Continuation Passing Style, Transformation, letrec

CS 421

Revision 1.0

1 Change Log

1.0 Initial Release.

2 Objectives

Your objectives are:

- Constructing data structures from algebraic data types
- Deconstructing data structures built from algebraic data types
- Implementing continuation passing style transformations

1. In the previous assignment you converted some expressions to use Continuation-Passing Style (CPS). In this section you will build a function `cps_exp : exp -> cps_cont -> exp_cps` to automatically transform expressions in our language into CPS.

   Mathematically, we represent CPS transformation by the function $[[e]]_\kappa$, which calculates the CPS form of an expression e when passed the continuation κ, where 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. In these rules f, x, v and v_i represent variables in our programming language, k is a continuation variable, c is a constant, e and e_i are expressions and t is a transformed expression. The variables f and x will represent variables that were already present in the expression to be transformed. The variables v and v_i are used to represent newly introduced variables used to pass a value from the previous computation forward into the current continuation. The variable k is used to represent a variable (such as a formal parameter to a function) to be instantiated by an as yet unknown continuation.

   By v being fresh for an expression e, we mean that v needs to be some variable that is NOT free in e. In `common.ml`, we have supplied a function `freshFor : string list -> string` that, when given a list of names, will generate a name that is not in the list. When implementing `cps_exp`, the names you use for these "fresh" variables do not have be the same as the ones we use, but they do have to satisfy the required freshness constraint.

   a. In PicoML, the only expressions that can be declared with `let rec` are functions. A (`let rec` f x `=` e_1 `in` e_2) expression creates a recursive function binding for f with formal parameter x and body e_1. The binding for f is then available for the evaluation of e_2. When e_1 is evaluated in the context of a function call in e_2, the environment for e_1 will need to be updated with this binding. Since we require `let rec` expressions to bind identifiers to functions, we do the CPS transform for this local declaration in a fairly similar way. We transform the body with respect to the continuation variable and paramaterize by that continuation variable. We need to convert the CPS transformed expression waiting for the binding into a continuation taking a value for f. The main difference at the end is that we wrap it all up with a constructor representing a fixed-point operator. Implement the following rule.

$$[[\texttt{let rec } f \ x \texttt{ = } e_1 \texttt{ in } e_2]]_\kappa = \texttt{(FN } f \texttt{ -> } [[e_2]]_\kappa\texttt{) (}\mu\, f.\, \texttt{FUN } x \ k \texttt{ -> } [[e_1]]_k\texttt{)}$$

```
# string_of_exp_cps (cps_exp (LetRecInExp ("f", "x",VarExp "x",
                                            ConstExp (IntConst 4)))
                             (ContVarCPS Kvar));;
- : string = "(FN f -> _k 4)(FIX f. FUN x _k -> _k x)"
```

In the text already present in `letrec.ml` and in the editor provided we have given all cases except the one you are to implement as calls to auxiliary functions in the precompiled module `Plsolution`. The recursive function will be complete once you add the clause for the one case requested in this problem.