CS 496: HW6 (Optional)
Due: 11 December 2022, 11:59pm

Contents

1 Introducing TSOOL with an Example

This assignment asks you to extend an interpreter for a simple typed object-oriented language called TSOOL. TSOOL builds on IMPLICIT-REFS.

A program in TSOOL consists of a (possibly empty) list of interface and class declarations and an expression, called the *main expression*, where evaluation starts. Figure 1 presents an example[1]. It consists of one interface declaration `tree` and two class declarations named `interior_node`, and `leaf_node`. Both classes implement the `tree` interface. An interface declaration consists of a list of abstract method declarations: name of a method, return type of the method and the types of the formal parameter of the method. These are abstract since no code is provided for them. Class declarations consist of a sequence of field declarations and a sequence of method declarations. For example, class `interior_node` has two fields, namely `left` and `right`, and five methods, namely `initialize(l:tree, r:tree)`, `getleft()`, `getright()`, `sum()`, and `equal(t:tree)`. The `initialize(l:tree, r:tree)` method is called when an object instance of class `interior_node` is created; it sets the values of fields `left` and `right`. The main expression of the example in Figure 1 is

```
let o1 = new interior_node (
               new interior_node (
                     new leaf_node(3),
                     new leaf_node(4)),
               new leaf_node(5))
in list(send o1 sum(),if send o1 equal(o1) then 100 else 200)
```

This expression creates an object `o1` instance of the class `interior_node`. It then returns a list whose first component is the sum of all the nodes in `o1` and whose second component is `100` since `o1` is equal to itself. An explanation on how to run this example will be given below.

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

2 The Syntax of TSOOL

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

$$\langle \text{Program} \rangle \quad ::= \quad \langle \text{Iface\_or\_Class\_Decl} \rangle^* \langle \text{Expression} \rangle$$

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

[1]This example is available as the file `src/ex2.sool` in the stub.

```
(* ex2.sool *)

(* interface and class declarations *)

interface tree {
   method int sum ()
   method bool equal (t:tree)
}

class interior_node extends object implements tree {
   field tree left
   field tree right
   method unit initialize(l:tree, r:tree) {
     begin
      set left = l;
      set right = r
     end
   }
   method tree getleft () { left }
   method tree getright () { right }
   method int sum () {send left sum() + send right sum() }
   method bool equal (t:tree) {
     if instanceof(t,interior_node)
     then if send left equal(send
                     cast(t,interior_node)
                     getleft())
      then send right equal(send
                    cast(t,interior_node)
                    getright())
      else zero?(1)
     else zero?(1)
     }
}

class leaf_node extends object implements tree {
   field int value
   method unit initialize (v:int) { set value = v }
   method int sum() { value }
   method int getvalue() { value }
   method bool equal (t:tree) {
     if instanceof(t,leaf_node)
     then zero?(value - send cast(t,leaf_node) getvalue())
     else zero?(1)
   }
}

(* main expression *)

let o2 = new leaf_node(3)
in let o1 = new interior_node ( new interior_node (
           o2,
           new leaf_node(4)),
         new leaf_node(5))
in list(send o1 sum(),
if send o1 equal(o2) then 100 else 200)
```

Figure 1: Example of a program in TSOOL

$$
\begin{array}{lcl}
\langle\text{Expression}\rangle & ::= & \texttt{list(}\langle\text{Expression}\rangle^{+(,)}\texttt{)} \\
\langle\text{Expression}\rangle & ::= & \texttt{hd(}\langle\text{Expression}\rangle\texttt{)} \\
\langle\text{Expression}\rangle & ::= & \texttt{tl(}\langle\text{Expression}\rangle\texttt{)} \\
\langle\text{Expression}\rangle & ::= & \texttt{empty?(}\langle\text{Expression}\rangle\texttt{)} \\
\langle\text{Expression}\rangle & ::= & \texttt{cons(}\langle\text{Expression}\rangle, \langle\text{Expression}\rangle\texttt{)} \\
 & & \\
\langle\text{Expression}\rangle & ::= & \texttt{new } \langle\text{Identifier}\rangle\texttt{(}\langle\text{Expression}\rangle^{*(,)}\texttt{)} \\
\langle\text{Expression}\rangle & ::= & \texttt{self} \\
\langle\text{Expression}\rangle & ::= & \texttt{send } \langle\text{Expression}\rangle\langle\text{Identifier}\rangle\texttt{(}\langle\text{Expression}\rangle^{*(,)}\texttt{)} \\
\langle\text{Expression}\rangle & ::= & \texttt{super } \langle\text{Identifier}\rangle\texttt{(}\langle\text{Expression}\rangle^{*(,)}\texttt{)} \\
\langle\text{Expression}\rangle & ::= & \texttt{instanceof?(}\langle\text{Expression}\rangle, \langle\text{Identifier}\rangle\texttt{)} \\
\langle\text{Expression}\rangle & ::= & \texttt{cast(}\langle\text{Expression}\rangle, \langle\text{Identifier}\rangle\texttt{)}
\end{array}
$$

The productions involving lists have been added to be able to facilitate some examples.

Class declarations have the following concrete syntax:

$$
\begin{array}{lcl}
\langle\text{Iface\_or\_Class\_Decl}\rangle & ::= & \langle\text{Iface\_Decl}\rangle \mid \langle\text{Class\_Decl}\rangle \\
\langle\text{Class\_Decl}\rangle & ::= & \texttt{class}\ \langle\text{Identifier}\rangle\ \texttt{extends}\ \langle\text{Identifier}\rangle\ [\texttt{implements}\ \langle\text{Identifier}\rangle]\ \{\langle\text{Field\_Decl}\rangle^{*}\ \langle\text{Method\_Dec} \\
\langle\text{Iface\_Decl}\rangle & ::= & \texttt{interface}\ \langle\text{Identifier}\rangle\ \{\langle\text{Field\_Decl}\rangle^{*}\ \langle\text{Abs\_Method\_Decl}\rangle^{*}\} \\
\langle\text{Field\_Decl}\rangle & ::= & \texttt{field}\ \langle\text{Type}\rangle\ \langle\text{Identifier}\rangle \\
\langle\text{Method\_Decl}\rangle & ::= & \texttt{method}\ \langle\text{Type}\rangle\ \langle\text{Identifier}\rangle\texttt{((}\langle\text{Identifier}\rangle : \langle\text{Type}\rangle\texttt{)}^{*(,)}\texttt{)}\ \{\langle\text{Expression}\rangle\} \\
\langle\text{Abs\_Method\_Decl}\rangle & ::= & \texttt{method}\ \langle\text{Type}\rangle\ \langle\text{Identifier}\rangle\texttt{((}\langle\text{Identifier}\rangle : \langle\text{Type}\rangle\texttt{)}^{*(,)}\texttt{)}
\end{array}
$$

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
  | IsInstanceOf of expr*string
  | Cast of expr*string
and
  cdecl =
  | Class of string*string*string option*(string*texpr option) list*mdecl list
  | Interface of string*abs_mdecl list
and
  mdecl = Method of string*texpr option*(string*texpr option) list*expr
and
  abs_mdecl = MethodAbs of string*texpr*(string*texpr option) list
```

```
and
  texpr =
  | UserType of string
  | IntType
  | BoolType
  | UnitType
  | FuncType of texpr*texpr
  | RefType of texpr
  | ListType of texpr
```
ast.ml

3 Trying Out the Parser and Interpreter in TSOOL

3.1 Trying Out the Parser

There are two ways you can parse TSOOL 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
 ([Interface ("tree",
    [MethodAbs ("sum", IntType, []);
     MethodAbs ("equal", BoolType, [("t", Some (UserType "tree"))])]);
   Class ("interior_node", "object", Some "tree",
    [("left", Some (UserType "tree")); ("right", Some (UserType "tree"))],
    [Method ("initialize", Some UnitType,
      [("l", Some (UserType "tree")); ("r", Some (UserType "tree"))],
      BeginEnd [Set ("left", Var "l"); Set ("right", Var "r")]);
     Method ("getleft", Some (UserType "tree"), [], Var "left");
     Method ("getright", Some (UserType "tree"), [], Var "right");
     Method ("sum", Some IntType, [],
      Add (Send (Var "left", "sum", []), Send (Var "right", "sum", [])));
     Method ("equal", Some BoolType, [("t", Some (UserType "tree"))],
      ITE (IsInstanceOf (Var "t", "interior_node"),
       ITE
        (Send (Var "left", "equal",
          [Send (Cast (Var "t", "interior_node"), "getleft", [])]),
        Send (Var "right", "equal",
         [Send (Cast (Var "t", "interior_node"), "getright", [])]),
        IsZero (Int 1)),
       IsZero (Int 1)))]);
   Class ("leaf_node", "object", Some "tree", [("value", Some IntType)],
    [Method ("initialize", Some UnitType, [("v", Some IntType)],
```

```
      Set ("value", Var "v"));
     Method ("sum", Some IntType, [], Var "value");
     Method ("getvalue", Some IntType, [], Var "value");
     Method ("equal", Some BoolType, [("t", Some (UserType "tree"))],
      ITE (IsInstanceOf (Var "t", "leaf_node"),
       IsZero
        (Sub (Var "value", Send (Cast (Var "t", "leaf_node"), "getvalue", []))),
       IsZero (Int 1)))])],
 Let ("o1",
  NewObject ("interior_node",
   [NewObject ("interior_node",
     [NewObject ("leaf_node", [Int 3]); NewObject ("leaf_node", [Int 4])]);
    NewObject ("leaf_node", [Int 5])]),
  List
   [Send (Var "o1", "sum", []);
    ITE (Send (Var "o1", "equal", [Var "o1"]), Int 100, Int 200)]))
```
utop

3.2 Trying Out the Interpreter

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

```
# interp "2+2";;
- : exp_val Tsool.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 "ex2";;
- :  : exp_val Tsool.ReM.result = Tsool.ReM.Ok (ListVal [NumVal 12; NumVal 100])
```
utop

4 Evaluating Programs in TSOOL

This assignment asks you to complete the implementation of an interpreter for TSOOL. Type information will play no role at all in the process; in particular, interfaces are irrelevant to evaluation.

Recall from above that a program in TSOOL is an expression of the form `AProg(cs,e)` where `cs` is a list of interface and 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 *)
```
interp.ml

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. 6. 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. Interfaces are discarded since they are not needed for evaluation, as mentioned above. 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;;
   - : class_env =
   [("leaf_node",
     ("object", ["value"],
      [("initialize", (["v"], Set ("value", Var "v"), "object", ["value"]));
       ("sum", ([], Var "value", "object", ["value"]));
       ("getvalue", ([], Var "value", "object", ["value"]));
       ("equal",
        (["t"],
         ITE (IsInstanceOf (Var "t", "leaf_node"),
          IsZero
           (Sub (Var "value", Send (Cast (Var "t", "leaf_node"), "getvalue", []))),
          IsZero (Int 1)),
   ```

[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.

```
      "object", ["value"]))]));
 ("interior_node",
  ("object", ["left"; "right"],
   [("initialize",
     (["l"; "r"], BeginEnd [Set ("left", Var "l"); Set ("right", Var "r")],
      "object", ["left"; "right"]));
    ("getleft", ([], Var "left", "object", ["left"; "right"]));
    ("getright", ([], Var "right", "object", ["left"; "right"]));
    ("sum",
     ([], Add (Send (Var "left", "sum", []), Send (Var "right", "sum", [])),
      "object", ["left"; "right"]));
    ("equal",
     (["t"],
      ITE (IsInstanceOf (Var "t", "interior_node"),
       ITE
        (Send (Var "left", "equal",
          [Send (Cast (Var "t", "interior_node"), "getleft", [])]),
        Send (Var "right", "equal",
         [Send (Cast (Var "t", "interior_node"), "getright", [])]),
        IsZero (Int 1)),
       IsZero (Int 1)),
      "object", ["left"; "right"]))]))]
```
utop

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

```
[("leaf_node",("object", ["value"],...))
```

Here:

- `leaf_node` is the name of the class
- `object` is the name of the superclass of `leaf_node`
- `["value""]` is the list of all the fields that are visible to `leaf_node`, from left-to-right. **NOTE**: if `object` had fields, this list would include those inherited fields.
- The ellipses ... is a list of all the methods that are visible to `lead_node`. **NOTE**: if `object` had fields, it would include 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 Extending Expressed Values with Objects

Programs can now return objects. Therefore, TSOOL 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. 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 `o1` from the example in Fig. 1. The string `"interior_node"` is the class of the object. If `"object"` had fields, they would be listed here too, at the beginning of the list. Also notice that, since TSOOL is an extension of IMPLICIT-REFS, environments map variables to references (i.e. to `RefVal`s).

```
- : exp_val Tsool.ReM.result =
Tsool.ReM.Ok
 (ObjectVal ("interior_node",
   ExtendEnv ("right", RefVal 18,
   ExtendEnv ("left", RefVal 19,
   EmptyEnv))))
```

The contents of the store is as follows:

```
Store:
0->NumVal 3,
1->ObjectVal(leaf_node,(value,RefVal (0))),
2->NumVal 3,
3->StringVal object,
4->NumVal 4,
5->ObjectVal(leaf_node,(value,RefVal (4))),
6->NumVal 4,
7->StringVal object,
8->ObjectVal(leaf_node,(value,RefVal (4))),
9->ObjectVal(leaf_node,(value,RefVal (0))),
10->ObjectVal(interior_node,(right,RefVal (8))(left,RefVal (9))),
11->ObjectVal(leaf_node,(value,RefVal (0))),
12->ObjectVal(leaf_node,(value,RefVal (4))),
13->StringVal object,
14->NumVal 5,
15->ObjectVal(leaf_node,(value,RefVal (14))),
16->NumVal 5,
17->StringVal object,
18->ObjectVal(leaf_node,(value,RefVal (14))),
19->ObjectVal(interior_node,(right,RefVal (8))(left,RefVal (9))),
20->ObjectVal(interior_node,(right,RefVal (18))(left,RefVal (19))),
21->ObjectVal(interior_node,(right,RefVal (8))(left,RefVal (9))),
22->ObjectVal(leaf_node,(value,RefVal (14))),
23->StringVal object,
24->ObjectVal(interior_node,(right,RefVal (18))(left,RefVal (19)))
```

Some of these are garbage resulting from function calls. For example, when the `initialize` methods are called.[3].

6 Your Task

Implement the following variants of `eval_expr`:

```
let rec eval_expr : expr -> exp_val ea_result =
  fun e ->
  match e with
  ...
  | IsInstanceOf(e,id) ->  failwith "implement"
  | Cast(e,id) ->
```

A description of each of these is provided below.

6.1 IsInstanceOf

An expression `IsInstanceOf(e,id)` should evaluate `e`, make sure it is an object and then check if the class of that object is a subclass of the class `id`. It should thus return a boolean (i.e. a `BoolVal`). You will need to implement a helper function

```
is_subclass : string -> string -> class_env -> exp_val ea_result
```

such that `is_subclass c1 c2 cenv` determines, returning a boolean (i.e. `BoolVal`), whether `c1` is a subclass of `c2`. If `c2` does not exist it should return an error with error message `"is_subclass: class c2 not found"`.

6.2 Cast

An expression `Cast(e,id)` evaluates just like `IsInstanceOf(e,id)` except that it returns the object itself resulting from evaluating `e`.

7 Submission Instructions

Make sure all your code (helper functions) is in the file `interp.ml`. Submit this file.

NOTE: This assignment is individual.

[3] In each of those calls an environment was set up, including special variables "_self" and "_super" which were mapped to fresh locations.