Lecture 16

CS131: COMPILERS

Announcements

- Midterm: November 19th
  - In class
  - One-page, letter-sized, double-sided "cheat sheet" of notes permitted
  - Coverage: interperters, x86, LLVMlite, lexing, parsing
  - See examples of previous exam on Blackboard

- HW4: OAT v. 1.0
  - Parsing & basic code generation
  - **Due: November 25th**

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.

See fun.ml

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

IMPLEMENTING THE INTERPRETER

CBV Operational Semantics

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

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

"Values evaluate to themselves"

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

"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

Eval2, Eval3

ENVIRONMENT BASED INTERPRETERS

Environment Based Interpreters

- Thread through an ***environment***, which maps variables to their values.
  - extend the environment when doing a function call
  - lookup variables in the current environment

- To properly handle first-class functions: use closures
  - a ***closure*** is a pair of a
    (1) a datastructure representing the saved environment, and
    (2) the function body definition

See cc.ml

CLOSURE CONVERSION

Closure Conversion Summary

- A ***closure*** is a pair of an environment and a code pointer
  - the environment is a map data structure binding variables to values
  - environment could just be a list of the values (with known indices)

- Building a closure value:
  - code pointer is a function that takes an extra argument for the environment: A → B becomes (Env * A → B)
  - body of the closure "projects out" then variables from the environment
  - creates the environment map by bundling the free variables

- Applying a closure:
  - project out the environment, invoke the function (pointer) with the environment and its "real" argument

- Hoisting:
  - Once closure converted, all functions can be lifted to the top level