CS 496: Extra Credit Assignment*
Due: 8 May 2021, 11:59pm

Contents

*v0.19

1 Introducing SOOL with an Example

This assignment asks you to implement an interpreter for a simple object-oriented language called SOOL. SOOL builds on IMPLICIT-REFS, to which support for lists has been added to be able to write more interesting examples.

A program in SOOL consists of a (possibly empty) list of class declarations and an expression, called the *main expression*, where evaluation starts. Figure 1 presents an example[1]. It consists of three class declarations named `c1`, `c2` and `c3`. Class `c2` is a subclass of `c1` and `c3` is a subclass of `c2`. Each class declaration contains a list of field declarations and a list of method declarations. For example, class `c1` has two fields, namely `x` and `y`, and three methods, namely `initialize()`, `m1()` and `m2()`. The `initialize()` method is called when an object instance of class `c1` is created; it sets the value of field `x` to 11 and field `y` to 12. The main expression of the example in Figure 1 is

```
let o3 = new c3() in send o3 m1(7,8)
```

This expression creates an object `o3` instance of the class `c3` via the expression `new c3()` and then sends it the message `m1(7,8)` via the expression `send o3 m1(7,8)`. The expressions `7` and `8` are the arguments to method `m1`.

Other examples are available in the file `test/test.ml`

2 The Syntax of SOOL

We briefly present the concrete syntax of SOOLand then discuss the abstract syntax. A program in SOOL consists of a (possibly empty) sequence of class declarations followed by a main expression:

⟨Program⟩ ::= ⟨Class_Decl⟩*⟨Expression⟩

Expressions are those of IMPLICIT-REFS together with the following new productions the first five of which involve list operations and the remaining four SOOL expressions proper:

⟨Expression⟩ ::= `list(`⟨Expression⟩*(,)`)`
⟨Expression⟩ ::= `hd(`⟨Expression⟩`)`
⟨Expression⟩ ::= `tl(`⟨Expression⟩`)`
⟨Expression⟩ ::= `empty?(`⟨Expression⟩`)`
⟨Expression⟩ ::= `cons(`⟨Expression⟩`,` ⟨Expression⟩`)`

⟨Expression⟩ ::= `new` ⟨Identifier⟩`(`⟨Expression⟩*(,)`)`
⟨Expression⟩ ::= `self`
⟨Expression⟩ ::= `send` ⟨Expression⟩⟨Identifier⟩`(`⟨Expression⟩*(,)`)`
⟨Expression⟩ ::= `super` ⟨Identifier⟩`(`⟨Expression⟩*(,)`)`

Class declarations have the following concrete syntax:

[1] This example is available as the file `src/ex1.sool` in the stub.

```
(* ex1.sool *)

(* class declarations *)
class c1 extends object {
 field x
 field y
 method initialize() {
  begin
   set x = 11;
   set y = 12
  end
  }
  method m1() { x+y }
  method m2() { send self m3() }
}

class c2 extends c1 {
 field y
 method initialize() {
  begin
   super initialize();
   set y = 22
  end
  }
  method m1(u,v) { x+y-v }
  method m3() { 7 }
}

class c3 extends c2 {
 field x
 field z
 method initialize() {
  begin
   super initialize();
   set x = 31;
   set z = 32
  end
  }
  method m3() { x+y+z }
}

(* main expression *)
let o3 = new c3()
in send o3 m1(7,8)
```

Figure 1: Example of a program in SOOL

⟨Class_Decl⟩ ::= `class` ⟨Identifier⟩ `extends` ⟨Identifier⟩ {⟨Field_Decl⟩* ⟨Method_Decl⟩*(;)}
⟨Field_Decl⟩ ::= `field` ⟨Identifier⟩
⟨Method_Decl⟩ ::= `method` ⟨Identifier⟩(⟨Identifier⟩*(,)) {⟨Expression⟩}

As for the abstract syntax, it is defined below:

```
type
  prog = AProg of (cdecl list)*expr
and
  expr =
  ...   (* the expressions of IMPLICIT-REFS *)
  | Self
  | Send of expr*string*expr list
  | Super of string*expr list
  | NewObject of string*expr list
  | Cons of expr*expr
  | Hd of expr
  | Tl of expr
  | IsEmpty of expr
  | List of expr list
and
  cdecl = Class of string*string*string list*mdecl list
and
  mdecl = Method of string*string list*expr
```

ast.ml

3 Trying Out the Parser and Interpreter in SOOL

3.1 Trying Out the Parser

There are two ways you can parse SOOL programs. You can either type them in as an argument to `parse`, as in the example below:

```
# parse "2+2";;
- : prog = AProg ([], Add (Int 2, Int 2))
```

utop

Or you can save them in a text file with extension `sool` and then use `parsef`. For example:

```
# parsef "ex1";;
- : prog =
AProg
 ([Class ("c1", "object", ["x"; "y"],
    [Method ("initialize", [], BeginEnd [Set ("x", Int 11); Set ("y", Int 12)]);
     Method ("m1", [], Add (Var "x", Var "y"));
     Method ("m2", [], Send (Self, "m3", []))]);
   Class ("c2", "c1", ["y"],
    [Method ("initialize", [],
      BeginEnd [Super ("initialize", []); Set ("y", Int 22)]);
```

```
     Method ("m1", ["u"; "v"], Sub (Add (Var "x", Var "y"), Var "v"));
     Method ("m3", [], Int 7)]);
   Class ("c3", "c2", ["x"; "z"],
    [Method ("initialize", [],
      BeginEnd [Super ("initialize", []); Set ("x", Int 31); Set ("z", Int 32)]);
     Method ("m3", [], Add (Add (Var "x", Var "y"), Var "z"))])],
 Let ("o3", NewObject ("c3", []), Send (Var "o3", "m1", [Int 7; Int
8])))
```

utop

3.2 Trying Out the Interpreter

There are two ways you can evaluate programs in SOOL. You can either type them in as an argument to `interp`, as in the example below:

```
# interp "2+2";;
- : exp_val Sool.Ds.result = Ok (NumVal 4)
```

utop

Or you can save them in a text file with extension `sool` and then use `interpf`. For example:

```
# interpf "ex1";;
- : exp_val Sool.Ds.result = Ok (NumVal 25)
```

utop

4 Evaluating Programs in SOOL

Recall from above that a program in SOOL is an expression of the form `AProg(cs,e)` where `cs` is a list of class declarations and `e` is the main expression. Evaluation of a program `AProg(cs,e)` takes place via the `eval_prog` function below:

```
let rec
  eval_expr : expr -> exp_val ea_result =
  ...
and
  eval_prog : prog -> exp_val ea_result =
    fun (AProg(cs,e)) ->
    initialize_class_env cs;    (* Step 1 *)
    eval_expr e                 (* Step 2 *)
```

This function performs two steps, Step 1 and Step 2, as may be seen above. Step 1 has been implemented for you already (the code for by `initialize_class_env` is supplied with the stub). Part of Step 2 has been implemented; your task is to complete the rest as outlined in Sec. 5. We next describe each of these steps in more detail below.

1. **Step 1: From class declarations to a class environment.** First the class declarations in `cs` are processed, producing a class environment. The aim is to have ready access not just to the fields and methods declared in a class, but also to all those it inherits. A class environment is a list of pairs:

```
type class_env = (string*class_decl) list
```

Each entry in this list consists of a pair whose first component is the name of the class and the second one is a *class declaration*. A class declaration is a tuple of type `string*string list*method_env` consisting of the name of the class, the list of the fields **visible** from that class and the list of methods **visible** from that class.

```
type method_decl = string list*Ast.expr*string*string list
type method_env = (string*method_decl) list
type class_decl = string*string list*method_env
```

The resulting class environment is placed in the global variable `g_class_env` of type `class_env ref` for future use. Thus `g_class_env` is a reference to an association list, that is, a list of pairs.

For the example from Fig. 1 the contents of `g_class_env` may be inspected as follows[2]. Please familiarize yourself with it since it will help you with the upcoming tasks.

```
# #print_length 2000;;
# interpf "ex1";;
- : exp_val Sool.Ds.result = Error "eval_expr: Not implemented: NewObj(c3,[])"
# !g_class_env;;
- : (string * class_decl) list =
[("c3",
  ("c2", ["x"; "z"; "y"; "x"; "y"],
   [("initialize",
     ([],
      BeginEnd [Super ("initialize", []); Set ("x", Int 31); Set ("z", Int 32)],
      "c2", ["x"; "z"; "y"; "x"; "y"]));
    ("m3",
     ([], Add (Add (Var "x", Var "y"), Var "z"), "c2",
      ["x"; "z"; "y"; "x"; "y"]));
    ("initialize",
     ([], BeginEnd [Super ("initialize", []); Set ("y", Int 22)], "c1",
      ["y"; "x"; "y"]));
    ("m1",
     (["u"; "v"], Sub (Add (Var "x", Var "y"), Var "v"), "c1", ["y"; "x"; "y"]));
    ("m3", ([], Int 7, "c1", ["y"; "x"; "y"]));
    ("initialize",
     ([], BeginEnd [Set ("x", Int 11); Set ("y", Int 12)], "object",
      ["x"; "y"]));
    ("m1", ([], Add (Var "x", Var "y"), "object", ["x"; "y"]));
    ("m2", ([], Send (Self, "m3", []), "object", ["x"; "y"]))]));
 ("c2",
  ("c1", ["y"; "x"; "y"],
   [("initialize",
     ([], BeginEnd [Super ("initialize", []); Set ("y", Int 22)], "c1",
```

[2]It is possible that the output is truncated by utop. The directive in utop `#print_length 2000;;` changes this to allow printing up to 2000 items.

```
      ["y"; "x"; "y"]));
   ("m1",
    (["u"; "v"], Sub (Add (Var "x", Var "y"), Var "v"), "c1", ["y"; "x"; "y"]));
   ("m3", ([], Int 7, "c1", ["y"; "x"; "y"]));
   ("initialize",
    ([], BeginEnd [Set ("x", Int 11); Set ("y", Int 12)], "object",
     ["x"; "y"]));
   ("m1", ([], Add (Var "x", Var "y"), "object", ["x"; "y"]));
   ("m2", ([], Send (Self, "m3", []), "object", ["x"; "y"]))]));
("c1",
 ("object", ["x"; "y"],
  [("initialize",
    ([], BeginEnd [Set ("x", Int 11); Set ("y", Int 12)], "object",
     ["x"; "y"]));
   ("m1", ([], Add (Var "x", Var "y"), "object", ["x"; "y"]));
   ("m2", ([], Send (Self, "m3", []), "object", ["x"; "y"]))]))]
```
utop

In particular, notice that the first entry in the list is of the form

```
("c3", ("c2", ["x"; "z"; "y"; "x"; "y"],...))
```

Here:

- `c3` is the name of the class
- `c2` is the name of the superclass of `c3`
- `["x"; "z"; "y"; "x"; "y"]` is the list of all the fields that are visible to `c3`, from left-to-right. **NOTE**: it includes the inherited fields.
- The ellipses `...` is a list of all the methods that are visible to `c3`. **NOTE**: it includes the inherited methods.

2. **Step 2: Evaluation of the main expression.** Second, we evaluate the main expression. This process consults `g_class_env` whenever it requires information from the class hierarchy. Evaluation takes place via the function `eval_expr`. Your task will be to complete some of the variants defining this function as explained in the next section.

5 Your Task

Implement the following variants of `eval_expr`:

```
let rec eval_expr : expr -> exp_val ea_result = fun e ->
  match e with
  ...
  | NewObject(c_name,args) -> failwith "implement"
  | Send(e,m_name,args) -> failwith "implement"
  | Self -> eval_expr (Var "_self")
  | Super(m_name,args) -> failwith "implement"
```

SOOL has two new expressed values:

```
type exp_val =
  ...
  | ObjectVal of string*env
  | StringVal of string
```
ds.ml

The use of strings will be explained later; we focus here on objects. Programs can now return objects. An object is represented as an expression `ObjectVal(c_name,env)`, where `c_name` is the class of the object and `env` is the value of its fields encoded as an environment. As an example, here is the object `o3` from the example in Fig. 1. The string `"c3"` is the class of the object. Notice that the environment has the fields higher-up in the hierarchy listed first. Also notice that, since SOOL is an extension of IMPLICIT-REFS, environments map variables to references (i.e. to `RefVal`s).

```
ObjectVal ("c3",
 ExtendEnv ("y", RefVal 4,
  ExtendEnv ("x", RefVal 3,
   ExtendEnv ("y", RefVal 2,
    ExtendEnv ("z", RefVal 1,
     ExtendEnv ("x", RefVal 0,
       EmptyEnv))))))
```

The contents of the store is as follows:

```
Store:
0->NumVal 31
1->NumVal 32
2->NumVal 22
3->NumVal 11
4->NumVal 12
5->ObjectVal(c3,(y,RefVal (4))(x,RefVal (3))(y,RefVal (2))
                (z,RefVal (1))(x,RefVal (0)))
6->StringVal c2
7->ObjectVal(c3,(y,RefVal (4))(x,RefVal (3))(y,RefVal (2))
                (z,RefVal (1))(x,RefVal (0)))
8->StringVal c1
9->ObjectVal(c3,(y,RefVal (4))(x,RefVal (3))(y,RefVal (2))
                (z,RefVal (1))(x,RefVal (0)))
10->StringVal object
```

Note: it is possible for the references in the environments in the ObjectVal's above to have been allocated in a different order. However, the order of the variables in those environments has to be the same as the one depicted above. For example, the environment cannot start with `"x"`.

Only the first five entries are relevant to the object. The rest is garbage. This garbage was generated when each of the `initialize` methods was called, upon evaluation of `new c3()`[3].

[3] In each of those calls an environment was set up, including special variables "_self" and "_super" which were mapped to locations 5 and 6 respectively, for the `initialize` method of `c3`, 7 and 8 for the `initialize` method of `c2` and 9 and 10 for the `initialize` method of `c1`.

5.1 Self

This is the easiest case. It has already been implemented for you. In SOOL the environment always contains at least two (reserved) variables "_self" and "_super". So the code for `Self` simply involves looking up the location for "_self" in the environment and then the contents of that location in the store. Of course, when methods are called using the `apply_method` function described below, the environment that includes these reserved variables has to be set up appropriately. In particular, "_self" will be mapped to an `ObjectVal` and "_super" to a `StringVal`.

5.2 NewObject

In the case of `NewObject(c_name,es)` a new object of class `c_name` should be created and should be initialized if there are any applicable `initialize` methods. We next describe the steps that should be followed. Along the way we indicate the helper functions you need to implement.

These are the steps you should follow in order to implement the `NewObject(c_name,args)` case.

1. Evaluate the arguments `args` producing a list of expressed values `evs`. These will be passed on to the initialization method, if there is one.

2. Lookup the class `c_name` in the class environment held in `g_class_env`. Do so through a helper function

   ```
   lookup_class : string -> class_env -> class_decl ea_result
   ```

   where `lookup_class c_name c_env` searches for the class `c_name` in the class environment `c_name`. This should either return an error `"lookup_class: class c_name not found"` if the class does not exist, or a triple `(super,fields,methods)` if it does. Recall from page 6 that `class_decl` is just defined to be a triple. Below we only use the `fields` and `methods` components.

3. Create an environment for the newly created object using `fields` and call it, say, `env`. To do so implement a helper function:

   ```
   new_env : string list -> env ea_result
   ```

   This function creates the environment and also allocates a dummy value, say `NumVal 0`, for all the entries in the store. More precisely, given a list of variables `ids`, the expression `new_env ids` returns an `ea_result` that ignores its environment and builds a new one whose domain is `ids`. For example,

```
# new_env ["x";"y";"z"] EmptyEnv;;
- : env Sool.Ds.result =
Ok
 (ExtendEnv ("z", RefVal 0,
   ExtendEnv ("y", RefVal 1, ExtendEnv ("x", RefVal 2, EmptyEnv))))
                                                                utop
```

and all of `RefVal 0`, `RefVal 1` and `RefVal 2` are initialized to `NumVal 0` in the store.

With the help of this function, your object can now be created as follows:

```
new_env fields >>= fun env ->
ObjectVal(c_name,env)
```

where `fields` are all the fields visible to class `c_name`. For example, the fields visible to class `c3` are `["x"; "z"; "y"; "x"; "y"]` and those visible to class `c2` are `["y"; "x"; "y"]`.

4. If there is an `initialize` method, then it should be called. In order to determine whether there is such a method, we simply use `List.assoc_opt "initialize" methods`. If `None` is returned, meaning the `initialize` method is not found, then you simply return `ObjectVal(c_name,env)`. If it is found, let us call it `m`, then you should first call it and then return `ObjectVal(c_name,env)`. For that you must use the helper function `apply_method` what is supplied with the stub. It should be called as follows in order to initialize the object:

   ```
   apply_method "initialize" self evs m
   ```

   where `self` denotes the object `ObjectVal(c_name,env)`.

Summary of helper methods you must implement:

```
lookup_class : string -> class_env -> class_decl ea_result
new_env : string list -> env ea_result
```

5.3 Send

In the case of `Send(e,m_name,es)` proceed as follows:

1. Evaluate `e` and make sure it is an object, say `ObjectVal (c_name,_)`. We'll only need the class name `c_name` of the object. Let us call this object `self`.
2. Evaluate the arguments `es` producing a list of expressed values `args`. These will be passed on to the method `m_name`, if there is one.
3. Look for and apply the method `m_name`. For that implement a helper function `lookup_method` whose type is:

   ```
   string -> string -> ((string*class_decl) list) -> method_decl option
   ```

   A typical call to this function will look like `lookup_method c_name m_name !g_class_env`. It should look for method `m_name` in the class environment `!g_class_env` in the class `c_name`. Note that given then way the class environment has been defined, if `c_name` is found then it there is not need to search its super classes since the entry for `c_name` already has all visible methods as described in Sec. 4.

   ```
   (match lookup_method c_name m_name !g_class_env with
   | None -> error "Method not found"
   | Some m -> apply_method m_name self args m)
   ```

Summary of helper methods you must implement:

```
lookup_method:string -> string -> ((string*class_decl) list) -> method_decl option
```

5.4 Super

In the case of `Super(m_name,es)` you proceed as follows.

1. Evaluate the arguments `es` producing a list of expressed values `args`. These will be passed on to the method `m_name`, if there is one.

2. Lookup who the super class is in the current environment and make sure it is a string. You can do so as follows:

```
eval_expr (Var "_super") >>=
string_of_stringVal >>= fun c_name ->
...
```

3. Lookup who self is in the current environment. You don't need to check that it is an object since we put it there ourselves. You can do so as follows:

```
eval_expr (Var "_self") >>= fun self ->
...
```

4. Finally, you lookup the method `m_name` and then apply it using `apply_method` as follows:

```
match lookup_method c_name m_name !g_class_env with
    | None -> error "Method not found"
    | Some m -> apply_method m_name self args m
```

Summary of helper methods you must implement:
None

6 Submission Instructions

Type `make zip` in the root folder of your stub. This will create a file `SOOL.zip`. Submit this file.