Extensions to Typed Lambda Calculus

Chapter 11, 13

Benjamin Pierce

Types and Programming Languages

Simple Typed Lambda Calculus

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

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

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

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

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

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

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

Type Rules

t ::= terms

x variable

λ x: T. t abstraction

$\Gamma \vdash t : T$

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

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

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

Properties of the type system

- Uniqueness of types
- Linear time type checking
- Type Safety
  - Well typed programs cannot go wrong
    - No undefined semantics
    - No runtime checks
  - 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]

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)

New derived forms

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

Ascription

New syntactic forms

t ::= ….
t as T

New typing rules

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

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

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

Let Bindings

t ::=	terms
x	variable
λ x: T. t	abstraction
let x = t_1 in t_2	let binding

New Evaluation Rules extend $t \rightarrow t'$

let x = v_1 in $t_2 \rightarrow [x \mapsto v_1]$ t2 (E-LETV)

$$\frac{t_1 \rightarrow t'_1}{\text{let } x = t_1 \text{ in } t_2 \rightarrow \text{let } x = t'_1 \text{ in } t_2} \quad \text{(E-LET)}$$

New Typing Rules extend $\Gamma \vdash t : T$

$$\frac{\Gamma \vdash t_1 : T \quad \Gamma, x{:}T \vdash t_2 : T_2}{\Gamma \vdash \text{let } x = t_1 \text{ in } t_2 : T_2} \quad \text{(T-LET)}$$

Pairs

New syntactic forms

t ::= ….	Terms:
{t, t}	pair
t.1	first projection
t.2	second projection
v ::= ….	Values:
{v, v}	pair value
T ::= ….	types:
$T_1 \times T_2$	pair type

New typing rules

$$\frac{\Gamma \vdash t_1 : T_1 \quad \Gamma \vdash t_2 : T_2}{\Gamma \vdash \{t_1 , t_2 \} : T_1 \times T_2} \text{ (T-Pair)}$$

$$\frac{\Gamma \vdash t : T_1 \times T_2}{\Gamma \vdash t.1 : T_1} \text{ (T-Proj1)}$$

$$\frac{\Gamma \vdash t : T_1 \times T_2}{\Gamma \vdash t.2 : T_2} \text{ (T-Proj2)}$$

New Evaluation Rules extends $\rightarrow$

$\{v_1, v_2\}.\ 1 \rightarrow v_1$ (E-PairBeta1)

$\{v_1, v_2\}.\ 2 \rightarrow v_2$ (E-PairBeta2)

$$\frac{t \rightarrow t'}{t.1 \rightarrow t'.1} \text{ (E-Proj1)}$$

$$\frac{t \rightarrow t'}{t.2 \rightarrow t'.2} \text{ (E-Proj2)}$$

$$\frac{t_1 \rightarrow t'_1}{\{t_1,t_2\} \rightarrow \{t'_1,t_2\}} \text{ (E-Pair1)}$$

$$\frac{t_2 \rightarrow t'_2}{\{v_1,t_2\} \rightarrow \{v_1,t'_2\}} \text{ (E-Pair2)}$$

Examples

- {pred 4, if true then false else false}.1
- (λx: Nat $\times$ Nat. x.2) {pred 4, pred 5}

Tuples

New syntactic forms

t ::= ….	Terms:
$\{t_i \ ^{i \in 1..n}\}$	tuple
t.i	projection
v ::= ….	Values:
$\{v_i \ ^{i \in 1..n}\}$	tuple
T ::= ….	types:
$\{ T_i \ ^{i \in 1..n} \}$	tuple type

New typing rules

$$\frac{\text{For each i } \Gamma \vdash t_i : T_i}{\Gamma \vdash \{t_i \ ^{i \in 1..n}\} : \{ T_i \ ^{i \in 1..n} \}} \quad \text{(T-Tuple)}$$

$$\frac{\Gamma \vdash t : \{ T_i \ ^{i \in 1..n} \}}{\Gamma \vdash t.j : T_j} \quad \text{(T-Proj)}$$

New Evaluation Rules extends $\rightarrow$

$\{v_i \ ^{i \in 1..n} \}.\ j \rightarrow v_j$ (E-ProjTuple)

$$\frac{t \rightarrow t'}{t.i \rightarrow t'.i} \quad \text{(E-Proj)}$$

$$\frac{t_j \rightarrow t'_j}{\{v_i \ ^{i \in 1..j-1}, t_i \ ^{i \in j..n}\} \rightarrow \{v_i \ ^{i \in 1..j-1}, t'_j, t_k \ ^{k \in j+1..n}\}} \quad \text{(E-Tuple)}$$

Records

New syntactic forms

t ::= ….		Terms:
	$\{l_i=t_i \ ^{i \in 1..n}\}$	record
	t.l	projection
v ::= ….		Values:
	$\{l_i=v_i \ ^{i \in 1..n}\}$	records
T ::= ….		types:
$\{l_i:T_i \ ^{i \in 1..n}\}$		record type

New Evaluation Rules extends $\rightarrow$

$\{l_i=v_i \ ^{i \in 1..n}\}.\ l_j \rightarrow v_j$ (E-ProjRCD)

$$\frac{t \rightarrow t'}{t.l \rightarrow t'.l} \quad \text{(E-Proj)}$$

$$\frac{t_j \rightarrow t'_j}{\{l_i=v_i \ ^{i \in 1..j-1},\ l_i=t_i \ ^{i \in j..n}\} \rightarrow \{l_i=v_i \ ^{i \in 1..j-1}, l_j=t'_j, l_k=t_k \ ^{k \in j+1..n}\}} \quad \text{(E-Tuple)}$$

New typing rules

$$\frac{\text{For each } i\ \Gamma \vdash t_i : T_i}{\Gamma \vdash \{l_i=t_i \ ^{i \in 1..n}\} : \{l_i: T_i \ ^{i \in 1..n}\}} \quad \text{(T-Tuple)}$$

$$\frac{\Gamma \vdash t: \{l_i: T_i \ ^{i \in 1..n}\}}{\Gamma \vdash t.l_j : T_j} \quad \text{(T-Proj)}$$

Pattern Matching

- An elegant way to access records
- Simultaneous cases
- Checked by the compiler
- Saves a lot of code
- Standard in ML, Haskel, Scala
- Can be expressed in the untyped lambda calculus

Sums

New syntactic forms

t ::= …. Terms:
- inl t tagging(left)
- inr t tagging(right)
- case t of inl x $\Rightarrow$ t | inr x $\Rightarrow$ t case

v ::= …. Values:
- inl v tagged value(left)
- inr v tagged value(right)

T ::= …. types:
- $T_1 + T_2$ sum type

New Evaluation Rules extends $\rightarrow$

$$\frac{t_2 \rightarrow t'_2}{\text{inr } t_2 \rightarrow \text{inr } t'_2} \quad \text{(E-INR)}$$

New typing rules

$$\frac{\Gamma \vdash t_1 : T_1}{\Gamma \vdash \text{inl } t_1 : T_1 + T_2} \quad \text{(T-INL)}$$

$$\frac{\Gamma \vdash t_2 : T_2}{\Gamma \vdash \text{inr } t_2 : T_1 + T_2} \quad \text{(T-INR)}$$

$$\frac{\Gamma, x1 : T1 \vdash t_1 : T \quad \Gamma, x2 : T2 \vdash t_1 : T \quad \Gamma \vdash t : T_1 + T_2}{\Gamma \vdash \text{case of t } x_1 \Rightarrow t_1 \mid \text{inr } x_2 \Rightarrow t_2 : T} \quad \text{(T-CASE)}$$

case (inl v) of inl $x_1 \Rightarrow t_1$ | inr $x_2 \Rightarrow t_2 \rightarrow [x_1 \mapsto v]\, t_1$ (E-CaseINL)

case (inr v) of inl $x_1 \Rightarrow t_1$ | inr $x_2 \Rightarrow t_2 \rightarrow [x_2 \mapsto v]\, t_2$ (E-CaseINR)

$$\frac{t \rightarrow t'}{\text{case t of inl } x_1 \Rightarrow t_1 \mid \text{inr } x_2 \Rightarrow t_2 \rightarrow \text{case t' of inl } x_1 \Rightarrow t_1 \mid \text{inr } x_2 \Rightarrow t_2} \quad \text{(E-Case)}$$

$$\frac{t_1 \rightarrow t'_1}{\text{inl } t_1 \rightarrow \text{inl } t'_1} \quad \text{(E-INL)}$$

A Simple Example

PhyisicalAddr = {firstlast: String, add:String}

VirtualAddr = {name: String, email:String}

Addr = PhisicalAddr+VirtualAddr

inl: PhisicalAddr $\rightarrow$Addr

inr: VirtualAddr $\rightarrow$Addr

```
getName = λa: Addr.
            case a of
                inl x ⇒ x.firstlast
                |
                inr y ⇒ y.name
```

▹ getName : Addr $\rightarrow$ String

Sums and Uniqueness of types

- T-INL and T-INR allow multiple types
- Potential solutions
  - Use “type reconstruction” (Chapter 12)
  - Use subtyping (Chapter 15)
  - User annotation

Sums with Unique Types

New syntactic forms

t ::= ....
- inl t as T — tagging(left)
- inr t as T — tagging(right)
- case t of inl x ⇒ t | inr x ⇒ t case

v ::= ....
- inl v as T — tagged value(left)
- inr v as T — tagged value(right)

T ::= ....
- $T_1 + T_2$ — sum type

Terms:

Values:

types:

New Evaluation Rules extends →

$$\text{case (inl v as T) of inl } x_1 \Rightarrow t_1 \mid \text{inr } x_2 \Rightarrow t_2 \rightarrow [x_1 \mapsto v]\, t_1 \quad \text{(E-CaseINL)}$$

$$\text{case (inr v as T) of inl } x_1 \Rightarrow t_1 \mid \text{inr } x_2 \Rightarrow t_2 \rightarrow [x_2 \mapsto v]\, t_2 \quad \text{(E-CaseINR)}$$

$$\frac{t \rightarrow t'}{\text{case } t \text{ of inl } x_1 \Rightarrow t_1 \mid \text{inr } x_2 \Rightarrow t_2 \rightarrow \text{case } t' \text{ of inl } x_1 \Rightarrow t_1 \mid \text{inr } x_2 \Rightarrow t_2} \quad \text{(E-Case)}$$

$$\frac{t_1 \rightarrow t'_1}{\text{inl } t_1 \text{ as } T \rightarrow \text{inl } t'_1 \text{ as } T} \quad \text{(E-INL)}$$

$$\frac{t_2 \rightarrow t'_2}{\text{inr } t_2 \text{ as } T \rightarrow \text{inr } t'_2 \text{ as } T} \quad \text{(E-INR)}$$

New typing rules

$$\frac{\Gamma \vdash t_1 : T_1}{\Gamma \vdash \text{inl } t_1 \text{ as } T_1 + T_2 : T_1 + T_2} \quad \text{(T-INL)}$$

$$\frac{\Gamma \vdash t_2 : T_2}{\Gamma \vdash \text{inr } t_2 \text{ as } T_1 + T_2 : T_1 + T_2} \quad \text{(T-INR)}$$

$$\frac{\Gamma, x_1 : T_1 \vdash t_1 : T \quad \Gamma, x_2 : T_2 \vdash t_1 : T \quad \Gamma \vdash t : T_1 + T_2}{\text{case of } t \; x_1 \Rightarrow t_1 \mid \text{inr } x_2 \Rightarrow t_2 : T} \quad \text{(T-CASE)}$$

Variants

New syntactic forms

t ::= ….
  <l=t> as T
  case t of $<l_i = x_i> \Rightarrow t_i^{\ i \in 1..n}$

Terms:
tagging
case

$$\frac{t_i \rightarrow t'_i}{<l_i=t_i> \text{ as } T \rightarrow <l_i=t'_i> \text{ as } T} \quad \text{(E-VARIANT)}$$

New typing rules

v ::= ….
  <l=v> as T

Values:
tagged value

$$\frac{\Gamma \vdash t_j : T_j}{\Gamma \vdash <l_j=t_j> \text{ as } <l_i = T_i^{\ i \in 1..n}> : <l_i = T_i^{\ i \in 1..n}>} \quad \text{(T-VARIANT)}$$

T ::= ….
  $<l_i = T_i^{\ i \in 1..n}>$

types:
type of variants

$$\frac{\text{For each } i\ \Gamma, x_i : T_i \vdash t_i : T \qquad \Gamma \vdash t : <l_i = T_i^{\ i \in 1..n}>}{\text{case } t \text{ of } <l_i = x_i> \Rightarrow t_i^{\ i \in 1..n} : T} \quad \text{(T-CASE)}$$

New Evaluation Rules extends $\rightarrow$

$$\text{case } (<l_j = v> \text{ as } T) \text{ of } <l_i = x_i> \Rightarrow t_i^{\ i \in 1..n} \rightarrow [x_j \mapsto v]\, t_j \quad \text{(E-CaseVariant)}$$

$$\frac{t \rightarrow t'}{\text{case } t \text{ of } <l_i = x_i> \Rightarrow t_i^{\ i \in 1..n} \rightarrow \text{case } t' \text{ of } <l_i = x_i> \Rightarrow t_i^{\ i \in 1..n}} \quad \text{(E-CASE)}$$

A Simple Example

PhyisicalAddr = {firstlast: String, add:String}

VirtualAddr = {name: String, email:String}

Addr = <physical: PhisicalAddr: virtual:VirtualAddr>

```
getName = λa: Addr.
            case a of
                <physical x> ⇒ x.firstlast
                |
                <virtual:y> ⇒ y.name
```

▹ getName : Addr → String

Options (Optional Values)

OptionalNat =<none: Unit, some: Nat>

Table = Nat →OptionalNat

emptyTable = λn: Nat. <none:unit> as OptionalNat

▹ emptyTable :Table

x = case t (5) of
<none:u> ⇒ 99
|
<some:v> ⇒ v

▹ x : Nat

extendTable = λt: Table. λ m: Nat. λv: Nat.
λn: Nat.
If equal n m then <some=v> as OptionalNat
else t n

▹ extendTable :Table → Nat → Nat → Table

Enumeration

Weekday= <monday:Unit, Tuesday:Unit, Wednesday Unit, thursday:Unit,Friday: Unit>

nextBuisnessDay = λd: Weekday.
case d of
<monday:u> ⇒ <tuesday:unit> as Weekday
|
<tuesday:u> ⇒ <wednesday:unit> as Weekday
|
<wednesday:u> ⇒ <thursday:unit> as Weekday
|
<thursday:u> ⇒ <friday:unit> as Weekday
|
<friday:u> ⇒ <monday:unit> as Weekday

▹ nextBuisnessDay :Weekday → Weekday

Single-Field Variant

V = <l: T>

Motivating Single-Field Variant

dollars2euros =λd : Float. timesfloat d 1.325

▹ dollars2euros : Float → Float

euros2dollars =λd : Float. timesfloat d 0.883

▹euros2dollars : Float → Float

mybankbalance =39.5

▹mybankbalance : Float

euros2dollars (dollars2euros mybankbalance)

▹39.49: Float

dollars2euros (dollars2euros mybankbalance)

▹50.660: Float

Using Single-Field Variant

DollarAmount = <dollars: float>

EuroAmount = <euros: float>

dollars2euros =λd : DollarAmount.
  case d of <dollars= x> timesfloat x 1.325 as EuroAmount

▹ dollars2euros : DollarAmount → EuroAmmount

euros2dollars =λd : EuroAmount.
  case d of <euros= x> timesfloat x 0.883 as DollarAmount

▹ euros2dollars : EuroAmount → DollarAmmount

mybankbalance =<dollars: 39.5> as DollarAmount

▹ mybalance : DollarAmount

euros2dollars (dollars2euros mybankbalance)

▹ 39.49 : DollarAmount

dollars2euros (dollars2euros mybankbalance)

▹ error type mismatch dollars2euros expect an argument of type DollarAmmount and not EuroAmmount

Dynamic

- Allow investigating the type at runtime
- Data which spans multiple machines
- Databases
- Files

General Recursion

ff = λ ie: Nat → Bool.
    λx. Nat.
        if iszero x then true
        else if iszero (pred x) then false
        else ie (pred (pred x))

▷ ff : Nat → Bool → (Nat → Bool)

iseven = fix ff

▷ iseven : Nat → Bool

iseven 7

▷ false: Bool

Recursion

New syntactic forms

t ::= ….
fix t

Terms:
fixedpoint of t

New Evaluation Rules extends $\rightarrow$

fix (λx. T_1: t_2) $\rightarrow$ [x $\mapsto$(fix (λx. T_1: t_2)] t_2 (E-FixBeta)

$$\frac{t \rightarrow t'}{\text{fix } t \rightarrow \text{fix } t'} \quad \text{(E-FIX)}$$

New typing rules

$$\frac{\Gamma \vdash t : T_1 \rightarrow T_1}{\Gamma \vdash \text{fix } t : T_1} \quad \text{(T-FIX)}$$

New derived forms

letrec x: T_1 = t_1 in t_2 $\doteq$ let x = (fix (λx: T_1. t_1)) in t_2

Lists

- Constructs a list of items
- Straightforward to define
- Becomes more interesting with polymorphism

References (Chapter 13)

- A mechanism to mutate the store
- Create aliases
- Type safety can be enforced if free is not allowed

Basics

```
r = ref 5;
r : Ref Nat

!r ;
5 :  Nat


r := 7;
unit :  Unit


!r ;
7 :  Nat
```

Side Effects and Sequencing

```
r := 7;
unit :  Unit
```

```
!r ;
7 :  Nat
```

```
(r :=succ(!r) ; !r)
8 :  Nat
```

```
(r :=succ(!r) ;  r : succ(!r); r := succ(!r) ; !r)
11 :  Nat
```

References and Aliasing

```
r := 7;
unit :  Unit


s = r ;
s :  ref Nat

(r :=succ(!r) ; !s)
8 :  Nat

(a := 1 ;  a := !b)

 a := !b
```

Dangling References

- Operations like free can create dangling references
  - Confusing
  - Violate type safety

(x = ref(5); free(x) ; y = ref(true))

Type Rules for References

$\Gamma \vdash t : T$

$$\frac{\Gamma \vdash t : T}{\Gamma \vdash \text{ref } t : \text{ref } T} \text{ (T-REF)}$$

$$\frac{\Gamma \vdash t : \text{ref } T}{\Gamma \vdash !t : T} \text{ (T-DEREF)}$$

$$\frac{\Gamma \vdash t_1 : \text{ref } T \qquad \Gamma \vdash t_2 : T}{\Gamma \vdash t_1 := t_2 : \text{Unit}} \text{ (T-DEREF)}$$

Evaluating References

- Need to allocate memory
- Record types of locations
- Can be recorded at compile-time
- Preserve type safety

Summary

- Typed lambda calculus can be extended to cover many programming language features
- Concise
- Not meant for programming
- It is possible to enforce type safety in realistic situations
- Combine with runtime techniques
- Impact language design