Lecture 14

CIS 4521/5521: COMPILERS

Announcements

- Midterm: March 6th
  - In class this Thursday!
  - One-page, letter-sized, ***hand-written***, double-sided "cheat sheet" of notes permitted
  - Coverage: interpreters, x86, IRs, LLVM IR, calling conventions, lexing, parsing (up to today)
  - See Ed post for previous exams

- Looking ahead: HW4: Oat compiler Frontend
  - released later week (i.e., before Spring Break)
  - Due: Wednesday, March 26th at 10:00pm

- Final exam:
  Thursday, May 8, 2025: 12:00pm to 2:00pm in TOWN 100

See HW4

OAT V. 1

Untyped lambda calculus

Substitution

Evaluation

FIRST-CLASS FUNCTIONS

"Functional" languages

- Oat (like C) has only top-level functions
- In languages like OCaml, Haskell, Scheme, Python, C#, Java, Swift
  - Functions can be passed as arguments (e.g., to `map` or `fold`)
  - Functions can be returned as values (e.g., from `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
  - OCaml: `(fun x -> e)`
  - lambda-calculus notation: λ 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 Scheme, ML, Haskell, etc.

Lambda calculus is the c. elegans of programming languages. Its minimal (but not too minimal!) form lets us deeply characterize its properties.

c. elegans – with 6 chromosomes, fully sequence DNA, 302 neurons, and extremely well-studied life cycle is a "model organism" used in biology.

Untyped Lambda Calculus Syntax

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*
| exp_1 exp_2	*function application*
| (exp)	*parentheses*

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. These can be encoded as lambda terms.

Lambda Calculus Operational Semantics

- Substitution function (in Math):

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

- Examples:

```
(x y){(fun z → z z)/y}
      =  x (fun z → z z)

(fun x → x y){(fun z → z z)/y}
      =  fun x → x (fun z → z z)

(fun x → x){(fun z → z z)/x}
      =  fun x → x    // x is not free!
```

Free Variables and Scoping

```
let add = fun x → fun y → x + y
let inc = add 1
```

- The result of `add 1` is itself 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*.

Free Variable Calculation

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

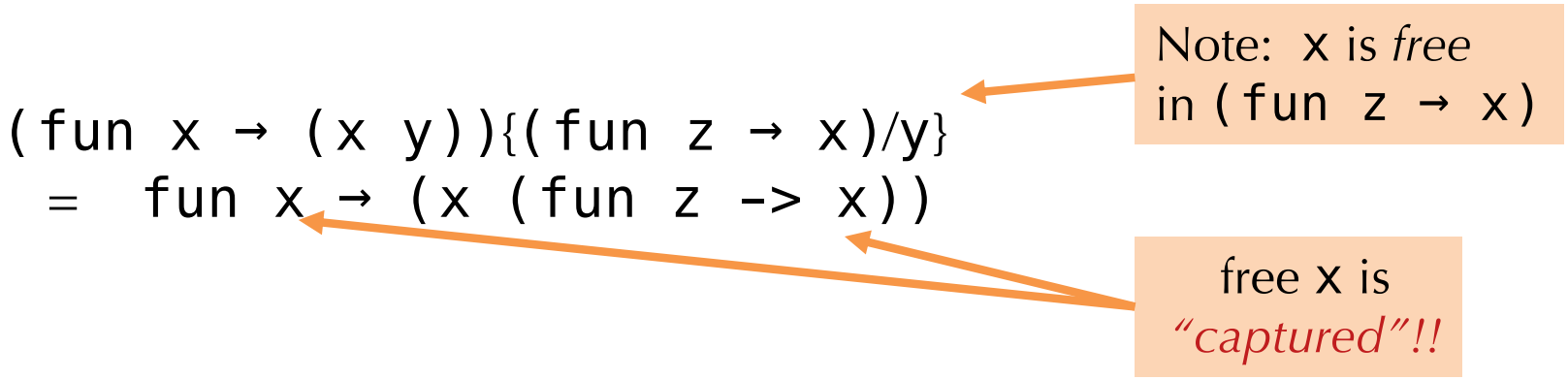
```
let rec free_vars (e:exp) : VarSet.t =
  begin match e with
    | Var x         -> VarSet.singleton x
    | Fun(x, body) -> VarSet.remove x (free_vars body)
    | App(e1, e2)  -> VarSet.union (free_vars e1) (free_vars e2)
  end
```

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

$$
\begin{aligned}
fv(x) &= \{x\} \\
fv(\texttt{fun}\ x \rightarrow exp) &= fv(exp) \setminus \{x\} \qquad \textit{(`x' is a bound in exp)} \\
fv(exp_1\ exp_2) &= fv(exp_1) \cup fv(exp_2)
\end{aligned}
$$

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 behavior
  - 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, if 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.
  - Harder said than done! (Many "obvious" implementations are wrong.)
  - Then do the "naïve" substitution.

For example: `(fun x → (x y))`{`(fun z → x)`/y}

= `(fun x' → (x' (fun z → x))`

rename x to x'

On the other hand, this requires no renaming:

`(fun x → (x y))`{`(fun x → x)`/y}

= `(fun x → (x (fun x → x))`

= `(fun a → (a (fun b → b))`

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"

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

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

See fun.ml

Examples of encoding Booleans, integers, conditionals, loops, etc., in untyped lambda calculus.

IMPLEMENTING THE INTERPRETER