Functional Programming

WS 2014/15

Cezary Kaliszyk (VO+PS)
Yann Savoye (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

Combinator Parsing

Notes

- decompose linear sequence (text) into structure (type)
- type `('a,'t)Parser.t` is `'t list -> ('a * 't list)option`
- primitive parser accepts/rejects single token
- parser combinators compose parsers, e.g., `(>>=)`, `(>>)`, `(<|>)`, `many`

Combinator Parsing

Notes

- decompose linear sequence (text) into structure (type)
- type `('a,'t)Parser.t` is `'t list -> ('a * 't list)option`
- primitive parser accepts/rejects single token
- parser combinators compose parsers, e.g., `(>>=)`, `(>>)`, `(<|>)`, `many`

Combinator Parsing

Notes

- decompose linear sequence (text) into structure (type)
- type `('a,'t)Parser.t` is `'t list -> ('a * 't list)option`
- primitive parser accepts/rejects single token
- parser combinators compose parsers, e.g., $\underbrace{\texttt{(>>=)}}_{\text{bind}}$, `(>>)`, `(<|>)`, `many`

Combinator Parsing

Notes

- decompose linear sequence (text) into structure (type)
- type `('a,'t)Parser.t` is `'t list -> ('a * 't list)option`
- primitive parser accepts/rejects single token
- parser combinators compose parsers, e.g., `(>>=)`, `(>>)` (then), `(<|>)`, `many`

Combinator Parsing

Notes

- decompose linear sequence (text) into structure (type)
- type `('a,'t)Parser.t` is `'t list -> ('a * 't list)option`
- primitive parser accepts/rejects single token
- parser combinators compose parsers, e.g., `(>>=)`, `(>>)`, `(<|>)` (choice), `many`

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, dynamic programming

Theory II
type checking, type inference

Advanced Topics
lazy evaluation, infinite data structures, monads, ...

Overview

Core ML

Definition (Expressions)

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

Core ML

Definition (Expressions)

$$e ::= \overbrace{x \mid e\ e \mid \lambda x.e}^{\lambda\text{-Calculus}} \mid \ c \ \mid \textbf{let}\ x = e\ \textbf{in}\ e \mid \textbf{if}\ e\ \textbf{then}\ e\ \textbf{else}\ e$$

Core ML

Definition (Expressions)

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

Core ML

Definition (Expressions)

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

Core ML

Definition (Expressions)

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

Core ML

Definition (Expressions)

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

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)

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

Convention

- type variables α, α_0, α_1, ..., β, β_0, ...
- function type constructor '$\to$' is right associative
- base data type constructors: int, bool (instead of int(), bool())

Preliminaries

Definition (Types)

$$\tau ::= \alpha \mid \overbrace{\tau \to \tau}^{\text{function type constructor}} \mid g(\tau, \ldots, \tau)$$

Convention

- type variables α, α_0, α_1, ..., β, β_0, ...
- function type constructor '$\to$' is right associative
- base data type constructors: int, bool (instead of int(), bool())

Preliminaries

Definition (Types)

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

Convention

- type variables α, α_0, α_1, ..., β, β_0, ...
- function type constructor '$\to$' is right associative
- base data type constructors: int, bool (instead of int(), bool())

Preliminaries

Definition (Types)

$$\tau ::= \alpha \mid \tau \to \tau \mid g(\tau, \ldots, \tau)$$

Convention

- type variables α, α_0, α_1, . . . , β, β_0, . . .
- function type constructor '$\to$' is right associative
- base data type constructors: int, bool (instead of int(), bool())

Example

int $\to$ bool, (int $\to$ list(int)) $\to$ bool, list(α_0) $\to$ int, . . .

Preliminaries (cont'd)

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

$$(e : \tau) \in E \qquad \text{“}e \textit{ is of type } \tau \textit{ in } E\text{”}$$

Preliminaries (cont'd)

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

$$e : \tau \ \in E \qquad \text{“}\textit{e is of type } \tau \textit{ in } E\text{”}$$

Preliminaries (cont'd)

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

$e : \tau \ \in E$ "*e is of type* τ *in E*"

(Typing) judgment:

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

Preliminaries (cont'd)

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

$e : \tau \in E$ "*e is of type* τ *in E*"

(Typing) judgment:

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

Example

- environment $P = \{+ : \mathsf{int} \to \mathsf{int} \to \mathsf{int}, \mathsf{nil} : \mathsf{list}(\alpha), \mathsf{true} : \mathsf{bool}, \ldots\}$
- judgement $P \vdash \mathsf{true} : \mathsf{bool}$
- judgement $P \nvdash \mathsf{true} : \mathsf{int}$

Preliminaries (cont'd)

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

$e : \tau \ \in E$ "*e is of type* τ *in E*"

(Typing) judgment:

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

Example

- environment $P = \{+ : \text{int} \to \text{int} \to \text{int}, \text{nil} : \text{list}(\alpha), \text{true} : \text{bool}, \ldots\}$
- judgement $P \vdash \text{true} : \text{bool}$
- judgement $P \nvdash \text{true} : \text{int}$

Convention

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

The Type Checking System $\mathcal{C}$

$$\frac{}{E, e : \tau \vdash e : \tau}\ (\mathsf{ref}) \qquad \frac{E \vdash e_1 : \tau_2 \to \tau_1 \quad E \vdash e_2 : \tau_2}{E \vdash e_1\ e_2 : \tau_1}\ (\mathsf{app})$$

$$\frac{E, x : \tau_1 \vdash e : \tau_2}{E \vdash \lambda x.e : \tau_1 \to \tau_2}\ (\mathsf{abs}) \qquad \frac{E \vdash e_1 : \tau_1 \quad E, x : \tau_1 \vdash e_2 : \tau_2}{E \vdash \textbf{let}\ x = e_1\ \textbf{in}\ e_2 : \tau_2}\ (\mathsf{let})$$

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

Example

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

Proof.

$$\dfrac{\dfrac{E, x : \mathsf{bool} \vdash x : \mathsf{bool}}{E \vdash \lambda x.x : \mathsf{bool} \to \mathsf{bool}}\ (\mathsf{abs}) \qquad E \vdash \mathsf{true} : \mathsf{bool}}{E \vdash (\lambda x.x)\ \mathsf{true} : \mathsf{bool}}\ (\mathsf{app})$$

□

Example

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

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:

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

Preliminaries

Type variables:

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

Type substitution: σ is mapping from type variables to types

Preliminaries

Type variables:

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

Type substitution: σ is mapping from type variables to types

Application:

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

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

Preliminaries

Type variables:

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

Type substitution: σ is mapping from type variables to types

Application:

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

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

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

Example

$$\tau = \alpha \to (\alpha_1 \to \alpha_3)$$
$$\sigma = \{\alpha/\mathsf{int} \to \mathsf{int}, \alpha_1/\mathsf{list}(\alpha_2)\}$$
$$\sigma_2 = \{\alpha_3/\alpha_4, \alpha_2/\alpha, \alpha/\alpha_1\}$$

$$\mathcal{TV}\mathsf{ar}(\tau) = \{\alpha, \alpha_1, \alpha_3\}$$
$$\tau\sigma = (\mathsf{int} \to \mathsf{int}) \to (\mathsf{list}(\alpha_2) \to \alpha_3)$$
$$\mathcal{TV}\mathsf{ar}(\tau\sigma) = \{\alpha_2, \alpha_3\}$$
$$\sigma\sigma_2 = \{\alpha/\mathsf{int} \to \mathsf{int}, \alpha_1/\mathsf{list}(\alpha), \alpha_3/\alpha_4, \alpha_2/\alpha\}$$

Unification Problems

Definition

- unification problem is (finite) sequence of equations

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

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

$$\tau_1\sigma = \tau_1'\sigma; \ldots; \tau_n\sigma = \tau_n'\sigma$$

- process of computing a unifier is called unification

Unification Problems

Definition

- unification problem is (finite) sequence of equations

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

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

$$\tau_1\sigma = \tau_1'\sigma; \ldots; \tau_n\sigma = \tau_n'\sigma$$

- process of computing a unifier is called unification

Unification Problems

Definition

- unification problem is (finite) sequence of equations

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

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

$$\tau_1\sigma = \tau_1'\sigma; \ldots; \tau_n\sigma = \tau_n'\sigma$$

- process of computing a unifier is called unification

Unification Problems

Definition

- unification problem is (finite) sequence of equations

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

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

$$\tau_1\sigma = \tau_1'\sigma; \ldots; \tau_n\sigma = \tau_n'\sigma$$

- process of computing a unifier is called unification

The Unification System $\mathcal{U}$

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

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

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

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

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

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 $E_1 \Rightarrow_{\sigma_1}^{(r_1)} E_2 \Rightarrow_{\sigma_2}^{(r_2)} \ldots \Rightarrow_{\sigma_{n-1}}^{(r_{n-1})} \square$ *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

$$\begin{aligned} \mathsf{list}(\mathsf{bool}) \approx \mathsf{list}(\alpha) \quad &\Rightarrow_{\iota}^{(\mathsf{d}_1)} \quad \mathsf{bool} \approx \alpha \\ &\Rightarrow_{\{\alpha/\mathsf{bool}\}}^{(\mathsf{v}_2)} \quad \square \end{aligned}$$

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

$$\begin{aligned} \mathsf{list}(\mathsf{bool}) \approx \mathsf{list}(\alpha) \quad &\Rightarrow_{\iota}^{(\mathsf{d}_1)} \quad \mathsf{bool} \approx \alpha \\ &\Rightarrow_{\{\alpha/\mathsf{bool}\}}^{(\mathsf{v}_2)} \quad \square \end{aligned}$$

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
- σ 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
- σ s.t., $E\sigma \vdash e : \alpha_0\sigma$ (if exists) is solution (otherwise: e not typable)

The Type Inference System $\mathcal{I}$

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

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

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

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 $E \rhd e : \alpha_0$ (fresh type variable α_0)
2. use $\mathcal{I}$ to transform $E \rhd e : \alpha_0$ into unification problem u
   (if at any point no rule applicable Not Typable)
3. if possible solve u (obtaining unifier σ) 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 α_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 σ) 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 □