SimPL Interpreter

Programming Language Course Project

Haoyi You(519030910193, yuri-you@sjtu.edu.cn)

Department of Computer Science,
Shanghai Jiao Tong University, Shanghai, China

1 Introduction

1.1 Briefly Introduction

In this project, it aims to implement an interpreter of the programming language SimPL, which is a simplified dialect of ML and supports both functional and imperative programming methods.

1.2 Interpreter Utility

The interpreter is allowed to execute any SimPL language and have type check and type inference utility. Besides, it has the ability to distinguish the TypeError,RuntimeError and SyntaxError of the program.

1.3 Project Structure

The project consists of 3 parts–interpreter,parser and typing parts. Parser part is used for generating abstract syntax tree(AST). Typing part is used to do type checking and interpreter part is used to execute the whole program according to AST. The details of the above three part will be introduced in the following sections.

2 Typing

The Typing parts is used for checking whether the program is well-typed. It has 3 abstract class $Type, TypeEnv$ and $TypeError$.

2.1 Type

The type class is the basis of all the types, it has structure

```
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;
}
```

The function $isEqualityType()$ is used to judge whether such type can compare equality (useful in operator Eq and Neq).
The function $contains()$ is used to judge whether the object of such type has type variable tv.
The function $replace()$ is used to replace the all the type variable a in such object into type t.
The function $unify()$ is one step of the unification algorithm, which means add $this = t$ into the substitution set S. And we extend $TypeVar, IntType, BoolTypeandsoon$ from type class, represents various type in the type check process respectively. This children type is required to overwrite the function in the abstract type. Since the overwrite methods are similar, we present one of it here and please refer to the code for more details

```
public final class ArrowType extends Type {
    public Type t1, t2;
    ...
    @Override
    public boolean isEqualityType() {
        return false;
    }
    @Override
    public Substitution unify(Type t) throws TypeError {
        if(t instanceof ArrowType){
            Substitution left=this.t1.unify(((ArrowType) t).t1);
            Substitution right=this.t2.unify(((ArrowType) t).t2);
            return left.compose(right);
        }
        else{
            if(t instanceof TypeVar){
                return t.unify(this);
            }
            else 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));
    }
}
```

Consider the ArrowType of $t_1 \to t_2$.
In unify function, we distinguish the type of the t. If it is also a arrowtype, we implement it based on the rule of the arrowtype that

$$\frac{t_1 \to t_2 = t_1' \to t_2'}{t_1 = t_1', t_2 = t_2'}$$

If it is a type variable, we implement the function based on the rule of typevariable that

$$\frac{a = b}{b = a}$$

Otherwise, there exists a Typemismatch error.
In contains function, we only need to whether t_1 or t_2 contains such type variable, and it is the same in replace function.
We cannot judge whether two function are equal, so the isEqualityType function return $false$.

2.2 TypeError

There are two TypeError cases, TypeMismatchError and TypeCircularityError.
TypeMismatchError is thrown when we encounter such cases in typechecking process

$$a = b,$$

where a, b are concrete type and not the same, for instance $IntType = BoolType$.
TypeCircularityError is thrown when we encounter such cases in typechecking process

$$a = t, a \in FV(t)$$

a is a type variable and $a \in FV(a)$ means t contains a, for instance $a = a \to IntType$.

2.3 TypeEnvironment

The typeEnvironment is used in typechecking process, we can regard it as the Γ in the lecture. It stores as a form of linknode. Since we have implement 7 functions in lib and pcf parts (it is the default function of the SimPL), we need to add it into the default environment that

```
public DefaultTypeEnv() {
    TypeVar env1 = new TypeVar(true), env2 = new TypeVar(true), env3 = new TypeVar(true
        env5 = new TypeVar(true), env6 = new TypeVar(true);
    E = TypeEnv.empty;
    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));
    E = TypeEnv.of(E, Symbol.symbol("succ"), new ArrowType(Type.INT, Type.INT));
    E = TypeEnv.of(E, Symbol.symbol("fst"), new ArrowType(new PairType(env1,env2),env1)
    E = TypeEnv.of(E, Symbol.symbol("snd"), new ArrowType(new PairType(env3,env4),env4)
    E = TypeEnv.of(E, Symbol.symbol("hd"), new ArrowType(new ListType(env5),env5));
    E = TypeEnv.of(E, Symbol.symbol("tl"), new ArrowType(new ListType(env6),new ListTyp
}
```

2.4 Substitution

The class substitution is used for unification algorithm. In that algorithm, we consider the pair (S, q). We need to begin from (I, q) and ends at $(S, \{\})$.
In this interpreter, we do not collect all q constraints. Instead, it generates the constraint from each expression on by one, eliminates it immediately and update the S set.

3 Interpreter

3.1 Value

The abstract class Value is the basis of the all the values in the program, and we extend $BoolValue, FunValueandsoon$ from it. For each concrete value type, we need to implement $equals()$ function.
In java, $a == b$ means both the value and the address of a, b are the same, and that is because for a immutable value in java (for instance $inta = 1$), it has one address to store 1 and the variable a are all pointers (which is different from C/C++). However, there may be two IntValue objects $i1, i2$, but the member m of it points to the same int value ($i1.n = 12.n$). If we simply use $i1 == i2$ to judge whether they are the same, it will return false. Therefore, we need to implement $equals()$ function.
Here we present two concrete Value as examples

```
public class FunValue extends Value {
    @Override
    public boolean equals(Object other) {
        return false;
    }
}
public class IntValue extends Value {
    @Override
    public boolean equals(Object other) {
        if(other instanceof IntValue) return this.n==((IntValue)other).n;
        else return false;
    }
}
```

If we want to judge whether two value are the same, firstly they mush have the same type. For function value, since there are not two function the same, it simply returns false. For IntValue, we need to judge whether the n of it are the same if they are both Int type.

3.2 Environment

The environment stores the temporary values and is used to expression evaluation. With the help of it, we can realize the name-value binding of the program. It is in the form of linknode, and we can get the value of anyvariable based on its symbol.

```
public Value get(Symbol y)throws RuntimeError  {
    //Symbol in this layer of the Enviornmnet
    if(this.x.toString().equals(y.toString()))return this.v;
    else{
        //Symbol not in this layer of Enviornmnet
        if(v==null)throw new RuntimeError("Free Variable");
        return this.E.get(y);
    }
}
```

3.3 State

It consists of the environment, memory and the pointer p. It is significant to mention that the memory is implemented as a form of hashtable rather than an array. So we need to store the last element pointer of the memory, which is the p.

4 Parse

Parse part is used to generate AST. The generation process has well-written previously, so we do not discuss it too much. Here we mainly introduce the operators of AST.

4.1 Expr

The Expr class is an abstract class, requiring to implement two function *typecheck*() and *eval*(). The former one is used for the expression typechecking and the latter one is for expression evaluation.
From the class Expr, we extend three chilren class, $BinaryExpr, UnaryExpr$ and $Unit$,corresponding to binary operators, unary operators and unit operator respectively.

4.2 Typecheck function

This function is to check whether the node of this object in AST is well-typed. It needs to check the legality of the input expression and return the output type of the operator. Here we use the ExExpr class as an example

```
public abstract class EqExpr extends BinaryExpr {
    @Override
    public TypeResult typecheck(TypeEnv E) throws TypeError {
        TypeResult left=this.l.typecheck(E);
        TypeResult right=this.r.typecheck(E);
        Substitution S=left.s.compose(right.s);
        Type typeleft=S.apply(left.t),typeright=S.apply(right.t);
        S=S.compose(typeleft.unify(typeright));
        typeleft=S.apply(typeleft);
        // typeright=S.apply(typeright);
        if(typeleft.isEqualityType()){
            return TypeResult.of(S,Type.BOOL);
        }
        else throw new TypeMismatchError();
    }
}
```

The EqExpr is a binary operator and contains Eq and Neq operators.
Firstly, we need to check the input type of two input expressions. During the typechecking, it will generate the constraints and type variables, and the constraints will be transformed into substitutions immediately according to unification algorithm. Therefore, the TypeResult contains both Substitution set and the result type.
If the operator is well-typed, it has two conditions

1. Two input expressions has the same type.
2. Such type is EqualityType.

For the first conditions, we need $l.t == r.t$, thus using unifying function that $typeleft.unify(typeright)$ and compose the result into total substitution S.
For the second condition, we only need to use the $isEqualityType()$ function.
Since the result of the operator is a boolean value, we return the S and $BoolValue$.

4.3 Evaluation function

The utility of the function is to evaluate the value of the operator based on the input values. Before the evaluation, we mush judge the whether the input value is legal. Here we use Add class asan example

```
public class Add extends ArithExpr {
    @Override
    public Value eval(State s) throws RuntimeError {
        Value v1=l.eval(s),v2=r.eval(s);
        //From rule, we need to eval both side of the Add operator first
        if((v1 instanceof IntValue)&&(v2 instanceof IntValue)){
            return new IntValue (((IntValue) v1).n+((IntValue) v2).n);
        }
        else throw new RuntimeError("One side of Add operator is not an integer");
    }
}
```

We first need to evaluate the input of the expression. Then we are required to judge whether the input is legal. Since we can only add integer to integer, so we need to judge whether the input is an instance of IntValue.
After that, we add two value and return the result.
It is important to mention that there is a particular case for Cons operator

```
public class Cons extends BinaryExpr {
    @Override
    public Value eval(State s) throws RuntimeError {
        Value left=l.eval(s),right=r.eval(s);
        return new ConsValue(left,right);
    }
}
```

In this case, we cannot judge whether the right is an instance of a ConsValue. That's because in the interpreter part, we regard the list and nil as two different class. For more, please refer to ConsValue.java and NilValue.java.

5 Result

Based on the given SimPL program, we test them in the main function that

```
public class Interpreter {
    public static void main(String[] args) {
        interpret("doc/examples/factorial.spl");
        interpret("doc/examples/gcd1.spl");
```

```
        interpret("doc/examples/gcd2.spl");
        interpret("doc/examples/letpoly.spl");
        interpret("doc/examples/map.spl");
        interpret("doc/examples/max.spl");
        interpret("doc/examples/pcf.even.spl");
        interpret("doc/examples/pcf.factorial.spl");
        interpret("doc/examples/pcf.fibonacci.spl");
        interpret("doc/examples/pcf.sum.spl");
        interpret("doc/examples/plus.spl");
        interpret("doc/examples/sum.spl");
    }
}
```

And the result is

```
doc/examples/factorial.spl
int
24
doc/examples/gcd1.spl
int
1029
doc/examples/gcd2.spl
int
1029
doc/examples/letpoly.spl
int
0
doc/examples/map.spl
((tv40 -> tv47) -> (tv41 list -> tv47 list))
fun
doc/examples/max.spl
int
2
doc/examples/pcf.even.spl
(int -> bool)
fun
doc/examples/pcf.factorial.spl
int
720
doc/examples/pcf.fibonacci.spl
int
5
doc/examples/pcf.sum.spl
(int -> (int -> int))
fun
doc/examples/plus.spl
int
3
doc/examples/sum.spl
int
6
```

Fig. 1. Caption

6 Acknowledgement

In this project, I got to know the programming language more deeply. Especially for the lambda calculus and the unification algorithm, previously I only know their theoretical forms, however, I realised it practically now. Indeed, I learned lot from the project and the class.
Firstly, I must show my appreciation to the professor Zhu. His brilliant lecture teaches me lots of useful knowledge and helps me make progress.
Secondly, I need to acknowledge the TAs, their tutorials eliminate my questions in class and their guidance provides me lots of help in the project.
Lastly, I need to thank my classmates, discussing with them gives me lots of ideas in the project.

7 Reference

https://github.com/wzh99/SimPL