Extending REC with References

CS496

A New Language

- We are going to extend REC with references
- As a consequence we obtain a functional language that supports variables as in imperative programming
- Note that all previous programming features shall still be available, including
  - let expressions
  - procedures
  - recursion

Implicit vs Explicit References

There are essentially two ways of doing this:

1. Treat every variable as a mutable reference
   - The resulting language is called IMPLICIT-REFS
   - References are implicit
2. Add mutable references to the non-mutable ones
   - The resulting language is called EXPLICIT-REFS
   - References are explicit

We are going to study both

Implicit vs Explicit References

IMPLICIT-REFS

```
let g = let count = 0
        in proc(d) {
            begin set count =  count + 1;
                  count
            end }
in (g 11) - (g 22)
```

EXPLICIT-REFS

```
let g = let counter = newref(0)
        in proc (d) {
            begin
               setref(counter, deref(counter) + 1);
               deref(counter)
            end }
in (g 11) - (g 22)
```

Concrete Syntax

Two new productions that are added to those of REC

$$
\begin{array}{lcl}
\langle \mathit{Expression} \rangle & ::= & \text{set } \langle \mathit{Identifier} \rangle = \langle \mathit{Expression} \rangle \\
\langle \mathit{Expression} \rangle & ::= & \text{begin } \langle \mathit{Expression} \rangle^{+(;)} \text{ end}
\end{array}
$$

- The `set` expression is assignment
- A `begin` ... `end` expression evaluates its subexpressions in order and returns the value of the last one.

Concrete Syntax – An Example

```
let g = let count = 0
        in proc(d) {
            begin
              set count = count + 1;
              count
            end
        }
in (g 11) - (g 22)
```

IMPLICIT-REFS: Abstract Syntax

```
type expr =
  | Var of string
  | Int of int
  | Add of expr*expr
  | Sub of expr*expr
  | Mul of expr*expr
  | Div of expr*expr
  | Let of string*expr*expr
  | IsZero of expr
  | ITE of expr*expr*expr
  | Proc of string*expr
  | App of expr*expr
  | Letrec of string*string*expr*expr
  | Set of string*expr
  | BeginEnd of expr list
```

Example - Abstract Syntax

```
let g = let count = 0
        in proc(d) {
            begin
              set count = count + 1;
              count
            end
        }
in (g 11) - (g 22)
```

```
AProg
 (Let ("g",
   Let ("count", Int 0,
    Proc ("d", BeginEnd [Set ("count", Add (Var "count
       ↪ ", Int 1)); Var "count"])),
  Sub (App (Var "g", Int 11), App (Var "g", Int 22))))
```

Unit Value

- An expression in REC can return one of:

$$\mathsf{ExpVal} \quad = \quad \mathsf{Int} + \mathsf{Bool} + \mathsf{Proc}$$

- What should the result of this be?

```
let x = 2 in set x =7
```

- Same question: what is the result of evaluating a `set` expression?

`set` is evaluated to cause an effect (modify the store/memory), not return a result

Unit Value

- We introduce a special value for this situation, namely `UnitVal`

$$\text{ExpVal} \quad = \quad \text{Int} + \text{Bool} + \text{Proc} + \text{Unit}$$

- `set` evaluates to a unit value

Motivating the Store

Consider this assignment statement in Java

$$x := x + 1$$

- As we know, variables have two readings:
  - An address (the blue occurrence of `x`)
  - A value (the red occurence of `x`)
- Environments therefore have to change their type:
  - Before: $\mathit{Vars} \longrightarrow \mathit{ExpVal}$
  - Now: $\mathit{Vars} \longrightarrow \mathit{Refs}$
- *Refs* is a set of references or locations
- References point to expressed values
  - Hence we typically write $\mathit{Refs}(\mathit{ExpVal})$ (rather than just *Refs*)

Motivating the Store

Consider this assignment statement in Java

$$x := x + 1$$

- Environments therefore have to change their type:
  - Before: $Vars \longrightarrow ExpVal$
  - Now: $Vars \longrightarrow Refs(ExpVal)$
- Revisiting the two readings of a variable
  - Blue `x`: we just look it up in the environment
  - Red `x`: we look it up in the environment and then use that location to get its value in a store or memory
- Our interpreter thus shall need a store

Environment and Store

Environment

RefVal 0	RefVal 1	RefVal 2
x	y	z

Store

NumVal 3	BoolVal true	NumVal 7
0	1	2

Summing Up Our Analysis

$$
\begin{array}{rcl}
\text{ExpVal} & = & \text{Int} + \text{Bool} + \text{Proc} \\
\text{DenVal} & = & \text{Ref(ExpVal)}
\end{array}
$$

- Recall: Denoted values = Values assigned to variables in the environment
- References exist only as the bindings of variables.
- Two options for the type of the interpreter
  1. Store is passed as argument

     ```
     eval_expr:: env -> store -> expr -> exp_val
     ```

  2. Store is held in global variable (we choose this one)

     ```
     eval_expr::env -> expr -> exp_val
     ```

Implementing the Store

- There are many ways to represent the store (eg. Lists, User-defined datatypes, etc.)
  - We choose arrays since it is simple
  - The references will just be indices
  - An example

`NumVal 3`	`BoolVal true`	`NumVal 7`	`NumVal 28`
0	1	2	3

- Sample operations we should support
  - Create a store
  - Lookup the value associated to a reference
  - Ask the store for a fresh reference and assign it a value (allocation)

Store Interface - `store.mli`

```
type 'a t (* t is the (opaque) type of the store *)

val empty_store : int -> 'a -> 'a t

val new_ref : 'a t -> 'a -> int

val deref: 'a t -> int -> 'a

val set_ref : 'a t -> int -> 'a -> unit

val store_to_list:  'a t -> 'a list
```

Implementing the Store

```
type 'a t = {mutable data: 'a array;
             mutable size: int}
(* data is declared mutable so the store may be resized *)

let empty_store i v = { data=Array.make i v; size=0 }

let get_size st = st.size
```

Implementing the Store

```
let enlarge_store st value =
  let new_array = Array.make (st.size*2) value
  in Array.blit st.data 0 new_array 0 st.size;
  st.data<-new_array

let new_ref st value =
  if Array.length (st.data)=st.size
  then enlarge_store st value
  else ();
  begin
    st.data.(st.size)<-value;
    st.size<-st.size+1;
    st.size-1
  end
```

Implementing the Store

`NumVal 3`	`BoolVal true`	`NumVal 7`
0	1	2

After `newref (NumVal 28)`:

`NumVal 3`	`BoolVal true`	`NumVal 7`	`NumVal 28`
0	1	2	3

Implementing the Store

```
let deref st ref = st.data.(ref)

let set_ref st ref value =
    if ref>=st.size
    then failwith "Index out of bounds"
    else st.data.(ref)<Value

let rec take n = function
  | [] -> []
  | x::xs when n>0 -> x::take (n-1) xs
  | x::xs -> []

let store_to_list st =
  take st.size @@ Array.to_list @@ st.data
```

Implementing the Store

`NumVal 3`	`BoolVal true`	`NumVal 7`	`NumVal 28`
0	1	2	3

After `setref st 1 (NumVal 42)`:

`NumVal 3`	`NumVal 42`	`NumVal 7`	`NumVal 28`
0	1	2	3

Specification of Behavior of the Interpreter

- We now specify how the interpreter `eval_expr` behaves
- Input:
  - expression
  - environment
  - store
- Output:
  - expressed value
  - updated store

```
eval_expr:: env -> store -> expr -> (expval * store)
```

- Note: As mentioned, in our implementation the store won't be passed as a parameter, it will be held in a global variable

Specification of Behavior of Interpreter on Constants

- First we revisit the interpreter's behavior for the simplest cases.

```
eval_expr ρ σ (Int n) = (NumVal n, σ)
```

- ρ is an environment
- σ is the store
- `Int` *var* is the expression to evaluate
- In this case the store σ is returned unaltered

Specification of Behavior of Interpreter on Difference

```
eval_expr ρ σ0 (Sub exp1 exp2) =
    let (val1,σ1) = eval_expr ρ σ0 exp1
    in let (val2,σ2) = eval_expr ρ σ1 exp2
    in NumVal ((expval_to_num val1) -
         (expval_to_num val2)), σ2)
```

Specification of Behavior of Interpreter on Variables

1. look up the identifier in the environment to find the location to which it is bound
2. look up in the store to find the value at that location

```
eval_expr ρ σ (Var var)= (σ(ρ(var)), σ)
```

Specification of Behavior of Interpreter w.r.t. Assignment

```
eval_expr ρ σ0 (Set (var,exp1)) =
  let (val1,σ1) = eval_expr ρ σ0 exp1
  in (UnitVal, [ρ(var) = val1]σ1)
```

- `UnitVal` is the value of an expression, such as `set`, whose sole objective is to cause an effect
- $[\rho(var) = val1]\sigma_1$ stands for "update the reference $\rho(var)$ in the store σ_1 with the new value `val1`

Implementation

```
eval_expr ρ σ (Var var) = (σ(ρ(var)),σ)
```

```
eval_expr (en:env) (e:expr) :exp_val =
 match e with
 | Int n            -> NumVal n
 | Var id           ->
   (match apply_env en id with
   | None -> failwith @@ "Variable "^id^" undefined"
   | Some ev -> Store.deref g_store @@ refVal_to_int
     ↪ ev)
```

Implementation

```
eval_expr ρ σ0 (Set (var,exp1)) =
  let (val1,σ1) = eval_expr ρ σ0 exp1
  in (NumVal 27, [ρ(var) = val1]σ1)
```

```
eval_expr (en:env) (e:expr) :exp_val =
  match e with
  ...
  | Set(id,e) ->
    let v=eval_expr en e
    in Store.set_ref g_store (refVal_to_int (from_some (apply_env g_store en id))) v;
       UnitVal
```

Updating the Implementation of Extant REC Features

- The implementation of features that were already present in REC also have to be updated
- We've already seen the cases for constants and difference
- Here is the specification of the behavior of the interpreter for `let`

```
eval_expr ρ σ0 (Let(var,exp1,body))=
  let (val1,σ1) = eval_expr ρ σ0 exp1
  in eval_expr [var = l]ρ [l = val1]σ1 body
```

- l denotes a fresh store location

Updating the Implementation of Extant REC Features

- The implementation for `let`

```
eval_expr (en:env) (e:expr) :exp_val =
  match e with
  ...
  | Let(x, e1, e2) ->
    let v1 = eval_expr en e1
    in let l = Store.new_ref g_store v1
    in eval_expr (extend_env en x (RefVal l)) e2
```

Specification of Behavior of Interpreter w.r.t. Procedures

```
eval_expr ρ σ (Proc(var,body)) =
    (procVal(var,body,ρ), σ)
```

- No changes w.r.t. REC here

Specification of Behavior of Interpreter w.r.t. Procedure Calls

```
eval_expr ρ σ0 (App(rator,rand)) =
    let (proc,σ1) = eval_expr ρ σ0 rator
        (arg,σ2) =  eval_expr ρ σ1 rand
    in (apply_proc proc arg, σ2)
```

- No changes w.r.t. REC here but must update `apply_proc`

Specifying the Behavior of Procedure Application

```
apply_proc (ProcVal(var, body, ρ)) val σ =
     eval_expr [var = l]ρ [l = val]σ body
```

- Here l denotes a fresh store location

Implementing Procedure Application

```
(* exp_val -> exp_val -> exp_val *)
let rec apply_proc f a =
  match f with
    ProcVal (x,b,env) ->
    let l = Store.new_ref g_store a
    in eval_expr (extend_env env x (RefVal l)) b
   | _ -> failwith "apply_proc: Not a procVal"
```

Environment Lookup

- Last we have to update environment lookup
- This operation was called `apply_env`
- Only the case for `letrec` has to be updated
- Before doing so we recall its definition

apply-env as Implemented in REC

```
let rec apply_env (env:env) (id:string):exp_val option =
  match env with
  | EmptyEnv -> None
  | ExtendEnv (key,value,env) ->
    if id=key
    then Some value
    else apply_env env id
  | ExtendEnvRec(key,param,body,en) ->
    if id=key
    then Some (ProcVal(param,body,env))
    else apply_env en id
```

- Note: `apply_env` would return a closure

Updating `apply-env`

- Before:
  - `apply_env` would return a closure
- Now:
  - `apply_env` should return a reference to a location in the store containing the appropriate closure
- The reason: how the interpreter manages variable lookup

```
eval_expr ρ σ (Var var) = (σ(ρ(var)),σ)
```

$\rho(var)$ is environment lookup of var in ρ

Updating `apply-env`

```
let rec apply_env (st:exp_val Store.t) (env:env) (id:string):exp_val option =
  match env with
  | EmptyEnv -> None
  | ExtendEnv (key,value,env) ->
    if id=key
    then Some value
    else apply_env st env id
  | ExtendEnvRec(key,param,body,en) ->
    if id=key
    then Some (RefVal (Store.new_ref st (ProcVal(param,body,env))))
    else apply_env st en id
```

- The closure is stored in the store: we return a reference to it
- This completes the implementation of IMPLICIT-REFS.

The Interpreter for IMPLICIT-REFS

- Code available in Canvas `Modules/Interpreters`
- Directory `implicit-refs`
- Compile with `ocamlbuild -use-menhir interp.ml`
- Make sure the `.ocamlinit` file is in the folder of your sources
- Run `utop`

EXPLICIT-REFS: A language with explicit references

1. We now define a new language EXPLICIT-REFS, which adds references as expressed values to our language.
2. Concrete and Abstract Syntax
3. Specification
4. Implementation

Implicit vs Explicit References

IMPLICIT-REFS

```
let g = let count = 0
        in proc(d)
            begin { set count = count + 1;
                    count
            end }
in (g 11) - (g 22)
```

EXPLICIT-REFS

```
let g = let counter = newref(0)
        in proc (d) {
            begin
               setref(counter, deref(counter) +1);
               deref(counter)
            end }
in (g 11) - (g 22)
```

Implicit vs Explicit Store

- In the implicit store design, every variable is mutable.
  - Allocation, dereferencing and mutation are built into the language.
- The explicit reference design gives a clear account of allocation, dereferencing, and mutation
  - All these operations are explicit in the programmer's code.

Expressed and Denoted Values

Before (IMPLICIT-REFS)

$$\begin{array}{lcl} \mathsf{ExpVal} & = & \mathsf{Int} + \mathsf{Bool} + \mathsf{Proc} + \mathsf{Unit} \\ \mathsf{DenVal} & = & \mathsf{Ref(ExpVal)} \end{array}$$

Now (EXPLICIT-REFS):

$$\begin{array}{lcl} \mathsf{ExpVal} & = & \mathsf{Int} + \mathsf{Bool} + \mathsf{Proc} + \mathsf{Unit} + \mathsf{Ref(ExpVal)} \\ \mathsf{DenVal} & = & \mathsf{ExpVal} \end{array}$$

Think of the result of evaluating:

```
newref(7)
```

Or

```
let x = newref(7) in x
```

Expressed and Denoted Values

Before (IMPLICIT-REFS)

$$\begin{array}{lcl} \mathsf{ExpVal} & = & \mathsf{Int} + \mathsf{Bool} + \mathsf{Proc} + \mathsf{Unit} \\ \mathsf{DenVal} & = & \mathsf{Ref(ExpVal)} \end{array}$$

Now (EXPLICIT-REFS):

$$\begin{array}{lcl} \mathsf{ExpVal} & = & \mathsf{Int} + \mathsf{Bool} + \mathsf{Proc} + \mathsf{Unit} + \mathsf{Ref(ExpVal)} \\ \mathsf{DenVal} & = & \mathsf{ExpVal} \end{array}$$

- Before: programs could only produce numbers, booleans, closures or unit as a result
- Now: programs can also produce references as a result (or even store them)

Expressed Values for EXPLICIT-REFS

- Expressed values before

```
type exp_val =
  | NumVal of int
  | BoolVal of bool
  | ProcVal of string*Ast.expr*env
  | UnitVal
```

- Now

```
type exp_val =
  | NumVal of int
  | BoolVal of bool
  | ProcVal of string*Ast.expr*env
  | UnitVal
  | RefVal of int
```

Environment and Store

Environment

NumVal 3	NumVal 3	RefVal 0
x	y	z

Store

NumVal 7	RefVal 9	BoolVal true
0	1	2

Yet Another Example – References to References

```
let x = newref(newref(0))
in begin
     setref(deref(x), 11);
     deref(deref(x))
   end
```

- Allocates a new reference containing 0.
- Then binds x to a reference containing the above reference.
- The value of `deref(x)` is a reference to the first reference.
- So when it evaluates the setref, it is the first reference that is modified, and the entire program returns 11.

EXPLICIT-REFS: Concrete Syntax

$$
\begin{array}{lll}
\langle \mathit{Expression} \rangle & ::= & \langle \mathit{Number} \rangle \\
\langle \mathit{Expression} \rangle & ::= & \langle \mathit{Expression} \rangle \text{ - } \langle \mathit{Expression} \rangle \\
\langle \mathit{Expression} \rangle & ::= & \text{zero? } (\langle \mathit{Expression} \rangle) \\
\langle \mathit{Expression} \rangle & ::= & \text{if } \langle \mathit{Expression} \rangle \\
 & & \text{then } \langle \mathit{Expression} \rangle \text{ else } \langle \mathit{Expression} \rangle \\
\langle \mathit{Expression} \rangle & ::= & \langle \mathit{Identifier} \rangle \\
\langle \mathit{Expression} \rangle & ::= & \text{let } \langle \mathit{Identifier} \rangle = \langle \mathit{Expression} \rangle \text{ in } \langle \mathit{Expression} \rangle \\
\langle \mathit{Expression} \rangle & ::= & \text{proc } (\langle \mathit{Identifier} \rangle) \ \{ \ \langle \mathit{Expression} \rangle \ \} \\
\langle \mathit{Expression} \rangle & ::= & (\langle \mathit{Expression} \rangle \ \langle \mathit{Expression} \rangle) \\
\langle \mathit{Expression} \rangle & ::= & \text{letrec } \langle \mathit{Identifier} \rangle(\langle \mathit{Identifier} \rangle) = \langle \mathit{Expression} \rangle \\
 & & \text{in } \langle \mathit{Expression} \rangle
\end{array}
$$

EXPLICIT-REFS: Concrete Syntax

$$\begin{array}{lcl}
\langle \mathit{Expression} \rangle & ::= & \texttt{newref}\ (\langle \mathit{Expression} \rangle) \\
\langle \mathit{Expression} \rangle & ::= & \texttt{deref}\ (\langle \mathit{Expression} \rangle) \\
\langle \mathit{Expression} \rangle & ::= & \texttt{setref}\ (\langle \mathit{Expression} \rangle) \\
\langle \mathit{Expression} \rangle & ::= & \texttt{begin}\ \langle \mathit{Expression} \rangle^{+(;)}\ \texttt{end}
\end{array}$$

We add four new operations

- `newref`: allocates a new location and returns a reference to it.
- `deref`: dereferences a reference: that is, it returns the contents of the location that the reference represents.
- `setref`: changes the contents of the location that the reference represents.
- `begin...end`, as before.

EXPLICIT-REFS: Abstract Syntax

```
type expr =
  | Var of string
  | Int of int
  | Add of expr*expr
  | Sub of expr*expr
  | Mul of expr*expr
  | Div of expr*expr
  | Let of string*expr*expr
  | IsZero of expr
  | ITE of expr*expr*expr
  | Proc of string*expr
  | App of expr*expr
  | Letrec of string*string*expr*expr
  | Set of string*expr
  | BeginEnd of expr list
  | NewRef of expr
  | DeRef of expr
  | SetRef of expr*expr
```

Example

```
let x = newref(newref(0))
in begin
     setref(deref(x), 11);
     deref(deref(x))
   end
```

```
AProg
 (Let ("x", NewRef (NewRef (Int 0)),
   BeginEnd [SetRef (DeRef (Var "x"), Int 11); DeRef (
       ↪ DeRef (Var "x"))]))
```

Specification – `newref`

```
eval_expr ρ σ0 (NewRef exp) =
    let  (val,σ1) = eval_expr ρ σ0 exp
    in (RefVal l, [l=val]σ1)
```

- l is fresh, that is $l \notin dom(\sigma_1)$
- `RefVal` is the tag that indicates that the result is a reference

Implementation – `newref`

```
eval_expr ρ σ0 (NewRef exp) =
   let  (val,σ1) = eval_expr ρ σ0 exp
   in (RefVal l, [l=val]σ1)
```

```
eval_expr (en:env) (e:expr) :exp_val =
  match e with
  ...
  | NewRef(e) ->
      RefVal(Store.new_ref g_store (eval_expr en e))
```

Specification – `deref` and `setref`

```
eval_expr ρ σ0 (DeRef exp) =
    let (RefVal l,σ1) = eval_expr ρ σ0 exp
    in (σ1(l),σ1)
```

```
eval_expr ρ σ0 (SetRef exp1 exp2) =
    let (RefVal l,σ1) = eval_expr ρ σ0 exp1
        (val,σ2) = eval_expr ρ σ1 exp2
    in (UnitVal, [l=val]σ2)
```

Implementation – deref

```
eval_expr ρ σ₀ (DeRef exp) =
    let  (RefVal l,σ₁) = eval_expr ρ σ₀ exp
    in (σ₁(l),σ₁)
```

```
eval_expr (en:env) (e:expr) :exp_val =
  match e with
  ...
  | DeRef(e) ->
    let v1 = eval_expr en e
    in Store.deref g_store (refVal_to_int v1)
```

Implementation – setref

```
eval_expr ρ σ0 (SetRef exp1 exp2) =
    let  (RefVal l,σ1) = eval_expr ρ σ0 exp1
         (val,σ2) = eval_expr ρ σ1 exp2
    in (UnitVal,[l=val]σ2)
```

```
eval_expr (en:env) (e:expr) :exp_val =
  match e with
  ...
  | SetRef(e1,e2) ->
    let v1=eval_expr en e1
    in let v2=eval_expr en e2
    in Store.set_ref g_store (refVal_to_int v1) v2;
    UnitVal
```

The Interpreter for EXPLICIT-REFS

- Code available in Canvas `Modules/Interpreters`
- Directory `explicit-refs`
- Compile with `ocamlbuild -use-menhir interp.ml`
- Make sure the `.ocamlinit` file is in the folder of your sources
- Run `utop`