Lecture 15

EECS 483: COMPILER CONSTRUCTION

Announcements

- Midterm
  - Grades will be released after we review the results in next Wednesday's class (3/20).
- HW4: OAT v.1.0
  - Parsing & translation to LLVM IR
  - Helps to start early!
  - **Due: Tuesday, March 26th**
- CSE Distinguished Lecture Series **today** :
  - Ion Stoica, UC Berkeley
  - *An AI stack: from cloud orchestration to LLM evaluation*
  - 3:30pm in Lurie Engineering Center, Johnson Rooms

UNTYPED LAMBDA CALCULUS

(Untyped) Lambda Calculus

- The lambda calculus is a minimal programming language.
  - Note: we're writing `(fun x -> e)` lambda-calculus notation: λ x. e

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*

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 an 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.

rename x to x'

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

- Key operation: *capture-avoiding substitution*: $e_2\{e_1/x\}$
  - replaces all free occurrences of x in e_2 by e_1
  - must respect scope and alpha equivalence (renaming)

- *Reduction Strategies*
  Various ways of simplifying (or "*reducing*") lambda calculus terms.
  - call-by-value evaluation:
    - simplify the function argument *before* substitution
    - *does not* reduce under lambda (a.k.a. fun)
  - call-by-name evaluation:
    - *does not* simplify the argument before substitution
    - *does not* reduce under lambda
  - weak-head normalization:
    - does not simplify the argument before substitution
    - does not reduce under lambda
    - works on open terms, "suspending" reduction at variables
  - normal order reduction:
    - *does* reduce under lambda
    - first does weak-head normalization and then recursively continues to reduce
    - works on open terms – guaranteed to find a "normal form" if such a form exists

A "normal form" is one that has no substitution steps possible, i.e., there are no subterms of the form (fun x → e1) e2 anywhere.

CBV Operational Semantics

- This is *call-by-value* semantics:
  function arguments are evaluated before substitution

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

"Values evaluate to themselves"

$$\frac{exp_1 \Downarrow (\text{fun } x \rightarrow exp_3) \qquad exp_2 \Downarrow v \qquad exp_3\{v/x\} \Downarrow w}{exp_1\ exp_2 \Downarrow w}$$

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

CBN Operational Semantics

- This is *call-by-name* semantics:
  function arguments are evaluated before substitution

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

"Values evaluate to themselves"

$$\frac{exp_1 \Downarrow (\text{fun } x \rightarrow exp_3) \qquad exp_3\{exp_2/x\} \Downarrow w}{exp_1\ exp_2 \Downarrow w}$$

"To evaluate function application: Evaluate the function to a value, substitute the argument into the function body, and then keep evaluating. "

See fun.ml

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

IMPLEMENTING THE INTERPRETER