Document

尹楚轩 517021910929 yinchuxuan@sjtu.edu.cn

2020 年 6 月 26 日

1 Introduction

In this project, we implement an interpreter for a programming language called SimPL. Following by the skeleton provided, we use the primitive language java to descripe the evaluation rule and type checking rule of SimPL, and make sure it can execute some SimPL program normally.

2 implementation

The implementation of this project follows the given skeleton and the given type checking rule and evaluation rule basically, but there are two places where I modified the design of the skeleton and I think these changes is rather reasonable.

2.1 predefined function

The first one is about predefined functions. As the skeleton imtimates, these predefined functions, which are some FunValue, should be loaded to the initial environment and initial type environment. But I find that if I adopt this strategy, I will get myself into trouble when the interpreter is doing the evaluation and type checking, since in the whole environment there is only one type and one symbol for these predefined functions, but in one program they can be polymorphic.
So I decide to follow another way. I do not bind any thing in the initial environment and type environmnet, instead, the program will dynamically bind the symbol of predefined functions

and its value when it evaluates, and the type checking is also produced dynamically.
To implement this, I need to add something in the evaluation rule and type checking rule in the Name expression:

```
public Value eval(State s) throws RuntimeError {
    if(x.is("fst")){
        return new fst(varcnt++);
    }

    if(x.is("snd")){
        return new snd(varcnt++);
    }

    if(x.is("hd")){
        return new hd(varcnt++);
    }

    if(x.is("tl")){
        return new tl(varcnt++);
    }

    if(x.is("iszero")){
        return new iszero(varcnt++);
    }

    if(x.is("pred")){
        return new pred(varcnt++);
    }

    if(x.is("succ")){
        return new succ(varcnt++);
    }
```

```
        return s.E.get(x);
    }

    public TypeResult typecheck(TypeEnv E) throws TypeError {
        if(x.is("fst")){
            TypeVar a = new TypeVar(false);
            TypeVar b = new TypeVar(false);
            return TypeResult.of(new ArrowType(new PairType(a, b), a));
        }

        if(x.is("snd")){
            TypeVar a = new TypeVar(false);
            TypeVar b = new TypeVar(false);
            return TypeResult.of(new ArrowType(new PairType(a, b), b));
        }

        if(x.is("tl") || x.is("hd")){
            TypeVar a = new TypeVar(false);
            return TypeResult.of(new ArrowType(new ListType(a), a));
        }

        if(x.is("iszero")){
            return TypeResult.of(new ArrowType(Type.INT, Type.BOOL));
        }

        if(x.is("pred") || x.is("succ")){
            return TypeResult.of(new ArrowType(Type.INT, Type.INT));
        }

        return TypeResult.of(E.get(x));
    }
```

where the "is" method is a method of symbol that used to check whether the symbol is such a name:

```
public boolean is(String n){
    return n.equals(name);
}
```

So in this case we can see that once there is a application of predefined functions, the interpreter will generate a binding of a symbol and a FunValue just like the normal functions.

2.2 the generation of substitution

During the process of type checking, we need to add some type variables into type environment and generate substitution to replace them when we meet type polymorphism. Usually, the generation of substituion follows the type constraint rule. There are cases in some type constraint that it requires the types of the two expressions that their tree expression contains must be equal. At first glance it seems like we only need to generate a substituion that one replace another if one of them is type variable. But the real situation is more complicated. Due to the polymorphism, we may meet two type in one expression like this:

$$TypeVar_1\ list \quad TypeVar_2\ list$$

It's natural to replace $TypeVar_1$ by $TypeVar_2$ if the two types are required to be same. But according our type checking rule, these two types are neither same nor type variables, so the interpreter will throw type error. It seems like we need a method to generate substituion for such two types which are required to be equal and contains type variables.
I add a static method *genSubstitution* in Type class:

```
public static Substitution genSubstitution(Type a, Type b) throws TypeError{
    if(a instanceof ListType && b instanceof ListType){
        return genSubstitution(((ListType)a).t, ((ListType)b).t);
    }

    if(a instanceof RefType && b instanceof RefType){
        return genSubstitution(((RefType)a).t, ((RefType)b).t);
    }

    if(a instanceof ArrowType && b instanceof ArrowType){
        return genSubstitution(((ArrowType)a).t1, ((ArrowType)b).t1).
        compose(genSubstitution(((ArrowType)a).t2, ((ArrowType)b).t2));
```

```
        }

        if(a instanceof PairType && b instanceof PairType){
            return genSubstitution(((PairType)a).t1, ((PairType)b).t1).
            compose(genSubstitution(((PairType)a).t2, ((PairType)b).t2));
        }

        if(a instanceof TypeVar){
            return Substitution.of((TypeVar)a, b);
        }

        if(b instanceof TypeVar){
            return Substitution.of((TypeVar)b, a);
        }

        if(a == b){
            return Substitution.IDENTITY;
        }

        throw new TypeError("the generation of Substitution is faulty!");
    }
```

So after that, we only need to apply this method to generate substituion:

```
s = s.compose(Type.genSubstitution(t1, t2));
```

then we can get correct substituion.

3 Explanation

It's very easy to use this interpreter. If you want to interpret a .spl file, you only need to apply interpret method in main function, whose parameter is the relative path of this file. The interpreter passes all the test provided rightnow, except that for pcf.fibonacci.spl, it can work until the given number is 19. But for 20, it may evaluate too long and crush.