CS383 Course Project Report
SimPL Interpreter

November 26, 2018

1 Overview

The objective of this course project is to achieve a interpreter for simPL, which is a simplified dialect of ML. The lexical and syntactic analyzer can be provided in the skeleton architecture.
The implementation is based on skeleton architecture, including the parser, interpreter and typing three parts. To implement the basic part, it's needed to complete all the java TODO functions in three parts. According to the semantic rules, they can be completed realtively easy in similiar ways.
For the bonus, there are six features optional. The mutually recursive combinator (Y^*) and infinite streams needs new keywords and corresponding rules, for lack of experience of using parser of java-cup and time limited, both are unconsummated. The polymorphic type is achieved by implement TypeVar class. The garbage collection is implemented with mark and sweep algorithm. The tail recursion optimization concerns about the recursive unfolding, and the transfrom to iterative loop needs to take many conditions into account, with time limitation, it was not implemented. The lazy evaluation is achieved by a history buffer table.

2 Implementation

The whole interpreter starts from Interpreter.java, where the .spl files get parsed and expressions get evaluated. The three statements in run function are corresponding to what we need to implement: Expr program, program.typecheck() and program.eval().

2.1 Typing

The implementation of typing is based on type inference rules. All the type classes inherit the Expr class and all the TODOs in this part are to achieve the isEqualityType, unify, contains and replace operations. In the process of typecheck, when there is new information about certain type, which is a TypeVar object there, we will use unify to bind

this new type to former type and create new substitution. Besides, the isEqualityType tells whether this type can be compared.

- Type.java
  Base class with abstract functions.

- TypeVar.java
  Used as a temp type class before substition, so it also need to implement the functions.

- Substitution.java
  Core class of typecheck. After there is more bind from TypeVar to certain Type, it will store the type binding information. It includes three subclasses:Identity, Replace, Compose. Identity is a static member,it binds every type with exactly the same type. Replace stores the bind for only one typevar and its actual type. Compose is used to compose two substitution into one. In each subclass there is a corresponding apply function to do replacement in a type using the binding information in substitution.

```
    private static final class Identity extends Substitution {
        public Type apply(Type t) {
            return t;
        }
    }

    private static final class Replace extends Substitution {
        private TypeVar a;
        private Type t;

        public Replace(TypeVar a, Type t) {
            this.a = a;
            this.t = t;
        }

        public Type apply(Type b) {
            return b.replace(a, t);
        }
    }

    private static final class Compose extends Substitution {
        private Substitution f, g;

        public Compose(Substitution f, Substitution g) {
            this.f = f;
            this.g = g;
        }

        public Type apply(Type t) {
            return f.apply(g.apply(t));
        }
    }
```

Substitution

- *Type.java
  Similar with TypeVar class, it need to implement the functions accorinding to its type.

```
    @Override
    public boolean isEqualityType() {
        return true;
    }

    @Override
    public Substitution unify(Type t) throws TypeError {
        if(t instanceof RefType)
            return this.t.unify(((RefType) t).t);
        else if(t instanceof TypeVar)
            return t.unify(this);
        else {
            throw new TypeMismatchError();
        }
    }

    @Override
    public boolean contains(TypeVar tv) {
        return t.contains(tv);
    }

    @Override
    public Type replace(TypeVar a, Type t) {
        return new RefType(t.replace(a, t));
    }
```

RefType

2.2 Interpreter

In this part, corresponding to package simple.interpreter, we implement the essential supports for interpreter, such as value, state, memory and environment.

- Value.java
  Just like Type.java, all value class inherit this class and achieve the equal operation. It includes NIL and UNIT, which are static method to acquire nil value or unit.

- Env.java
  It stores all the binding of symbols and values for every given variable. Every Env (except empty one) is regarded as smaller one associated with a new x-v binding. It has get and clone operation. We can use get to find value for given variable symbol. But if variable not exists, it will return null. And we can use clone to get a copy of this environment.

```
    public Value get(Symbol y) {
        // if y==x, return v, else find the value in E
        if(y.toString().equals(x.toString())){
            return v;
        }
        return E.get(y);
    }

    public Env clone() {
        return new Env(E,x,v);
    }
```

Env

- State.java
  It's a unit of Env and Mem. It keeps changing, so it use a int p as a conference of now.

- Mem.java
  Mem is a binding with pointer and its stuff, which can be realized in a hashmap.

- Basic operations
  There are 7 built-in function to be implemented, which are subclass of FunValue. Besides the contructor, they also need to implement the typecheck and eval operation.

```

public class iszero extends FunValue {

    public iszero() {
        super(Env.empty,Symbol.symbol("x"),new Expr(){

            @Override
            public TypeResult typecheck(TypeEnv E) throws TypeError {
                return null;
            }

            @Override
            public Value eval(State s) throws RuntimeError {
                IntValue iv = (IntValue)(s.E.get(Symbol.symbol("x")));
                return new BoolValue(iv.n==0);
            }

        });
    }
}
```

iszero

- *Value.java
  inherit Value class, it need to implement the equal function accorinding to its type.

2.3 Expressions

With all the types and values ready we can achieve the typecheck and evaluation of each expression. All the expression classes inherits the Expr class and all the TODOs in this part are to achieve the typecheck and eval operations of corresponding expression class.

- App.java
  It need to apply l on r, so it need eval l and r first, then create a FunValue.
- Name.java
  It get the value from the environment. For recursion, it construct a new RecValue to continue the recursion.
- *.java
  For other expressions, the implementations of typecheck and eval are silmilar.

```
    @Override
    public TypeResult typecheck(TypeEnv E) throws TypeError {
        TypeResult typeResult = e.typecheck(E);
        Substitution s = typeResult.s;

        Type type = typeResult.t;
        type = s.apply(type);

        return TypeResult.of(s,new RefType(type));
    }

    @Override
    public Value eval(State s) throws RuntimeError {
        int pointer = s.get_pointer();
        Value v = e.eval(s);
        //put pointer as a key for value v
        s.M.put(pointer, v);
        return new RefValue(pointer);
    }
```

Ref.java

3 Features

3.1 Polymorphic type

Actually, with class TypeVar implemented, the simPL can be regarded as polymorphic language, or rather, parametric polymorphism.

```
(* using polymorphic types *)
rec map =>
  fn f => fn l =>
    if l=nil
    then nil
    else (f (hd l))::(map f (tl l))
```

map.spl

And we can see the result:

```
doc/examples/map.spl
((tv53 -> tv60) -> (tv53 list -> tv60 list))
fun
```

Result

3.2 Garbage collection (of ref cells)

There we use mark and sweep algorithm to achieve garbage collection. So at first, to represent the value and mark flag of memory, we create the MemUse class:

```
public class MemUse{
    public Value value;
    public boolean mark;
    public MemUse(Value value) {
        this.value = value;
        mark = false;
    }
}
```

MemUse.java

For the memory can only be allocated to ref, so it is easy to mark by check the ref in current environment.

```
    public void mark(Env E){
        if(E==null)
            return;
        Symbol x = E.get_symbol();
        Value v = E.get_value();
        if(x !=null && v instanceof RefValue){
            int pointer = ((RefValue)v).p;
            this.M.mark(pointer);
        }
    }
```

State.java

As for sweep, we have to rewrite mem to refresh allocated information. For the collected free spaces's pointers, there we use a stack freelist to store .

```

   public void sweep(){
      for(Integer p:alloList){
          MemUse m = memMap.get(p);
          if(m.mark) {
              demark(p);
```

```
            }else{
                tmp.push(p);
            }
        }
        while(!tmp.isEmpty()){
            Integer p = tmp.pop();
            delete(p);
        }
    }
```

Mem.java

There we run a test .spl file to test the garbage collection:

```
let y=ref 0 in
  let y=ref 1 in
    let y=ref 2 in
      let x=ref 3 in !x
      end
    end
  end
end
```

mem.spl

And we get the result:

```
doc/examples/mem.spl
int
3
```

Result

Obviously,Mem[2] is marked and the space of Mem[0] and Mem[1] is collected.

3.3 Lazy evaluation

Lazy evaluation is an evaluation strategy which delays the evaluation of an expression until its value is needed and also avoids repeated evaluations. So there must be a structure to store the expressions and also a buffer to store the latest evaluations and results.
To store the expression, we define the FuncEntry class, where fun is this function,and para is its value.

```
public class FuncEntry {

        Value fun;
        Value para;
        Value result;

        public FuncEntry(Value expr,Value p) {
            fun = expr;
            para = p;
```

```
        }

        public boolean equal(FuncEntry f){
            return para.equals(f.para)&& fun.equals(f.fun);
        }

        public void set_result(Value r){
            this.result = r;
        }
}
```

FuncEntry.java

To store the evaluations, we define the LazyTable, which is FuncEntry stack only store recursive function's value done:

```
public class LazyTable {
    private Stack<FuncEntry> table ;

        public LazyTable() {
            table = new Stack<FuncEntry>();
        }

        public Value get_result(FuncEntry fe){
            for (FuncEntry f:table){
                if (f.equal(fe)){
                        //System.out.println("found previos result"+fe.fun+
                            fe.para);
                        return f.result;
                }
            }
            return null;
        }

        public void put(FuncEntry fe,Value result) {
            fe.set_result(result);
            table.push(fe);
            clear();
        }

        public void clear(){
            if(table.size()>200){
                while(table.size()>200){
                    table.pop();
                }
            }
        }


}
```

LazyTable.java

We use fibonacci test it:

```
let plus = rec p =>
      fn x => fn y => if iszero x then y else p (pred x) (succ y)
in
  let fibonacci = rec f =>
  fn n => if iszero n then
      0
    else if iszero (pred n) then
      1
    else
      plus (f (pred n)) (f (pred (pred n)))
  in
    fibonacci 6
  end
end
```

pcf.fibonacci.spl

Then we can get the result:

```
doc/examples/pcf.fibonacci.spl
int
Reuse stored evaluations: fun 1
Reuse stored evaluations: fun 1
Reuse stored evaluations: fun 0
Reuse stored evaluations: fun 2
Reuse stored evaluations: fun 1
Reuse stored evaluations: fun 0
Reuse stored evaluations: fun 3
Reuse stored evaluations: fun 2
Reuse stored evaluations: fun 1
Reuse stored evaluations: fun 0
Reuse stored evaluations: fun 4
Reuse stored evaluations: fun 3
Reuse stored evaluations: fun 2
Reuse stored evaluations: fun 1
Reuse stored evaluations: fun 0
8
```

Result

We can see that the evaluation is delayed, and after one evaluation, the next evaluation will just get from the table.

4 Results

For the example .spl files we can get these resules:

```
doc/examples/true.spl
type error
doc/examples/plus.spl
int
3
```

```
doc/examples/factorial.spl
int
24
doc/examples/gcd1.spl
int
1029
doc/examples/gcd2.spl
int
1029
doc/examples/max.spl
int
2
doc/examples/mem.spl
int
3
doc/examples/sum.spl
int
6
doc/examples/map.spl
((tv53 -> tv60) -> (tv53 list -> tv60 list))
fun
doc/examples/pcf.sum.spl
(int -> (int -> int))
fun
doc/examples/pcf.even.spl
(int -> bool)
fun
doc/examples/pcf.minus.spl
int
46
doc/examples/pcf.factorial.spl
int
720
doc/examples/pcf.fibonacci.spl
int
6765
doc/examples/pcf.twice.spl
type error
doc/examples/pcf.lists.spl
type error
```

Result

5 Summary

It's really benefit a lot from this project. Although the implementations are similar, the attemptation at the begining and the process of debug were quite tough. Only after the success run of the first expression, then all gets easier. When it comes to bonus part, the implementation gets harder again. With a lot of trying, there still some features unfinised, which is a small regret. Above all this project gives me a clear understanding

on how the interpreter run. Finally really appreciate the instructions from teacher and the helps from TA, really thanks a lot!