
MP 5 – Working with ADTs

CS 421 – Fall 2014

Revision 1.0

Assigned Sep 23, 2014

Due Sep 30, 2014 23:59

Extension 48 hours (20% penalty)

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 PicoML, is mainly a simplification of OCaml. In this assignment we shall build a translator from abstract syntax for PicoML 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 PicoML AST

Expressions in PicoML are mostly a subset of expressions in OCaml. The Abstract Syntax Trees for PicoML expressions are given by the following OCaml type:

```
type 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 *)
  | FunExp of string * exp          (* fun x -> exp1 *)
  | LetInExp of string * exp * exp  (* let x = exp1 in exp2 *)
  | LetRecInExp of string * string * exp * exp (* let rec f x = exp1 in exp2 *)
```

This type makes use of the auxiliary types:

```
type const =
  | BoolConst of bool              (* for true and false *)
  | IntConst of int                (* 0,1,2, ... *)
  | FloatConst 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 PicoML.

```
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       (* _ / _ *)
  | FloatPlusOp    (* _ +. _ *)
  | FloatMinusOp   (* _ -. _ *)
  | FloatTimesOp   (* _ *. _ *)
  | FloatDivOp     (* _ /. _ *)
  | 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`, `FunExp` and `LetInExp`) 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` expressions, `FunExp` is for function expressions, and `AppExp` is for the application of one expression to another. `LetInExp` is for introducing local bindings in expressions. PicoML differs from OCaml in not overloading the keyword `let` for global declarations and local bindings within an expression. PicoML only uses `let_in_end` for the latter.

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

5.2 OCaml Types for CPS transformation type

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

```

type cps_cont =
  External
  | ContVarCPS of int (* _ki *)
  | FnContCPS of string * exp_cps (* FN 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 (* x y k *)
  | FunCPS of cps_cont * string * int * exp_cps (* k (FUN x _ki -> [[exp]]_ki) *)
  | FixCPS of cps_cont * string * string * int * exp_cps (* k (FIX f. FUN x _ki -> [[exp]]_ki) *)

```

Each constructor (except `FixCPS`) in the type `exp_cps` corresponds to one in the direct style representation type `exp`. You will note that all declarations are now gone. Also you will note that every construct except `IfCPS` takes an additional argument of `cps_cont` for the continuation to receive the result of the expression immediately being constructed. With `IfCPS`, this continuation is missing because it is buried one step down in each of the branches. It is also worth noting that, with the exception of `IfCPS` and `FunCPS`, 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 performed, 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 `FunCPS`, 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 replacement 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 take 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: (int * int) list -> exp`, that takes a list of pairs and converts it into an expression in our language that is equivalent to it.

```
# let rec import_list lst = ...;;
val import_list : (int * int) list -> Mp5common.exp = <fun>
# import_list [(7,1); (4,2); (6,3)];;
- : Mp5common.exp =
BinOpAppExp (ConsOp,
  BinOpAppExp (CommaOp, ConstExp (IntConst 7), ConstExp (IntConst 1)),
  BinOpAppExp (ConsOp,
    BinOpAppExp (CommaOp, ConstExp (IntConst 4), ConstExp (IntConst 2)),
    BinOpAppExp (ConsOp,
      BinOpAppExp (CommaOp, ConstExp (IntConst 6), ConstExp (IntConst 3)),
      ConstExp NilConst)))
```

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

```
let rec pair_sums lst =
  if lst = [] then []
  else let x = (hd lst) in (fst x + snd x) :: (pair_sums (tl lst))
in pair_sum [(7,1); (4,2); (6,3)]

# let pair_sum = ...
val pair_sum : exp =
# string_of_exp pair_sum;;
- : string =
"let rec pair_sum lst = if lst = [] then [] else let x = hd lst
  in ((fst x) + (snd x)) :: (pair_sum (tl lst))
  in pair_sum (((7,1)) :: (((4,2)) :: (((6,3)) :: [])))"
```

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

```
# #load "mp5common.cmo";;
# #load "mp5eval.cmo";;
# open Mp5eval;;
# open Mp5common;;
# #use "mp5.ml";;
# eval_exp (pair_sum, []);
- : Mp5eval.value = ListVal [IntVal 8; IntVal 6; IntVal 9]
```

3. (5 pts) Write a function `cal_max_exp_height : exp -> int` that counts the maximum height of the input `exp` 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 : Mp5common.exp -> int = <fun>
# cal_max_exp_height (BinOpAppExp (ConsOp,
  BinOpAppExp (CommaOp, ConstExp (IntConst 7), ConstExp (IntConst 1)),
  BinOpAppExp (ConsOp,
    BinOpAppExp (CommaOp, ConstExp (IntConst 4), ConstExp (IntConst 2)),
    BinOpAppExp (ConsOp,
      BinOpAppExp (CommaOp, ConstExp (IntConst 6), ConstExp (IntConst 3)),
      ConstExp NilConst)))));;
- : int = 5
```

4. (20 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 fun s -> a x s end)` the variables `a` and `y` are free but `x` and `s` are not. Notice that to understand the free variables in the previous example, we also need 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 function `freeVarsInExp : exp -> string list` that calculates the names of the free variables of an expression (where we represent sets via lists). 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 .

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

$$\begin{aligned}\varphi(v) &= \{v\} \\ \varphi(c) &= \emptyset\end{aligned}$$

The function `freeVarsInExp` should behave in a similar manner, returning no names for a constant, and the singleton name of the variable for 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 : Mp5common.exp -> string list = <fun>
# freeVarsInExp (VarExp "x");;
- : string list = ["x"]
```

- b. (8 pts.) The set of free variables of an expression that is at the top-most level 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.

$$\varphi(\text{if } e_1 \text{ then } e_2 \text{ else } e_3) = \varphi(e_1) \cup \varphi(e_2) \cup \varphi(e_3)$$

$$\varphi(e \oplus) = \varphi(e)$$

For unary operator \oplus

$$\varphi(e_1 \oplus e_2) = \varphi(e_1) \cup \varphi(e_2)$$

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 in 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.

$$\varphi(\text{fun } x \rightarrow e) = \varphi(e) - \{x\}$$

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

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

Note: You can implement set subtraction using the library function `List.filter : ('a -> bool) -> 'a list -> 'a list`.

- d. (3 pts.) The free variables of a `let`-expression are restricted by the variable being locally declared, and hence bound. In `let x = e1 in e2` end the `x` in the `let` part binds any occurrence of `x` in `e2`, but not in `e1`.

$$\varphi(\text{let } x = e_1 \text{ in } e_2) = \varphi(e_1) \cup (\varphi(e_2) - \{x\})$$

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

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

- e. (4 pts) The most complicated case for computing the free variables of an expression is that of a `let rec`-expression. In `let rec`-expressions, there are two bindings taking place, and they have two different scopes. In `let rec f x = e1 in e2`, the `f` binds all the occurrences of `f` in both `e1` and `e2`, but the `x` only binds occurrences of `x` in `e1`; if `x` is a free variable of `e2` it will be a free variable of `let rec f x = e1 in e2`.

$$\varphi(\text{let rec } f \text{ } x = e_1 \text{ in } e_2) = (\varphi(e_1) - \{f, x\}) \cup (\varphi(e_2) - \{f\})$$

Write the clause for `freeVarsInExp` for `let rec`-expressions.

```
# freeVarsInExp (LetRecInExp("f", "x", AppExp(VarExp "f", VarExp "x"),
                                     AppExp(VarExp "f", VarExp "y")));;
- : string list = ["y"]
```

5. (28 pts total) In MP4 you converted some expressions to use Continuation-Passing Style (CPS), and in HW4 you worked with the recursive algorithm for transforming code to CPS by hand. In this section you will build a function

`cps_exp : exp -> cps_cont -> 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 , 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, t is a transformed expression and d a declaration. The variables f and x will represent variables that were already presented 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`, the names you use for these “fresh” variables do not have to 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).

$$\begin{aligned} [[v]]_\kappa &= \kappa \ v \\ [[c]]_\kappa &= \kappa \ c \end{aligned}$$

The code for the function `cps_exp` should behave in 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 .

$$[[\text{if } e_1 \text{ then } e_2 \text{ else } e_3]]_\kappa = [[e_1]]_{\text{FN } v \rightarrow \text{IF } v \text{ THEN } [[e_2]]_\kappa \text{ ELSE } [[e_3]]_\kappa}$$

Where v is fresh for e_2, e_3 , and κ

With `FN $v \rightarrow \text{IF } v \text{ THEN } [[e_2]]_\kappa \text{ ELSE } [[e_3]]_\kappa$` we are creating a new continuation from our old. This is not a function at the level of expressions, but rather at the level of continuations, and, as a result, should use the constructor `FnContCPS`.

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 = "(FN a -> IF a THEN _k0 2 ELSE _k0 5) b"
```

- c. (3 pts) The CPS transformation for application mirrors its evaluation order. In PicoML, we will uniformly use right-to-left evaluation. Therefore, to evaluate an application, first evaluate e_2 to a value then evaluate the function, e_1 , to a closure, and finally, evaluate the application of the closure to the value. To transform the application of e_1 to e_2 , we create a new continuation that takes the result of e_2 and binds it to v_2 , then evaluates e_1 and binds it to v_1 , then finally, applies v_1 to v_2 and, since the CPS transformation makes all functions take a continuation, it is also applied to the current continuation κ . This is the continuation that is used in transforming e_2 . Implement this rule:

$$[[e_1 e_2]]\kappa = [[e_2]]_{\text{FN } v_2 \rightarrow [[e_1]]_{\text{FN } v_1 \rightarrow v_1 v_2 \kappa} \quad \text{Where } v_2 \text{ is fresh for } e_1 \text{ and } \kappa \\ v_1 \text{ is fresh for } v_2 \text{ and } \kappa$$

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

- d. (3 pts) The CPS transformation for a binary operator mirrors its evaluation order. It first evaluates its second argument, then its first before evaluating the binary operator applied to those two values. We create a new continuation that takes the result of the second argument, e_2 , binds it to v_2 then evaluates the first argument, e_1 , and binds that result to v_1 . As a last step it applies the current continuation to the result of $v_1 \oplus v_2$. Implement the following rule.

$$[[e_1 \oplus e_2]]\kappa = [[e_2]]_{\text{FN } v_2 \rightarrow [[e_1]]_{\text{FN } v_1 \rightarrow \kappa (v_1 \oplus v_2)} \quad \text{Where } v_2 \text{ is fresh for } e_1 \text{ and } \kappa \\ v_1 \text{ is fresh for } \kappa, \text{ and } v_2$$

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

- 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.

$$[[\oplus e]]\kappa = [[e]]_{\text{FN } v \rightarrow \kappa (\oplus v)} \quad \text{Where } v \text{ is fresh for } \kappa$$

```
# string_of_exp_cps (fst (cps_exp (MonOpAppExp (HdOp, ConstExp NilConst))
                                   (ContVarCPS 0) 1))) ;
- : string = "(FN 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.

$$[[\text{fun } x \rightarrow e]]_{\kappa} = \kappa (\text{FUN } x \ k \rightarrow [[e]]_k) \quad \text{Where } k \text{ is new (fresh for } \kappa)$$

The syntax for a function in PicoML, as in OCaml is `(fun _ -> _)`. Write the clause for the case for functions.

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

- g. (3 pts) A `(let x = e1 in e2)` expression first evaluates the expression e_1 generating a collection of local bindings, and then evaluates e_2 in the context of those new bindings. Note that, in order to transform e_2 into CPS, we already have the necessary continuation κ because e_2 computes the value to be given as the final result. To transform the `(let x = e1 in e2)` expression, we transform e_2 with respect to our current continuation κ , and then give this transformed expression to our procedure for transforming declarations. Implement the following rule.

$$[[\text{let } x = e_1 \text{ in } e_2]]_{\kappa} = [[e_1]]_{\text{FN } x \rightarrow} [[e_2]]_{\kappa}$$

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

6.1 Extra Credit

- h. (3 pts) In PicoML, the only expressions that can be declared with `let rec` are functions. A `(let rec f x = e1 in e2)` expression creates a recursive function binding for f and 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` 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 parameterize 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.

$$[[\text{let rec } f \ x = e_1 \text{ in } e_2]]_{\kappa} = (\text{FN } f \rightarrow [[e_2]]_{\kappa}) (\mu f. \text{FUN } x \ k \rightarrow [[e_1]]_k) \\ \text{Where } k \text{ is new (fresh for } \kappa).$$

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