CSE 341: Programming Languages

Dan Grossman

Winter 2008

Lecture 17— Implementing languages, especially higher-order functions

Where are we

- Today:
  - Finish static vs. dynamic typing (arguments 2–5)
  - Learn how closures are actually implemented (key to hw 5)
- Friday: Modularity in Scheme
- Monday: Ruby basics
- Later: More concepts and contrasts
  - At least as important as programming details
  - (for life and, say, the final)

Implementing Languages

Mostly 341 is about language meaning, not "how can an implementation do that", but it's important to "dispel the magic".

At super high-level, there are two ways to implement a language $\boldsymbol{A}$:

- Write an *interpreter* in language $\boldsymbol{B}$ that evaluates a program in $\boldsymbol{A}$
- Write a *compiler* in langage $\boldsymbol{B}$ that translates a program in $\boldsymbol{A}$ to a program in language $\boldsymbol{C}$ (and have an implementation of $\boldsymbol{C}$)

In theory, this is just an implementation decision.

HW5: An interpreter for MUPL in Scheme.

Most interesting thing about MUPL: higher-order functions.

An interpreter

A "direct" language implementation is often just writing our evaluation rules for our language in another language.

- "eval" takes an environment and an expression and returns a value (the subset of expressions that we define to be answers)
- "eval" uses recursion
  - Example: To evaluate an addition expression, evaluate the two subexpressions under the same environment, then...
- For homework 5, expressions & environments are all we need
  - Exceptions or mutation can require more inputs/outputs to "eval"

Implementing Higher-Order Functions

The magic: How is the “right environment” around for lexical scope (the environment from when the function was defined)?

Lack of magic: Implementation keeps it around!

Interpreter:

- The interpreter has a “current environment”
- To evaluate a function (expression), create a closure (value), a *pair* of the function and the environment.
- Application will now apply a closure to an argument: Interpret function body, but instead of using “current environment”, use closure’s environment extended with the argument.

Note: This is directly implementing the semantics from week 3.

Is that expensive?

Building a closure is easy; you already have the environment.

Since environments are immutable, it's easy to share them.

Still, a given closure doesn't need most of the environment, so for *space efficiency* it can be worth it to make a new smaller environment holding only the function's free variables.

- Challenge problem in homework 5

Compiling Higher-Order Functions

The key to the interpreter approach: The interpreter has an explicit environment and can "change" it to implement lexical scope.

We can also *compile* to a language without free variables:
Instead of an *implicit* environment, we pass an *explicit* environment to every function.

- As with interpreter, we build a closure to evaluate functions.
- But all functions now take one extra argument.
- Application passes a closure's code its own environment for the extra argument.
- Evaluating variables uses this extra argument.
  - Compiler translates them to environment-reads.

Plus: Lots of data-structure optimizations so variable-lookup is fast (often a read from a known-size record).