PL'18 Exam Preparation: Lambda Calculus, JavaScript, Type Inference

Oded Padon & Mooly Sagiv

Untyped Lambda Calculus - Syntax

t ::=	terms
x	variable
λ x. t	abstraction
t t	application

- Terms can be represented as abstract syntax trees
- Syntactic Conventions:
  - Applications associates to left :
    $e_1\ e_2\ e_3 \equiv (e_1\ e_2)\ e_3$
  - The body of abstraction extends as far as possible:
    $\lambda x.\ \lambda y.\ x\ y\ x \equiv \lambda x.\ (\lambda y.\ ((x\ y)\ x))$

Non-Deterministic Operational Semantics

(E-AppAbs) $(\lambda\, x.\, t_1)\, t_2 \Rightarrow [x \mapsto t_2]\, t_1$

$$\frac{t \Rightarrow t'}{\lambda\, x.\, t \Rightarrow \lambda\, x.\, t'}$$ (E-Abs)

(E-App$_1$) $$\frac{t_1 \Rightarrow t'_1}{t_1\ t_2 \Rightarrow t'_1\ t_2}$$

$$\frac{t_2 \Rightarrow t'_2}{t_1\ t_2 \Rightarrow t_1\ t'_2}$$ (E-App$_2$)

Why is this semantics non-deterministic?

Different Evaluation Orders

(E-AppAbs) $(\lambda\ x.\ t_1)\ t_2 \Rightarrow [x \mapsto t_2]\ t_1$

(E-Abs)
$$\frac{t \Rightarrow t'}{\lambda\ x.\ t \Rightarrow \lambda\ x.\ t'}$$

(E-App$_1$)
$$\frac{t_1 \Rightarrow t'_1}{t_1\ t_2 \Rightarrow t'_1\ t_2}$$

(E-App$_2$)
$$\frac{t_2 \Rightarrow t'_2}{t_1\ t_2 \Rightarrow t_1\ t'_2}$$

$(\lambda x.\ (\text{add}\ x\ x))\ (\text{add}\ 2\ 3) \Rightarrow (\lambda x.\ (\text{add}\ x\ x))\ (5) \Rightarrow \text{add}\ 5\ 5 \Rightarrow 10$

$(\lambda x.\ (\text{add}\ x\ x))\ (\text{add}\ 2\ 3) \Rightarrow (\text{add}\ (\text{add}\ 2\ 3)\ (\text{add}\ 2\ 3)) \Rightarrow$

$(\text{add}\ 5\ (\text{add}\ 2\ 3)) \Rightarrow (\text{add}\ 5\ 5) \Rightarrow 10$

This example: same final result but lazy performs more computations

Different Evaluation Orders

(E-AppAbs) $(\lambda\, x.\, t_1)\ t_2 \Rightarrow [x \mapsto t_2]\ t_1$

$$\frac{t \Rightarrow t'}{\lambda\, x.\, t \Rightarrow \lambda\, x.\, t'}\ \text{(E-Abs)}$$

$$\text{(E-App}_1\text{)}\ \frac{t_1 \Rightarrow t'_1}{t_1\ t_2 \Rightarrow t'_1\ t_2} \qquad \frac{t_2 \Rightarrow t'_2}{t_1\ t_2 \Rightarrow t_1\ t'_2}\ \text{(E-App}_2\text{)}$$

$(\lambda x.\ \lambda y.\ x)\ 3\ (\text{div}\ 5\ 0) \Rightarrow$ Exception: Division by zero

$(\lambda x.\ \lambda y.\ x)\ 3\ (\text{div}\ 5\ 0) \Rightarrow (\lambda y.\ 3)\ (\text{div}\ 5\ 0) \Rightarrow 3$

This example: lazy suppresses erroneous division and reduces to final result

Can also suppress non-terminating computation.

Many times we want this, for example:

```
if i < len(a) and a[i]==0: print “found zero”
```

Strict	Lazy	Normal Order
(E-App$_1$) $\dfrac{t_1 \Rightarrow t'_1}{t_1\ t_2 \Rightarrow t'_1\ t_2}$	**(E-AppAbs)** $(\lambda\ x.\ t_1)\ t_2 \Rightarrow [x \mapsto t_2]\ t_1$	**(E-AppAbs)** $(\lambda\ x.\ t_1)\ t_2 \Rightarrow [x \mapsto t_2]\ t_1$
precedence		precedence
(E-App$_2$) $\dfrac{t_2 \Rightarrow t'_2}{t_1\ t_2 \Rightarrow t_1\ t'_2}$	**(E-App$_1$)** $\dfrac{t_1 \Rightarrow t'_1}{t_1\ t_2 \Rightarrow t'_1\ t_2}$	**(E-App$_1$)** $\dfrac{t_1 \Rightarrow t'_1}{t_1\ t_2 \Rightarrow t'_1\ t_2}$
precedence		precedence
(E-AppAbs) $(\lambda\ x.\ t_1)\ t_2 \Rightarrow [x \mapsto t_2]\ t_1$		**(E-App$_2$)** $\dfrac{t_2 \Rightarrow t'_2}{t_1\ t_2 \Rightarrow t_1\ t'_2}$
		(E-Abs) $\dfrac{t \Rightarrow t'}{\lambda\ x.\ t \Rightarrow \lambda\ x.\ t'}$

Call-by-value Operations Semantics via Inductive Definition (no precedence)

t ::=	terms	v ::= λ x. t	abstraction values
x	variable		
λ x. t	abstraction		
t t	application		

$$(\lambda\ x.\ t_1)\ v_2 \Rightarrow [x \mapsto v_2]\ t_1$$ (E-AppAbs)

$$\frac{t_1 \Rightarrow t'_1}{t_1\ t_2 \Rightarrow t'_1\ t_2}$$ (E-APPL1)

$$\frac{t_2 \Rightarrow t'_2}{v_1\ t_2 \Rightarrow v_1\ t'_2}$$ (E-APPL2)

Summary Order of Evaluation

- Full-beta-reduction
  - All possible orders
- Applicative order call by value (strict, eager)
  - Left to right
  - Fully evaluate arguments before function application
- Normal order
  - The leftmost, outermost redex is always reduced first
- Call by name (lazy)
  - Evaluate arguments as needed
- Call by need
  - Evaluate arguments as needed and store for subsequent usages
  - Implemented in Haskell

2016 Moed B

שאלה 4 - λ-calculus

נתון הביטוי הבא ב λ-calculus :

$(\lambda t.\ \lambda f.\ f)\ ((\lambda s.\ \lambda z.\ s\ z)\ (\lambda x.x))\ (\lambda x.x)\ (\lambda x.x)$

א. ציירי את ה AST המתאים לביטוי.

ב. כתבי סדרת חישוב (reduction) לביטוי תחת call by value semantics.

ג. כתבי סדרת חישוב (reduction) לביטוי תחת lazy evaluation semantics.

ד. הסבירי בקצרה את ההבדל בין call by value ו lazy evaluation.

Typed Lambda Calculus

Chapter 9

Benjamin Pierce

Types and Programming Languages

Simple Types

T ::=		types
	Bool	type of Booleans
	$T \rightarrow T$	type of functions

$$T_1 \rightarrow T_2 \rightarrow T_3 = T_1 \rightarrow (T_2 \rightarrow T_3)$$

Simple Typed Lambda Calculus

t ::=	terms	T::=	types
x	variable	Bool	atomic type for Booleans
λ x: T. t	abstraction	T $\to$ T	types of functions
t t	application		

Tying Relation $\vdash$

- Type fact: t:T means term t has type T (e.g. 1+3:Int, f:Int$\to$Int)
- Typing relation $\vdash$: relates sets of type facts and type facts
- $\Gamma \vdash$ t:T means under Γ (context, environment), term t has type T
- $\vdash$ t:T means term t has type T under the empty environment (no assumptions)
  - Can t have free variables?

Typing Relation Examples

- Type fact: $t{:}T$ means term t has type T (e.g. $1+3{:}Int$, $f{:}Int \to Int$)
- $\Gamma \vdash t{:}T$ means under Γ (context, environment), term t has type T
- $\vdash t{:}T$ means closed term t has type T under the empty environment

Examples

- $x{:}Int, y{:}Int \vdash x+y : ?$
- $x{:}Int, y{:}Bool \vdash x+y : ?$
- $x{:}Int \vdash x+y : ?$
- $x{:}Int, y{:}Int, b{:}Bool \vdash$ if b then x else y : ?
- $x{:}Int, y{:}Bool, b{:}Bool \vdash$ if b then x else y : ?
- $x{:}Int, b{:}Bool \vdash$ if b then x else y : ?
- $x{:}Int, f{:}Int \to Bool \vdash f\ x : ?$
- $x{:}Int, f{:}\ Int \to Int \to Bool \vdash f\ x\ x : ?$
- $x{:}Int, f{:}\ Int \to Int \to Bool \vdash f\ x{:}\ ?$
- $f{:}\ Int \to Int \to Bool \vdash f\ x\ x : ?$
- $\vdash 1+2 : ?$
- $\vdash$ true : ?
- $x{:}Int, y{:}Int, f{:}\ Int \to Int \to Bool \vdash 1+2 : ?$

Formally Defining $\vdash$

t ::=	terms
x	variable
λ x: T. t	abstraction
t t	application
T::=	types
$T \to T$	types of functions
Γ::=	context
$\varnothing$	empty context
Γ, x : T	term variable binding

$$\frac{x : T \in \Gamma}{\Gamma \vdash x : T} \quad \text{(T-VAR)}$$

$$\frac{\Gamma, x : T_1 \vdash t_2 : T_2}{\Gamma \vdash \lambda x : T_1.\ t_2 : T_1 \to T_2} \quad \text{(T-ABS)}$$

$$\frac{\Gamma \vdash t_1 : T_{11} \to T_{12} \qquad \Gamma \vdash t_2 : T_{11}}{\Gamma \vdash t_1\ t_2 : T_{12}} \quad \text{(T-APP)}$$

Adding Booleans

t ::=	terms
x	variable
λ x: T. t	abstraction
t t	application
true	constant true
false	constant false
if t then t else t	conditional

T::=	types
Bool	type of Booleans
$T \to T$	types of functions

Γ::=	context
$\varnothing$	empty context
Γ, x : T	term variable binding

$$\frac{x : T \in \Gamma}{\Gamma \vdash x : T} \quad \text{(T-VAR)}$$

$$\frac{\Gamma, x : T_1 \vdash t_2 : T_2}{\Gamma \vdash \lambda x : T_1 .\ t_2 : T_1 \to T_2} \quad \text{(T-ABS)}$$

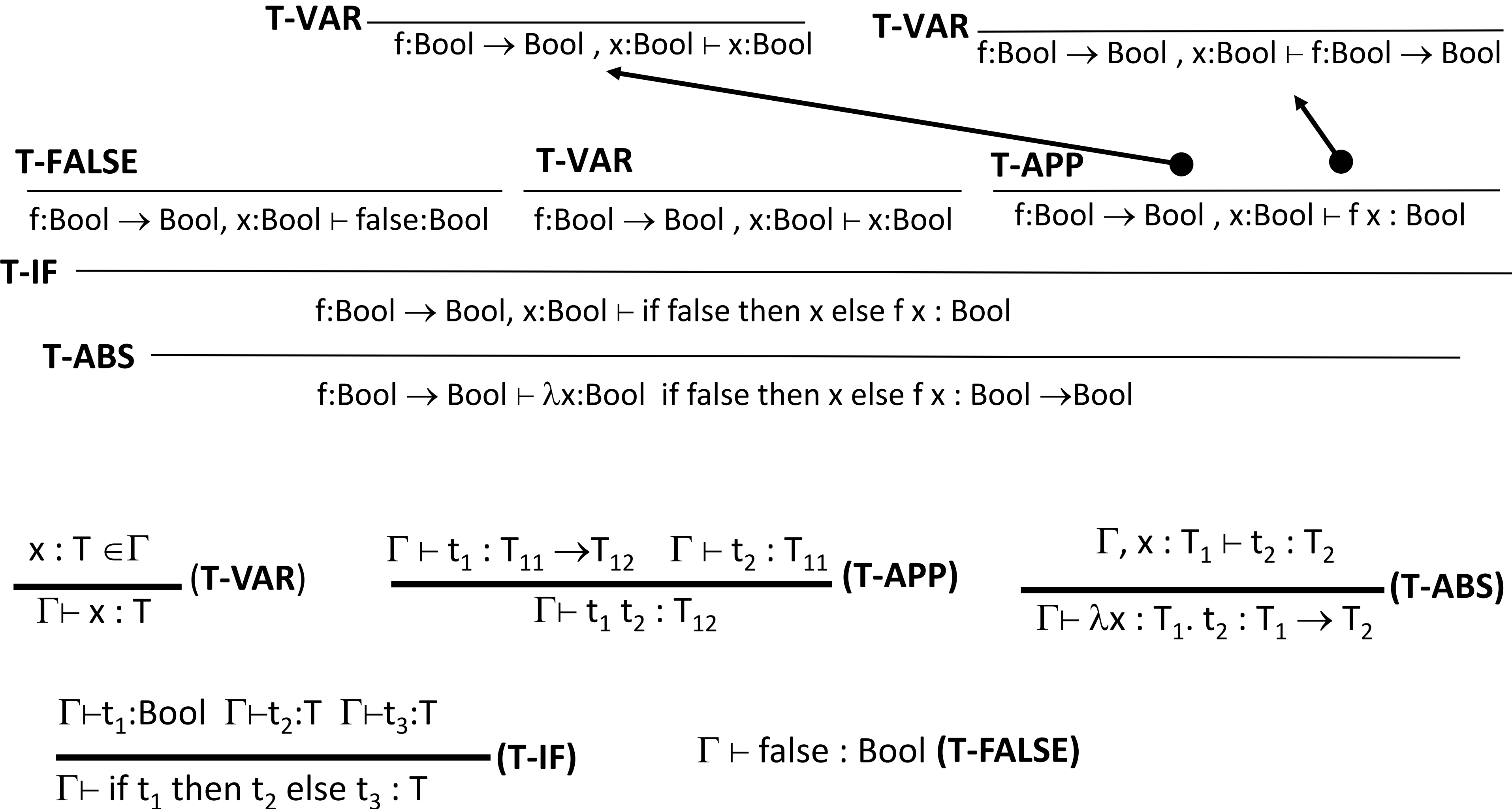
$$\frac{\Gamma \vdash t_1 : T_{11} \to T_{12} \qquad \Gamma \vdash t_2 : T_{11}}{\Gamma \vdash t_1\ t_2 : T_{12}} \quad \text{(T-APP)}$$

$$\Gamma \vdash \text{true} : \text{Bool} \quad \text{(T-TRUE)}$$

$$\Gamma \vdash \text{false} : \text{Bool} \quad \text{(T-FALSE)}$$

$$\frac{\Gamma \vdash t_1 : \text{Bool} \quad \Gamma \vdash t_2 : T \quad \Gamma \vdash t_3 : T}{\Gamma \vdash \text{if } t_1 \text{ then } t_2 \text{ else } t_3 : T} \quad \text{(T-IF)}$$

Example Typing Tree



2017 Moed B

ד. מצאי טיפוס T כך שקביעת הטיפוס הבא תתקיים, וכתבי עץ גזירה שמוכיח אותה לפי כללי הטיפוס (typing rules):

f:T ⊢ (λx:bool. f (f x)) : T

JavaScript

2015 A

שאלה 6 - JavaScript

אחת הדרכים ליצור אובייקט עם שדות פרטיים ב JavaScript היא ע"י שימוש ב closures. הקוד הבא מדגים זאת, וממש אובייקט Point שיש לו מתודות move, toString, ואובייקט ColoredPoint שיש לו מתודות move, darken, toString:

```
function Point(x, y) {
   var obj = {};
   obj.move = function(dx, dy) { x += dx; y += dy; };
   obj.toString = function() {
       return "[Point with x=" + x + " and y=" + y + "]";
   };
   return obj;
}
function ColoredPoint(x, y, color) {
   var obj = Point(x, y);
   obj.darken = function(tint) { color += tint; };
   obj.toString = function() {
       return "[ColoredPoint with x=" + x + ", y=" + y +
              ", and color=" + color + "]";
   };
   return obj;
}
p = Point(3, 4);
p.move(1, 2);
console.log(p.toString());
```

א. הסבירי בקצרה את הרעיון של מימוש שדות פרטיים ע"י closures.

ב. בקוד הנ"ל יש באג. כתבי 3 שורות של JavaScript שמדגימות את הבאג, והסבירי מהו הבאג וממה הוא נובע.

ג. האם ניתן לתקן את הבאג ע"י שינוי הפונקציה toString ב ColoredPoint בלבד?

ד. תקני את הבאג, ע"י תוספות ל Point ושינוי הפונקציה toString ב ColoredPoint.

Type Inference

Type Inference Algorithm

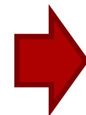
- Parse program to build parse tree
- Assign type variables to nodes in tree
- Generate constraints:
  - From environment: literals (2), built-in operators (+), known functions (tail)
  - From form of parse tree: e.g., application and abstraction nodes
- Solve constraints using *unification*
- Determine types of top-level declarations

```
let isempty l = match l with
    |[] -> true
    | _ -> false
```

fun
isempty : t_0
l : t_1
match : t_2
l : t_1
Case 1
Case 2
[] : [t_3]
true : bool
_ : t_4
false : bool

Unification Constrains:

```
t_0 = t_1 -> t_2   (Fun)
t_1 = [t_3]        (Case 1)
t_2 = bool         (Case 1)
t_1 = t_4          (Case 2)
t_2 = bool         (Case 2)
```



```
isempty : [t_3] -> bool
```

2016 B

שאלה 7 - Types

נתון המימוש הבא של פונקציית OCaml בשם absent שאמורה לבדוק האם איבר <u>אינו</u> מופיע ברשימה,

כלומר להחזיר true אם האיבר אינו מופיע ברשימה ו false אם הוא כן מופיע:

```
let rec absent x xs = match xs with
 | [] -> true
 | h::t -> absent x t
```

א. נתחי את הטיפוס של הפונקציה absent ע"י אלגוריתם Hindley-Milner. עלייך לפרט את אופן פעולת האלגוריתם עד להגעה לטיפוס הכללי ביותר של הפונקציה.

ב. כיצד ניתן להבין מהטיפוס שיש באג במימוש?

ג. תקני את הבאג.

ד. נתחי את הטיפוס של הפונקציה המתוקנת. עלייך להסביר בפירוט מה ישתנה בפעולת אלגוריתם Hindley-Milner על הקוד המתוקן לעומת סעיף א'.

```
let rec absent:t_1 x:t_2 xs:t_3 =
  (match xs:t_3 with
    | []:[t_5] -> true:bool
    | (h:t_6 :: t:t_7):t_8 ->
        (absent:t_1 x:t_2 t:t_7):t_9
  ):t_4
```

Unification Constrains

$t_1 = t_2 \text{ -> } t_3 \text{ -> } t_4$

$t_3 = [t_5]$

$t_4 = \text{bool}$

$t_7 = [t_6]$

$t_8 = [t_6]$

$t_8 = t_3$

$t_1 = t_2 \text{ -> } t_7 \text{ -> } t_9$

$t_9 = t_4$

----

absent : t_2 -> $[t_5]$ -> bool

```
let rec absent:t1 x:t2 xs:t3 =
  (match xs:t3 with
    | []:[t5] -> true:bool
    | (h:t6 :: t:t7):t8 ->
        (absent:t1 x:t2 t:t7):t9
  ):t4
```

```
let rec absent:t_1 x:t_2 xs:t_3 =
  (match xs:t_3 with
    | []:[t_5] -> true:bool
    | (h:t_6 :: t:t_7):t_8 ->
        ((h:t_6 <> x:t_2):bool and
        (absent:t_1 x:t_2 t:t_7):t_9):t_10
  ):t_4
```

Unification Constrains

$t_1 = t_2$ -> t_3 -> t_4

$t_3 = [t_5]$

t_4 = bool

$t_7 = [t_6]$

$t_8 = [t_6]$

$t_8 = t_3$

$t_1 = t_2$ -> t_7 -> t_9

$t_6=t_2$

t_9=bool

t_{10}=bool

$t_{10} = t_4$

----

absent : t_5 -> $[t_5]$ -> bool

```
let rec absent:t1 x:t2 xs:t3 =
  (match xs:t3 with
    | []:[t5] -> true:bool
    | (h:t6 :: t:t7):t8 ->
        ((h:t6 <> x:t2):bool and
        (absent:t1 x:t2 t:t7):t9):t10
   ):t4
```