

Typed Lambda Calculus

Chapter 9

Benjamin Pierce

Types and Programming Languages

Call-by-value small step operational Semantics

$t ::=$	terms	$v ::=$	values
x	variable	$\lambda x. t$	abstraction values
$\lambda x. t$	abstraction		
$t t$	application		

$$(\lambda x. t_{12}) v_2 \Rightarrow [x \mapsto v_2] t_{12} \text{ (E-AppAbs)}$$

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

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

Call-by-value big-step Operational Semantics

$t ::=$	terms	$v ::=$	values
x	variable	$\lambda x. t$	abstraction values
$\lambda x. t$	abstraction		
$t t$	application		other values

$$\lambda x. t \rightarrow \lambda x. t \quad (\text{V-Value})$$

$$t_1 \rightarrow \lambda x. t_3 \quad t_2 \rightarrow v_1 \quad [x \mapsto v_1] t_3 \rightarrow v_2$$

$$t_1 \ t_2 \rightarrow v_2$$

(V-App)

A Pseudocode for call-by value interpreter

```
Lambda eval(Lambda t) {  
    switch(t) {  
        case t= λx. t1: // A value  
            return t  
        case t = t1 t2:  
            Lambda temp = eval(t1);  
            assert temp = λx. t3;  
            Lambda v1 = eval(t2) ; // v1 must be a value  
            return eval([x ↦ v1] t3) ;  
        default: assert false;  
    }  
}
```

Consistency of Function Application

- Prevent runtime errors during evaluation
- Reject inconsistent terms
- What does ‘ $x x$ ’ mean?
- Cannot be always enforced
 - if $\text{if } <\!\!\text{tricky computation}\!> \text{ then true else } (\lambda x. x)$

A Naïve Attempt

- Add function type \rightarrow
- Type rule $\lambda x. t : \rightarrow$
 - $\lambda x. x : \rightarrow$
 - If true then $(\lambda x. x)$ else $(\lambda x. \lambda y y) : \rightarrow$
- Too Coarse

Simple Types

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

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

Explicit vs. Implicit Types

- How to define the type of λ abstractions?
 - **Explicit**: defined by the programmer

$t ::=$	Type λ terms
x	variable
$\lambda x : T. t$	abstraction
$t t$	application

- **Implicit**: Inferred by analyzing the body
- The **type checking problem**: Determine if typed term is well typed
- The **type inference problem**: Determine if there exists a type for (an untyped) term which makes it well typed

Simple Typed Lambda Calculus

$t ::=$ terms

x variable

$\lambda x: T. t$ abstraction

$t t$ application

$T ::=$ types

$T \rightarrow T$ types of functions

Typing Function Declarations

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

A typing context Γ maps free variables into types

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

Typing Free Variables

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

Typing Function Applications

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

Typing Conditionals

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

if true then ($\lambda x: \text{Bool}. x$) else ($\lambda y: \text{Bool}. \text{not } y$)

SOS for Simple Typed Lambda Calculus

$t ::=$	terms	$t_1 \rightarrow t_2$
x	variable	
$\lambda x: T. t$	abstraction	$\frac{t_1 \rightarrow t'_1}{t_1 \ t_2 \rightarrow t'_1 \ t_2} \text{ (E-APP1)}$
$t \ t$	application	
$v ::=$	values	$\frac{t_2 \rightarrow t'_2}{v_1 \ t_2 \rightarrow v_1 \ t'_2} \text{ (E-APP2)}$
$\lambda x: T. t$	abstraction values	$(\lambda x: T_{11}. t_{12}) v_2 \rightarrow [x \mapsto v_2] t_{12} \text{ (E-APPABS)}$
$T ::=$	types	
$T \rightarrow T$	types of functions	

Type Rules

$t ::=$

terms

x

variable

$\lambda x : T. t$

abstraction

$\Gamma \vdash t : T$

$x : T \in \Gamma$

$\Gamma \vdash x : T$

(T-VAR)

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

$T ::=$

types

$T \rightarrow T$

types of functions

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

$\Gamma ::=$

context

\emptyset

empty context

$\Gamma, x : T$

term variable binding

$t ::=$ terms

x variable

$\lambda x : T. t$ abstraction

$t t$ application

true constant true

false constant false

if t then t else t conditional

$\Gamma \vdash t : T$

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

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

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

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

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

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

$T ::=$ types

Bool Boolean type

$T \rightarrow T$ types of functions

$\Gamma ::=$ context

\emptyset empty context

$\Gamma, x : T$ term variable binding

Simple Example

$(\lambda x:\text{Bool}. \ x) \text{ true}$

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

$$\frac{x : \text{Bool} \vdash x : \text{Bool}}{\vdash \lambda x : \text{Bool}. \ x : \text{Bool} : \text{Bool} \rightarrow \text{Bool}} \text{ (T-ABS)}$$

$$\Gamma \vdash t : T$$

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

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

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

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

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

$$\vdash (\lambda x:\text{Bool}. \ x) \text{ true} : \text{Bool}$$

$$\text{(T-APP)}$$

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

Another Example

if true then ($\lambda x:\text{Bool}. x$) else ($\lambda x:\text{Bool}. x$)

$$\Gamma \vdash t : T$$

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

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

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

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

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

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

Another Example

if true then $(\lambda x:\text{Bool}. x)$ else
 $(\lambda x:\text{Bool}. \lambda y:\text{Bool}. x)$

$$\Gamma \vdash t : T$$

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

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

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

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

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

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

The Typing Relation

- Formally the typing relation is the smallest ternary relation on contexts, terms and types
 - in terms of inclusion
- A term t is **typable** in a given context Γ (**well typed**) if there exists some type T such that
$$\Gamma \vdash t : T$$
- Interesting on closed terms (empty contexts)

Uniqueness of Types

- Each term t has at most one type in any given context
 - If t is typable then
 - its type is unique
 - There is a unique type derivation tree for t

Top-Down Type Checking

```
Type compute_type(Tenv:  $\Gamma$ , Lambda t) {  
    switch (t) {  
        case x : return  $\Gamma$ x;  
        case  $\lambda x : T. t$  :  
            return compute_type( $\Gamma[x \rightarrow T]$ , t);  
        case t1 t2:  
            T = compute_type( $\Gamma$ , t1) ;  
            assert T ==  $T_{11} \rightarrow T_{12}$ ;  
            T' = compute_type(t2);  
            assert T' ==  $T_{11}$ ;  
            return T12
```

Type Safety

- Well typed programs cannot go wrong
- If t is well typed then either t is a value or there exists an evaluation step $t \rightarrow t'$
[Progress]
- If t is well typed and there exists an evaluation step $t \rightarrow t'$ then t' is also well typed
[Preservation]

Expressive Power

- How expressive is typed lambda calculus
- Can we code divergence?
- How about loops?

Summary

- Constructive rules for preventing runtime errors in a Turing complete programming language
- Efficient type checking
- Unique types
- Type safety
- But limits programming

Summary: Lambda Calculus

- Powerful
- The ultimate assembly language
- Useful to illustrate ideas
- But can be counterintuitive
- Usually extended with useful syntactic sugars
- Other calculi exist
 - pi-calculus
 - object calculus
 - mobile ambients
 - ...