Typed Lambda Calculus

Chapter 9

Benjamin Pierce

Types and Programming Languages

Call-by-value Operational Semantics

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

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

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

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

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)

A Naïve Attempt

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

Simple Types

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

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

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

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

Typing Function Applications

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

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 (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

T::= types

$T \to T$ types of functions

$t_1 \to t_2$

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

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

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

Type Rules

t ::= terms

x variable

λ x: T. t abstraction

T::= types

$T \rightarrow T$ types of functions

Γ::= context

$\varnothing$ empty context

$\Gamma, x : T$ term variable binding

$\Gamma \vdash t : T$

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

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

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

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	Boolean type
$T \rightarrow T$	types of functions

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

$\Gamma \vdash t : T$

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

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

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

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

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

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

Examples

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

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)

Inversion of the typing relation

- $\Gamma \vdash x : R \Rightarrow x : R \in \Gamma$
- $\Gamma \vdash \lambda x : T_1.\ t2 : R \Rightarrow R = T_1 \rightarrow R_2$ for some R_2 with $\Gamma \vdash t_2 : R_2$
- $\Gamma \vdash t_1\ t_2 : R \Rightarrow$ there exists T_{11} such that $\Gamma \vdash t_1 : T_{11} \rightarrow R$ and $\Gamma \vdash t_2 : T_{11}$
- $\Gamma \vdash \text{true} : R \Rightarrow R = \text{Bool}$
- $\Gamma \vdash \text{false} : R \Rightarrow R = \text{Bool}$
- $\Gamma \vdash$ if t_1 then t_2 else $t_3 : R \Rightarrow \Gamma \vdash t_1 : \text{Bool}$, $\Gamma \vdash t_2 : R$, $\Gamma \vdash t_3 : R$

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

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]

Canonical Forms

- If v is a value of type Bool then v is either true or false
- If v is a value of type $T_1 \to T_2$ then $v = \lambda x{:}\ T_1.t_2$

Progress Theorem

- Does not hold on terms with free variables
- For every closed well typed term t, either t is a value or there exists t’ such that $t \to t'$

Preservation Theorem

- If $\Gamma \vdash$ t : T and Δ is a permutation of Γ then $\Delta \vdash$ t : T [Permutation]
- If $\Gamma \vdash$ t : T and x $\notin$ dom(Γ) then Γ,t $\vdash$ t : T with a proof of the same depth [Weakening]
- If Γ, x: S $\vdash$ t : T and $\Gamma \vdash$ s: S
  then $\Gamma \vdash$ [x $\mapsto$ s] t : T
  [Preservation of types under substitution]
- $\Gamma \vdash$ t : T and t $\rightarrow$ t' then $\Gamma \vdash$ t' : T

The Curry-Howard Correspondence

- Constructive proofs
- The proof of a proposition P consists of a concrete evidence for P
- The proof of $P \supset Q$ can be viewed as a mechanical procedure for proving Q using the proof of P
- The proof of $P \wedge Q$ consists of a proof of P and a proof of Q
- An analogy between function introduction and function application(elimination)

The Curry-Howard Correspondence

Logic	Programming Languages
propositions	types
proposition $P \supset Q$	type $P \to Q$
proposition $P \wedge Q$	type $P \times Q$
proof of proposition P	term t of type P
proposition P is provable	Type P is inhabited

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

T::= types

$T \to T$ types of functions

$$t_1 \to t_2$$

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

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

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

Erasure and Typability

- Types are used for preventing errors and generating more efficient code
- Types are not used at runtime

$$\text{erase}(x) = x$$
$$\text{erase}(\lambda x{:}\ T_1.\ t_2) = \lambda x.\text{erase}(t_2)$$
$$\text{erase}(t_1\ t_2) = \text{erase}(t_1)\ \text{erase}(t_2)$$

- If $t \rightarrow t'$ under typed evaluation relation, then $\text{erase}(t) \rightarrow \text{erase}(t')$
- A term t in the untyped lamba calculus is typable if there exists a typed term t’ such that erase(t’) = t

Different Ways for formulating semantics

- Curry-style
  - Define a semantics of untyped terms
  - Provide a type system for rejecting bad programs
- Church-style
  - Define semantics only on typed terms

Simple Extensions (Chapter 11)

- Base Types
- The Unit Type
- Ascription
- Let bindings
- Pairs
- Tuples
- Records
- Sums
- Variants
- General recursion
- Lists

Unit type

New syntactic forms

t ::= ….	Terms:
unit	constant unit
v ::= ….	Values:
unit	constant unit
T ::= ….	types:
Unit	unit type

New typing rules

$\Gamma \vdash \text{unit} : \text{Unit}$ (T-Unit)

$$\frac{\Gamma \vdash t_1 : \text{Unit} \quad \Gamma \vdash t_2 : T}{\Gamma \vdash t_1 ; t_2 : T} \quad \text{(T-SEQ)}$$

New derived forms

$t_1 ; t_2 \doteq (\lambda x.\ \text{Unit}\ t_2)\ t_1$
where $x \notin FV(t_2)$

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

$\text{unit} ; t_2 \rightarrow t_2$ (E-SEQ)

Two ways for language extensions

- Derived forms (syntactic sugars)
- Explicit extensions

Ascription

- Explicit types for subterms
- Documentation
- Identify type errors
- Handle type shorthand

Ascription

New syntactic forms

t ::= ....
  t as T

New typing rules

$$\frac{\Gamma \vdash t : T}{\Gamma \vdash t \text{ as } T : T} \quad \text{(T-ASCRIBE)}$$

$$v \text{ as } T \rightarrow v \quad \text{(E-ASCRIBE)}$$

$$\frac{t \rightarrow t'}{t \text{ as } T \rightarrow t'} \quad \text{(E-ASCRIBE1)}$$

Interesting Extensions

- References (Chapter 13)
- Exceptions (Chapter 14)
- Subtyping (Chapters 15-16)
  - Most general type
- Recursive Types (Chapters 20, 21)
  - NatList = <Nil: Unit, cons: {Nat, NatList}>
- Polymorphism (Chapters 22-28)
  - length:list $\alpha \rightarrow$ int
  - Append: list $\alpha \rightarrow \alpha \rightarrow$ list α
- Higher-order systems (Chapters 29-32)

Imperative Programs

- Linear types
- Points-to analysis
- Typed assembly language

Summary

- Constructive rules for preventing runtime errors in a Turing complete programming language
- Efficient type checking
  - Code is described in Chapter 10
- Unique types
- Type safety
- But limits programming