MP 5 – Working with ADTs

CS 421 – Spring 2014

Revision 1.0

Assigned Feb 20, 2014
Due Mar 2, 2014 11:59pm

1 Change Log

1.0 Initial Release.

2 Caution

This assignment can appear quite complicated at first. It is essential that you understand how all the code you will write will eventually work together. Please read through all of the instructions and the given code thoroughly before you start, so you have some idea of the big picture.

3 Objectives

Your objectives are:

- Constructing algebraic data types
- Deconstructing algebraic data types
- Continuation passing style transformations

4 Background

Throughout this MP we will be working with a (very) simple functional language. It is the seed of the language with which we will be working on MPs throughout the rest of this semester. In this MP, instead of writing our programs in text files and parsing them, we will represent the structure of our programs via terms made from a set of five algebraic data types.

5 Given Code

This semester the language for which we shall build an interpreter, which we call MicroML, is mainly a simplification of Standard ML (also known as SML). In this assignment we shall build a translator from abstract syntax for MicroML expressions to abstract syntax for a variant that enforces Continuation Passing Style. The file `mp5common.cmo` contains compiled code to support your construction of this translator. Its contents are described here.

5.1 OCaml Types for MicroML AST

Expressions in MicroML are quite similar to expressions in OCaml. The Abstract Syntax Trees for MicroML expressions are given by the following OCaml type:

```
type dec =  (* This type will be expanded in later MPs *)
     Val of string * exp             (* val x = exp *)
   | Rec of string * string * exp    (* val rec f x = exp *)
   | Seq of dec * dec                (* dec1 dec2 *)
and exp =  (* Exceptions will be added in later MPs *)
   | VarExp of string                    (* variables *)
   | ConstExp of const                   (* constants *)
   | MonOpAppExp of mon_op * exp         (* % exp1
                  where % is a builtin monadic operator *)
   | BinOpAppExp of bin_op * exp * exp   (* exp1 % exp2
                  where % is a builtin binary operator *)
   | IfExp of exp * exp * exp            (* if exp1 then exp2 else exp3 *)
   | AppExp of exp * exp                 (* exp1 exp2 *)
   | FnExp of string * exp               (* fn x => exp *)
   | LetExp of dec * exp                 (* let dec in exp end *)
```

This type makes use of the auxiliary types:

```
type const =
     BoolConst of bool        (* for true and false *)
   | IntConst of int          (* 0,1,2, ... *)
   | RealConst of float       (* 2.1, 3.0, 5.975, ... *)
   | StringConst of string    (* "a", "hi there", ... *)
   | NilConst                 (* [ ] *)
   | UnitConst                (* ( ) *)
```

for representing the constants. The `mon_op` type represents monomorphic unary operators in SML.

```
type mon_op =
     IntNegOp      (* integer negation *)
   | HdOp          (* hd *)
   | TlOp          (* tl *)
   | FstOp         (* fst *)
   | SndOp         (* snd *)
```

The primitive binary operators are given by the Ocaml data type `bin_op`.

```
type bin_op =
     IntPlusOp        (* _ + _ *)
   | IntMinusOp       (* _ - _ *)
   | IntTimesOp       (* _ * _ *)
   | IntDivOp         (* _ / _ *)
   | RealPlusOp       (* _ +. _ *)
   | RealMinusOp      (* _ -. _ *)
   | RealTimesOp      (* _ *. _ *)
   | RealDivOp        (* _ /. _ *)
   | ConcatOp         (* _ ^ _ *)
   | ConsOp           (* _ :: _ *)
   | CommaOp          (* _ , _ *)
   | EqOp             (* _ = _ *)
   | GreaterOp        (* _ > _ *)
```

Any of these types may be expanded in future MPs in order to enrich the language.

Most of the constructors of `exp` should be self-explanatory. Names of constants are represented by the type `const`. Names of variables are represented by strings. The constructors that take `string` arguments (`VarExp`,

FnExp, LetExp, and RecExp) use the strings to represent names of variables that they bind. BinOpAppExp takes the binary operator, represented by the type bin_op, together with two operands. Similarly, MonOpAppExp takes the unary operator of the mon_op type and an operand. IfExp is for if_then_else_ expressons, FnExp is for function expressions, and AppExp is for the application of one expression to another. LetExp is for introducing local bindings in expressions. MicroML differs from OCaml in not overloading the keyword let for global declarations and local bindings within an expression. MicroML only uses let_in_end for the latter.

Inside the first part of a let_in_end expression is a declaration (of type dec), and any declaration allowed at top level is allowed here, too. We have simplified the type dec a bit from the full type you will evetually use to reduce the complexity of this assignment. For this assignment, there are three types of declarations:

- val x = exp: binds x to the value of the expression exp,
- val rec x = exp: recursively binds x to the value of the expression exp and
- dec_1 dec_2 allows for sequences of declarations.

Later, we will add local declarations, and wildcards.

There are companion functions string_of_exp, string_of_dec, print_exp and print_dec for viewing expressions and declarations in a more readable form.

A function eval:exp -> string that will execute your code, generating a string that is what the top-level loop would print as a value if you were to execute the corresponding code in OCaml. To use eval in OCaml, execute

```
#load "mp5common.cmo"
#load "mp5eval.cmo";;
```

and then import the needed modules:

```
open Mp5common
open Mp5eval.
```

5.2 OCaml Types for CPS transformation type

In addition to having abstract syntax trees for the expressions of MicroML, we need to have abstract syntax trees for the type of continuations and expressions in CPS.

```
type cps_cont =
   External
 | ContVarCPS of int                                    (* _ki *)
 | ContCPS of string * exp_cps                          (* FUN x -> exp_cps *)

and exp_cps =
   VarCPS of cps_cont * string                          (* k x *)
 | ConstCPS of cps_cont * const                         (* k c *)
 | MonOpAppCPS of cps_cont * mon_op * string            (* k (% x) *)
 | BinOpAppCPS of cps_cont * bin_op * string * string   (* k (x % y) *)
 | IfCPS of string * exp_cps * exp_cps             (* IF x THEN exp_cps1 ELSE exp_cps2 *)
 | AppCPS of cps_cont * string * string
 | FnCPS of cps_cont * string * int * exp_cps          (* k (FN x _ki -> [[exp]]_ki) *)
 | FixCPS of cps_cont * string * string * int * exp_cps
                                      (* (FUN f -> k) (FIX f. FN x _ki -> [[exp]]_ki) *)
```

Each contructor (except FixCPS) in the type exp_cps corresponds to one in the direct style represention type exp. You will note that all decalarations are now gone. Also you will note that every construct except IfCPS takes an additional argument of cps_cont for the continuation to recieve the result of the expression immediately being constructed. With IfCPS, this continuation is missing because it is burried one step down in each of the branches. It is also worth noting that, with the exception of IfCPS and FnCPS, where we had exp arguments before, now we have only the names of variables. This reflects that fact that in CPS, before you express an operation to be preformed,

you must first compute, and store the values of all the components of that operation (with the exception of `IfCPS`, which must wait until it tests its boolean guard before touching either of its branches, and `FnCPS`, which must protect its function body from being executed until it is applied). The augmentation of the constructors with a place for a continuation, and the replacment of general expression arguments by variable arguments are the changes necessary to guarantee that terms built in this type represent expressions in CPS.

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 represent this variable by an integer rather than a string. It really is a different type of variable because it is always internally generated and it is to supply a continuation and not an expression. When transforming an expression, we will taken in and hand back an integer giving the next integer available to be used for a continuation variable.

6 Problems

The problems below have sample executions that suggest how to write answers. Students have to use the same function name, but the name and number of the parameters that follow the function name need not be the same as the ones we give. That is, the students are free to choose different names for the arguments to the functions from the ones given in the example execution. You may use any library functions you wish.

1. (4 pts) Write a function `import_list: (string * int) list -> exp`, that takes an list of pairs and converts it into an expression in our language that is equivalent.

```
# let rec import_list lst = ...;;
val import_list : (string * int) list -> exp = <fun>
# import_list [("a",1);("b",2);("c",3)];;
- : exp =
BinOpAppExp (ConsOp,
 BinOpAppExp (CommaOp, ConstExp (StringConst "a"), ConstExp (IntConst 1)),
 BinOpAppExp (ConsOp,
  BinOpAppExp (CommaOp, ConstExp (StringConst "b"), ConstExp (IntConst 2)),
  BinOpAppExp (ConsOp,
   BinOpAppExp (CommaOp, ConstExp (StringConst "c"), ConstExp (IntConst 3)),
   ConstExp NilConst)))
```

2. (4 pts) Write a term in our language (an abstract syntax tree) that represents the funciton declaration that implements the following MicroML function `list_all` (which is basically the same function as the one by this name from MP2):

```
val rec list_all p =
     fn xs =>
      if xs = [] then true
      else if p (hd xs)
           then if list_all p (tl xs) then true else false
           else false
```

The syntax `fn` x `=>` e corresponds to the OCaml `fun` x `->` e.

```
# let list_all = ...
val list_all : dec = ...
# string_of_dec list_all;;
- : string =
```

```
"val rec list_all p = fn xs => if xs = [] then true else
if p (hd xs) then if list_all p (tl xs) then true else false else false"
```

You can test out your implementation by evaluating it on various input as follows:

```
# #load "mp5common.cmo";;
# #load "mp5eval.cmo";;
# open Mp5eval;;
# open Mp5common;;
# #use "mp5.ml";;
# eval_exp (LetExp (list_all,(AppExp ( AppExp (VarExp "list_all",FnExp
("x", BinOpAppExp (EqOp,(VarExp "x"), ConstExp (IntConst 1)))),BinOpAppExp
(ConsOp,ConstExp (IntConst 1),ConstExp NilConst))), []);;
- : Mp5eval.value = BoolVal true
# eval_exp (LetExp (list_all,(AppExp ( AppExp (VarExp "list_all",FnExp
("x", BinOpAppExp (EqOp,(VarExp "x"), ConstExp (IntConst 1)))),(BinOpAppExp
(ConsOp,ConstExp (IntConst 1),BinOpAppExp (ConsOp,ConstExp
(IntConst 1),ConstExp NilConst))))), []);;
- : Mp5eval.value = BoolVal true
# eval_exp (LetExp (list_all,(AppExp ( AppExp (VarExp "list_all",FnExp
("x", BinOpAppExp (EqOp,(VarExp "x"), ConstExp (IntConst 1)))),(BinOpAppExp
(ConsOp,ConstExp (IntConst 1),BinOpAppExp (ConsOp,ConstExp
(IntConst 2),ConstExp NilConst))))), []);;
- : Mp5eval.value = BoolVal false
```

3. (5 pts) Write a pair of functions `cal_max_exp_height : exp -> int` and `cal_max_dec_height : dec -> int` that count the maximum height of the input `exp` or `dec` by viewing the `exp` or `dec` as a tree structure. Assume that VarExp and ConstExp have height 1.

```
# let rec cal_max_exp_height exp = ...
val cal_max_exp_height : exp -> int = <fun>
val cal_max_dec_height : dec -> int = <fun>
# cal_max_exp_height (BinOpAppExp (ConsOp,
 BinOpAppExp (CommaOp, ConstExp (StringConst "a"), ConstExp (IntConst 1)),
 BinOpAppExp (ConsOp,
  BinOpAppExp (CommaOp, ConstExp (StringConst "b"), ConstExp (IntConst 2)),
  BinOpAppExp (ConsOp,
   BinOpAppExp (CommaOp, ConstExp (StringConst "c"), ConstExp (IntConst 3)),
   ConstExp NilConst))));;
- : int = 5
```

4. (25 pts total) A free variable in an expression is a variable that isn't bound in that expression. In our setting, free variables are the variables that had to be given a value previously for the expression to be able to be evaluated. As an example, in (`let val x = y in fn s => a x s end`) the variables `a` and `y` are free but `x` and `s` are not. Notice that to undesrstand the free variables in the previous example, we also need to to know for the inner declaration both the free variables of the expression in it, but also the variables that are being bound by it.

   Write a pair of functions `freeVarsInExp : exp -> string list` and `freeAndBindingVarsInDec : dec -> string list * string list` that calculate the names of the free variables of an expression (represent sets via lists) for `freeVarsInExp`, and calculates the pair of lists where the first list gives the set of

names of free variables, and the second gives the set of names of the variables being bound. The grader will cope with answers that have duplicate entries or the result list in a different order than our reference solution.

To assist you in writing this function, we have broken the problem down into groups of similar cases. We also give the precise mathematical definition (in cases) for a function φ calculating the free variables of an expression e, and a function ψ calculating the pair of the free varaibles the binding variables of a declaration.

In `mp5common.ml`, we have supplied you with a related pair of functions `freeVarsInContCPS : cps_cont -> string list` and `freeVarsInExpCPS : exp_cps -> string list` for calculating the free variables in a continuation and CPS-transformed expression respectively. You should feel free to examine these definitions for inspiration in writing the code for this problem.

a. (2 pts.) We can define a function $\varphi(e)$ that calculates the free variables of an expression, where the expression is a variable v, or a constant c by

$$\varphi(v) = \{v\}$$
$$\varphi(c) = \emptyset$$

The function `freeVarsInExp` should behave in a similar manner, returning no names for a constant, and the singleton name of a variable. Write the appropriate clause for `freeVarsInExp` to return the free variables of expressions that are constants or variables.

```
# let rec freeVarsInExp = ... ;;
val freeVarsInExp : exp -> string list = <fun>
val freeAndBindingVarsInDec : dec -> string list * string list = <fun>
# freeVarsInExp (VarExp "x");;
- : string list = ["x"]
```

b. (8 pts.) The set of free variables of an expression that is top-most an if-then-else, the use of a unary or binary operator, or the application of one expression to another is just the union of the free variables of all the immediate subexpressions.

$$\varphi(\texttt{if } e_1 \texttt{ then } e_2 \texttt{ else } e_3) = \varphi(e_1) \cup \varphi(e_2) \cup \varphi(e_3)$$
$$\varphi(\oplus e) = \varphi(e) \qquad \text{For unary operator } \oplus$$
$$\varphi(e_1 \oplus e_2) = \varphi(e_1) \cup \varphi(e_2) \qquad \text{For binary operator } \oplus$$
$$\varphi(e_1\ e_2) = \varphi(e_1) \cup \varphi(e_2)$$

Write the clauses for `freeVarsInExp` for expressions that are top-most an if-then-else, the use of a unary or binary operator, or the application of one expression to another.

```
# freeVarsInExp (IfExp(ConstExp (BoolConst true), VarExp "x", VarExp "y"));;
- : string list = ["x"; "y"]
```

c. (3 pts.) The free variables of a function expression are all the free variables in the body of the expression except the variable that is the formal parameter. Any occurrence of that variable in the body of the function is bound by the formal parameter, and not free.

$$\varphi(\texttt{fn } x \texttt{ => } e) = \varphi(e) - \{x\}$$

Add clauses to `freeVarsInExp` to compute the free variables of a function expression.

```
# freeVarsInExp (FnExp("x", VarExp "x"));;
- : string list = []
```

d. (9 pts.) In a declaration of the form `val` x `=` e, the free variables are the free variables of the expressions e, while the bound variables are just the singleton x. A declaration of the form `val rec` f x `=` e is almost the same, except that we must remove both f and x from the free variables of the expression since the recursion causes the declaration to bind f there, and x is the formal parameter of the function with e as its body. The binding variables of a sequence of two declarations is jus the union of the binding variables of each. However, since the earlier declaration binds variables in the later one, the free variables of the sequence is the union of the free variables of the first together with the reduction of the free variables of the second by the binding variables of the first.

$$\begin{aligned}
\psi(\texttt{val } x = e) &= (\varphi(e), \{x\}) \\
\psi(\texttt{val rec } f\ x = e) &= (\varphi(e) - \{f,\ x\}, \{f\}) \\
\psi(dec_1\ \ dec_2) &= (fv_1 \cup (fv_2 - bv_1), bv_1 \cup bv_2) \\
&\qquad \text{where } \psi(dec_1) = (fv_1, bv_1) \text{ and } \psi(dec_2) = (fv_2, bv_2)
\end{aligned}$$

Add clauses to compute `freeAndBindingVarsInDec`.

```
freeAndBindingVarsInDec
  (Rec ("f", "x", BinOpAppExp(IntPlusOp, VarExp "x",
                              AppExp(VarExp  "f", VarExp "y"))));;
  - : string list * string list = (["y"], ["f"])
```

e. (3 pts.) The free variables of a `let`-expression are restricted by the bindings the dec introduces. In `let` dec `in` e `end` any x bound in dec binds any occurrence of x in e.

$$\varphi(\texttt{let val } dec \texttt{ in } e \texttt{ end}) = fv \cup (\varphi(e) - bv) \qquad \text{where } \psi(dec) = (fv, bv)$$

Add the clause to `freeVarsInExp` to compute the free variables of `let`-expressions.

```
# freeVarsInExp (LetExp(Val("x", VarExp "y"), VarExp "x"));;
- : string list = ["y"]
```

5. (28 pts total) In MP4 you converted some expressions to use Continuation-Passing Style (CPS). In this section you will build a pair of functions `cps_exp : exp -> cps_cont -> int -> exp_cps * int` and `val cps_dec : dec -> exp_cps -> int -> exp_cps * int` to automatically transform expressions in our language into CPS. The `int` argument in each function represents the next continuation variable available for use. Each time you use this integer to create a new continuation variable, you should increment this integer and return it as part of your result.

   Mathematically we represent CPS transformation by the functions $[[e]]_\kappa$, which calculates the CPS form of an expression e when passed the continuation κ, and $\langle\langle d \rangle\rangle_t$, which calculates the CPS form of an expression `let` dec `in` e `end` where $t = [[e]]_\kappa$. 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 or e_i are expression, t is a transformed expression and d a declaration. 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 `mp5common.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` and `cps_dec`, 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. (4 pts) 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).

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

The code for the function `cps_exp` should behave is a similar manner, creating the application of the continuation to the variable or constant. Add code to `cps_exp` to implement the CPS-transformation of an expression that is a constant or variable.

```
# string_of_exp_cps (fst (cps_exp (VarExp "x") (ContVarCPS 0) 1));;
- : string = "_k0 x"
```

b. (3 pts) 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 e_1 with the continuation with a formal parameter v that is fresh for e_2, e_3 and 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.

$$[[\texttt{if}\ e_1\ \texttt{then}\ e_2\ \texttt{else}\ e_3]]_\kappa = [[e_1]]_{\texttt{fun}\ v\ \texttt{->}\ \texttt{if}\ v\ \texttt{then}\ [[e_2]]_\kappa\ \texttt{else}\ [[e_3]]_\kappa}$$

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 funciton at the level of expressions, but rather at the level of continuations, and, as a result, should use the constructor `ContCPS`.

Add a clause to `cps_exp` for the case for if-then-else operators.

```
# string_of_exp_cps (fst (cps_exp (IfExp (VarExp "b", ConstExp (IntConst 2),
                                          ConstExp (IntConst 5)))
                                  (ContVarCPS 0) 1));;
- : string = "(FUN a -> IF a THEN _k0 2 ELSE _k0 5) b"
```

c. (3 pts) 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. Implement this rule.

$$[[e_1\ e_2]]_\kappa = [[e_1]]_{\texttt{fun}\ v_1\ \texttt{->}\ [[e_2]]_{\texttt{fun}\ v_2\ \texttt{->}\ v_1\ v_2\ \kappa}}$$

Where v_1 is fresh for e_2 and κ; v_2 is fresh for v_1 and κ

```
# string_of_exp_cps (fst (cps_exp (AppExp (VarExp "f", VarExp "x"))
                                  (ContVarCPS 0) 1));;
- : string = "(FUN a -> (FUN b -> a b _k0)) x) f"
```

d. (3 pts) 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 $v_1 \oplus v_2$. Implement the following rule.

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

```
# string_of_exp_cps (fst (cps_exp (BinOpAppExp (IntPlusOp, ConstExp(IntConst 5),
                                                ConstExp(IntConst 1)))
                                  (ContVarCPS 3) 4));;
- : string = "(FUN a -> (FUN b -> _k3(a + b)) 1) 5"
```

e. (3 pts) The CPS transformation for a unary operator mirrors its evaluation order. It first evaluates the argument of the operator and then applies the continuation to the result of applying that operator to the value. Thus we create a continuation that takes the result of evaluating the argument, e, and binds it to v then applies the continuation to the result of $\oplus v$. Implement the following rule.

$$[[\oplus e]]\kappa = [[e]]_{\texttt{fun } v \texttt{ -> } \kappa\ (\oplus v)} \qquad \text{Where } v \text{ is fresh for } \kappa$$

```
# string_of_exp_cps (fst (cps_exp (MonOpAppExp (HdOp, ConstExp NilConst))
                                  (ContVarCPS 0) 1));;
- : string = "(FUN a -> _k0(hd a)) []"
```

f. (3 pts) 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.

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

The syntax for a function in MicroML is (`fn _ => _`). Write the clause for the case for functions.

```
# string_of_exp_cps (fst (cps_exp (FnExp ("x", VarExp "x"))
                                  (ContVarCPS 0) 1));;
- : string = "_k0(FN x _k1 -> _k1 x)"
```

g. (3 pts) In the context of CPS transforming an expression, a declaration is always local to some expression for which it is supplying variable bindings. The arguments to `cps_dec : dec -> exp_cps -> int -> exp_cps * int` are the declaration to be transformed, the expresion to which the declaration is supplying bindings and an integer giving the next unused continutation variable. To evaluate a simple, single declaration of the form `val x = e` in the context where we wish to evaluate a CSP transformed expression t, we want to convert t into a continuation taking a value for x, and then transform e with respect to that continuation. More formally, you should implement the following rule:

$$\langle\langle \texttt{val } x \texttt{ = } e \rangle\rangle t = [[e]]_{\texttt{fun } x \texttt{ -> } t}$$

```
# string_of_exp_cps (fst (cps_dec (Val ("x", ConstExp(IntConst 2)))
                                  (VarCPS (ContVarCPS 0, "x")) 1));;
- : string = "(FUN x -> _k0 x) 2"
```

h. (3 pts) To transform a sequence of declarations with respect to a CPS expression, transform the first with respect to the result of transforming the second with respect to the CPS expression. Implement the following rule.

$$\langle\langle dec_1\ dec_2\rangle\rangle_t = \langle\langle dec_1\rangle\rangle_{\langle\langle dec_2\rangle\rangle_t}$$

```
# string_of_exp_cps (fst (cps_dec (Seq (Val ("x", ConstExp(IntConst 2)),
                                        Val ("y", VarExp "x")))
                                  (VarCPS (ContVarCPS 0, "x")) 1));;
- : string = "(FUN x -> (FUN y -> _k0 x) x) 2"
```

i. (3 pts) A (`let` dec `in` e `end`) expression first evaluates the declaration d, generating a collection of local bindings, and then evaluates e in the context of those new bindings. Note that, in order to transform e into CPS, we already have the necessary continuation κ because e computes the value to be given as the final result. To transform the (`let` dec `in` e `end`) expression, we transform e with respect to our current continuation κ, and then give this transformed expression to our procedure for transforming declarations. Implement the following rule.

$$[[\texttt{let}\ dec\ \texttt{in}\ e\ \texttt{end}]]_\kappa = \langle\langle dec\rangle\rangle_{[[e]]_\kappa}$$

```
# string_of_exp_cps (fst (cps_exp (LetExp (Val ("x", ConstExp(IntConst 2)),
                                           VarExp "x"))
                          (ContVarCPS 0) 1));;
- : string = "(FUN x -> _k0 x) 2"
```

6.1 Extra Credit

j. (3 pts) In MicroML, the only expressions that can be declared with `val rec` are functions. A (`val rec` f x `=` e) declaration creates a recursive function binding for f and with formal parameter x and body e. When e is later evaluated in the context of a function call, the environment for e will need to be updated with this binding. Since we require `val rec` declarations to bind identifiers to functions, we do the CPS transform for this declaration in a fairly similar way. We need to make up a new continuation variable and transform the body with respect to that, and paramaterize by that continuation varaible. 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.

$$[[\texttt{val rec}\ f\ x\ \texttt{=}\ e]]_t = (\texttt{fun}\ f\ \texttt{->}\ t)\,(\mu\, f.\,\texttt{fun}\ x\ \texttt{->}\ \texttt{fn}\ k\ \texttt{=>}\ [[e]]_k)$$

Where k is new (fresh for κ).

```
# string_of_exp_cps (fst (cps_exp (LetExp (Rec("f", "x",VarExp "x"),
                                    ConstExp (IntConst 4)))
                                  (ContVarCPS 1) 2));;
- : string = "(FUN f -> _k1 4)(FIX f. FN x _k2 -> _k2 x)"
```