MP 11 – A Transition Semantics Evaluator for CPS

CS 421 – Spring 2014
Revision 1.0

Assigned April 24, 2014
Due May 4, 2014 11:59PM

1 Change Log

1.0 Initial Release.

2 Overview

In MP10, you saw how to write an interpreter for MicroML based on Natural Semantics. The general form of evaluation (for nonrecursive entities) was to walk the abstract syntax tree (in a mostly forward-recursive fashion) gathering up values for all the sub-trees as we went to form a cummulative value to return for the given expression. To use transition semantics as the basis of our evalutor, again we generaly need to walk the abstract syntax trees, but this time we are looking only for the single next step of compution to be performs. However, we have actually performed that precise actively once before, in MP5, when we transformed MicroML abstract syntax into CPS. So, rather then replicating that work again in this MP, let's use it. Instead of implementing a transition semantics for MicroML, we will use our translation of MicroML into CPS, and implement a transition semantics for CPS instead. The advantage to doing the transition semantics on CPS is that the CPS is explicitly designed so that "the next sinlge step of computation" is specified immediately at the root of the abstract syntax tree of an expression, and there is no need to structurally recurse through the abstract syntax tree to find the next unit of work.

As before, we will make use of the lexer, parser and type inferencer you wrote in earlier MPs, but we will add in the CPS transformation you wrote in MP5 - revised to handle exceptions, as well as normal computations. In this assignment, you are to write a function that will perform one step of transition for the transition semantics for the CPS expressions. This function will be called repeatedly by a loop until it returns a value. As in MP10, once you supply the function for performing one step of transition semantics, you will be able to build an interactive top-level loop to test out evaluating MicroML programs, `micromlint` and `micromlintSol`

3 Types

The basic language that we a implementing transition semantics for is that of CPS term, extended with continuations for handling exceptions. The type in OCaml for the abstract syntax of this language are given as follows:

```
type cps_cont =
   ContVarCPS of int                          (* _ki *)
 | External
 | ContCPS of string * exp_cps                (* funk x --> exp_cps *)
 | ExnMatch of exn_cont                       (* i1 |-> ec1; ... in |-> ecn *)

and exn_cont =
   ExnContVarCPS of int
```

```
 | EmptyExnContCPS
 | UpdateExnContCSP of (int option * exp_cps) list * exn_cont

and exp_cps =
   VarCPS of cps_cont * string
 | ConstCPS of cps_cont * const
 | MonOpAppCPS of cps_cont * mon_op * string * exn_cont
 | BinOpAppCPS of cps_cont * bin_op * string * string * exn_cont
 | IfCPS of string * exp_cps * exp_cps
 | AppCPS of cps_cont * string * string * exn_cont
 | FnCPS of cps_cont * string * int * int * exp_cps
 | FixCPS of cps_cont * string * string * int * int * exp_cps
```

This is an extension of the type `exp_cps` and its auxiliary types we used in MP5. It has been extended by adding a type `exn_cont` of exception continuations for handling raised exceptions, in the expansion of those constructs in `exp_cps` that could directly raise an exception. These include plain application and application of monadic (head of an empty list) and binary operators (division by zero). Plain and recursive functions are expanded not only to take a second argument for their continuation, but also a third argument or their exception handler continuations.

In MP10, we had a type of values that included closures and recursive closures, each of which included an expression from MicroML for the body of the function represented. Since we are evaluating starting with CPS expressions, we need to revise our closures to use CPS expressions as their bodies, and to take the extra arguments for continuations and exception handler continuations the CPS functions now take. They also need to take a new kind of environment for interpreting (integer) variables ranging over continuations and exception continuations, as well. The environments we put in closures and recursive closures for continuations and exceptions continuations will store enough information with each return result to be able to resolve any free variables in the result. For values, there are no free variables. For continuations and exception handle continuations we store a memory along with the continuation so that it is a essentially a closed entity, not requiring any external information for interpretation.

Another change to our type of values is that, for the most part, we no longer need to segregate exceptions as a separate case of values; instead they will be ordinary values handed to an exception handler continuation instead of the main continuation. Our new types of `value` and `cps_env_entry` then are:

```
type value =
    UnitVal
  | BoolVal of bool
  | IntVal of int
  | RealVal of float
  | StringVal of string
  | PairVal of value * value
  | ListVal of value list
  | CPSClosureVal of string * int * int * exp_cps * cps_env_entry list
  | CPSRecClosureVal of string * string * int * int * exp_cps * cps_env_entry list

and cps_env_entry =
     ValueBinding of (string * value)
   | ContBinding of (int * (cps_cont * cps_env_entry list))
   | ExnContBinding of (int * (exn_cont * cps_env_entry list))
```

We will associate values with identifier names in environments, as in previous MPs. We use integers for variables for continuations and exception handler continuations. We will use a form of environment that incorporates each of these using lists of `cps_env_entry`. We include the following look-up functions for retrieving results new type of environment, or memory:

```
val lookup_value : cps_env_entry list -> string -> value option = <fun>
val lookup_cont :
```

```
  cps_env_entry list -> int -> (cps_cont * cps_env_entry list) option = <fun>
val lookup_exn_cont :
  cps_env_entry list -> int -> (exn_cont * cps_env_entry list) option = <fun>
```

An exception to the remark that we no longer "need" exceptions as a separate class of values arises in the handling of primitive (monadic and binary) operators. Here it will be useful to distinguish an integer as an ordinary value from its use as a code for an exception. To restore this ability for this limited case, we have added the type `val_or_exn` used by `monOpApply` and `binOpApply`.

```
type val_or_exn = Value of value | Exn of int
```

We have included the appropriately modified versions of `monOpApply` and `binOpApply` in `mp11common.ml`. Their new types are:

```
val binOpApply : bin_op -> value -> value -> val_or_exn = <fun>
val monOpApply : mon_op -> value -> val_or_exn = <fun>
```

One last type we need is for the result of a single step in the transition semantics. This calls for a separate type because we must account of several kinds of results. In general, a single step in the transition semantics will move us from a CPS expression in some value environment, continuation environment and exception environment, to a new CPS expression in a new value environment, new continuation environment and new exception environment. However, we there is also the desirable possibility that we transition to a final value. There are also the two less fortunate cases, the first where we transition to an exception for which their is no handler, and thus we want to signal the user that an exception was raised, and the least desirable case where evaluation has just gone wrong because we tried to access the value of a variable for which we have none, or some other reason why we have no ability to transition.

```
type step_result =
   Intermediate of (cps_env_entry list * exp_cps)
 | Final of value
 | UncaughtException of int
 | Failed
```

4 Compiling, etc...

For this MP, you will only have to modify **mp11.ml**, adding the functions requested. To test your code, type `make` and the three needed executables will be built: `micromlint`, `micromlintSol` and `grader`. The first two are explained below. `grader` checks your implementation against the solution for a fixed set of test cases as given in the `tests` file.

You may also work interactively with your code in OCaml. To facilitate your doing this, and because there are more files than usual to load than usual, we have included a file `.ocamlinit` that is executed by `ocaml` every time it is started in the mp directory. The contents of the file are:

```
#load "mp11common.cmo";;
#load "micromlparse.cmo";;
#load "micromllex.cmo";;
#load "solution.cmo";;
open Mp11common;;
#use "mp11.ml";;

let parse s = Micromlparse.main Micromllex.token (Lexing.from_string s);;
```

4.1 Running MicroML

The given `Makefile` builds executables called `micromlint` and `micromlintSol`. The first is an executable for an interactive loop for the evaluator built from your solution to the assignment and the second is built from the standard solution. If you run `./micromlint` or `./micromlintSol`, you will get an interactive screen, much like the OCaml interactive screen. You can type in MicroML declarations (followed by a semicolon), and they will be evaluated, and the resulting binding will be displayed.

At the command prompt, the programs will be evaluated (or fail evaluation) starting from the initial memory, which is empty. Each time, if evaluation is successful, the resulting memory will be displayed. Note that a program can fail at any of several stages: lexing, parsing, type inferencing, or evaluation itself. Evaluation itself will tend to fail until you have solved at least some of the problems to come.

5 Notation

Your main assignment in this MP in to implement the function

```
val one_step_exp_cps_eval : cps_env_entry list -> exp_cps -> step_result = <fun>
```

You will be given a specification of the function as a set of transition semantics rules for CPS terms. In our interpreter, we only have abstract syntax for our CPS terms, while the rules will be expressed in a form of concrete syntax. In giving the rules we will use the following conventions:

e = CPS expression

$x,\ y,\ z$ = identifier/variable

b = boolean valued variable

c = constant

v = value

κ = continuation

k = continuation variable (`int`)

ε = an exception handler continuation (`exn_cont`)

ek = an exception handler continuation variable (`int`)

ρ = memory stored as a `cps_env_entry list`

The following table is intended to allow you to translate from concrete syntax to the corresponding abstract syntax for CPS terms, and values.

concrete syntax	abstract syntax
(x, κ)	$\texttt{VarCPS}(\kappa, x)$
(c, κ)	$\texttt{ConstCPS}(\kappa, c)$
$((\otimes x), \kappa/\varepsilon)$	$\texttt{MonOpAppCPS}(\kappa, \otimes, x, \varepsilon)$
$((x \oplus y), \kappa/\varepsilon)$	$\texttt{BinOpAppCPS}(\kappa, \oplus, x, y, \varepsilon)$
$(\texttt{if}\ b\ \texttt{then}\ e_1\ \texttt{else}\ e_2)$	$\texttt{IfCPS}(b, e_1, e_2)$
$(f\ x, \kappa/\varepsilon)$	$\texttt{AppCPS}(\kappa, f, x, \varepsilon)$
$((\lambda x, k, ek.\ e), \kappa)$	$\texttt{FnCPS}(\kappa, x, k, ek, e)$
$((\mu f, x, k, ek.\ e), \kappa)$	$\texttt{FixCPS}(\kappa, f, x, k, ek, e)$

A CPS expression represents the evaluation that is to compute one step of a total computation, either yielding a new CPS expression in the case of a conditional, or yielding a value that is passed to a continuation. In turn, a continuation represents a function that receives a value and generates the next CPS expression. The act of "applying" a continuation to the corresponding value in the memory ρ will be written as $\kappa \cdot v|_\rho$. We will use $\langle x, k, ek, e, \rho\rangle$ for closures and $\langle\langle f, x, k, ek, e, \rho\rangle\rangle$ for recursive closures.

6 Problems

These problems ask you to create an evaluator for `MicroML` by writing the functions `app_cont_to_value` and `one_step_exp_cps_eval` as specified. The evaluator works first by lexing, parsing, type checking and translating the users' code into CPS form (a `exp_cont`). From here, `one_step_exp_cps_eval` is repeatedly called until a `Final` result is returned.

The functions you are to implement

```
val app_cont_to_value :
  cps_env_entry list -> cps_cont -> value -> step_result = <fun>
```

and

```
val one_step_exp_cps_eval :
  cps_env_entry list -> exp_cps -> step_result = <fun>
```

take a memory (*a.k.a* a `cps_env_entry list`) giving values for the various different sorts of identifiers possibly occurring in the other arguments to the functions. The function `app_cont_to_value` additionally takes a continuation and a value and generates the net result of applying the continuation to the value in the given context. It implements $(\kappa \cdot v)|_\rho$. The function `one_step_exp_cps_eval` additionally takes a CPS expression and using the function `app_cont_to_value` computes the result of taking one step of evaluation.

For each problem, you should refer to the list of rules given as part of the problem. The rules specify how evaluation should be carried out, using transition semantics for CPS expressions. Transition semantics were covered in class; see the lecture notes for details. If any expression raises an exception n that is not caught by any exception handler, then `UncaughtException`(n) should be returned. All other cases that the rules fail to cover should return `Failed`.

The problems are ordered such that simpler and more fundamental concepts come first. For this reason, it is recommended that you solve the problems in the order given. Doing so may make it easier for you to test your solution before it is completed.

You are allowed, and even encouraged to create and use helper functions in this MP. In particular, you may want to create functions that implement some of the auxiliary math notation, and some of the implicit coercions.

As mentioned, you can test your code in the executable MicroML environment, or directly in OCaml. The problem statements that follow include some examples in MicroML. However, the problem statements also contain test cases that can be used to test your implementation in the OCaml environment. The former may be more useful in helping you gain an overall understanding of what is happening in this MP, while the latter are more likely to help you implement each individual section.

1. **Continuation Application** (12 pts)

   Implement `app_cont_to_value`. If the continuation is of the form `External`, you should return `Final` of the value. If it is a continuation variable, look it up in the environment and apply the resulting continuation. If the continuation is of the form `ContCPS`(y, e), then create the one-step intermediate result given by adding the binding of y to the value to the memory, and bundling this with the CPS expression e. If the continuation is an exception handler continuation (`ExnMatch`), and the value is an integer, then apply the exception handler continuation to the integer to yield the result.

   To apply an exception handler continuation (`exn_cont`) that is a exception handler continuation variable, look the variable up in the environment and apply the resulting exception handler continuation to the integer. If the handler continuation is empty, the result is an `UncaughtException`. If the handler continuation is an update of an earlier handler continuation, ε, by a partial mapping of integer options to exception continuations, and either the given integer or `None` is assigned an exception handler continuation by the partial mapping, then return the one-step intermediate result using the given memory and the first expression continuation just found. If no match exists in the partial mapping, then recursively apply ε.

   ```
   # app_cont_to_value [] External (IntVal 6);;
   - : step_result = Final (IntVal 6)
   ```

2. **Constants** (2 pt)

Extend `one_step_exp_cps_eval` to handle constants (i.e. integers, bools, real numbers, strings, nil, and unit). For this question you will need to use `const_to_val: const -> value`, which is provided in `mp11common.ml` and is a function you implemented in the previous MP. This function takes a constant and returns the corresponding `value`.

$$\frac{}{((c, \kappa), \rho) \longrightarrow (\kappa \cdot (\mathsf{const\_val}(c)))|_\rho}$$

A sample test case for the OCaml environment:

```
# one_step_exp_cps_eval [] (ConstCPS(External, IntConst 2));;
- : step_result = Final (IntVal 2)
```

In the MicroML environment, this enables

```
> val x = 2;

result:
val x = 2
```

3. **Identifiers** (3 pts)

Extend `one_step_exp_cps_eval` to handle identifiers (i.e. variables).

$$\frac{\rho(x) = v}{((x, \kappa), \rho) \longrightarrow (\kappa \cdot v)|_\rho}$$

Here is a sample test case.

```
# one_step_exp_cps_eval [ValueBinding("x", IntVal 2)] (VarCPS(External, "x"));;
- : step_result = Final (IntVal 2)
```

In the MicroML environment, if you have previously successfully done Problem 1 and run the sample code, you can test this problem with:

```
> x;

result:
val it = 2
```

4. **Monadic Operator Application** (7 pts)

Extend `one_step_exp_cps_eval` to handle application of monadic operators `~`, `hd`, `tl`, `fst`, `snd` and `print_string`. For this question, you will want to use the function `monOpApply: mon_op -> value -> val_or_exn` that is a modest modification of the function you implemented in the previous MP.

$$\frac{\rho(x) = v \quad \mathsf{monOpApply}(\otimes, v) = v'}{(\otimes x, \kappa/\varepsilon), \rho) \longrightarrow (\kappa \cdot v')|_\rho} \qquad \frac{\rho(x) = v \quad \mathsf{monOpApply}(\otimes, v) = \mathsf{Exn}(n)}{((\otimes x, \kappa/\varepsilon), \rho) \longrightarrow (\varepsilon \cdot n)|_\rho}$$

where $\otimes$ is a monadic constant function value. A sample test case in the OCaml environment:

```
# one_step_exp_cps_eval [ValueBinding("x", IntVal 2)]
  (MonOpAppCPS(External, IntNegOp, "x", EmptyExnContCPS));;
- : step_result = Final (IntVal (-2))
```

A sample test case in the MicroML interpreter:

```
> val y = ~3;

result:
val y = -3
```

5. **Binary Operators** (8 pts)

Extend `one_step_exp_cps_eval` to handle the application of binary operators. For this question, you need to use the `binOpApply : bin_op -> value -> value -> val_or_exn` function.

$$\frac{\rho(x) = v_1 \quad \rho(y) = v_2 \quad \mathsf{binOpApply}(\oplus, v_1, v_2) = v}{((x \oplus y, \kappa/\varepsilon)\rho) \longrightarrow (\kappa \cdot v)|_\rho} \qquad \frac{\rho(x) = v_1 \quad \rho(y) = v_2 \quad \mathsf{binOpApply}(\oplus, v_1, v_2) = \mathsf{Exn}(n)}{((x \oplus y, \kappa/\varepsilon), \rho) \longrightarrow (\varepsilon \cdot n)|_\rho}$$

A sample test case.

```
# one_step_exp_cps_eval [ValueBinding("b", IntVal 3);ValueBinding("a", IntVal 2)]
  (BinOpAppCPS(External, IntPlusOp, "a", "b", EmptyExnContCPS));;
- : step_result = Final (IntVal 5)
```

In the MicroML environment, you can test this problem with:

```
> 2 + 3;

result:
val it = 5
```

6. **If constructs** (5 pts)

Extend `one_step_exp_cps_eval` to handle `if` constructs.

$$\frac{\rho(b) = \texttt{true}}{(\texttt{if}\ b\ \texttt{then}\ e_1\ \texttt{else}\ e_2), \rho) \longrightarrow (e_1, \rho)} \qquad \frac{\rho(b) = \texttt{false}}{(\texttt{if}\ b\ \texttt{then}\ e_1\ \texttt{else}\ e_2), \rho) \longrightarrow (e_2, \rho)}$$

A sample test case.

```
# one_step_exp_cps_eval [ValueBinding("a", BoolVal true)]
  (IfCPS("a", ConstCPS(External,IntConst 1), ConstCPS(External,IntConst 0)));;
- : step_result =
Intermediate
 ([ValueBinding ("a", BoolVal true)], ConstCPS (External, IntConst 1))
```

In the MicroML environment,

```
> if true then 1 else 0;

result:
val it = 1
```

7. **Functions** (3 pts)

Extend `one_step_exp_cps_eval` to handle functions. You will need to create a `CPSClosureVal` represented by $\langle x, k, ek, e, \rho \rangle$ in the rule below.

$$\frac{}{(((\lambda x, k, ek.\ e), \kappa), \rho) \longrightarrow (\kappa \cdot \langle x, k, ek, e, \rho \rangle)|_\rho}$$

A sample test case.

```
# one_step_exp_cps_eval []
 (FnCPS (External, "x", 1, 0,
   VarCPS
    (ContCPS ("a",
      ConstCPS
       (ContCPS ("b",
         BinOpAppCPS (ContVarCPS 1, IntPlusOp, "a", "b", ExnContVarCPS 0)),
       IntConst 5)),
    "x")))
- : step_result =
Final
 (CPSClosureVal ("x", 1, 0,
   VarCPS
    (ContCPS ("a",
      ConstCPS
       (ContCPS ("b",
         BinOpAppCPS (ContVarCPS 1, IntPlusOp, "a", "b", ExnContVarCPS 0)),
       IntConst 5)),
    "x"),
   []))
```

In the MicroML environment,

```
> val plus5 = fn x => x + 5;

result:
val plus5 = <some closure>
```

8. **Recursive Expressions** (3 pts)

Extend `one_step_exp_cps_eval` to handle recursive CPS expressions (`FixCPS`). In this case, you will need to create a `CPSRecClosureVal` represented by $\langle\langle f, x, k, ek, e, \rho \rangle\rangle$ in the rule below.

$$\frac{}{(((\mu f, x, k, ek.\ e), \kappa), \rho) \longrightarrow (\kappa \cdot \langle\langle f, x, k, ek, e, \rho \rangle\rangle)|_\rho}$$

A sample test case.

```
# let dec = parse "val rec f n = if n = 0 then 1 else f (n - 1);" in
  one_step_exp_cps_eval []
   (cps_dec dec (VarCPS(External, "f"))
   EmptyExnContCPS);;
- : Mp11common.step_result =
Intermediate
```

```
([ValueBinding
   ("f",
    CPSRecClosureVal ("f", "n", 1, 0,
     VarCPS
      (ContCPS ("5",
        ConstCPS
         (ContCPS ("6",
           BinOpAppCPS
            (ContCPS ("0",
              IfCPS ("0", ConstCPS (ContVarCPS 1, IntConst 1),
               VarCPS
                (ContCPS ("1",
                  VarCPS
                   (ContCPS ("3",
                     ConstCPS
                      (ContCPS ("4",
                        BinOpAppCPS
                         (ContCPS ("2",
                           AppCPS (ContVarCPS 1, "1", "2", ExnContVarCPS 0)),
                         IntMinusOp, "3", "4", ExnContVarCPS 0)),
                      IntConst 1)),
                    "n")),
                 "f"))),
             EqOp, "5", "6", ExnContVarCPS 0)),
         IntConst 0)),
      "n"),
     []))],
 VarCPS (External, "f"))
```

In the MicroML environment, you should now be able to do

```
> val rec f n = if n = 0 then 1 else f (n - 1);

result:
val f = <some recvar>
```

9. **Function application** (10 pts)

Extend `one_step_exp_cps_eval` to handle function application.

$$\frac{\rho(f) = \langle y, k, ek, e, \rho' \rangle \quad \rho(x) = v}{((f\ x, \kappa/\varepsilon), \rho) \longrightarrow (e, ([y \mapsto v] + [k \mapsto (\kappa, \rho)] + [ek \mapsto (\varepsilon, \rho)] + \rho'))}$$

$$\frac{\rho(f) = \langle\langle g, y, k, ek, e, \rho' \rangle\rangle \quad \rho(x) = v}{((f\ x, \kappa/\varepsilon), \rho) \longrightarrow (e, ([y \mapsto v; g \mapsto \langle\langle g, y, k, ek, e, \rho' \rangle\rangle] + [k \mapsto (\kappa, \rho)] + [ek \mapsto (\varepsilon, \rho)] + \rho'))}$$

Caution: In general, the order in which bindings are added to the environment matters, so you will be required to implement the order given by the rules, even though other orders would give equivalent semantics. A sample test case.

```
# one_step_exp_cps_eval
 [ValueBinding("plus5", (CPSClosureVal ("x", 1, 0,
```

```
  VarCPS
   (ContCPS ("a",
     ConstCPS
      (ContCPS ("b",
        BinOpAppCPS (ContVarCPS 1, IntPlusOp, "a", "b", ExnContVarCPS 0)),
       IntConst 5)),
    "x"),
    [])));
  ValueBinding("c",IntVal 2)]
 (AppCPS(External, "plus5", "c", EmptyExnContCPS));;
- : Mp11common.step_result =
Intermediate
 ([ValueBinding ("x", IntVal 2);
   ContBinding
    (1,
     (External,
      [ValueBinding
        ("plus5",
         CPSClosureVal ("x", 1, 0,
          VarCPS
           (ContCPS ("a",
             ConstCPS
              (ContCPS ("b",
                BinOpAppCPS (ContVarCPS 1, IntPlusOp, "a", "b",
                 ExnContVarCPS 0)),
               IntConst 5)),
            "x"),
           [])));
       ValueBinding ("c", IntVal 2)]));
   ExnContBinding
    (0,
     (EmptyExnContCPS,
      [ValueBinding
        ("plus5",
         CPSClosureVal ("x", 1, 0,
          VarCPS
           (ContCPS ("a",
             ConstCPS
              (ContCPS ("b",
                BinOpAppCPS (ContVarCPS 1, IntPlusOp, "a", "b",
                 ExnContVarCPS 0)),
               IntConst 5)),
            "x"),
           [])));
       ValueBinding ("c", IntVal 2)]))],
  VarCPS
   (ContCPS ("a",
     ConstCPS
      (ContCPS ("b",
        BinOpAppCPS (ContVarCPS 1, IntPlusOp, "a", "b", ExnContVarCPS 0)),
       IntConst 5)),
```

```
   "x"))
```

In the MicroML environment,

```
> plus5 2;

result:
val it = 7
```

Final Remark: Please add numerous test cases to the test suite. Try to cover obscure cases.