typing constraint rules.

## Exams

### 1st Exam

- 12:15–14:00 in HSB 1 on February 1, 2019
- online [registration](#) required before 23:59 on January 18, 2019

### 2nd Exam

- 12:15–14:00 in HSB 1 on March 1, 2019
- online [registration](#) required before 23:59 on February 15, 2019

### 3rd Exam

- 12:15–14:00 in HSB 1 on September 27, 2019
- online [registration](#) required before 23:59 on September 13, 2019

### Some Old Exams

- exam of February 2, 2018 + solutions
- exam of March 2, 2018 + solutions