Typed Lambda Calculus Revisited

Chapter 9

Benjamin Pierce

Types and Programming Languages

Topics

- Reminder call by value interpreter
- What is typed lambda calculus and what does it provide
- Limitations
- Relations to OCaml

Call-by-value small step perational Semantics

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

v ::=	values
λ x. t	abstraction values

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

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

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

Call-by-value large-step Operational Semantics

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

v ::=	values
λ x. t	abstraction values
other values	

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

$$\frac{t_1 \rightarrow \lambda\, x.\ t_3 \quad t_2 \rightarrow v_1 \quad [x \mapsto v_1]\, t_3 \rightarrow v_2}{t_1\ t_2 \rightarrow v_2} \qquad \text{(V-App)}$$

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 <tricky computation> then true else (λx. x)

Summary Lambda Calculus

Pros

- Expressive
- Compact
- Demonstrate programming concepts
  - Lazy vs eager
  - Divergence
  - Higher order programming
  - OO programing
  - Continuation
  - …

Cons

- Counterintuitive
- Runtime errors

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
λ 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
λ x: T. t	abstraction
t t	application

T::=	types
$T \rightarrow T$	types of functions

Typing Function Declarations

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

A typing context Γ maps free variables into types

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

Typing Free Variables

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

Typing Function Applications

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

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 (λx: Bool. x) else (λy: Bool. not y)

SOS for Simple Typed Lambda Calculus

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

$$\frac{t_1 \rightarrow t_2}{t_1\ t_2 \rightarrow t'_1\ t_2}\ \text{(E-APP1)}$$

$$\frac{t_2 \rightarrow t'_2}{v_1\ t_2 \rightarrow v_1\ t'_2}\ \text{(E-APP2)}$$

$$(\lambda\ x{:}\ T_{11}.\ t_{12})\ v_2 \rightarrow [x \mapsto v_2]\ t_{12}\ \text{(E-APPABS)}$$

T::=	types
$T \rightarrow T$	types of functions

Type Rules

$\Gamma \vdash t : T$

t ::= terms

x variable

λ x: T. t abstraction

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

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

T::= types

$T \to T$ types of functions

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

Γ::= context

$\varnothing$ empty context

$\Gamma, x : T$ term variable binding

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

$$\Gamma \vdash t : T$$

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

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

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

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

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

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

T::=	types
Bool	Boolean type
$T \to T$	types of functions
Γ::=	context
$\varnothing$	empty context
Γ, x : T	term variable binding

Simple Example

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

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

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

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

$$\frac{}{\vdash (\lambda x{:}\mathrm{Bool}.\ x)\ \mathrm{true} : \mathrm{Bool}}\ (\text{T-APP})$$

$$\Gamma \vdash t : T$$

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

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

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

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

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

Another Example

if true then (λx:Bool. x) else (λx:Bool. x)

$$\Gamma \vdash t : T$$

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

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

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

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

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

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

Another Example

if true then (λx:Bool. x) else
(λx:Bool. λy:Bool. x)

$$\Gamma \vdash t : T$$

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

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

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

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

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

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

The Typing Relation

- 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 type_check(Tenv: Γ, Lambda t)  {
        switch (t) {
            case x : return Γx;
            case λx : T. t :
                    return  T → type_check(Γ[x →T], t);
            case t1 t2:
                    T = type_check(Γ,   t1) ;
                    assert T == T11 → T12;
                    T'=type_check(t2);
                    assert T' == T11;
                    return T12
```

Ocaml Types

```
type Basic_type   = Primitive of String |
                    FunctionType of Basic_Type * Basic_Type

type Typed_lambda = Abstraction of String * Basic_Type * Typed_Lambda|
                    Application of Typed_lambda * Typed_lambda |
                    Variable of String
```

Type Checking

```
let rec type_check env = function
  Variable(v) ->  var_find env v
  |
  Abstraction(v, ty, t) ->  FunctionType(ty, type_check (v, ty) :: env t)
  |
  Application(t1, t2) -> let ty1 = type_check env t1 in
                         let ty2 = type_check env t1 in
                         match ty1 with
                         Function(ty11, ty12) ->
                                  if ty11 = ty2 then ty12
                                  else failwith “unmatched types”
                          |
                          Primitive(ty) ->     failwith “must be a function”
let rec var_find env v = match e with
           [] -> failwith “undefined type for a variable”
            |
          (v1, t) :: e’ -> if v = v1 then t else var_find e’ v
```

Type Safety
Well typed programs cannot go wrong

For every well typed term t:

1. $t \rightarrow^* t$
2. t loops

Type Safety

- 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
  - tru = λt. λf. t
  - fls = λt. λf. f
  - test = λl. λm. λn. l m n

- Can we code divergence?

  (λ x.y) ((λ x.(x x)) (λ x.(x x)))

- 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
  - …