Computing Science (CMPUT) 325

Nonprocedural Programming

Department of Computing Science
University of Alberta

An Interpreter based on Context and Closure

- We will build an interpreter for a Lisp-like language
- No named functions, only lambda functions
- Similar to the `eval` function in Lisp:
- `(eval expr)`
  Take an s-expression and keep reducing it
- Main issues: reduction order, how to do efficient computations
- We will introduce a new, efficient approach different from basic NOR, AOR
- Based on two concepts: **context** and **closure**

Context and Closure Main Idea

- Context = current variables and their bindings
- Closure = a pair: an s-expression, together with a context
- We will write `eval` for:
  - The s-expression to evaluate...
  - ...in the current context
    (which might contain some closures)

Language - Simple Lisp Variant

- Variables: e.g. `x, y, z`
- Constant expressions: `(quote e)`
- Arithmetic: `(+ e1 e2), (- e1 e2), (* e1 e2), (/ e1 e2)`
- Relations and Logic: `(eq e1 e2), (and e1 e2), (not e)`
- Primitives for s-expressions: `(car e), (cdr e), (cons e1 e2), (atom e), (null e)`

Language - continued

- `(if e1 e2 e3)`
- lambda function `(lambda (x1 ... xk) e)`
- function call `(e e1 ... ek)`
- simple block `(let (x1.e1) ... (xk.ek) e)`
- (optional) recursive block
  `(letrec (x1.e1) ... (xk.ek) e )`

Notes for `let`

- We use `(let (x1.e1) ... (xk.ek) e)`
- Lisp uses `(let ((x1 e1) ... (xn en)) e)`
- Our form is a little simpler to implement, same meaning
- We can also write it as a lambda function application:
- `((lambda (x1 ... xk) e) e1 ... ek)`
- It does exactly the same!

Why Not Just Use Beta Reductions?

- For β-reduction we need to:
- Determine the scope of each parameter
- Detect potential name conflicts
- Implement variable renaming (α-reductions)
- Implement direct substitution
- It is possible but not very efficient
- Main problem: need to check all the above repeatedly after each substitution step

New Approach

- Key idea: *delay* the substitutions
  by using Contexts and Closures
- A technique used in real Lisp interpreters
- Will help us understand compilation as well

Context - Main Idea

- Remember function application
- Example: `(λx | (+ x 4)) 2`
- Need to replace the x in the body by 2
- So far, we have done this immediately by substitution: `(+ 2 4)`
- Instead, we can keep the body as-is, and **remember the binding** $x \rightarrow 2$
- A **context** is a data structure that keeps track of such variable bindings

Definition of Context

- A context is a list of bindings
- $n_1 \rightarrow v_1, ...n_k \rightarrow v_k$
- where n_i are identifiers and v_i are expressions
- A v_i can also be a “closure” containing all information about applying a lambda function

Evaluation with a Context

- Start of evaluation: always begin with an empty context
  - Compare with other programming languages, where we may have some global variables already bound to values before we start computing
- In the middle of evaluating an expression,
  the context is usually non-empty

Example

- Application `(`λ`x | (+ x 4)) 2`
- To evaluate:
- Build a context $x \to 2$
- $x \to 2$ means that x is bound to 2
- Now evaluate `(+ x 4)` in this context
- When we need the arguments for +,
  we get the binding for x from the context

Evaluation with a Context - Observations

- Substitutions are delayed to the point where the value of a variable is really needed for the evaluation to continue
- Variables are left “free” (such x in `(+ x 4)` above)
- Variable is bound “as needed”, if binding can be found in the context

Definition of Context

- A Context is a list of pairs of the form $n \to v$
- n is a name
- v is either an expression or a closure
- A context is used to record and lookup name bindings
- A context can be *extended*
  when a new pair $n \to v$ is created
  in a function application

Definition of Closure

- A closure is a pair [f, CT]
- f is a lambda function
- CT is a (possibly empty) context
- Remember - a lambda function consists of two parts
- function parameters e.g. `(x y)`
- the body - the definition of the function e.g. `(+ x y)`

More about Closure

- How to use the information in a closure [f, CT]:
- When function f is applied...
- we know its parameters and definition
- From the context CT, we get values
  for the variables in f's body
- Next: details about the process of interpretation

Mini-History of Closures

- Why is a closure called a closure?
- Concept developed in the 1960s by Landin, when he developed the concept of SECD machine (see later)
- He was one of the first to realize that abstract lambda calculus can be used as a basis for real computation
- What we call “free” variables now,
  were called “open” variables then
- A closure “closes” an open variable
  by binding it to a value

Function Application in a Context

- When interpretation of a program starts,
  the context is empty
- When a function is applied:
- Evaluate the arguments in the current context
- Evaluate the functional part in the current context
- Extend the context

Extending a Context

- Steps to extend the context:
- Bind parameter names to the evaluated arguments
- Add these bindings to current context to form the next context
- Evaluate the body of the function in this extended context

Example

- Evaluate `(λx | (+ x 4)) 2`
- Start in empty context, [].
- Evaluate argument 2 in current context, []. Result is 2
- Evaluate the function part, `(λx | (+ x 4))`, in current context. Result is `(λx | (+ x 4))`
- Note: these two steps are trivial here. But in general, both for the argument(s) **and** the function part could be function applications which we need to reduce

Example Continued

- Extend the context:
- bind parameter name x to evaluated argument 2:
  $x \rightarrow 2$
- Add binding to current (empty) context:
  $[] \cup x \rightarrow 2 = [x \rightarrow 2]$
- Evaluate body `(+ x 4)` in extended context $[x \rightarrow 2]$
- More about evaluation later

An Implementation of Context for Interpreter

- First, need a data structure to represent a context
- One possible choice: Two lists, name list and value list
- Both lists are “in sync” - for each name there is a corresponding value in the same location in the other list

Name List and Value List

- Each is a list of lists
- One sublist corresponds to the names and values in one function call
- Name list is a list of lists of atoms
- Value list is a list of lists of s-expr
  that the names are bound to

Example - Name and Value Lists

- Name list `((x y) (z) (w s))`
- Value list `((1 2) ((lambda (x) (* x x))) ((a b) e))`
- List of three sublists corresponding to three (nested) lambda function applications
- In previous notation, this implements the context `{x→1, y→2, z→(lambda (x) (* x x)), w→(a b), s→e}`
- Compare to call stack, stack frames in most programming languages' runtime model

Name Lookup

- Search for a name:
- Walk synchronously over both name and value lists
- If a name is found:
- The s-exp in the same position in the value list is its binding
- Next slide:
  function `assoc(x, n, v)` for name lookup

- `assoc` iterates over sublists of n and v (in sync)
- `locate` iterates over elements in one such pair of sublists

```
assoc(x, n, v)
  = if null(n) then nil  /* x not in n */
    else if member(x, car(n))
              then locate(x, car(n), car(v))
    else assoc(x, cdr(n), cdr(v))

  locate(x, l, m)
  = if eq(x, car(l)) then car(m)
    else locate(x, cdr(l), cdr(m))
```

The Interpreter Evaluator

- We will define a function called `eval` that can evaluate any s-expression
- Note: our `eval` function is **not** part of the language that we interpret
- To avoid confusion between the two languages, we will use square brackets: `eval[e, n, v]`

The `eval` Function - Preliminaries

- `eval[e, n, v]`: the result of applying our evaluator to expression `e`, in the context defined by name list `n` and value list `v`
- Notation:
- `e, e1, e2, ...` well-formed expressions
  `x, x1, x2, ...` atoms used as variables
  `n, n1, n2, ...` names
  `v, v1, v2, ...` values
  `a, b, s` and other letters ... arbitrary S-exprs
  `(a . b)` for `cons(a, b)`
- We define `eval[e, n, v]` for each of the 18 cases that we support in our language, as per the list in last lecture (repeated on next slide)

Language - Simple Lisp Variant

- Variables: e.g. `x, y, z`
- Constant expressions: `(quote e)`
- Arithmetic: `(+ e1 e2)`, `(- e1 e2)`, `(* e1 e2)`, `(/ e1 e2)`
- Relations and Logic: `(eq e1 e2)`, `(and e1 e2)`, `(not e)`
- Primitives for s-expressions: `(car e)`, `(cdr e)`, `(cons e1 e2)`, `(atom e)`, `(null e)`
- `(if e1 e2 e3)`
- **lambda function** `(lambda (x1 ... xk) e)`
- **function call** `(e e1 ... ek)`
- simple block `(let (x1.e1) ... (xk.ek) e)`
- (optional) recursive block `(letrec (x1.e1) ... (xk.ek) e )`

Evaluation of Variables and Constants

- We use Fun here but the translation to Lisp is straightforward (see code on eClass)
- Evaluation of a variable `x`: lookup in name list `n`, return corresponding value in `v`
  - `eval[x, n, v] = assoc(x, n, v)`
- Evaluation of a constant: just return it.
  - `eval[(quote s), n, v] = s`

Evaluation of Arithmetic, Relational and Structural Expressions

- General idea: call `eval` on all arguments first
- Then call through to the corresponding built-in function to do the work
- Example:
  `eval[(+ e1 e2), n, v] = eval[e1, n, v] + eval[e2, n, v]`
- Same for `-, *, /`
- Same for single-argument functions:
- Example: `eval[(car e), n, v] = car(eval[e, n, v])`

More Examples

```
eval[(cdr e), n, v] = cdr(eval[e, n, v])
eval[(cons e1 e2), n, v] = cons(eval[e1, n, v],
                                eval[e2, n, v])
eval[(atom e), n, v] = atom(eval[e, n, v])
eval[(null e), n, v] = null(eval[e, n, v])
eval[(and e1 e2), n, v] = and(eval[e1, n, v],
                              eval[e2, n, v])
eval[(not e), n, v] = not(eval[e, n, v])
eval[(eq e1 e2), n, v] = eq(eval[e1, n, v],
                            eval[e2, n,v])
```

Evaluation of Conditional Expressions

- `(if e1 e2 e3)` where `e1` is the test, `e2` is the then-part, and `e3` the else-part
- The first argument is always evaluated.
  Then either the second or the third argument is evaluated, depending on the value of the first argument

```
eval[(if e1 e2 e3), n, v] =
    if eval[e1, n, v] then
        eval[e2, n, v]
    else
        eval[e3, n, v]
```

Evaluation of Lambda Functions

- A lambda function evaluates **to a closure** which contains:
- The body of the lambda function
- The variable list - names of function parameters, such as `(x y)` in `(lambda (x y) ...)`
- The context in which the body should be evaluated **when the function is eventually applied**
- Remember: the context is implemented as name list and value list

Notation and One Implementation

- `C` .. a closure
- The four parts contained in a closure:
- `parms(C)`, `body(C)`, `names(C)` and `values(C)`
- For example, we can use dotted pairs to build the closure:
- `eval[(lambda y e), n, v]`
  `= cons(cons(y, e), cons(n, v))`
  `= ((y . e). (n . v))`
- Here, if the resulting closure is `C`, then `y` is `parms(C)`, `e` is `body(C)`, `n` is `names(C)` and `v` is `values(C)`
- Implementing these 4 functions is just `caar, cadr, cdar, cddr`

`evalList`

- A helper function for function application:
- Call `eval` on a whole list of expressions and collect results
- (We could use map here)

```
evalList[L, n, v] =
     if null(L) then nil
     else cons(eval[car(L), n, v],
               evalList[cdr(L), n, v])
```

Eval for Function Application

```
eval[(e e1 ... ek), n, v] =
   eval[body(c),
        cons(parms(c), names(c)),
        cons(z, values(c))]
```

- Here, `c = eval[e, n, v]` is the closure from evaluating the function `e`
- `z = evalList[(e1 ...  ek), n, v]` is the list of given arguments in the function application, each evaluated in the current context
- The two `cons` statements **extend the context** with the arguments of the current function, and their bindings
- Finally, we call `eval` for `body(c)` in this extended context
- That's it! If you understand this clearly, then you understand the interpreter. We will do some examples soon.

Evaluation of `let` Expressions

Recall that `let` is just a special case of function application:

```
   (let (x1.e1) ... (xk.ek) e)
= ((lambda (x1 ... xk) e) e1 ...ek)
```

- Therefore `eval` for `let` is very similar to function application:

```
eval[(let (x1.e1) ... (xk.ek) e), n, v]
  = eval[e, cons((x1 ... xk), n), cons(z, v)]

where z = evalList[(e1 ... ek), n, v]
```

Summary of Interpreter

- We developed a design for an interpreter based on context and closure
- We chose some data structures and wrote code in Fun
- The interesting parts are: evaluating lambda functions as closures, and function application
- Next, we look at examples of evaluation, and an interpreter written in Lisp
- (we skipped recursive let for now)