CSE 341: Programming Languages

Dan Grossman
Spring 2004
Lecture 17—Closure Conversion
Today

- Some terminology and motivation for language translation
- The closure-conversion source-to-source translation
- Some specifics for your homework

Why learn closure conversion:

- Help reason about functional programs
- Explicit closure construction is an idiom in languages without first-class functions
- A great example of source-to-source transformation
Language Translation

One way to implement a language is to translate it to another language (that presumably has an implementation).

Equivalence is, of course, key.

In translation, there are 3 languages involved (source, target, translation-implementation (a.k.a. meta))

If source-language = target-language, called a source-to-source translation. If result is a subset, it can simplify the implementation.

HW5:

- source-language = target-language = “minfun”
- meta-language = Scheme
- target has no functions with free variables
Embedded Language

To add a bit more confusion:

• “minfun” abstract-syntax is written with Scheme expressions (using a bunch of define-struct definitions)

• the implementation for the target is written in Scheme (a function called evaluate)

But we saw this in HW3 too: We embedded the “propositional logic” language in ML and implemented it with an ML function (called eval)

One new twist because I’m nice:

• There’s also a Scheme function parse for converting “minfun” concrete syntax to “minfun” abstract syntax.

• But this is just for writing tests; closure-conversion operates on abstract syntax (using make- and selector functions).
Closure Conversion

For any program, we need an equivalent program where any function body accesses data only through its parameters.

So `(fun (x) (fun (y) (+ x y)))` is no good.

Key idea: Change the program to keep track of environments itself, rather than relying on the implementation.

(This is roughly what compilers for functional languages do.)
The key ideas

This is the rough idea for \((\text{fun } (x) \ (\text{fun } (y) \ (+ \ x \ y)))\).

1. Have code take an extra argument.
   - \((\text{fun } (y) \ e)\) becomes \((\text{fun } (\text{env } y) \ e')\)

2. Translate functions to pairs of code and environment.
   - \((\text{fun } (y) \ e)\) becomes \((\text{pr } (\text{fun } (\text{env } y) \ e') \ \text{lst})\) where \(\text{lst}\) is a list of the variables in scope (where the function is defined).

3. Free-variables become environment-access expressions.
   - \(e\) becomes \((+ \ (\text{fst } \text{env}) \ y)\).

4. Function application must pass environment (next page)
Function Application

Given (app e1 e2), e1 will be translated to a pair of code and environment.

So we want something like (in pseudocode)

let closure = e1 
let arg = e2
((#1 closure) (#2 closure) arg)

But we don't have let, so you have to do this with a new function of two arguments (no big deal).

In other words, we extract the code and pass it the environment and the "real" argument.
Arbitrary Depth

When we are translating a program and we reach a function or free-variable, we may already be inside any number of outer functions.

For variables:

- We need an ordered list (a stack) of free variables and the environment-variable for the result, so we can create an expression that, at run-time, gets the right element from the list.
- We also need the local variable(s) so we don’t do anything to them.

For functions, we build a pair:

- The pair’s environment is a list made out of the local variable(s) added to the (outer) environment-variable for the result.
- The pair’s code is a function with one more argument and the body translated (with appropriate free-variable stack, etc.)
Everything Else and Fresh Variables

For all the other cases, just recursively convert.

Homework complicated slightly by “what if you’re not in a function”. The fun and app cases require us to make up new variable names.

• They better not shadow or get shadowed.
• Scheme has a primitive gensym that is just what we need.
  – Every time you evaluate (gensym) you get a symbol that has never been used before.
  – Example: make an increment function with a fresh name:
    (let ([x (gensym)])
      (make-fun (list x) (add x 1)))