

# Functional Programming

WS 2015/16

Cezary Kaliszyk (VO+PS)

Yann Savoye (PS)

Łukasz Czajka (PS)

Computational Logic  
Institute of Computer Science  
University of Innsbruck

week 10



# Overview

- Week 10 - Types
  - Summary of Week 9
  - Core ML
  - Type Checking
  - Type Inference



# Overview

- Week 10 - Types
  - Summary of Week 9
    - Core ML
    - Type Checking
    - Type Inference



# Efficiency of Functional Programs

Avoid unnecessary recomputations by ...

- tupling

Introduce tail recursion by ...

- parameter accumulation

# Overview

- Week 10 - Types
  - Summary of Week 9
  - Core ML
  - Type Checking
  - Type Inference



# This Week

## Practice I

OCaml introduction, lists, strings, trees

## Theory I

lambda-calculus, evaluation strategies, induction, reasoning about functional programs

## Practice II

efficiency, tail-recursion, combinator-parsing,

## Theory II

type checking, type inference

## Advanced Topics

lazy evaluation, infinite data structures, dependent types, monads

# Overview

- Week 10 - Types
  - Summary of Week 9
  - **Core ML**
  - Type Checking
  - Type Inference



# Core ML

## Definition (Expressions)

$$e ::= x \mid e e \mid \lambda x. e \mid c \mid \mathbf{let} \ x = e \ \mathbf{in} \ e \mid \mathbf{if} \ e \ \mathbf{then} \ e \ \mathbf{else} \ e$$



## Core ML

## Definition (Expressions)

$\lambda$ -Calculus

$$e ::= x \mid e e \mid \lambda x. e \mid c \mid \mathbf{let} \ x = e \ \mathbf{in} \ e \mid \mathbf{if} \ e \ \mathbf{then} \ e \ \mathbf{else} \ e$$

## Core ML

## Definition (Expressions)

$$e ::= x \mid e \ e \mid \lambda x. e \mid \underbrace{c}_{\text{primitives/constants}} \mid \mathbf{let} \ x = e \ \mathbf{in} \ e \mid \mathbf{if} \ e \ \mathbf{then} \ e \ \mathbf{else} \ e$$

# Core ML

## Definition (Expressions)

$$e ::= x \mid e e \mid \lambda x. e \mid c \mid \underbrace{\text{let } x = e \text{ in } e}_{\text{let binding}} \mid \text{if } e \text{ then } e \text{ else } e$$

# Core ML

## Definition (Expressions)

$e ::= x \mid e e \mid \lambda x. e \mid c \mid \mathbf{let} \ x = e \ \mathbf{in} \ e \mid \underbrace{\mathbf{if} \ e \ \mathbf{then} \ e \ \mathbf{else} \ e}_{\text{conditional}}$

# Core ML

## Definition (Expressions)

$$e ::= x \mid e \ e \mid \lambda x. e \mid c \mid \mathbf{let} \ x = e \ \mathbf{in} \ e \mid \mathbf{if} \ e \ \mathbf{then} \ e \ \mathbf{else} \ e$$

## Primitives

**Boolean:** true, false, <, >, ...

**Arithmetic:**  $\times$ , +,  $\div$ , -, 0, 1, ...

**Tuples:** pair, fst, snd

**Lists:** nil, cons, hd, tl

# Overview

- Week 10 - Types
  - Summary of Week 9
  - Core ML
  - **Type Checking**
  - Type Inference



# What is Type Checking?

Given some **environment** (assigning types to primitives) together with a core ML **expression** and a **type**, check whether the expression really has that type with respect to the environment.

# Preliminaries

## Definition (Types)

$$\tau ::= \underbrace{\alpha}_{\text{type variable}} \mid \tau \rightarrow \tau \mid g(\tau, \dots, \tau)$$

## Convention

- **type variables**  $\alpha, \alpha_0, \alpha_1, \dots, \beta, \beta_0, \dots$
- function type constructor ' $\rightarrow$ ' is right associative
- base data type constructors: `int, bool` (instead of `int()`, `bool()`)



# Preliminaries

## Definition (Types)

function type constructor

$$\tau ::= \alpha \mid \overbrace{\tau \rightarrow \tau} \mid g(\tau, \dots, \tau)$$

## Convention

- type variables  $\alpha, \alpha_0, \alpha_1, \dots, \beta, \beta_0, \dots$
- **function type** constructor ' $\rightarrow$ ' is right associative
- base data type constructors: `int`, `bool` (instead of `int()`, `bool()`)

# Preliminaries

## Definition (Types)

$$\tau ::= \alpha \mid \tau \rightarrow \tau \mid \underbrace{g(\tau, \dots, \tau)}_{\text{data type constructor}}$$

## Convention

- type variables  $\alpha, \alpha_0, \alpha_1, \dots, \beta, \beta_0, \dots$
- function type constructor ' $\rightarrow$ ' is right associative
- **base data type** constructors: `int`, `bool` (instead of `int()`, `bool()`)

# Preliminaries

## Definition (Types)

$$\tau ::= \alpha \mid \tau \rightarrow \tau \mid g(\tau, \dots, \tau)$$

## Convention

- type variables  $\alpha, \alpha_0, \alpha_1, \dots, \beta, \beta_0, \dots$
- function type constructor ' $\rightarrow$ ' is right associative
- base data type constructors: `int`, `bool` (instead of `int()`, `bool()`)

## Example

`int  $\rightarrow$  bool`, `(int  $\rightarrow$  list(int))  $\rightarrow$  bool`, `list( $\alpha_0$ )  $\rightarrow$  int`, ...

## Preliminaries (cont'd)

(Typing) environment  $E$ : maps (variables and) primitives to types  
 $(e : \tau) \in E$       “ $e$  is of type  $\tau$  in  $E$ ”

## Preliminaries (cont'd)

(Typing) environment  $E$ : maps (variables and) primitives to types

$e : \tau \in E$       “ $e$  is of type  $\tau$  in  $E$ ”

# Preliminaries (cont'd)

(Typing) environment  $E$ : maps (variables and) primitives to types

$e : \tau \in E$       “ $e$  is of type  $\tau$  in  $E$ ”

(Typing) judgment:

$E \vdash e : \tau$       “it can be *proved* that expression  $e$  has type  $\tau$  in environment  $E$ ”

# Preliminaries (cont'd)

(Typing) environment  $E$ : maps (variables and) primitives to types

$e : \tau \in E$       “ $e$  is of type  $\tau$  in  $E$ ”

(Typing) judgment:

$E \vdash e : \tau$       “it can be *proved* that expression  $e$  has type  $\tau$  in environment  $E$ ”

## Example

- environment  $P = \{+ : \text{int} \rightarrow \text{int} \rightarrow \text{int}, \text{nil} : \text{list}(\alpha), \text{true} : \text{bool}, \dots\}$
- judgement  $P \vdash \text{true} : \text{bool}$
- judgement  $P \not\vdash \text{true} : \text{int}$

# Preliminaries (cont'd)

(Typing) environment  $E$ : maps (variables and) primitives to types

$e : \tau \in E$       “ $e$  is of type  $\tau$  in  $E$ ”

(Typing) judgment:

$E \vdash e : \tau$       “it can be *proved* that expression  $e$  has type  $\tau$  in environment  $E$ ”

## Example

- environment  $P = \{+ : \text{int} \rightarrow \text{int} \rightarrow \text{int}, \text{nil} : \text{list}(\alpha), \text{true} : \text{bool}, \dots\}$
- judgement  $P \vdash \text{true} : \text{bool}$
- judgement  $P \not\vdash \text{true} : \text{int}$

## Convention

$E, e : \tau$  abbreviates  $E \cup \{e : \tau\}$



# The Type Checking System $\mathcal{C}$

$$\frac{}{E, e : \tau \vdash e : \tau} \text{ (ref)}$$

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

$$\frac{E, x : \tau_1 \vdash e : \tau_2}{E \vdash \lambda x. e : \tau_1 \rightarrow \tau_2} \text{ (abs)}$$

$$\frac{E \vdash e_1 : \tau_1 \quad E, x : \tau_1 \vdash e_2 : \tau_2}{E \vdash \mathbf{let} \ x = e_1 \ \mathbf{in} \ e_2 : \tau_2} \text{ (let)}$$

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

# Example

- environment  $E = \{\text{true} : \text{bool}, + : \text{int} \rightarrow \text{int} \rightarrow \text{int}\}$
- judgment  $E \vdash (\lambda x.x) \text{ true} : \text{bool}$

## Proof.

$$\frac{\frac{E, x : \text{bool} \vdash x : \text{bool}}{E \vdash \lambda x.x : \text{bool} \rightarrow \text{bool}} \text{ (abs)} \quad E \vdash \text{true} : \text{bool}}{E \vdash (\lambda x.x) \text{ true} : \text{bool}} \text{ (app)}$$



# Example

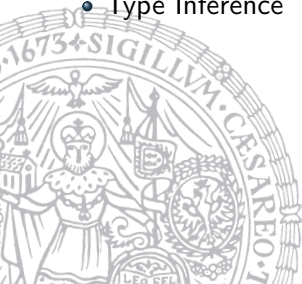
- environment  $E = \{\text{true} : \text{bool}, + : \text{int} \rightarrow \text{int} \rightarrow \text{int}\}$
- judgment  $E \vdash \lambda x.x + x : \text{int} \rightarrow \text{int}$

Proof.

Blackboard

# Overview

- Week 10 - Types
  - Summary of Week 9
  - Core ML
  - Type Checking
  - Type Inference



# What is Type Inference?

Given some **environment** together with a core ML **expression** and a **type**, infer a **unifier** (type substitution)—if possible—such that the **most general type** of the expression is obtained.

# Preliminaries

Type variables:

$$\mathcal{TVar}(\tau) \stackrel{\text{def}}{=} \begin{cases} \{\alpha\} & \text{if } \tau = \alpha \\ \mathcal{TVar}(\tau_1) \cup \mathcal{TVar}(\tau_2) & \text{if } \tau = \tau_1 \rightarrow \tau_2 \\ \bigcup_{1 \leq i \leq n} \mathcal{TVar}(\tau_i) & \text{if } \tau = g(\tau_1, \dots, \tau_n) \end{cases}$$

# Preliminaries

Type variables:

$$\mathcal{TVar}(\tau) \stackrel{\text{def}}{=} \begin{cases} \{\alpha\} & \text{if } \tau = \alpha \\ \mathcal{TVar}(\tau_1) \cup \mathcal{TVar}(\tau_2) & \text{if } \tau = \tau_1 \rightarrow \tau_2 \\ \bigcup_{1 \leq i \leq n} \mathcal{TVar}(\tau_i) & \text{if } \tau = g(\tau_1, \dots, \tau_n) \end{cases}$$

Type substitution:  $\sigma$  is mapping from type variables to types

# Preliminaries

Type variables:

$$\mathcal{TVar}(\tau) \stackrel{\text{def}}{=} \begin{cases} \{\alpha\} & \text{if } \tau = \alpha \\ \mathcal{TVar}(\tau_1) \cup \mathcal{TVar}(\tau_2) & \text{if } \tau = \tau_1 \rightarrow \tau_2 \\ \bigcup_{1 \leq i \leq n} \mathcal{TVar}(\tau_i) & \text{if } \tau = g(\tau_1, \dots, \tau_n) \end{cases}$$

Type substitution:  $\sigma$  is mapping from type variables to types

Application:

$$\tau\sigma \stackrel{\text{def}}{=} \begin{cases} \sigma(\alpha) & \text{if } \tau = \alpha \\ \tau_1\sigma \rightarrow \tau_2\sigma & \text{if } \tau = \tau_1 \rightarrow \tau_2 \\ g(\tau_1\sigma, \dots, \tau_n\sigma) & \text{if } \tau = g(\tau_1, \dots, \tau_n) \end{cases}$$

$$E\sigma \stackrel{\text{def}}{=} \{e : \tau\sigma \mid e : \tau \in E\}$$



# Preliminaries

Type variables:

$$\mathcal{TVar}(\tau) \stackrel{\text{def}}{=} \begin{cases} \{\alpha\} & \text{if } \tau = \alpha \\ \mathcal{TVar}(\tau_1) \cup \mathcal{TVar}(\tau_2) & \text{if } \tau = \tau_1 \rightarrow \tau_2 \\ \bigcup_{1 \leq i \leq n} \mathcal{TVar}(\tau_i) & \text{if } \tau = g(\tau_1, \dots, \tau_n) \end{cases}$$

Type substitution:  $\sigma$  is mapping from type variables to types

Application:

$$\tau\sigma \stackrel{\text{def}}{=} \begin{cases} \sigma(\alpha) & \text{if } \tau = \alpha \\ \tau_1\sigma \rightarrow \tau_2\sigma & \text{if } \tau = \tau_1 \rightarrow \tau_2 \\ g(\tau_1\sigma, \dots, \tau_n\sigma) & \text{if } \tau = g(\tau_1, \dots, \tau_n) \end{cases}$$

$$E\sigma \stackrel{\text{def}}{=} \{e : \tau\sigma \mid e : \tau \in E\}$$

Composition:  $\sigma_1\sigma_2 \stackrel{\text{def}}{=} \sigma_2 \circ \sigma_1$ , i.e.,  $\alpha \mapsto \sigma_2(\sigma_1(\alpha))$

## Example

$$\tau = \alpha \rightarrow (\alpha_1 \rightarrow \alpha_3)$$

$$\sigma = \{\alpha/\text{int} \rightarrow \text{int}, \alpha_1/\text{list}(\alpha_2)\}$$

$$\sigma_2 = \{\alpha_3/\alpha_4, \alpha_2/\alpha, \alpha/\alpha_1\}$$

$$\mathcal{TVar}(\tau) = \{\alpha, \alpha_1, \alpha_3\}$$

$$\tau\sigma = (\text{int} \rightarrow \text{int}) \rightarrow (\text{list}(\alpha_2) \rightarrow \alpha_3)$$

$$\mathcal{TVar}(\tau\sigma) = \{\alpha_2, \alpha_3\}$$

$$\sigma\sigma_2 = \{\alpha/\text{int} \rightarrow \text{int}, \alpha_1/\text{list}(\alpha), \alpha_3/\alpha_4, \alpha_2/\alpha\}$$

# Unification Problems

## Definition

- **unification problem** is (finite) sequence of equations

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

- $\square$  denotes empty sequence
- type substitution  $\sigma$  is unifier of unification problem if

$$\tau_1\sigma = \tau'_1\sigma; \dots; \tau_n\sigma = \tau'_n\sigma$$

- process of computing a unifier is called unification

# Unification Problems

## Definition

- unification problem is (finite) sequence of equations

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

- $\square$  denotes **empty sequence**
- type substitution  $\sigma$  is unifier of unification problem if

$$\tau_1\sigma = \tau'_1\sigma; \dots; \tau_n\sigma = \tau'_n\sigma$$

- process of computing a unifier is called unification

# Unification Problems

## Definition

- unification problem is (finite) sequence of equations

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

- $\square$  denotes empty sequence
- type substitution  $\sigma$  is **unifier** of unification problem if

$$\tau_1\sigma = \tau'_1\sigma; \dots; \tau_n\sigma = \tau'_n\sigma$$

- process of computing a unifier is called unification

# Unification Problems

## Definition

- unification problem is (finite) sequence of equations

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

- $\square$  denotes empty sequence
- type substitution  $\sigma$  is unifier of unification problem if

$$\tau_1\sigma = \tau'_1\sigma; \dots; \tau_n\sigma = \tau'_n\sigma$$

- process of computing a unifier is called **unification**

# The Unification System $\mathcal{U}$

$$\frac{E_1; g(\tau_1, \dots, \tau_n) \approx g(\tau'_1, \dots, \tau'_n); E_2}{E_1; \tau_1 \approx \tau'_1; \dots; \tau_n \approx \tau'_n; E_2} \quad (d_1)$$

$$\frac{E_1; \tau_1 \rightarrow \tau_2 \approx \tau'_1 \rightarrow \tau'_2; E_2}{E_1; \tau_1 \approx \tau'_1; \tau_2 \approx \tau'_2; E_2} \quad (d_2)$$

$$\frac{E_1; \alpha \approx \tau; E_2 \quad \alpha \notin \mathcal{TVar}(\tau)}{(E_1; E_2)\{\alpha/\tau\}} \quad (v_1)$$

$$\frac{E_1; \tau \approx \alpha; E_2 \quad \alpha \notin \mathcal{TVar}(\tau)}{(E_1; E_2)\{\alpha/\tau\}} \quad (v_2)$$

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

# Unification Problem (cont'd)

## Notation

$E \Rightarrow_{\sigma}^{(r)} E'$       if rule  $r$  from  $\mathcal{U}$  applied to equations  $E$  yields  $E'$



# Unification Problem (cont'd)

## Notation

$E \Rightarrow_{\sigma}^{(r)} E'$  if rule  $r$  from  $\mathcal{U}$  applied to equations  $E$  yields  $E'$

## Theorem

if  $E_1 \Rightarrow_{\sigma_1}^{(r_1)} E_2 \Rightarrow_{\sigma_2}^{(r_2)} \dots \Rightarrow_{\sigma_{n-1}}^{(r_{n-1})} \square$  then  $E_1$  has unifier  $\sigma_1 \cdots \sigma_{n-1}$

# Unification Problem (cont'd)

## Notation

$E \Rightarrow_{\sigma}^{(r)} E'$  if rule  $r$  from  $\mathcal{U}$  applied to equations  $E$  yields  $E'$

## Theorem

if  $E_1 \Rightarrow_{\sigma_1}^{(r_1)} E_2 \Rightarrow_{\sigma_2}^{(r_2)} \dots \Rightarrow_{\sigma_{n-1}}^{(r_{n-1})} \square$  then  $E_1$  has unifier  $\sigma_1 \cdots \sigma_{n-1}$

## Example

$$\begin{array}{l} \text{list}(\text{bool}) \approx \text{list}(\alpha) \Rightarrow_{\iota}^{(d_1)} \text{bool} \approx \alpha \\ \qquad \qquad \qquad \qquad \qquad \qquad \qquad \qquad \Rightarrow_{\{\alpha/\text{bool}\}}^{(v_2)} \square \end{array}$$

# Unification Problem (cont'd)

## Notation

$E \Rightarrow_{\sigma}^{(r)} E'$  if rule  $r$  from  $\mathcal{U}$  applied to equations  $E$  yields  $E'$

## Theorem

if  $E_1 \Rightarrow_{\sigma_1}^{(r_1)} E_2 \Rightarrow_{\sigma_2}^{(r_2)} \dots \Rightarrow_{\sigma_{n-1}}^{(r_{n-1})} \square$  then  $E_1$  has unifier  $\sigma_1 \cdots \sigma_{n-1}$

## Example

$$\begin{aligned} \text{list}(\text{bool}) \approx \text{list}(\alpha) &\Rightarrow_{\iota}^{(d_1)} \text{bool} \approx \alpha \\ &\Rightarrow_{\{\alpha/\text{bool}\}}^{(v_2)} \square \end{aligned}$$

## Remarks

- unification always terminates
- the order of applying inference rules has no (dramatic) effect

# Type Inference Problems

- $E \triangleright e : \alpha_0$  is **type inference problem**
- $\sigma$  s.t.,  $E\sigma \vdash e : \alpha_0\sigma$  (if exists) is solution (otherwise:  $e$  not typable)

# Type Inference Problems

- $E \triangleright e : \alpha_0$  is type inference problem
- $\sigma$  s.t.,  $E\sigma \vdash e : \alpha_0\sigma$  (if exists) is **solution** (otherwise:  $e$  not typable)

# The Type Inference System $\mathcal{I}$

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

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

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

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

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

# Recipe - Type Inference

## Input

core ML expression  $e$  and typing environment  $E$

# Recipe - Type Inference

## Input

core ML expression  $e$  and typing environment  $E$

## Algorithm

1. start with  $E \triangleright e : \alpha_0$  (fresh type variable  $\alpha_0$ )
2. use  $\mathcal{I}$  to transform  $E \triangleright e : \alpha_0$  into unification problem  $u$   
(if at any point no rule applicable **Not Typable**)
3. if possible solve  $u$  (obtaining unifier  $\sigma$ ) otherwise **Not Typable**



# Recipe - Type Inference

## Input

core ML expression  $e$  and typing environment  $E$

## Algorithm

1. start with  $E \triangleright e : \alpha_0$  (fresh type variable  $\alpha_0$ )
2. use  $\mathcal{I}$  to transform  $E \triangleright e : \alpha_0$  into unification problem  $u$   
(if at any point no rule applicable **Not Typable**)
3. if possible solve  $u$  (obtaining **unifier**  $\sigma$ ) otherwise **Not Typable**

## Output

the **most general** type of  $e$  w.r.t.  $E$  is  $\alpha_0\sigma$

# Example

find most general type of **let**  $id = \lambda x.x$  **in**  $id$  1 w.r.t.  $P$

Proof.

Blackboard