CS 496: Homework Assignment 4
Due: 13 November 2022, 11:55pm

1 Assignment Policies

Collaboration Policy. It is acceptable for students to collaborate in understanding the material but not in solving the problems or programming. Use of the Internet is allowed, but should not include searching for existing solutions.

Under absolutely no circumstances code can be exchanged between students. Excerpts of code presented in class can be used.

Assignments from previous offerings of the course must not be re-used. Violations will be penalized appropriately.

2 Assignment

This assignment has two parts. Part I consist in adding records with mutable fields to EXPLICIT-REFS to obtain EXPLICIT-REFS-MF. Part II consists in writing a series of programs in the language EXPLICIT-REFS-MF that make use of this new language construct.

2.1 Part I

As seen in class in OCaml, a mutable field in a record is one whose contents may be updated through assignment. One can declare a field to be mutable by using "`<=`" instead of "`=`". In the next example, a record is defined that holds two items of personal data, a social security number and the age. The `ssn` field in the example below is immutable but the `age` field is mutable; `age` is then updated to 31:

```
let p = {ssn = 10; age <= 30}
in begin
     p.age <= 31;  (* updates age to 31 *)
     p.age
   end
```

Evaluating this expression should produce `Ok (NumVal 31)`. Consider the evaluation of this other expression:

```
let p = {ssn = 10; age <= 30}
in begin
     p.age <= 31;
     p
   end
```

It should produce `Ok (RecordVal [("ssn", (false, NumVal 10)); ("age", (true, RefVal 1))])`. Notice two things here:

1. `false` indicates the field is immutable and `true` that it is mutable.
2. Mutable fields are always assigned a reference. In this example, the store will hold the expressed value `NumVal 31` at location 1.

Updating an immutable field should not be allowed. For example, the following expression should report an error `Error "Field not mutable"`:

```
let p = { ssn = 10; age <= 20}
in begin
    p.ssn <= 11;
    p.age
   end
```

EXPLICIT-REFS-MF also has a predicate to check whether a run-time value is a number or not. Here is how it should behave:

```
# interp "number?(2)";;
- : exp_val Explicit_refs.Ds.result = Ok (BoolVal true)
# interp "number?(2+1)";;
- : exp_val Explicit_refs.Ds.result = Ok (BoolVal true)
utop # interp "number?(zero?(0))";;
- : exp_val Explicit_refs.Ds.result = Ok (BoolVal false)
utop # interp "number?(proc(x) {x+1})";;
- : exp_val Explicit_refs.Ds.result = Ok (BoolVal false)
```
utop

The updated concrete syntax is:

<Expression> ::= { <Identifier> <FieldType> <Expression>$^{+(;)}$}
<Expression> ::= <Expression>. <Identifier>
<Expression> ::= <Expression>. <Identifier> <= <Expression>
<Expression> ::= number?(<Expression>)

<FieldType> ::= = | <=

The updated abstract syntax is as follows:

```
type expr =
  ...
  | Record of (string*(bool*expr)) list
  | Proj of expr*string
  | SetField of expr*string*expr
  | IsNumber of expr
```

For example,

```
# parse "
let p = {ssn = 10; age <= 30}
in begin
     p.age <= 31;
     p
   end";;
- : expr =
Let ("p", Record [("ssn", (false, Int 10)); ("age", (true, Int 30))],
 BeginEnd [SetField (Var "p", "age", Int 31); Var "p"])
```

utop

You are asked to implement the interpreter extension. The `RecordVal` constructor has added for you.

```
type exp_val =
  | NumVal of int
  | BoolVal of bool
  | ProcVal of string*Ast.expr*env
  | RecordVal of (string*(bool*exp_val)) list
```

ds.ml

As for `eval_expr`, the case for `Record` has already been implemented for you. You are asked to implement `Proj`, `SetField` and `IsNumber`:

```
let rec  eval_expr : expr -> exp_val ea_result = fun e ->
  match e with
  | IsNumber(e) ->
     failwith "implement"
  | Record(fs) ->
    sequence (List.map process_field fs) >>= fun evs ->
    return (RecordVal (addIds fs evs))
  | Proj(e,id) ->
     failwith "implement"
  | SetField(e1,id,e2) ->
     failwith "implement"
  | IsNumber(e) ->
     failwith "implement"
 and
 process_field (_id,(is_mutable,e))  =
  eval_expr e >>= fun ev ->
  if is_mutable
  then return (RefVal (Store.new_ref g_store ev))
  else return ev
```

where `sequence` is define here:

```
let rec sequence : ('a ea_result) list -> ('a list) ea_result  =
  fun cs ->
  match cs with
  | [] -> return []
  | c::t ->
    c >>= fun v ->
    sequence t >>= fun vs ->
    return (v::vs)
```

ds.ml

2.2 Part II: Implementing Linked Lists in EXPLICIT-REFS-MF

This part of the assignment asks you to implement linked-list operations in the language EXPLICIT-REFS-MF. Since we do not have a print operation, in order to test your code you will have to insert a `debug` instruction and inspect the contents of the environment and store. Following the discussion in class using OCaml, a linked-list will be implemented as a record with two fields `head` and `size`, both of which are mutable. The `head` field may either contain `0` for null or may be a node record. Node records have fields `data` and `next`, both of which are mutable. We will start with some examples. EXPLICIT-REFS-MF source code must be placed in text files with extension `.exr`.

Example 1 below creates a linked-list with two nodes.

```
(* Example 1 *)

let l1 = { head <= 0; size <= 0}         (* 0 in head signals null *)
in let add_front = proc (x) { proc (l) {
          begin
             l.head <={ data <=x; next <= l.head };
             l.size <= l.size+1
          end
          } }
in begin
     ((add_front 2) l1);
     ((add_front 3) l1);
     debug(l1)  (* required to inspect the list *)
   end
```

ll_add_front.exr

You may run this in utop as follows:

```
# interpf "ll_add_front";;
>>Environment:
l1->{head <= RefVal (0); size <= RefVal (1)},
add_front->ProcVal(x,Proc(l,BeginEnd(Var l.head<={data=Var x;next=Var l.head};
                    Var l.size<=Add(Var l.size,Int 1))),
                   [l1->{head <= RefVal (0); size <= RefVal (1)}])
>>Store:
0->{data <= RefVal (4); next <= RefVal (5)},
1->NumVal 2,
2->NumVal 2,
3->NumVal 0,
4->NumVal 3,
5->{data <= RefVal (2); next <= RefVal (3)}
- : exp_val Explicit_refs.Ds.result = Error "Reached breakpoint"
```

utop

Example 2 below creates a linked list with two nodes and then adds one to each data element in each node.

```
(* Example 2 *)

let l1 = { head <= 0; size <= 0}
in let add_front = proc (x) { proc (l) {
          begin
             l.head <={ data <=x; next <= l.head };
             l.size <= l.size+1
```

```
              end } }
in letrec bump_helper (node) =
   if number?(node)
   then 0
   else (begin
              node.data <= node.data + 1;
              (bump_helper node.next)
         end)
in let bump = proc (ll) { (bump_helper ll.head) }
in begin
     ((add_front 2) ll);
     ((add_front 3) ll);
     (bump ll);
     debug(ll)
   end
```

ll_bump.exr

You may run this in utop as follows:

```
utop # interpf "ll_bump";;
>>Environment:
ll->{head <= RefVal (0); size <= RefVal (1)},
add_front->ProcVal(x,Proc(l,BeginEnd(Var l.head<={data=Var x;next=Var l.head};
  Var l.size<=Add(Var l.size,Int 1))),[ll->{head <= RefVal (0); size <= RefVal (1)}]),
bump_helper->Rec(node,IfThenElse(Number?(Var node),Int 0,BeginEnd(
  Var node.data<=Add(Var node.data,Int 1);App(Var bump_helper,Var node.next)))),
bump->ProcVal(ll,App(Var bump_helper,Var ll.head),[ll->{head <= RefVal (0);
  size <= RefVal (1)},add_front->ProcVal(x,Proc(l,BeginEnd(Var l.head<={data=Var x;next=Var l.head}
>>Store:
0->{data <= RefVal (4); next <= RefVal (5)},
1->NumVal 2,
2->NumVal 3,
3->NumVal 0,
4->NumVal 4,
5->{data <= RefVal (2); next <= RefVal (3)}
- : exp_val Explicit_refs.Ds.result = Error "Reached breakpoint"
```

utop

You are asked to implement the following exercises. The stubs are provided for you in the `src` folder. You may include helper functions.

1. `ll_add_last.exr`. For example,

```
utop # interpf "ll_add_last";;
>>Environment:
ll->{head <= RefVal (0); size <= RefVal (1)},
add_last_helper->...,
add_last->...,
>>Store:
0->{data <= RefVal (2); next <= RefVal (3)},
1->NumVal 3,
2->NumVal 2,
3->{data <= RefVal (4); next <= RefVal (5)},
4->NumVal 3,
5->{data <= RefVal (6); next <= RefVal (7)},
6->NumVal 4,
7->NumVal 0
- : exp_val Explicit_refs.Ds.result = Error "Reached breakpoint"
```

utop

2. `ll_remove_first.exr` For example,

```
utop # interpf "ll_remove_first";;
>>Environment:
l1->{head <= RefVal (0); size <= RefVal (1)},
add_front->ProcVal(x,Proc(l,BeginEnd(Var l.head<={data=Var x;next=Var l.head};
  Var l.size<=Add(Var l.size,Int 1))),[l1->{head <= RefVal (0); size <= RefVal (1)}]),
remove_first-> ...,
>>Store:
0->{data <= RefVal (4); next <= RefVal (5)},
1->NumVal 2,
2->NumVal 2,
3->NumVal 0,
4->NumVal 3,
5->{data <= RefVal (2); next <= RefVal (3)},
6->NumVal 4,
7->{data <= RefVal (4); next <= RefVal (5)}
- : exp_val Explicit_refs.Ds.result = Error "Reached breakpoint"
```

utop

You may assume the list is not empty.

3. `ll_remove_last.exr` For example,

```
utop # interpf "ll_remove_last";;
>>Environment:
l1->{head <= RefVal (0); size <= RefVal (1)},
add_front->ProcVal(x,Proc(l,BeginEnd(Var l.head<={data=Var x;next=Var l.head};
  Var l.size<=Add(Var l.size,Int 1))),[l1->{head <= RefVal (0); size <= RefVal (1)}]),
remove_last_helper->...,
remove_last->...,
>>Store:
0->{data <= RefVal (6); next <= RefVal (7)},
1->NumVal 2,
2->NumVal 2,
3->NumVal 0,
4->NumVal 3,
5->NumVal 0,
6->NumVal 4,
7->{data <= RefVal (4); next <= RefVal (5)}
- : exp_val Explicit_refs.Ds.result = Error "Reached breakpoint"
```

utop

You may assume the list is not empty.

3 Submission instructions

Submit a file named `HW4.zip` through Canvas. Include the original stub provided. Please write the names of the members of the team in the as a comment at the top of the `interp.ml` file.