



EXTENSIONS TO SIMPLY-TYPED LAMBDA CALCULUS

1

BASIC TYPES

- Practical programming needs numerical and Boolean values and types. (Of course these can be encoded in lambda calculus.)

$e ::= \dots$
| v
| $e_1 \text{ bop } e_2$ (binary ops: $+$, $-$, $*$, $/$, $<$, $>$, $=$, $<=$, $>=$, and, or)
| $\text{uop } e$ (unary ops: \sim , not, pred, succ)

$v ::= \backslash x.e$
| $\dots, -1, 0, 1, 2, \dots$ [all integers]
| true | false

$t ::= \dots$
| int
| bool

- Semantics and typing rules for all the binary ops and unary ops are straight forward
- We dropped the type annotation from abstraction for brevity

ASSOCIATIVITY AND PRECEDENCE

- A grammar can be used to define associativity and precedence among the operators in an expression.
 - E.g., + and - are left-associative operators in mathematics;
 - * and / have higher precedence than + and - .
 - $a + b + c = (a + b) + c$; $a ** b ** c = a ** (b ** c)$
- Consider the more interesting grammar G_1 for arithmetic:

Expr ::= Expr + Term
 | Expr - Term
 | Term

Term ::= Term * Factor
 | Term / Factor
 | Term % Factor
 | Factor

Factor ::= Primary ** Factor
 | Primary

Primary ::= 0 | ... | 9 | (Expr)

AN AMBIGUOUS EXPRESSION GRAMMAR G_2

$Expr \rightarrow Expr \ Op \ Expr \mid (Expr) \mid Integer$

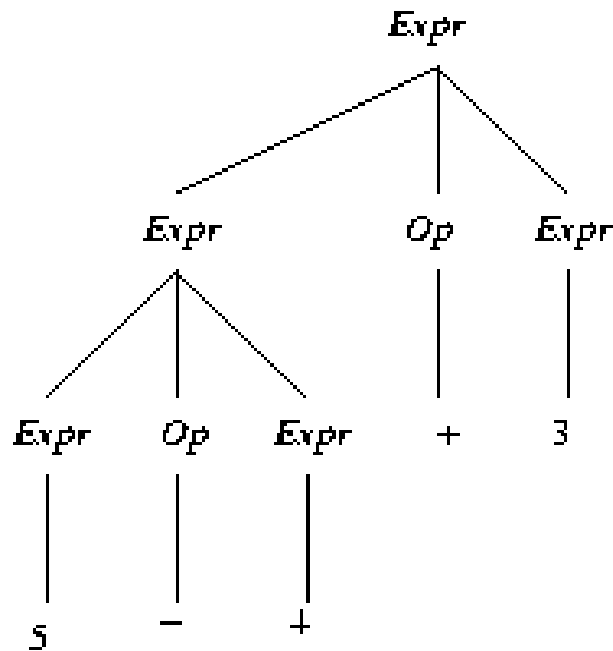
$Op \rightarrow + \mid - \mid * \mid / \mid \% \mid **$

Notes:

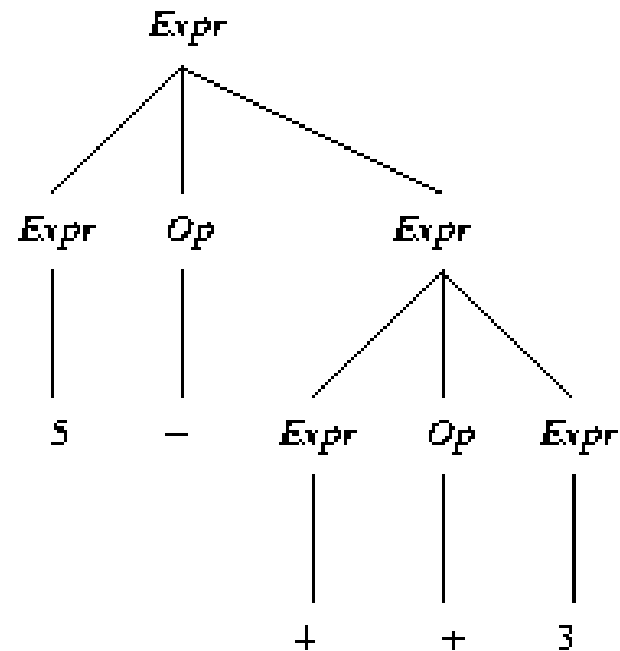
- G_2 is equivalent to G_1 , *i.e.*, its language is the same.
- G_2 has fewer productions and non-terminals than G_1 .
- However, G_2 is ambiguous.
- Ambiguity can be resolved using the associativity and precedence table



AMBIGUOUS PARSE OF 5-4+3 USING GRAMMAR G_2



(a)



(b)

LET BINDING

- It is useful to bind intermediate results of computations to variables:

New syntax:

<code>e ::= x</code>	(a variable)
<code>true</code> <code>false</code>	(a boolean value)
<code>if e1 then e2 else e3</code>	(conditional)
<code>\x.e</code>	(a nameless function)
<code>e1 e2</code>	(function application)
<code>let x = e1 in e2</code>	(let expression)

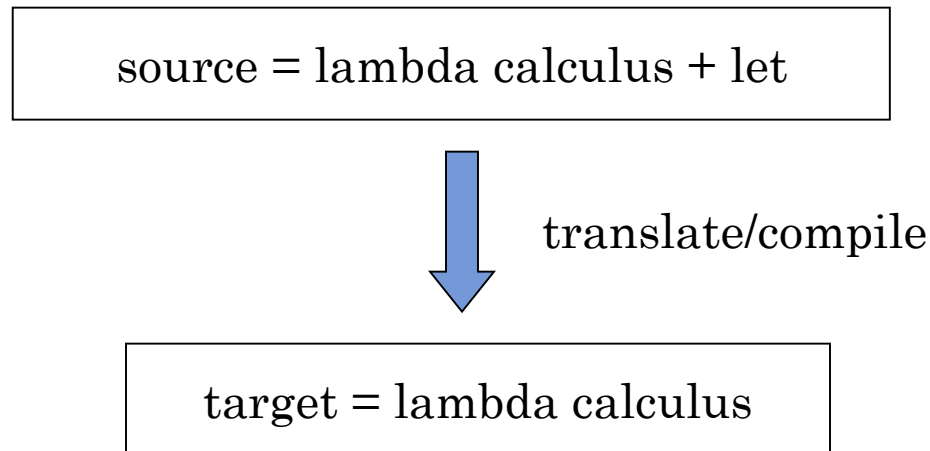
 `x` is bound in `e2` (which is the scope of `x`)

CALL-BY-VALUE SEMANTICS AND TYPING

$$e1 \rightarrow e1'$$
$$\frac{}{\text{let } x=e1 \text{ in } e2 \rightarrow \text{let } x =e1' \text{ in } e2} \text{ [e-let]}$$
$$\frac{}{\text{let } x=v \text{ in } e2 \rightarrow e2 [v/x]} \text{ [e-letv]}$$
$$\frac{G \vdash e1 : t1 \quad G, x:t1 \vdash e2 : t2}{G \vdash \text{let } x=e1 \text{ in } e2 : t2} \text{ [t-let]}$$

IMPLEMENTATION OF LET EXPRESSIONS

- Question: can we implement this idea in pure lambda calculus?



LET EXPRESSIONS

- Question: can we implement this idea in the lambda calculus?

translate (let $x = e1$ in $e2$) =
 $(\lambda x.e2) e1$

LET EXPRESSIONS

- Question: can we implement this idea in the lambda calculus?

$$\text{translate (let } x = e1 \text{ in } e2) =$$
$$(\lambda x. \text{translate } e2) (\text{translate } e1)$$

LET EXPRESSIONS

- Question: can we implement this idea in the lambda calculus?

translate (let x = e1 in e2) =

$(\lambda x. \text{translate } e2) (\text{translate } e1)$

translate (x) = x

translate $(\lambda x.e)$ = $\lambda x.\text{translate } e$

translate (e1 e2) = (translate e1) (translate e2)

THE PRINCIPLE OF “BOUND VARIABLE NAMES DON’T MATTER”

When you write

```
let x = \z.z z in  
  let y = \w.w in (x y)
```

you assume you can change the declaration of y to a declaration of v (or other name) provided you systematically change the uses of y . E.g.:

```
let x = \z.z z in  
  let v = \w.w in (x v)
```

provided that the name you pick doesn’t conflict with the free variables of the expression. E.g.:

```
let x = \z.z z in  
  let x = \w.w in (x x) ← bad, original x captured
```

STATIC VS. DYNAMIC SCOPING

- The **scope** of a name is the collection of expressions and/or statements which can access the name binding.
- In static scoping, a name is bound for a collection of statements according to its position in the source program → determined at compile time (static)
- In dynamic scoping, the valid association for a name X, at any point P of a program, is the most recent (in the temporal sense) association created for X which is still active when control flow arrives at P → determined at run time (dynamic)
- Most modern languages use static (or *lexical*) scoping.

STATIC VS. DYNAMIC SCOPING (II)

let **x** = v1 in

let y = (let **x** = v2 in x)

in **x**

- This expression evaluates to
 - v1 (static scoping)
 - v2 (dynamic scoping)

PAIRS

- Programming languages offer compound types.
- Simplest is *pairs*, or *2-tuples*.
- We introduce one new value $\{v1, v2\}$
- One new product type: $t1 * t2$.

PAIRS (SYNTAX)

$e ::= \dots$
| $\{e1, e2\}$
| $e.1$
| $e.2$

$v ::= \dots$
| $\{v1, v2\}$

$t ::= \dots$
| $t1 * t2$

expressions:
pair
first projection
second projection

values:
pair value

types:
product type

PAIRS (EVALUATION)

$[e \rightarrow e']$

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

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

$$\frac{e \rightarrow e'}{e.1 \rightarrow e'.1} \text{ (E - Proj1)}$$

$$\frac{e \rightarrow e'}{e.2 \rightarrow e'.2} \text{ (E - Proj2)}$$

$$\frac{e_1 \rightarrow e_1'}{\{e_1, e_2\} \rightarrow \{e_1', e_2\}} \text{ (E - Pair1)}$$

$$\frac{e_2 \rightarrow e_2'}{\{v_1, e_2\} \rightarrow \{v_1, e_2'\}} \text{ (E - Pair2)}$$

EXAMPLE EVALUATIONS

Left to right evaluation:

- $\{\text{if } 3+2 > 0 \text{ then true else false, succ } 0\}.1$
- $\rightarrow \{\text{if } 5 > 0 \text{ then true else false, succ } 0\}.1$
- $\rightarrow \{\text{if true then true else false, succ } 0\}.1$
- $\rightarrow \{\text{true, succ } 0\}.1$
- $\rightarrow \{\text{true, 1}\}.1$
- $\rightarrow \text{true}$

Pairs must be evaluated to values before passing to functions:

- $(\backslash x:\text{int}*\text{int}. x.2) \{\text{pred } 1, 6/2\}$
- $\rightarrow (\backslash x:\text{int}*\text{int}. x.2) \{0, 6/2\}$
- $\rightarrow (\backslash x:\text{int}*\text{int}. x.2) \{0, 3\}$
- $\rightarrow \{0, 3\}.2$
- $\rightarrow 3$

PAIRS (TYPING)

$[\Gamma \vdash e : t]$

$$\frac{\Gamma \mid - e_1 : t_1 \quad \Gamma \mid - e_2 : t_2}{\Gamma \mid - \{e_1, e_2\} : t_1 \times t_2} \quad (\text{T - Pair})$$

$$\frac{\Gamma \mid - e : t_1 \times t_2}{\Gamma \mid - e.1 : t_1} \quad (\text{T - Proj1})$$

$$\frac{\Gamma \mid - e : t_1 \times t_2}{\Gamma \mid - e.2 : t_2} \quad (\text{T - Proj2})$$

TUPLES

- Tuples generalize from pairs: binary product \rightarrow n-ary product

$e ::= \dots$
| $\{e_1, \dots, e_n\}$ (or $\{e_i^{i \in 1..n}\}$)
| $e.i$

expressions:
tuple
projection

$v ::= \dots$
| $\{v_1, \dots, v_n\}$

values:
tuple value

$t ::= \dots$
| $t_1 * \dots * t_n$ (or $\{t_i^{i \in 1..n}\}$)

types:
tuple type

TUPLE EVALUATION AND TYPING

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

$$\frac{e \rightarrow e'}{e.i \rightarrow e'.i} \quad (\text{E - ProjTuple1})$$

$$\frac{e_j \rightarrow e_j'}{\{v_1, \dots, v_{j-1}, e_j, \dots, e_n\} \rightarrow \{v_1, \dots, v_{j-1}, e_j', \dots, e_n\}} \quad (\text{E - Tuple})$$

$$\frac{\text{for each } i : \Gamma \mid - e_i : t_i}{\Gamma \mid - \{e_i^{i \in 1..n}\} : \{t_i^{i \in 1..n}\}} \quad (\text{T - Tuple})$$

$$\frac{\Gamma \mid - e : \{t_i^{i \in 1..n}\}}{\Gamma \mid - e.j : t_j} \quad (\text{T - Proj})$$

- Note that order of elements in tuple is significant.
- Evaluation is from left to right.
- Projection is done after tuple becomes value.

RECORDS

- Straightforward to extend tuples into records
- Elements are indexed by labels:
 - $\{y=10\}$
 - $\{id=1, salary=50000, active=true\}$
- The order of the record fields is often insignificant in most PL
 - $\{y=10, x=5\}$ is the same as $\{x=5, y=10\}$
- To access fields of a record:
 - `a.id`
 - `b.salary`
- Syntax and semantic rules left as an exercise.

SUMS

- Program needs to deal with heterogeneous collection of values – values that can take different shapes:
 - A binary tree node can be:
 - A leaf node, or
 - An interior node
 - An abstract syntax tree node of λ -calculus can be:
 - A variable
 - A function abstraction, or
 - An application, etc.
- *Sum* type: union of two types
- More generally, *variant* type: union of n types.

SUM (SYNTAX)

$e ::= \dots$
| $\text{inl } e$
| $\text{inr } e$
| $\text{case } e \text{ of } \text{inl } x \Rightarrow e1 \mid \text{inr } x \Rightarrow e2$

$v ::= \dots$
| $\text{inl } v$
| $\text{inr } v$

$t ::= \dots$
| $t1 + t2$

expressions:
injection (left)
injection (right)
case

values:
injection value (left)
injection value (right)

types:
sum type

SUMS (EXAMPLE)

- There are two types:
 - $\text{faculty} = \{\text{empid: int, position: string}\}$
 - $\text{student} = \{\text{stuid: int, level: int}\}$
- Define a sum type:
 - $\text{personnel} = \text{faculty} + \text{student}$
- We can “inject” element of *faculty* or *student* type into *personnel* type. Think of *inl* and *inr* as functions:
 - $\text{inl} : \text{faculty} \rightarrow \text{personnel}$
 - $\text{inr} : \text{student} \rightarrow \text{personnel}$
- To use a elements of sum type, we use the case expression:
 $\text{getid} = \lambda p : \text{personnel} .$
 $\text{case } p \text{ of}$
 $\text{inl } x \Rightarrow x.\text{empid}$
 $| \text{inr } x \Rightarrow x.\text{stuid}$

SUMS (SEMANTICS)

$$\frac{}{\text{case (inl } v) \text{ of inl } x_1 \Rightarrow e_1 \mid \text{inr } x_2 \Rightarrow e_2 \rightarrow e_1[v/x_1]} \quad (\text{E - CaseInl})$$

$$\frac{}{\text{case (inr } v) \text{ of inl } x_1 \Rightarrow e_1 \mid \text{inr } x_2 \Rightarrow e_2 \rightarrow e_2[v/x_2]} \quad (\text{E - CaseInr})$$

$$\frac{e \rightarrow e'}{\text{case } e \text{ of inl } x_1 \Rightarrow e_1 \mid \text{inr } x_2 \Rightarrow e_2 \rightarrow \text{case } e' \text{ of inl } x_1 \Rightarrow e_1 \mid \text{inr } x_2 \Rightarrow e_2} \quad (\text{E - Case})$$

$$\frac{e \rightarrow e'}{\text{inl } e \rightarrow \text{inl } e'} \quad (\text{E - Inl})$$

$$\frac{e \rightarrow e'}{\text{inr } e \rightarrow \text{inr } e'} \quad (\text{E - Inr})$$