Computing Science (CMPUT) 325

Nonprocedural Programming

Martin Müller

Department of Computing Science
University of Alberta
mmueller@ualberta.ca

Winter 2016

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** $x \to 2$
- 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” representing the state of an incomplete evaluation (see later)

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 `(λx | (+ 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 $n \rightarrow v$
- 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 $n \rightarrow v$ 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 in next lecture