C311 – Homework 6: Removing Recursion and Symbolic Variables from Jam

Corky Cartwright

Produced: November 22, 2005— Due: noon, Monday, November 21, 2005

Overview

Your assignment is to transform a dialect of untyped Jam to continuation passing style (CPS) and to convert the Jam environment representation to use static distance coordinates. The assignment should be done in three separate phases: (*i*) modifying your interpreter for Step I of Assignment 5 by restricting the interpreter to *eager* evaluation and splitting the recursive `let` construct into separate recursive `letrec` and non-recursive `let` constructs; (*ii*) modifying the parser to rename variables to eliminate the shadowing of variables by nested variable declarations (`map` parameters, `let` bindings, `reclet` bindings), and (*iii*) transforming Jam programs to CPS, leaving the interpreter from phase (*i*) unchanged; and (*iv*) the conversion of Jam program text to static distance coordinate form, which must be supported by modifying the environment representation in the Jam interpreter.

This is a challenging assignment and will be worth 200 points instead of 100 points.

Phase *I*

As a first phase of this assignment, you will modify your Phase 1 parser and interpreter from Assignment 5 to support only *eager* evaluation and split recursive `let` into two constructs: (*i*) `letrec`, which is *recursive* `let` with right hand sides limited to `map` constructions, and (*ii*) `let`, which is ordinary non-recursive `let` abbreviating the application of a `map` to the right hand sides. These modifications to the Jam language simplifies the transformation required to implement Phase *II*.

In Java, the only public evaluation method in the `Interpreter` class must be called `eval()` instead of the former name `eagerEval()` used in As-

signment 5. In Scheme, the functions `make-eval-file` and `make-eval-string` must return a function of *no* arguments (a thunk) instead of a function of one argument as in Assignment 5.

Phase *II*

Modify your parser to prevent shadowing variable names by converting the name of each Jam variable from x to $x{:}d$ where d is the lexical depth of the declaration of x. If x is free (not allowed in legal programs) then it has lexical depth 0. If it occurs inside one level of lexical nesting, it has lexical depth 1. For example, the program

```
(map x to x)(7)
```

becomes

```
(map x:1 to x:1)(7)
```

after renaming. The renamed variables be confused with existing variable names because `:` is not a legal character in variable names read by the parser.

In Java, add a method

```
public AST unshadow()
```

in the `Interpreter` class that performs this transformation on the Jam AST associated with `this`.

In Scheme, define a top-level function

```
;; unshadow: AST -> AST
(define (unshadow an-ast) ...)
```

Since the `unshadow` operation is not bundled with either parsing or syntax checking, you will subsequently be able to support both ordinary interpretation (as performed in Phase I above) and interpretation after converting a program to CPS form.

The unshadowing transformation permits `let` constructs to be re-interpreted as `let*` constructs without affecting program semantics. The correctness of our rules for CPS transformation hinges on this identity.

Phase *III*

Write a postprocessor (function from abstract syntax to abstrat syntax) for your parser that transforms a Phase I Jam program to equivalent CPS

form. Specifically, given a program M, your postprocessor will generate the program *Cps*[`map x to x`, M'] where M' is M converted to unshadowed form and *Cps* is a binary function mapping Jam program text to Jam program defined by the transformation rules given below. These rules are a loosely based on the exposition in Friedman, *et al.*, Chapter 8.

These rules presume that variables have been renamed to prevent "holes in scope" (nested variables with the same name) as described in Phase II. They will not work correctly on programs that shadow variable names without Phase *II* being performed first.

For the purposes of this assignment, we will consider operator applications (both unary and binary) as syntactic sugar for applications of corresponding primitive operations. Hence operator applications are treated just like primitive applications.

In Java, formulate the CPS postprocessor as a method

```
public AST convertToCPS()
```

in your `Interpreter` class. In Scheme, formulate it as a top-level function with the header

```
;; convert-to-cps: AST -> AST
(define (convert-to-cps an-ast) ...)
```

These interfaces are important because we will use them to test your code.

You must also provide new method/function names for performing interpretation of the CPS'ed program. In Java, you must add a method

```
public JamVal cpsEval()
```

to the `Interpreter` class that converts the associated program to CPS and then interprets the transformed program.

In Scheme, you must provide top-level functions

```
;; make-cps-eval-file: String -> ( -> Value)
(define (make-cps-eval-file filename) ...)

;; make-cps-eval-string: String -> ( -> Value)
(define (make-cps-eval-string filename) ...)
```

You can test that your implementation of the CPS transformation preserves the meaning of programs in Java by comparing the results produced by

`eval()` and `cpsEval()`. In Scheme, you can simply compare the behavior of the thunks produced by `make-eval-file/make-eval-string` and `make-cps-eval-file/make-cps-eval-string`.

To state the CPS transformation rules, we need to introduce a few technical definitions. Study them until your thoroughly understand them.

A Jam application $E(E_1, ..., E_n)$ is *primitive* if E is a primitive Jam function. Recall that we are interpreting operator applications as syntactic sugar for applications of corresponding primitive operations. So an application is primitive iff the rator of the application is either a primitive function or an operator. For example, the applications `first(append(x,y))` and `square(x) * y` are both primitive while the applications `square(4)` and `append(x,y)` are not.

A Jam expression E is *simple* iff all applications except those nested inside `map` constructions are *primitive*, *i.e.*, have a primitive function or operator as the rator. For example,

```
let x := first(a) * b * first(c);
    Y := map f to let g := map z to f(z(z)) in g(g);
in cons(x, cons(Y, null))
```

and

```
x+(y*z)
```

are both simple. In contrast,

```
f(1)
```

and

```
let Y := map f to let g := map z to f(z(z)) in map x to (g(g))(x);
in Y(map fact to map n to if n=0 then 1 else n*fact(n-1))
```

are not simple because `f` is not primitive and `Y)` is not primitive.

The following rules define two syntactic tranformers (functions) on Jam program text: the binary transformer $Cps : Jam \times Jam \rightarrow Jam$ and the unary transformer $Rsh : Simp \rightarrow Simp$, where *Jam* is the set of Jam expressions and *Simp* is the set of simple Jam expressions (*Rsh* stands for "reshape"). The binary transformer $Cps[k, M]$ takes a Jam expression k denoting a unary function, and an unshadowed Jam expression M as input and produces a non-recursive Jam expression with the same meaning as $k(M)$.

The unary transformer *Rsh* is a "help" function for *Cps* that take an unshadowed simple expression as input and adds a continuation parameter to

the `map` expressions and function constants embedded in simple expressions. *Rsh* also adjusts applications of the `arity` primitive function to ignore the added continuation argument.

In the following rules, $S, S_1, S_2, ...$ denote simple Jam expressions; $k, A, B, C, E, E_1, E_2, ..., T$ denote arbitrary Jam expressions; $x_1, x_2, ...$ denote ordinary Jam identifiers, and $v, v_1, v_2, ...$ denote fresh Jam identifiers that do not appear in any other program text. The variable names `x`, `y`, and `k` denote themselves.

Definition of *Cps*.

The following clauses define the textual tranformation $Cps[k,\ M]$:

- If M is a simple Jam expression S:

$$Cps[k,\ S]\ \Rightarrow\ k(Rsh[S])$$

- If M is an application $(\texttt{map}\ x_1, ..., x_n\ \texttt{to}\ B)(E_1, ..., E_n)$:

$$Cps[k,\ (\texttt{map}\ x_1, ..., x_n\ \texttt{to}\ B)(E_1, ..., E_n)]\ \Rightarrow\ Cps[k,\ \texttt{let}\ x_1 := E_1;\ ...;\ x_n := E_n;\ \texttt{in}\ B]$$

- If M is an application $S(S_1, ..., S_n)$:

$$Cps[k,\ S(S_1, ..., S_n)]\ \Rightarrow\ Rsh[S](Rsh[S_1], ..., Rsh[S_n], k)$$

- If M is an application $S(E_1, ..., E_n)$:

$$Cps[k,\ S(E_1, ..., E_n)]\ \Rightarrow\ Cps[k,\ \texttt{let}\ v_1 := E_1;\ ...\ v_n := E_n;\ \texttt{in}\ S(v_1, ..., v_n)]$$

- If M is an application $B(E1, ..., En)$ where B is *not* simple:

$$Cps[k,\ B(E1, ..., En)]\ \Rightarrow\ Cps[k,\ \texttt{let}\ v := B;\ v_1 := E_1;\ ...\ v_n := E_n;\ \texttt{in}\ v(v_1, ..., v_n)]$$

- If M is a conditional construction `if` S `then` A `else` C:

$$Cps[k,\ \texttt{if}\ S\ \texttt{then}\ A\ \texttt{else}\ C]\ \Rightarrow\ \texttt{if}\ Rsh[S]\ \texttt{then}\ Cps[k,\ A]\ \texttt{else}\ Cps[k,\ C]$$

- If M is a conditional construction `if` T `then` A `else` C:

$$Cps[k,\ \texttt{if}\ T\ \texttt{then}\ A\ \texttt{else}\ C]\ \Rightarrow\ Cps[k,\ \texttt{let}\ v := T\ \texttt{in if}\ v\ \texttt{then}\ A\ \texttt{else}\ C]$$

- If M is a block $\{E_1;\ E_2; ...;\ E_n\}$:

$$Cps[k,\ \{E_1;\ E_2; ...;\ E_n\}]\ \Rightarrow\ Cps[k,\ (\texttt{let}\ v_1 := E_1; ...; v_n := E_n;\ \texttt{in}\ v_n]$$

- If M is `let` $x_1 := S_1;$ `in` B:

$$Cps[k,\ \texttt{let}\ x_1 := S_1;\ \texttt{in}\ B]\ \Rightarrow\ \texttt{let}\ x_1 := Rsh[S_1];\ \texttt{in}\ Cps[k,\ B]$$

- If M is $\texttt{let}\ x_1 := S_1;\ x_2 := E_2;\ \ldots\ x_n := E_n;\ \texttt{in}\ B$:

$$Cps[k,\ \texttt{let}\ x_1 := S_1;\ x_2 := E_2;\ \ldots\ x_n := E_n;\ \texttt{in}\ B]\ \Rightarrow$$
$$\texttt{let}\ x_1 := Rsh[S_1];\ \texttt{in}\ Cps[k,\ \texttt{let}\ x_2 := E_2;\ \ldots;\ x_n := E_n;\ \texttt{in}\ B]$$

- If M is $\texttt{let}\ x_1 := E_1;\ \ldots\ x_n := E_n;\ \texttt{in}\ B$:

$$Cps[k,\ \texttt{let}\ x_1 := E_1;\ \ldots\ x_n := E_n;\ \texttt{in}\ B]\ \Rightarrow$$
$$Cps[\texttt{map}\ v\ \texttt{to}\ Cps[k,\ \texttt{let}\ x_1 := v;\ \ldots\ x_n := E_n;\ \texttt{in}\ B],\ E_1]$$

- If M is $\texttt{letrec}\ p_1 := \texttt{map}\ \ldots\ \texttt{to}\ E_1;\ \ldots;\ p_n := \texttt{map}\ \ldots\ \texttt{to}\ E_n;\ \texttt{in}\ B$:

$$Cps[k, \texttt{letrec}\ p_1 := \texttt{map}\ \ldots\ \texttt{to}\ E_1;\ \ldots;\ p_n := \texttt{map}\ \ldots\ \texttt{to}\ E_n;\ \texttt{in}\ B]\ \Rightarrow$$
$$\texttt{letrec}\ p_1 := Rsh[\texttt{map}\ldots\ \texttt{to}\ E_1];\ \ldots;\ p_n := Rsh[\texttt{map}\ldots\ \texttt{to}\ E_n];\ \texttt{in}\ Cps[k, B]$$

Note: in any instantiation of the preceding rules where a `let` expression has no bindings, the `let` expression should be collapsed to its body, *i.e.*

$$\texttt{let in}\ B\ \Rightarrow\ B$$

Definition of *Rsh***.**

The helper transformer $Rsh[S]$ is defined by the following rules:

- If S is a ground constant C (value that is not a `map`):

$$Rsh[C] \Rightarrow C$$

- If S is a variable x:

$$Rsh[x] \Rightarrow x$$

- If S is a primitive application $\texttt{arity}(S1)$:

$$Rsh[\texttt{arity}(S1)] \Rightarrow \texttt{arity}(Rsh[S1]) - \texttt{1}$$

- If S is a primitive application $f(S1, ..., Sn)$ where f is not `arity`:

$$Rsh[f(S1, ..., Sn)] \Rightarrow f(Rsh[S1], ..., Rsh[Sn])$$

- If S is $\texttt{map}\ x_1, ..., x_n\ \texttt{to}\ E$:

$$Rsh[\texttt{map}\ x_1, ..., x_n\ \texttt{to}\ E]\ \Rightarrow\ \texttt{map}\ x_1, ..., x_n, v\ \texttt{to}\, Cps[v,\ E]$$

- If S is the primitive function `arity`:

$$Rsh[\texttt{arity}]\ \Rightarrow\ \texttt{map x,k to k(arity(x) - 1)}$$

- If S is a unary primitive function f other than `arity`:
$$Rsh[f] \Rightarrow \texttt{map x,k to k(f(x))}$$

- If S is a binary primitive function g:
$$Rsh[g] \Rightarrow \texttt{map x,y,k to k(g(x,y))}$$

- If S is a conditional construct `if` S_1 `then` S_2 `else` S_3:
$Rsh[\texttt{if } S_1 \texttt{ then } S_2 \texttt{ else } S_3] \Rightarrow \texttt{if } Rsh[S_1] \texttt{ then } Rsh[S_2] \texttt{ else } Rsh[S_3]$

- If S is `let` $x_1 := S_1;\ ...;\ x_n := S_n;$ `in` S:
$$Rsh[\texttt{let } x_1 := S_1;\ ...;\ x_n := S_n;\ \texttt{in } S] \Rightarrow$$
$$\texttt{let } x_1 := Rsh[S_1];\ ...;\ x_n := Rsh[S_n];\ \texttt{in } Rsh[S]$$

- If S is `letrec` $p_1 :=$ `map` ... `to` $E_1;\ ...;\ p_n :=$ `map` ... `to` $E_n;$ `in` S:
$Rsh[\texttt{letrec } p_1 := \texttt{map } ... \texttt{ to } E_1;\ ...;\ p_n := \texttt{map } ... \texttt{ to } E_n;\ \texttt{in } S] \Rightarrow$
$\texttt{letrec } p_1 := Rsh[\texttt{map}... \texttt{ to} E_1];\ ...;\ p_n := Rsh[\texttt{map}... \texttt{ to} E_n]; \texttt{in } Rsh[S]$

- If S is a block $\{S_1;\ ...;\ S_n\}$:
$$Rsh[\{S_1;\ ...;\ S_n\}] \Rightarrow \{Rsh[S_1];\ ...;\ Rsh[S_{n-1}];\ Rsh[S_n]\}$$

For the purposes of testing your programs we require the following standardization. The top-level continuation must have the syntactic form

```
map x to x
```

using the variable name `x`. In some transformations, you must generate a new variable name. For this purpose, use variable names of the form `:`k where k is an integer. These name cannot be confused with the names of variables that already exist in the program. The sequence of variable names generated by your CPS tranformer must be `:0`, `: 1`, `:2`, ... so that your CPS tranformer has exactly the same behavior as our solution. Note that you must transform a program by making the leftmost possible reduction given that match variables S and E can only match raw program text (any embedded calls on *Cps* and *Rsh* must have already been reduced).

Phase IV

As the fourth part of the assignment, you will write another processor for Jam abstract syntax that transforms conventional Jam abstract syntax into static distance format.

In Java, add a method

```
public AST convertToSD()
```

in the `Interpreter` class that performs this transformation on the Jam AST associated with `this`.

In Scheme, define a top-level function

```
;; convert-to-sd: AST -> AST
(define (convert-to-sd an-ast) ...)
```

You must also write a new interpreter for static distance format (sharing as much existing code as possible) that represents environments as lists of activation records where activation records are represented as arrays in Java and vectors in Scheme.

In Java, you must add methods

```
public JamVal sdEval();
public JamVal sdCpsEval();
```

to the `Interpreter` class. The method `sdEval()` converts the program associated with `this` to static distance format and then interprets it. The method `sdCpsEval()` converts the program first to CPS and then to static distance format and interprets the result.

In Scheme, you must provide four top-level functions

```
;; make-sd-eval-file: String -> ( -> Value)
(define (make-sd-eval-file filename) ...)

;; make-sd-eval-string: String -> ( -> Value)
(define (make-sd-eval-string filename) ...)

;; make-sd-cps-eval-file: String -> ( -> Value)
(define (make-sd-cps-eval-file filename) ...)

;; make-sd-cps-eval-string: String -> ( -> Value)
(define (make-sd-cps-eval-string filename) ...)
```

where the first two functions convert a program to static distance format before evaluating it and the second two functions convert a program first to CPS and then to static distance format before interpreting it.

Hint: In Java, it is tempting to try to refactor the `AST` composite hierarchy into a generic composite hierarchy parameterized by the form of

variables (symbols or static distance coordinates) and environments (lists of binding pairs or lists of activation records) so that the distinctions between the two program representations are reflected in the typing of program text. While such a refactoring is possible, it is massive since every AST class (even including the constant classes) must be modified and every visitor class that processes ASTs must be modified. It is much easier (albeit less precise in the typing of program expressions) to simply add some new subclasses to the AST hierarchy that are static distance variants of the existing AST classes. Of course, this code requires some type casts. Note that static distance code in this representation is also conventional code since the static distance information is added to the conventional representation (which still has symbolic variable names).

Extra Credit (20 points): Supporting `letcc`

Extend the Jam source language to include the new construct

```
<exp> ::= ... | letcc x in M
```

The new construct `letcc x in M` binds the identifier `x` to the current continuation, and evaluates `M` in the extended environment. A continuation is a closure of one argument, reshaped to take an auxiliary continuation argument (like all other closures after CPS) which it discards. Since continuations are ordinary values, they can be stored in data structures, passed as argments, and returned as values.

The `letcc` construct is only supported in the interpreters that perform CPS conversion on the code. The conventional interpreters abort execution with an error if they encounter a use of `letcc`.

To perform CPS conversion on program containing `letcc`, we extend our rules for CPS conversion as follows.

First, a Jam expression E is *simple* iff all occurrences of the `letcc` construct and non-primitive applications appear nested within¡ `map` constructions.

Second, we add the following clause to the definition of the *Cps* syntax transformer:

- If M is `letcc` x `in` B:

$$Cps[k, \texttt{letcc}\ x\ \texttt{in}\ B] \Rightarrow \texttt{let}\ x := \texttt{map}\ v, k_1\ \texttt{to}\ k(v)\ in\ Cps[k, B]$$