HW 4 – CSP Transformation; Working with Mathematical Specifications

CS 421 – Fall 2013
Revision 1.0

Assigned Wednesday, September 18, 2013
Due Wednesday, September 25, 2013, 19:59pm
Extension 48 hours (20% penalty)

1 Change Log

1.0 Initial Release.

2 Objectives and Background

The purpose of this HW is to help your understanding of:

- The basic CSP transformation algorithm
- How to use a formal mathematically recursive definition

3 Turn-In Procedure

Using your favorite tool(s), you should put your solution in a file named `hw4.pdf` (the same name as this file has on the website). If you have problems generating a pdf, please seek help from the course staff. Your answers to the following questions are to be submitted electronically via the `handin` script as though an MP. This assignment is named `hw4`.

4 Background and Definitions

Throughout this HW, we will be working with a (very) simple functional language. It is a fragment of MicorML (a frgament of SML, an OCaml-like language), and the seed of the language that we will be using in MPs throughout the rest of this semester. Using a mix of MicroML concrete syntax (expression constructs as you would type them in MicroML's top-level loop) and recursive mathematical functions, we will describe below the algorithm for CSP transformation for this fragment. You should compare this formal definition with the description given on examples in class.

The language fragment we will work with in this assignment is given by the following Context Free Grammar:

$$
\begin{array}{rl}
e \rightarrow & c \mid v \mid e \oplus e \\
& \mid \texttt{if}\ e\ \texttt{then}\ e\ \texttt{else}\ e \\
& \mid \texttt{fn}\ v\ \texttt{=>}\ ; \mid e\ e
\end{array}
$$

The symbol e ranges recursively over all expressions in MicroML, c ranges over constants, v ranges over program variables, and $\oplus$ ranges over infixed binary primitive operations. The MicroML construct `fn` v`=>` e corresponds the OCaml's construct `fun` v`->` e. This language will expand over the course of the semester.

Mathematically we represent CPS transformation by the functions $[[e]]_\kappa$, which calculates the CPS form of an expression e when passed the continuation κ. The symbol κ does not represent a programming language variable, but rather a complex expression describing the current continuation for e.

The defining equations of this function are given below. Recall that when transforming a function into CPS, it is necessary to expand the arguments to the function to include one that is for passing the continuation to it. We will use κ to represent a continuation that has already been calculated or given to us, and k, k_i, k' etc as the name of variables that can be assigned continuations. We will use v, v_i, v' etc for the ordinary variables in our program.

By v being fresh for an expression e, we mean that v needs to be some variable that is NOT in e. In MP5, you will implement a function for selecting one, but here you are free to choose a name as you please, subject to being different from all the other names that have already been used.

- The CPS transformation of a variable or constant expression just applies the continuation to the variable or constant, since during execution, when this point in the code is reached, both variables and constants are already fully evaluated (except for being looked up).

$$[[v]]_\kappa = \kappa\ v$$
$$[[c]]_\kappa = \kappa\ c$$

  Example: $[[\texttt{x}]]_{\texttt{(FUN y -> report y)}} = \texttt{(FUN y -> report y) x}$ This may be read as "load register `y` with the value for `x`, then do a procedure call to `report`".

- The CPS transformation for a binary operator mirrors its evaluation order. It first evaluates its first argument then its second before evaluating the binary operator applied to those two values. We create a new continuation that takes the result of the first argument, e_1, binds it to v_1 then evaluates the second argument, e_2, and binds that result to v_2. As a last step it applies the current continuation to the result of $v_1 \oplus v_2$. This is formalized in the following rule:

$$[[e_1 \oplus e_2]]_\kappa = [[e_1]]_{\texttt{FUN } v_1 \texttt{ -> } [[e_2]]_{\texttt{FUN } v_2 \texttt{ -> } \kappa\ (v_1 \oplus v_2)}} \quad \text{Where } \begin{array}{l} v_1 \text{ is fresh for } e_1, e_2, \text{ and } \kappa \\ v_2 \text{ is fresh for } e_1, e_2, \kappa, \text{ and } v_1 \end{array}$$

  Example:
$$\begin{aligned} & [[\texttt{x + 1}]]_{\texttt{(FUN w -> report w)}} \\ &= [[\texttt{x}]]_{\texttt{(FUN y -> }[[\texttt{1}]]_{\texttt{(FUN z -> (FUN w -> report w) (y + z))}}\texttt{)}} \\ &= [[\texttt{x}]]_{\texttt{(FUN y -> ((FUN z -> (FUN w -> report w) (y + z)) 1))}} \\ &= \texttt{(FUN y -> ((FUN z -> (FUN w -> report w) (y + z)) 1)) x} \end{aligned}$$

- Each CPS transformation should make explicit the order of evaluation of each subexpression. For if-then-else expressions, the first thing to be done is to evaluate the boolean guard. The resulting boolean value needs to be passed to an if-then-else that will choose a branch. When the boolean value is true, we need to evaluate the transformed then-branch, which will pass its value to the final continuation for the if-then-else expression. Similarly, when the boolean value is false we need to evaluate the transformed else-branch, which will pass its value to the final continuation for the if-then-else expression. To accomplish this, we recursively CPS-transform the boolean guard e_1 with the continuation with a formal parameter v that is fresh for the then branch e_2, the else branch e_3 and the final continuation κ, where, based on the value of v, the continuation chooses either the CPS-transform of e_2 with the original continuation κ, or the CPS-transform of e_3, again with the original continuation κ.

$$[[\texttt{if } e_1 \texttt{ then } e_2 \texttt{ else } e_3]]_\kappa = [[e_1]]_{\texttt{FUN } v \texttt{ -> IF } v \texttt{ THEN } [[e_2]]_\kappa \texttt{ ELSE } [[e_3]]_\kappa}$$
Where v is fresh for e_2, e_3, and κ

With `FUN` v `->` `IF` v `THEN` $[[e_2]]_\kappa$ `ELSE` $[[e_3]]_\kappa$ we are creating a new continutation from our old. This is not a function at the level of expressions, but rather at the level of continuations, hence the use of a different, albeit related, syntax.

Example:

$[[$`if x > 0 then 2 else 3`$]]$`(FUN w -> report w)`
$=[[$`x > 0`$]]$`(FUN y -> IF y THEN` $[[2]]$`(FUN w -> report w) ELSE` $[[3]]$`(FUN w -> report w))`
`= (FUN y -> IF y THEN` $[[2]]$`(FUN w -> report w) ELSE` $[[3]]$`(FUN w -> report w)) (x > 0)`
`= (FUN y -> IF y THEN ((FUN w -> report w) 2) ELSE ((FUN w -> report w) 3)) (x > 0)`

- A function expression by itself does not get evaluated (well, it gets turned into a closure), so it needs to be handed to the continuation directly, except that, when it eventually gets applied, it will need to additionally take a continuation as another argument, and its body will need to have been transformed with respect to this additional argument. Therefore, we need to choose a new continuation variable k to be the formal parameter for passing a continuation into the function. Then, we need to transform the body with k as its continuation, and put it inside a continuation function with the same original formal parameter together with k. The original continuation κ is then applied to the result.

$$[[\texttt{fn } x \texttt{ => } e]]_\kappa = \kappa\,(\texttt{FN } x\ k \texttt{ -> } [[e]]_k) \qquad \text{Where } k \text{ is new (fresh for } \kappa)$$

Notice that we are not yet evaluating anything, so $(\texttt{FN } x\ k \texttt{ -> } [[e]]_k)$ is a CPS function expression, not actually a closure.

Example:

$[[$`fn x => x + 1`$]]$`(FUN w -> report w)`
`= (FUN w -> report w) (FN x k -> (FUN y -> ((FUN z -> k (y + z)) 1)) x)`

- The CPS transformation for application mirrors its evaluation order. In MicroML, we will uniformly use left-to-right evaluation. Therefore, to evaluate an application, first evaluate the function, e_1, to a closure, then evaluate e_2 to a value to which that closure is applied. We create a new continuation that takes the result of e_1 and binds it to v_1, then evaluates e_2 and binds it to v_2. Finally, v_1 is applied to v_2 and, since the CPS transformation makes all functions take a continuation, it is also applied to the current continuation κ. This rule is formalized by:

$$[[e_1\ e_2]]_\kappa = [[e_1]]_{(\texttt{FUN } v_1 \texttt{ -> } [[e_2]]_{(\texttt{FUN } v_2 \texttt{ -> } v_1\ v_2\ \kappa)})} \qquad \text{Where } \begin{array}{l} v_1 \text{ is fresh for } e_2 \text{ and } \kappa \\ v_2 \text{ is fresh for } v_1 \text{ and } \kappa \end{array}$$

Example: $[[$`(fn x => x + 1) 2`$]]$`(FUN w -> report w)`
$=[[$`(fn x => x + 1)`$]]$`(FUN y -> [[2]](FUN z -> y z (FUN w -> report w)))`
`= (FUN y -> [[2]](FUN z -> y z (FUN w -> report w)))`
`  (FN x k -> (FUN a -> ((FUN b -> k (a + b)) 1)) x)`
`= (FUN y -> ((FUN z -> y z (FUN w -> report w)) 2))`
`  (FN x k -> (FUN a -> ((FUN b -> k (a + b)) 1)) x)`

5 Problem

1. **(35 pts)** Compute the following CPS transformation. All parts should be transformed completely.

$[[$`fn f => fn x => if x > 0 then f x else f ((-1) * x)`$]]$`(FUN w -> report w)`