Interpreter for SimPL
CS383 Project

Yaoming Zhu

5140219355

Contents

1 Overview

1.1 Objective

Implement an interpreter for the programming language SimPL. SimPL is a simplified dialect of ML, which can be used for both functional and imperative programming. Lexical and syntactic analyzer have already been provided in the *skeleton.zip* file.

1.2 Skeleton Architecture

Parser

Majority of work in this part has been provided.

The node for the AST(abstract syntax tree) should be implemented, which means type checking process and evaluation process will be done here.

Interpreter

Main function is in this part at Interpreter.java, where it will start interpret some given *spl* codes.

Type value classes is implemented in this part. It will help to evaluate the variables or values.

Built-in function like *iszreo* is defied in this part, as subclasses of *FunValue*.

In addition, *class Env* and *class InitialState* are also implemented here.

The details are in the part 2.

Typing

Type inference and substitution are implemented in the part, which help to type check.

A special type class *TypeVar* will implement polymorphism, while *class DefaultTypeEnv* will initialize type environment.

2 Interpreter

Let's first begin with interpreter part.

2.1 Value

Value is the super class for all value class.

```java
public abstract class Value {

    public static final Value NIL = new NilValue();
    public static final Value UNIT = new UnitValue();

    public abstract boolean equals(Object other);
}
```

Value.java

NIL and *UNIT* are static method to acquire nil value or unit. nil value is used to represent nil-list, and unit is result of imperative operations (such as assign).
Here abstract method "equal" needs to be implemented. It is not very complex: for each type value class , there is a final member variables. Therefore we just need to tell whether all the member variables of "this" class equals to "other" class.Take *BoolValue* as example:

```java
public class BoolValue extends Value {
    public final boolean b;

    @Override
    public boolean equals(Object other) {
        return this.b == ((BoolValue)other).b;
    }
}
```

BoolValue.java (parts)

For *NIL* and *UNIT*, tell by this way:

```
    @Override
    public boolean equals(Object other) {
        return (other instanceof  UnitValue);
    }
```

equal method in UnitValue.java

Note. The Specification requires:
If the result is a list, output list@ followed by its length.
However, no method has been provided to get the length. So it will be implemented as follows:

```
    public String toString() {
        return "list@" + this.getLength();
    }

    private int getLength(){
        if(this.v2 instanceof NilValue){
            return 0;
        }
        return ((ConsValue)this.v2).getLength() + 1;
    }
```

getLength and toString ConsValue.java

2.2 Env

Class *Env* stands for environment, by Project Specification:

$$\mathbf{Env} = \mathbf{Var} \rightarrow \mathbf{Val} \cup \mathbf{Rec}$$

That is for every given variable, its symbol and value should be associated in the environment. It has 3 member variable.

```
    private final Env E;
    private final Symbol x;
    private final Value v;
```

member variables in Env.java

Every *Env* (except *empty* one) is regarded as smaller one associated with a new *x*-*v* binding. The *empty Env* has no variable or value:

```
    public static Env empty = new Env() {
        public Value get(Symbol y) {
            return null;
        }

        public Env clone() {
            return this;
        }
    };
```

empty Env in Env.java

get and *clone* method is implemented here. *get* finds value for given variable symbol or return null if variable not exists. It will look up symbol in *Env* "layer by layer" and return outermost one:

```
    public Value get(Symbol y) {
        if(this.x.toString().equals(y.toString())){
            return this.v;
        }
        return this.E.get(y);
    }
```

get in Env.java

clone return a new same *Env* as this one.

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

clone in Env.java

2.3 Built-In Function(I)

There are 7 built-in function to be implemented, here I will take *iszero* as example:
iszero is subclass of *FunValue*, whose constructor will call *super* method, where *Env* is initialized to be *empty*, and class *Symbol* helps to associate any variable(denoted by "x") with the unique symbol.

```java
    public iszero() {
        super(Env.empty, Symbol.symbol("x"), new Expr() {
            @Override
            public TypeResult typecheck(TypeEnv E) throws
                TypeError {
                /* details in the AST parts */
            }
            @Override
            public BoolValue eval(State s) throws RuntimeError {
                /* details in the AST parts */
            }
        });
    }
```

iszero.java (parts)

2.4 InitialState

In class *InitialState*, here let's start with *empty Env*, and then add built-in function name associated with its value.

```java
public class InitialState extends State {
    public InitialState() {
        super(initialEnv(Env.empty), new Mem(), new Int(0));
    }
    private static Env initialEnv(Env E) {
        E = new Env(E, Symbol.symbol("iszero"),new iszero());
        E = new Env(E,Symbol.symbol("pred"),new pred());
        /* other built-in functions */
        return E;
    }
}
```

InitialState.java

3 Typing

Note. For convenience, in the part I will call int, bool and unit as "***basic***" types; arrow, list, pair and ref as "***complex***" types.

3.1 Type

Type is the super class for all type class.

```java
public abstract class Type {
    public abstract boolean isEqualityType();
    public abstract Type replace(TypeVar a, Type t);
    public abstract boolean contains(TypeVar tv);
    public abstract Substitution unify(Type t) throws TypeError;
    public static final Type INT = new IntType();
    public static final Type BOOL = new BoolType();
    public static final Type UNIT = new UnitType();
}
```

Type.java

Method *isEqualityType* tells if a variable is equality type. From Project Specification, they are

$$\begin{array}{lcl} \textbf{Equality type } \alpha & ::= & \texttt{int} \\ & \mid & \texttt{bool} \\ & \mid & \alpha\ \texttt{list} \\ & \mid & t\ \texttt{ref} \\ & \mid & \alpha \times \alpha \end{array}$$

Figure 1: equality type

Method *replace* will replace type variable(*TypeVar*, or other types contains *TypeVar*) with its "true" type name (if it is not polymorphic).
Method *contains* tells whether a complex type contains certain *TypeVar* in it.
Method *unify* unify two types. For instance, unify *int* with *TypeVar1*, and returns a substitution.
Three final method correspond to three basic types.

3.1.1 TypeVar

Class *TypeVar* stands for type variable. It has three member variables: *tvcnt* count the total number of type variables so far; *equalityType* stands for whether it is equality type; name is associated with *tvcnt*(tv1, tv2, ...) and can be regarded as identical ID.

In method *unify*, if two type have contain relation but not the same, by type inference, we know it will get type error of *TypeCircularityError*.

```
@Override
public Substitution unify(Type t) throws
    TypeCircularityError {
    if( t.equals(this)){
        return Substitution.IDENTITY;
    }
    if( t.contains(this) ){
        throw new TypeCircularityError();
    }
    return Substitution.of(this, t);
}

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

@Override
public Type replace(TypeVar a, Type t) {
    return (this.name.equals(a.name)) ? t : this;
}
```

some methods in TypeVar.java

3.1.2 "Basic" Type

Basic types cannot contain any type variable or be replaced. When it is unified with other types, call the *unify* method of that one.
Here take int type as example.

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

@Override
```

```java
    public boolean contains(TypeVar tv) {
        return false;
    }

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

some methods in IntType.java

3.1.3 "Complex" Type

Since complex types can be regarded as combination of "basic" ones, the methods can be implemented recursively. Take arrow type as instance:

```java
    @Override
    public Substitution unify(Type t) throws TypeError {
        if (t instanceof TypeVar) {
            return t.unify(this);
        }
        if (t instanceof ArrowType){
            return this.t1.unify(((ArrowType) t).t1).compose(
                this.t2.unify(((ArrowType) t).t2));
        }
        throw new TypeMismatchError();
    }

    @Override
    public boolean contains(TypeVar tv) {
        return this.t1.contains(tv) || this.t2.contains(tv);
    }

    @Override
    public Type replace(TypeVar a, Type t) {
        return new ArrowType(this.t1.replace(a, t), this.t2.
            replace(a, t));
    }
```

some methods in ArrowType.java

3.2 Substitution

The class *Substitution* has already been provided. I will give a brief review on some important methods.
IDENTITY returns most basic substitution (I).
compose compose two substitution together.($S_1 \circ S_2$)
apply helps to get "true" type of certain type in certain substitution.($S[tv1] = int$)

3.3 DefaultTypeEnv

Class *DefaultTypeEnv* gives default type environments (for built-in function). It is much like *InitialState* in part one.

```java
public class DefaultTypeEnv extends TypeEnv {
    private TypeEnv E;

    public DefaultTypeEnv() {
        E = TypeEnv.empty;
        TypeVar tV1 = new TypeVar(false);
        TypeVar tV2 = new TypeVar(false);
        TypeVar tV = new TypeVar(false);

        PairType pair = new PairType(tV1, tV2);
        ListType list = new ListType(tV);

        E = TypeEnv.of(E, Symbol.symbol("iszero"), new ArrowType
            (Type.INT, Type.BOOL));
        E = TypeEnv.of(E, Symbol.symbol("pred"), new ArrowType(
            Type.INT, Type.INT));
        /* other built-in functions */

    @Override
    public Type get(Symbol x) {
        return E.get(x);
    }
}
```

DefaultTypeEnv.java(parts)

3.4 TypeResult

Class *TypeResult* is already provided. It is used in type checking process to return substitution status and type of certain AST node.

4 AST of Parser

4.1 Expr

```java
public abstract class Expr {

    public abstract TypeResult typecheck(TypeEnv E) throws
        TypeError;

    public abstract Value eval(State s) throws RuntimeError;
}
```

Expr.java

Obviously, *typecheck* do type checking for the expression while *eval* evaluates the expression.
The semantic rules is given in the specification.

4.1.1 typecheck

Let's take *Let* as an example.
Method *isEqualityType* tells if a variable is equality type. From Project Specification, they are

$$\frac{\Gamma \vdash e_1 : t_1 \quad \Gamma[x : t_1] \vdash e_2 : t_2}{\Gamma \vdash \mathbf{let}\ x = e_1\ \mathbf{in}\ e_2\ \mathbf{end} : t_2} \qquad \text{(T-LET)}$$

Figure 2: typing rule of Let

It will be implemented in Java as follows:

```java
public class Let extends Expr {

    public Symbol x;
    public Expr e1, e2;

    @Override
    public TypeResult typecheck(TypeEnv E) throws TypeError {
        TypeResult res1 = this.e1.typecheck(E);
        TypeResult res2 = this.e2.typecheck(TypeEnv.of(E, x,
            res1.t));

        Substitution sub = res1.s.compose(res2.s);

        return TypeResult.of(sub, sub.apply(res2.t));
    }
```

typecheck in Expr.java

Type environment Γ and $\Gamma[x : t_1]$ is implemented by *TypeEnv E* and *TypeEnv.of(E, x, res1.t)* respectively, where *res1.t* is the type of e_1.
The conclusion's substitution will be the composition of premises' substitutions, method *compose* from class *Substitution* is called to implement the process.
Finally, it will return the new substitution and the type of e_2 as *TypeResult*.

4.1.2 eval

Let's take *App* as an example.

$$\frac{E,M,p;e_1 \Downarrow M',p';(\mathtt{fun},E_1,x,e) \quad E,M',p';e_2 \Downarrow M'',p'';v_2 \quad E_1[x \mapsto v_2],M'',p'';e \Downarrow M''',p''';v}{E,M,p;e_1\ e_2 \Downarrow M''',p''',v} \quad \text{(E-App)}$$

Figure 3: Evaluation rule of App

It will be implemented in Java as follows:

```java
public class App extends BinaryExpr {

    @Override
    public Value eval(State s) throws RuntimeError {
        FunValue app = (FunValue) l.eval(s);
        Value val = r.eval(s);
        Env newEnv = new Env(app.E,app.x, val);
        State newState = State.of(newEnv, s.M, s.p);
        return app.e.eval(newState);
    }
}
```

eval in App.java

The left expression will be evaluated as *FunValue*. The current environment will be updated with new symbol-value bindings. We apply the new environment into the state.
Here, class *State* have two member variables other than *Env* one. They are concerned with **imperative** features and will be discussed in the next section.

4.2 Imperative

4.2.1 State

```java
public class State {

    public final Env E;
    public final Mem M;
    public final Int p;

    protected State(Env E, Mem M, Int p) {
        this.E = E;
        this.M = M;
        this.p = p;
    }

    public static State of(Env E, Mem M, Int p) {
        return new State(E, M, p);
    }
}
```

eval in App.java

Class *State* has three member variables.
Env E has been analyzed in the previous parts.
Mem M is actually class *HashMap*, which is used to simulate memory cells with Hash.

- *put* and *get* method in *HashMap* will be used in order to simulate memory store and memory load process.

Int p works as pointer of the memory.

4.2.2 imperative operations

The only operations related to imperative features are *Ref*, *Assign* and *Deref*. The implement of *typecheck* method is like others, therefore the report will concentrate on *eval* method:

$$\frac{E, M, p+1; e \Downarrow M'', p'; v \quad M' = M''[p \mapsto v]}{E, M, p; \texttt{ref}\ e \Downarrow M', p'; (\texttt{ref}, p)} \quad \text{(E-REF)}$$

Figure 4: evaluation rule of Ref

That is to say for every *Ref* operation, the ref cell of current pointer will get a new value while the pointer will point to a new cell, it will be implemented as:

```java
    @Override
    public Value eval(State s) throws RuntimeError {
        int ptr = s.p.get();
        s.M.put(ptr, this.e.eval(s));
        s.p.set(ptr+1);
        return new RefValue(ptr);
    }
```

eval in Ref.java

Similar process is done with *Assign* and *Deref*, no more repeat here. Please refer to project code for detailed information.

4.3 Built-In Function(II)

As demonstrated in the previous part, super constructor method is called. Here the *Expr* parameter will be overrode.
Since they are functions, the type checking process will be done in the *typecheck* of class *App*. Only *eval* should be finished. Here take *hd* as example:

```java
public class hd extends FunValue {

    public hd() {
        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 {
                if( (s.E.get(Symbol.symbol("x"))).toString().
                    equals("nil")){
                    throw new RuntimeError("hd(nil) is invalid!"
                        );
                }
                return ((ConsValue) (s.E.get(Symbol.symbol("x"))
                    )).v1;
            }
        });
    }
}
```

hd.java

5 Special Features (Bonus)

5.1 Lazy Evaluation

Given a spl program:

```
let a = (10 + 1) in
    let b = 5 in
        b + 1
    end
end
```

LasyEval.spl

In our original interpreter, it will first evaluate a and then evaluate b in the new environment. However, the evaluation of a is useless. By the thought of lazy evaluation(call-by-need), interpreter won't evaluate unless needed, which avoids useless processes.
In order to implement lazy evaluation, the environment sometimes need to store "symbol-expression-state" binding other than "symbol-value" bindings, so we will do some modification to environment.
State and *Expr* will now be member variables, and value is no more constant:

```
public class Env {
    private final Env E;
    private final Symbol x;
    private Value v;
    private final Expr e;
    private final State S;

    private Env() {
        E = null;
        x = null;
        v = null;
        e = null;
        S = null;
    }
    /* methods... */
}
```

member variables in Env.java

New constructor will be used to initialize current state and expression for the new symbol:

```
    public Env(Env E, Symbol x, Expr e, State S) {
        this.E = E;
        this.x = x;
        this.v = null;
        this.e = e;
        this.S = S;
    }
}
```

new constructor in Env.java

Then, some modification will be applied to *get* method: if current symbol is needed but not evaluated, evaluation will be done with expression and status:

```
    public Value get(Symbol y) throws RuntimeError {
        if(this.x.toString().equals(y.toString())){
            if (this.v == null){
                try {
                    this.v = this.e.eval(this.S);
                }
                catch (Exception ex){
                    throw new RuntimeError("RuntimeError");
                }
            }
            return this.v;
        }
        return this.E.get(y);
    }
```

modified get in Env.java

The only operations that create new environment are *Rec*, *Let* and *App*.
Rec shall not be modified since the value is actually a "state-symbol-expression" binding.
eval methods in *Let* and *App* now are:

```
    @Override
    public Value eval(State s) throws RuntimeError {
        Env newEnv = new Env(s.E, this.x, this.e1, s);
        return e2.eval(State.of(newEnv, s.M, s.p));
    }
```

eval in Let.java

```
    @Override
    public Value eval(State s) throws RuntimeError {
        FunValue app = (FunValue) l.eval(s);
        Env newEnv = new Env(app.E, app.x, this.r, s);
        State newState = State.of(newEnv, s.M, s.p);
        return app.e.eval(newState);
    }
```

eval in App.java

Note. The modified *eval* in App.java may cause StackOverflowError if the the times of recursive calls is too large.(e.g. in the pcf.fibonacci.spl, the 20th fibonacci number is evaluated by recursion). This is because class *Env* will consume more memory space compared to original ones.
You can set the stack of Java Virtual Machine larger to avoid the exception.

Now, the output of "LazyEval.spl" is still right, and variable a won't be evaluated.

5.2 Garbage Collection

Look at this spl program:

```
let a = ref 9 in
  let b = ref 7 in
    let c = ref 12 in
      (b := !c + !b);
            fst
    end
    let d = ref 0 in
        (d,b)
    end
  end
end
```

GC.spl

It might be wield, but it is right. With original interpreter, it will output "ref@3", meaning variable d will be at memory cell @3.(indexed from 0). But actually it can use cell @2, for @2's previous owner(variable c) has vanished when allocating cell for d.
In this part, **mark and sweep algorithm** is adapted and used to implement garbage collection(GC).
The key idea is to "mark and sweep" the first cell whose RC(reference counter) is 0 and allocate to new variable when the *Ref* operation is called.
First, the member variable *E* and *v* in class *Env* should be changed form private into public one so that the status of cells can be acquired.
Class *Ref* is another class to be modified. Previous, the pointer always increase 1 to find unused cell. Now, the pointer will choose the cell with least index number that is not refer by other variables.
New method *newPtr* helps the process. (*List* of Java is used)

```
    @Override
    public Value eval(State s) throws RuntimeError {
        int ptr = s.p.get();
        s.M.put(ptr, this.e.eval(s));
        s.p.set(newPtr(s));
        return new RefValue(ptr);
    }

    public int newPtr(State s){
        int ptr = 0;
        Env env = s.E;
        List l=new ArrayList();
        while(env != null){
            if(env.v instanceof RefValue){
                if(l.contains(((RefValue) env.v).p) == false) {
                    l.add(((RefValue) env.v).p);
                }
            }
            env = env.E;
        }
        l.sort(null);
        for(int i = 0; i < l.size(); i++){
            ptr++;
            if( (ptr - 1) != ((int) l.get(i))){
                break;
            }
        }
        return ptr;
    }
```

newPtr and eval Ref.java

Now, the output of "GC.spl" will be "ref@2". The memory cell of variable c is reused!

5.3 Polymorphic Type

Actually, with class *TypeVar* implemented, the simPL can be regarded as polymorphic language.
Taken map.spl as example

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

map.spl

By run

```
 interpret("map.spl");
```

We get

```
((tv3 -> tv10) -> (tv3 list -> tv10 list))
fun
```

6 Acknowledgement

I would like to express my sincere thanks to instructor Kenny Zhu and teaching assistant Xusheng Luo. CS383 has taught me a lot about Programming Languages. Thank you for your contribution!

Appendix

The output for the given spl program:

```
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/sum.spl
int
6
doc/examples/map.spl
((tv43 -> tv50) -> (tv43 list -> tv50 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
```

The result for LazyEval.spl is ”6”.
The result for GC.spl is ”ref@3” without GC, ”ref@2” with GC.