Introduction to ML

Mooly Sagiv

Cornell CS 3110 Data Structures and Functional Programming

Typed Lambda Calculus

Chapter 9

Benjamin Pierce

Types and Programming Languages

Call-by-value Operational Semantics

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

v ::=	values
λ x. t	abstraction values

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

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

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

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)

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

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

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

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

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

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

$$\Gamma \vdash t : T$$

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

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

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

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

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

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

Another Example

$\Gamma \vdash t : T$

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

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

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

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

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

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

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

Another Example

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

$$\Gamma \vdash t : T$$

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

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

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

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

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

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

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]

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 $t \rightarrow t'$

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 $x \notin dom(\Gamma)$ then $\Gamma, t \vdash t : T$ with a proof of the same depth [Weakening]
- If $\Gamma, x: S \vdash t : T$ 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 $\Gamma \vdash t' : T$

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

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

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

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

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

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

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
  - …

The ML Programming Language

- General purpose programming language designed by Robin Milner in 1970
  - Meta Language for verification
- Impure Functional Programming Language
  - Eager call by value evaluation
- Static strongly typed (like Java unlike C)
  - Protect its abstraction via type checking and runtime checking
- Polymorphic Type Inference
- Dialects: OCaml, Standard ML, F#
- Can be viewed as typed λ calculus with type inference

C is not Type Safe

```
int j;
union { int  i, int * p  } x;
x.i = 17 ;
j = *(x.p);
```

```
int i, *p;
i =17
p = (int *)  i;
```

Factorial in ML

```
let rec fac n = if n = 0 then 1 else n * fac (n - 1)

// val fac : int -> int = <fun>
```

```
int fac(int n) {
    if (n = 0) return 1 ; else return n * fac (n - 1) ;
  }
```

Factorial in ML

```
let rec fac n = if n = 0 then 1 else n * fac (n - 1)

// val fac : int -> int = <fun>

let rec fac  n : int = if n = 0 then 1 else n * fac (n - 1)

 let rec fac = function
     | 0 -> 1
     | n -> n * fac(n - 1)

let fac n =
    let rec ifac n acc =
        if n=0 then acc else ifac n-1, n * acc
  in ifac n, 1
```

Why Study ML?

- Functional programming will make you think differently about programming
  - Mainstream languages are all about state
  - Functional programming is all about values
- ML is “cutting edge”
  - Polymorphic Type inference
  - References
  - Module system
- Practical (small) Programming Language
- Useful for Java/Scala
- New ideas can help make you a better programmer, in any language

Plan

- Basic Programming in ML
- ML Modules & References
- Type Inference for ML

Simple Types

- Booleans

```
true -: bool = true
false -: bool = false
if …  then … else …types must match
```

- Integers

```
0, 1, 2,… -: int =0, 1, …
+, * ,    -: int * int  -> int
```

- Strings

  “I am a string” -: string =“I am a string”

- Floats

```
1.0, 2., 3.14159, … :- float = 1, 2, 3.14159
```

Scope Rules

- ML enforces static nesting on identifiers
  - To be explained later
- let x = e1 in e2 ≡ (λx.e2) e1
- let x: T = e1 in e2 ≡ (λx:T.e2) e1

Tuples

```
4, 5, "abc" :- (int*int*string)=(4, 5, "abc")
```

```
let max1 (r1, r2) : float =
  if r1 < r2 then r2 else r1
val max1: float * float -> float = fun
```

```
let args = (3.5, 4.5)
val args: float * float = (3.5, 4.5)
```

```
max1 args
:-   float = 4.5
```

let $y(x_1: t_1, x_2: t2, \ldots, x_n : t_n) = e$

Pattern-Matching Tuples

let x_1: t_1, x_2: t_2, …, x_n :t_n = e

```
let max1 (pair : float * float) : float =
  let (r1, r2) = pair in
    if r1 < r2 then r2 else r1
val max1: float * float -> float = fun
```

```
let minmax (a, b) : float * float =
  if a < b then (a, b) else (b, a)
val minmax: float * float -> float *float= fun
```

```
let (mn, mx) = minmax (2.0, 1.0)
val mn float 1
val mx float 2
```

The compiler guarantees the absence of runtime errors

User-Defined Types

```
type day = Sun | Mon | Tue | Wed | Thu | Fri | Sat
```

```
let int_to_day (i : int) : day =
  match i mod 7 with
    0 -> Sun
  | 1 -> Mon
  | 2 -> Tue
  | 3 -> Wed
  | 4 -> Thu
  | 5 -> Fri
  | _ -> Sat
```

User-Defined Types C

```
enum day { Sun, Mon, Tue, Wed, Thu, Fri, Sat };
```

```
day int_to_day (int i)  {
      switch i % 7 {
     case 0: return Sun ;
     case 1: return Mon ;
     case 2: return Tue ;
     case 3: return Wed ;
     case 4: return Thu ;
     case 5: return Fri ;
     default: return Sat ;
    }
}
```

Records

```
type person = {first:string; last:string; age:int}
```

```
{first="John"; last="Amstrong"; age=77}
:- person ={first="John; last="Amstrong"; age=77}
```

```
{first="John"; last="Amstrong"; age=77}.age
 :- int = 77
```

```
let ja = {first="John"; last="Amstrong"; age=77}
 val ja : person = {first="John"; last="Amstrong"; age=77}
```

```
let {first=first; last=last}  = ja
 val first:string="John"
 val last:string ="Amstrong"
```

C structs

```
struct person {char * first; char * last; int age} ;
```

```
struct person ja ={“John”, ”Amstrong”,77} ;
```

```
printf(“%d”, ja.age);
```

```
fn = ja.first; ln = ja.last ; ag = ja.age
```

```
ja.age = 78;
```

Variant Records

- Provides a way to declare Algebraic data types

```
type expression = Number of  int | Plus of expression * expression
```

```
let rec eval_exp (e : expression) : int =
  match e with
     Number(n) -> n
   | Plus (left, right) -> eval_exp(left) + eval_exp(right)
val eval_exp : expression -> int = <fun>
```

```
 eval_exp (Plus(Plus(Number(2), Number(3)), Number(5)))
:- int = 10
```

Variant Records in C

```
struct exp {
    enum {Number, Binop} etype ; /* Select between cases */
    union {
        struct  number { int : num; }
        struct  plus { struct exp *left, *right; }
            }
   }
```

```
int eval_exp (struct exp e)  {
  switch e.etype {
     case Number: return e.number.num ;
     case Binop :  return eval_exp(e.plus.left) + eval_exp(e.plus.right);
   }
}
```

Wrong Variant Records in C

```
struct exp {
    enum {Number, Binop} etype ; /* Select between cases */
    union {
        struct  number { int : num; }
        struct  plus { struct exp *left, *right; }
            }
  }
```

```
int eval_exp (struct exp e)  {
  switch expression.etype {
     case Binop: return e.number.num ;
     case Number :  return eval_exp(e.plus.left) + eval_exp(e.plus.right);
   }
}
```

Scope

- Local nested scopes
- Let constructs introduce a scope

```
let f x = e1 in e2
```

```
let x = 2
and y = 3
in   x + y
```

```
let rec even x = x = 0 || odd (x-1)
    and odd x = not (x = 0 || not (even (x-1)))
in
  odd 3110
```

Polymorphism

- A Polymorphic expression may have many types
- There is a “most general type”
- The compiler infers types automatically
- Programmers can restrict the types
- Pros:
  - Code reuse
  - Guarantee consistency
- Cons:
  - Compile-time
  - Some limits on programming

```
let max1 (r1, r2)  =
  if r1 < r2 then r2 else r1
val max1: ‘a * ‘a -> ‘a = fun
```

```
max1 (5, 7)
: - int = 7
```

```
max1 (5, 7.5)
```

Polymorphic Lists

```
[ ]
 - : 'a list = []
```

```
[2; 7; 8 ]
 - : int list = [2; 7; 8]
```

```
2 :: (7 ::  (8 ::[ ]))
 - : int list = [2; 7; 8]
```

```
[(2, 7) ; (4, 9) ; 5]
                    Error: This expression has type int but an expression
was expected of type
         int * int
```

Functions on Lists

```
let rec length l =
    match l with
   [] -> 0
  | hd :: tl  -> 1 + length tl
val length : 'a list -> int = <fun>
```

length [1; 2; 3] + length [“red”; “yellow”; “green”]
:- int = 6

length [“red”; “yellow”; 3]

Higher Order Functions

- Functions are first class objects
  - Passed as parameters
  - Returned as results
- Practical examples
  - Google map/reduce

Map Function on Lists

- Apply function to every element of list

```
let rec map f arg = function
     [] -> []
   | hd :: tl -> f hd :: (map f tl)

val map : ('a -> 'b) -> 'a list -> 'b list = <fun>
```

```
map (fun x -> x+1) [1;2;3]      [2,3,4]
```

- Compare to Lisp

```
(define map
    (lambda (f  xs)
      (if   (eq? xs ())   ()
         (cons (f  (car xs))  (map f  (cdr xs)))
   )))
```

More Functions on Lists

- Append lists

```
let rec append l1 l2 =
 match l1 with
  | []   -> l2
  | hd :: tl -> hd :: append (tl l2)
val append 'a list -> 'a list -> 'a list
```

```
let rec append l1 l2 =
 match l1 with
  | []   -> []
  | hd :: tl -> hd :: append (tl l2)
val append 'a list -> 'b -> 'a list
```

More Functions on Lists

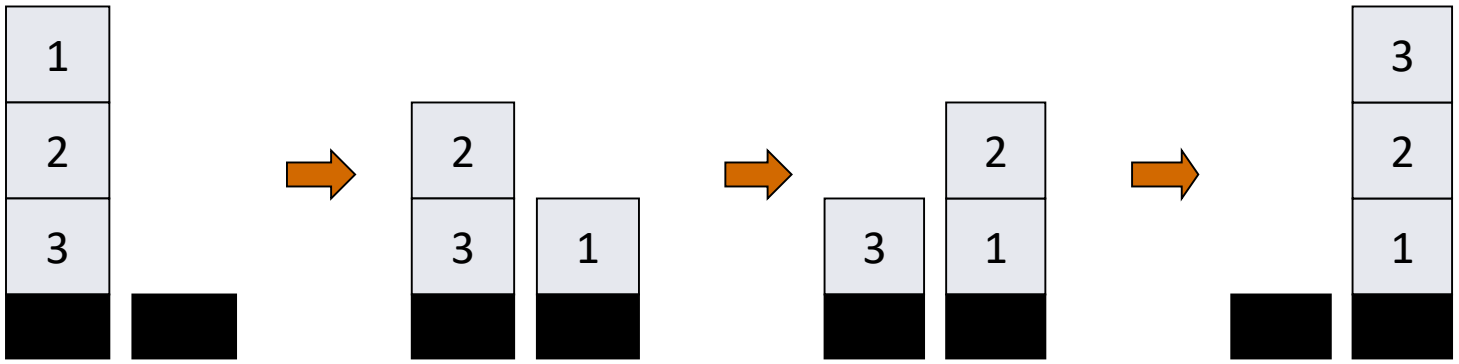
- Reverse a list

```
let rec reverse l =
    match l with
   | [] -> []
   | hd :: tl -> append (reverse tl)    [hd]
val reverse 'a list -> 'a list
```

- Questions
  - How efficient is reverse?
  - Can it be done with only one pass through list?

More Efficient Reverse

```
let rev list =
    let rec aux acc = function
      | [] -> acc
      | h::t -> aux (h::acc) t
    in
      aux [] list
val rev : 'a list -> 'a list = <fun>
```



Currying

```
let plus (x, y) = x + y
val plus : int * int -> int = fun
```

```
let plus (z : int * int) = match z with (x, y) -> x + y
val plus : int * int -> int = fun
```

```
let plus = fun (z : int * int) -> match z with (x, y) -> x + y
val plus : int * int -> int = fun
```

```
let plus x y = x + y
val plus : int -> int -> int
```

```
let  p1 = plus 5
val p1 :  int -> int = fun
```

```
let  p2 = p1 7
val p2 :   int = 12
```

Functional Programming Languages

PL	types	evaluation	Side-effect
scheme	Weakly typed	Eager	yes
ML OCAML F#	Polymorphic strongly typed	Eager	References
Haskell	Polymorphic strongly typed	Lazy	None

Things to Notice

- Pure functions are easy to test

```
prop_RevRev l = reverse(reverse l) == l
```

- In an imperative or OO language, you have to
  - set up the state of the object and the external state it reads or writes
  - make the call
  - inspect the state of the object and the external state
  - perhaps copy part of the object or global state, so that you can use it in the post condition

Things to Notice

Types are everywhere.

```
reverse:: [w] -> [w]
```

- Usual static-typing panegyric omitted...
- In ML, types express high-level design, in the same way that UML diagrams do, with the advantage that the type signatures are machine-checked
- Types are (almost always) optional: type inference fills them in if you leave them out

Recommended ML Textbooks

- L. C. PAULSON: ML for the Working Programmer
- J. Ullman: Elements of ML Programming
- R. Harper: Programming in Standard ML

Recommended Ocaml Textbooks

- Xavier Leroy: The OCaml system release 4.02
  - Part I: Introduction
- Jason Hickey: Introduction to Objective Caml
- Yaron Minsky, Anil Madhavapeddy, Jason Hickey: Real World Ocaml

Summary

- Functional programs provide concise coding
- Compiled code compares with C code
- Successfully used in some commercial applications
  - F#, ERLANG, Jane Street
- Ideas used in imperative programs
- Good conceptual tool
- Less popular than imperative programs