

CS4212: Compiler Design

Week 8: First-Class Functions

Ilya Sergey

ilya@nus.edu.sg

ilyasergey.net/CS4212/

First-Class Functions

Untyped lambda calculus

Substitution

Evaluation

“Functional” languages

- Languages like OCaml, Scala, Haskell, Scheme, Python, C#, Java 8, Swift
- Functions can be passed as arguments (e.g. map or fold)
- Functions can be returned as values (e.g. compose)
- Functions nest: inner function can refer to variables bound in the outer function

```
let add = fun x -> fun y -> x + y
```

```
let inc = add 1
```

```
let dec = add -1
```

```
let compose = fun f -> fun g -> fun x -> f (g x)
```

```
let id = compose inc dec
```

- How do we implement such functions?
 - in an interpreter? in a compiled language?

(Untyped) Lambda Calculus

- The **lambda calculus** is a minimal programming language.
 - Note: we're writing (fun x -> e) lambda-calculus notation: $\lambda x. e$
- It has **variables**, **functions**, and **function application**.
 - That's it!
 - It's Turing Complete.
 - It's the foundation for a *lot* of research in programming languages.
 - Basis for “functional” languages like Scala, OCaml, Haskell, etc.

Abstract syntax in OCaml:

```
type exp =  
  | Var of var      (* variables *)  
  | Fun of var * exp (* functions: fun x → e *)  
  | App of exp * exp (* function application *)
```

Concrete syntax:

```
exp ::=  
  | x          variables  
  | fun x → exp functions  
  | exp1 exp2 function application  
  | ( exp )    parentheses
```

Free Variables and Scoping

```
let add = fun x → fun y → x + y
```

```
let inc = add 1
```

- The result of `add 1` is a function
- After calling `add`, we can't throw away its argument (or its local variables) because those are needed in the function returned by `add`.
- We say that the variable `x` is *free* in `fun y → x + y`
 - Free variables are defined in an outer scope
- We say that the variable `y` is *bound* by “`fun y`” and its scope is the body “`x + y`” in the expression `fun y → x + y`
- A term with no free variables is called *closed*.
- A term with one or more free variables is called *open*.

Values and Substitution

- The only values of the lambda calculus are (closed) functions:

```
val ::=
    | fun x → exp      functions are values
```

- To *substitute* a (closed) value v for some variable x in an expression e
 - Replace all *free occurrences* of x in e by v .
 - In OCaml: written `subst v x e`
 - In Math: written $e\{v/x\}$
- Function application is interpreted by *substitution*:

```
(fun x → fun y → x + y) 1
= subst 1 x (fun y → x + y)
= (fun y → 1 + y)
```

Note: for the sake of examples we may add integers and arithmetic operations to the “pure” untyped lambda calculus.

Operational Semantics of Lambda Calculus

- Substitution function (in Math):

$x\{v/x\} = v$	<i>(replace the free x by v)</i>
$y\{v/x\} = y$	<i>(assuming $y \neq x$)</i>
$(\text{fun } x \rightarrow \text{exp})\{v/x\} = (\text{fun } x \rightarrow \text{exp})$	<i>(x is bound in exp)</i>
$(\text{fun } y \rightarrow \text{exp})\{v/x\} = (\text{fun } y \rightarrow \text{exp}\{v/x\})$	<i>(assuming $y \neq x$)</i>
$(e_1 e_2)\{v/x\} = (e_1\{v/x\} e_2\{v/x\})$	<i>(substitute everywhere)</i>

Operational Semantics of Lambda Calculus

- Substitution function (in Math):

$x\{v/x\} = v$	<i>(replace the free x by v)</i>
$y\{v/x\} = y$	<i>(assuming $y \neq x$)</i>
$(\text{fun } x \rightarrow \text{exp})\{v/x\} = (\text{fun } x \rightarrow \text{exp})$	<i>(x is bound in exp)</i>
$(\text{fun } y \rightarrow \text{exp})\{v/x\} = (\text{fun } y \rightarrow \text{exp}\{v/x\})$	<i>(assuming $y \neq x$)</i>
$(e_1 e_2)\{v/x\} = (e_1\{v/x\} e_2\{v/x\})$	<i>(substitute everywhere)</i>

- Examples:

$$\begin{aligned} (x y) \{(\text{fun } z \rightarrow z z)/y\} \\ = x (\text{fun } z \rightarrow z z) \end{aligned}$$

$$\begin{aligned} (\text{fun } x \rightarrow x y) \{(\text{fun } z \rightarrow z z)/y\} \\ = \text{fun } x \rightarrow x (\text{fun } z \rightarrow z z) \end{aligned}$$

$$\begin{aligned} (\text{fun } x \rightarrow x) \{(\text{fun } z \rightarrow z z)/x\} \\ = \text{fun } x \rightarrow x \quad // x \text{ is not free!} \end{aligned}$$

Demo: Programming in Lambda Calculus

- <https://github.com/cs4212/week-08-lambda-2023>
- lambda.ml – untyped lambda-calculus
- lambda_int.ml – untyped lambda-calculus with integers
- stlc.ml – simply-typed lambda-calculus

Free Variable Calculation

- An OCaml function to calculate the set of free variables in a lambda expression:

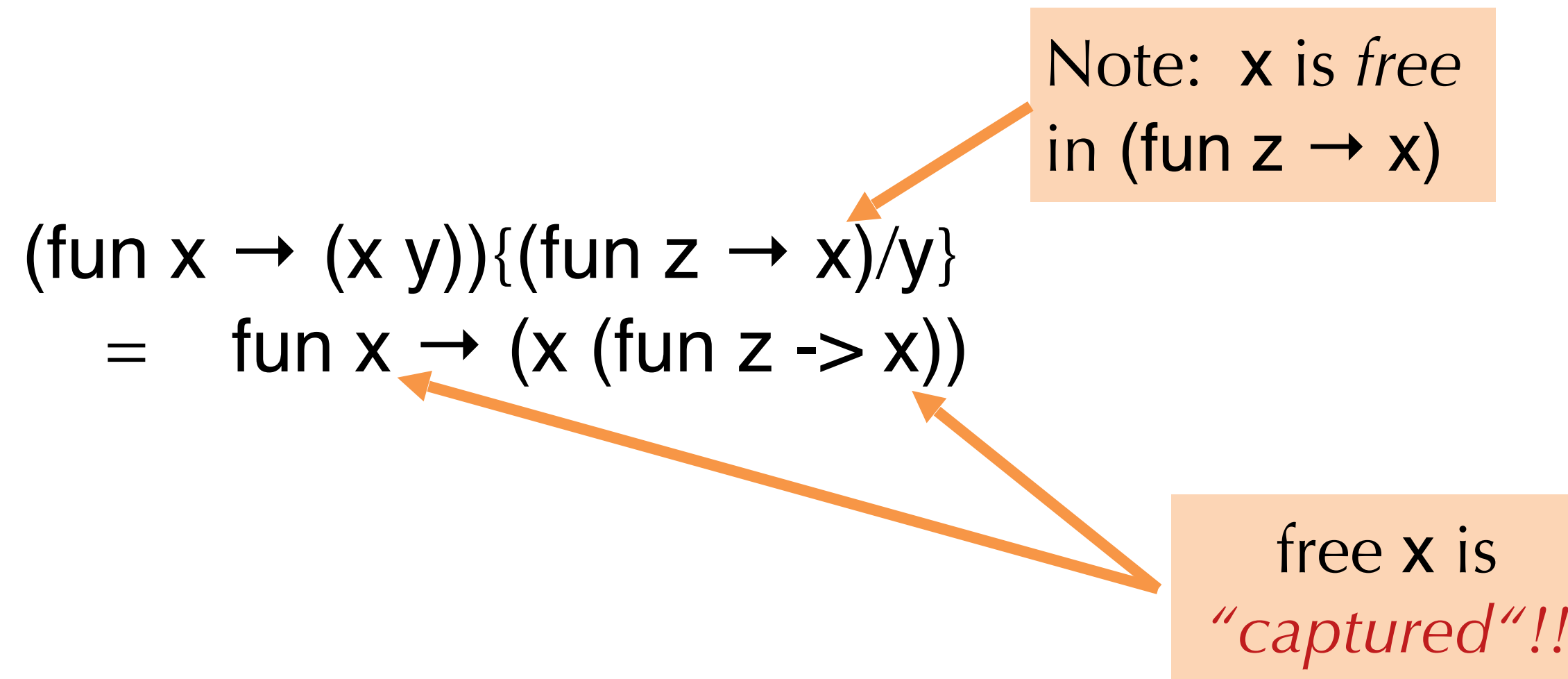
```
let rec free_vars (e:exp) : VarSet.t =  
  begin match e with  
  | Var x      -> VarSet.singleton x  
  | Fun(x, body) -> VarSet.remove x (free_vars body)  
  | App(e1, e2) -> VarSet.union (free_vars e1) (free_vars e2)  
  end
```

- A lambda expression e is *closed* if `free_vars e` returns `VarSet.empty`
- In mathematical notation:

$$\begin{aligned} \text{fv}(x) &= \{x\} \\ \text{fv}(\text{fun } x \rightarrow \text{exp}) &= \text{fv}(\text{exp}) \setminus \{x\} \quad (\text{'x' is a bound in exp}) \\ \text{fv}(\text{exp}_1 \text{ exp}_2) &= \text{fv}(\text{exp}_1) \cup \text{fv}(\text{exp}_2) \end{aligned}$$

Variable Capture

- Note that if we try to naively "substitute" an open term, a bound variable might *capture* the free variables:



- Usually *not* the desired behaviour
 - This property is sometimes called "dynamic scoping"
The meaning of " x " is determined by where it is bound dynamically, not where it is bound statically.
 - Some languages (e.g. emacs lisp) are implemented with this as a "feature"
 - But: it leads to hard-to-debug scoping issues

Alpha Equivalence

- Note that the names of bound variables don't matter to the semantics
 - i.e. it doesn't matter which variable names you use, as long as you use them consistently:

(fun **x** → y **x**) is the "same" as (fun **z** → y **z**)

the choice of "x" or "z" is arbitrary, so long as we consistently rename them

Two terms that differ only by consistent renaming of *bound* variables are called *alpha equivalent*

- The names of *free* variables **do** matter:
 - (fun x → **y** x) is *not* the "same" as (fun x → **z** x)

Intuitively: y and z can refer to different things from some outer scope

Students who cheat by “renaming variables” are trying to exploit alpha equivalence...

Fixing Substitution

- Consider the substitution operation:

$$e_1\{e_2/x\}$$

- To avoid capture, we define substitution to pick an alpha equivalent version of e_1 such that the bound names of e_1 don't mention the free names of e_2 .
 - Then do the "naïve" substitution.

For example: $(\text{fun } x \rightarrow (x \ y))\{(\text{fun } z \rightarrow x)/y\}$
 $= (\text{fun } x' \rightarrow (x' (\text{fun } z \rightarrow x)))$

rename x to x'

This is fine:

$$\begin{aligned} & (\text{fun } x \rightarrow (x \ y))\{(\text{fun } x \rightarrow x)/y\} \\ &= (\text{fun } x \rightarrow (x (\text{fun } x \rightarrow x))) \\ &= (\text{fun } a \rightarrow (a (\text{fun } b \rightarrow b))) \end{aligned}$$

Operational Semantics

- Specified using just two inference rules with judgments of the form $\text{exp} \Downarrow \text{val}$
 - Read this notation as “program exp evaluates to value val ”
 - This is *call-by-value* semantics: function arguments are evaluated before substitution

$$\frac{}{v \Downarrow v}$$

“Values evaluate to themselves”

$$\frac{\text{exp}_1 \Downarrow (\text{fun } x \rightarrow \text{exp}_3) \quad \text{exp}_2 \Downarrow v \quad \text{exp}_3\{v/x\} \Downarrow w}{\text{exp}_1 \text{ exp}_2 \Downarrow w}$$

“To evaluate function application: Evaluate the function to a value, evaluate the argument to a value, and then substitute the argument for the function. ”

Demo: Implementing the Interpreter

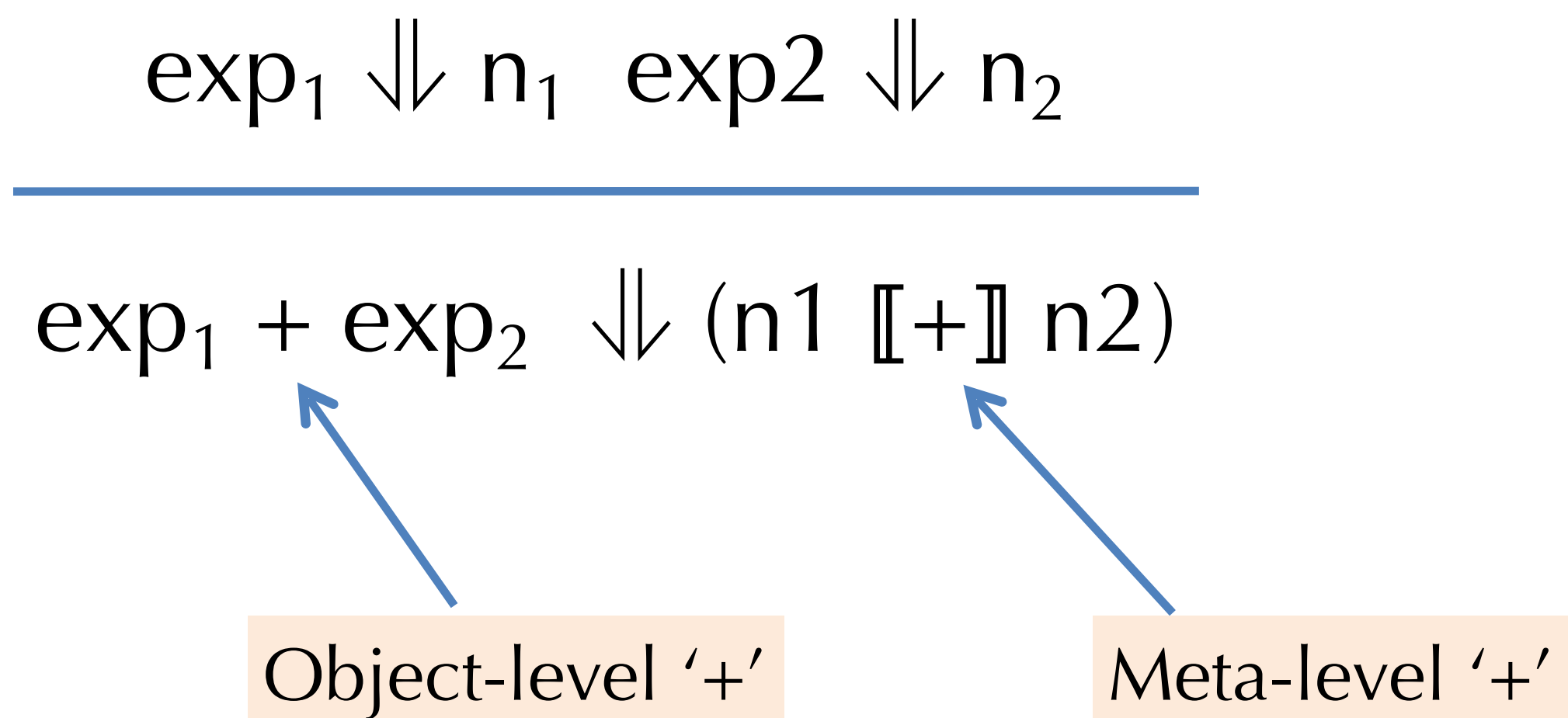
- <https://github.com/cs4212/week-08-lambda-2023>
- **lambda.ml** – **untyped lambda-calculus**
- lambda_int.ml – untyped lambda-calculus with integers
- stlc.ml – simply-typed lambda-calculus

Adding Integers to Lambda Calculus

exp ::=
| ...
| n *constant integers*
| exp₁ + exp₂ *binary arithmetic operation*

val ::=
| fun x → exp *functions are values*
| n *integers are values*

n{v/x} = n *constants have no free vars.*
(e₁ + e₂){v/x} = (e₁{v/x} + e₂{v/x}) *substitute everywhere*



Next Week

Semantic Analysis via Types