CS383: Programming Language Project

Dong Qing
5130309298
1005983364@qq.com

1 Abstract

In this project, I finished the implementation of an interpreter for the programming language SimPL. SimPL is a simplified dialect of ML, which can be used for both functional and imperative programming. With the provided skeleton, my work mainly focuses on the type system and evaluation system. The former is more challenging. After this project, I have known more about type inference, λ calculus, and garbage collection.

2 Evaluation System

I finished the Evaluation system firstly since it is simple. The main idea is recursively dealing with every expression. To do this, you need to fully understand the concepts about Memory and Environment.

2.1 Environment

An environment is a (variable, value) mapping (set of bindings):

```
     E ::= . | E, x v
Define E[x v] (add a binding into the environment):
     .[x v] = x v
     (E, x'  v')[x v] = E, x v  if x = x'
                          or E, x' v', x v   if x     x'
```

At first, I didn't know about why the class Env need to include another Env. Finally I realize the Sublety. For nested function calls, e.g. $(\backslash x.\backslash y.\backslash z.\ x+y+z)$ 1 2 3, the environment for each function call is organized in a stack, i.e. the call stack. Also Env will make lazy evaluation easy to implement. Every time we meet a symbol, we will check if it is the symbol in the current Env. If not, we will new a Variable-Value Pair to the Env of current Env. Along with such actions, eventually we will push all (Variable-Value) pair to the Env.

2.2 Memory

Another challenge for me is the Memory. In our skeleton, A memory use a Hash Table. Every time we meet a RefValue, we put the Value to Mem. The RefValue only stores the pointer p.

When it comes to the pointer, again, I have to say our skeleton is really well-designed. It uses states to store current Env, Mem and p. So when we want use a new location in memory, we only need to new a State, which p = p+1. Therefore, the interpreter will remember how many memory blocks have been used. You can clearly see how I make it according to the following code.

```
        public Value eval(State s) throws RuntimeError {
         /*find the current p*/
         Int p = new Int(s.p.get());
         /*p = p+1*/
         s.p.set(s.p.get()+1);
         /*Evaluate v1*/
         Value v1 = e.eval(s);
         /*store v1 in block p(before add)*/
         s.M.put(p.get(), v1);
         /*return the new refValue*/
         RefValue p1 = new RefValue(p.get());
         return p1;
    }
```

2.3 Implementation

I will illustrate some typical implementation in this part. Most of Evaluation designs are simple and I will not explain more.

2.3.1 Apply

I will illustrate how I finished "Apply" evaluation in this part. First, attach the code here.

```
FunValue fv = (FunValue)l.eval(s);
Value v2 = r.eval(s);
return fv.e.eval(State.of(new Env(fv.E, fv.x,v2),s.M,s.p));
```

It seems very simple, but still worth careful consideration. First, we need to evaluate left value because we need to know the Env and symbol of this funValue. After that, we evaluate the right value. Then what we need to do is replace "x" of the FunValue with v2. It's easy to do with Env, we just produce a new State, with a new Env. In this new Env, all "x" is binding to v2.

Now you may have a question: What will happen if a Function have more than one parameters? This is just where the Env's advantage is. Remember that we will return a FunValue. This FunValue will again apply to other parameters. And always note that we will **NEVER** evaluate the result of FunValue until we need. And that shows the strategy "Call By Need" (i.e. Lazy evaluation) And we will discuss this more in Section 5.2 .

2.3.2 Assign

I will illustrate how I finished "Assign" evaluation in this part. First, attach the code here.

```
public Value eval(State s) throws RuntimeError {
    RefValue p1 = (RefValue)l.eval(s);
    Value p2 = r.eval(s);
    s.M.put(p1.p, p2);
    return Value.UNIT;
    }
```

If you have viewed section 2.2, it's not hard to understand this part. First we need to evaluate left value. According to our grammar, left value must be a RefValue. We find "p"(i.e the address it pointing to). Then evaluate right value. Put the pair(Address, Value) to memory.

3 Type System

The implementation of Type System is so hard that I spent most of my time on it. Nevertheless, I felt a great sense of achievement when I overcame those bugs. First, I will introduce the design of my Type System.

3.1 Introduction

The type checking system is mainly relying on Type Inference. The main idea is:
1. introduce new type variables for unknown types whenever necessary.
2. walk over the program keep track of the type equations t1 = t2 that must hold in order to type check the expressions according to the normal typing rules.

3.2 Substitution

Fortunately, the hardest component was given in the skeleton. It is so important that all type checking rely on it. So I will firstly discuss this part.

A substitution is a function from TypeVar to Type Scheme. It will define on all typeVar, but only some of variables are actually changed. How to do this substitution are decided what evaluation you are doing.

When running a program, many substitutions will happen. And the best way to combine this substitution is define a new Compose. e.g. $(U\ o\ S)$ applies the substitution S and then applies the substitution U:

$$(UoS)(a) = U(S(a))$$

And also we need a function $Unify$ to tell us how to deal with constraints.

Unification systematically simplifies a set of constraints, yielding a substitution. Without unification, we will never get the principal solution.

3.3 Implementation

I will explain some typical implementation in this part.

3.3.1 AndAlso

```
    public TypeResult typecheck(TypeEnv E) throws TypeError {
        TypeResult t1 = l.typecheck(E);
        TypeResult t2 = r.typecheck(t1.s.compose(E));
        Substitution sout = t2.s.compose(t1.s);
        Substitution s1 = sout.apply(t1.t).unify(Type.BOOL);
        sout = sout.compose(s1);
        Substitution s2 = sout.apply(t2.t).unify(Type.BOOL);
        sout = sout.compose(s2);
        Type tout = Type.BOOL;
        return TypeResult.of(sout,tout);
    }
```

First of all, we check the left value in the current TypeEnv and return the result in t1. After that, we will check right value. But now we need add the t1 to the TypeEnv. Then we compose t2's substitution and t1's substitution, and get a return Substitution "sout". Now since *andalso* apply to two boolean type. So we unify t1 and t2 with *Type.bool*. Finally we have return TypeResult with the new Substitution and bool type.

3.3.2 AndAlso

```
    public TypeResult typecheck(TypeEnv E) throws TypeError {
        TypeResult t1 = l.typecheck(E);
        TypeResult t2 = r.typecheck(t1.s.compose(E));
        ArrowType t3 = new ArrowType(new TypeVar(true),
        new TypeVar(true));

        Substitution sout = t2.s.compose(t1.s);
        try{
```

```
        Substitution s1 = sout.apply(t1.t).unify(
                    sout.apply(t3));
        sout = sout.compose(s1);
        }catch (TypeMismatchError e){
        }
        try{
        Substitution s2 = sout.apply(t2.t).unify(
                        sout.apply(t3.t1));
        sout = sout.compose(s2);
        }catch (TypeMismatchError e){
        }

    }
```

The setup is same as *apply*. But now we need a new ArrowType. Sometimes we don't know about the function Type yet and the right value's type have been known. Whatever, what we need to do is unify t1 and t3, then unify t2 and t3. Generally, we will reduce the number of arrows by 1. eg. $a- > (b- > c)$ apply to int, we will get $a = int$ and return $b- > c$.

4 Basic Functions

I finished $fst.java$, *hd.java*, *snd.java*, *tl.java*, *iszero.java*, *pred.java*, *succ.java*.They are simple. I don't want to explain more, just attaching a example here.

```
    public iszero() {
        // TODO
        super(Env.empty, Symbol.symbol("iszero␣argument"),
            new Cond(
                new Eq(
                    new Name(Symbol.symbol("iszero␣argument")),
                    new IntegerLiteral(0)),
                new BooleanLiteral(true),
                new BooleanLiteral(false)));
    }
```

5 Bonus

5.1 Garbage Collection

When I finished the basic part, actually I have formed a basic idea of garbage collection.

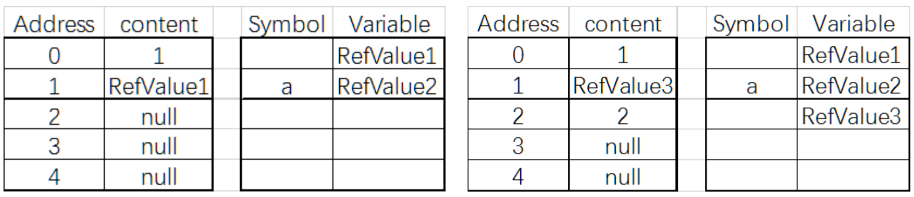
5.1.1 When will garbage be created?

In modern programming languages, many instructions may produce garbage. For example, assignment of pointers and ”free” action of some nodes. However, our grammar is relatively simple and assignments are only allowed in this form: $x : aref := y : a$ But it is also possible to create garbage. Consider the following instructions:

```
let x = ref ref 1 in
     x := ref 2; x
end
```

We let x = ref ref 1, the state shows in Fig1.a. When the interpreter find ref 1, he will put 1 in the address 0,and return a new RefVaule(we name it RefValue1). Then he finds another ref, and he will put RefValue1 to address 1, return RefValue2. When he knows he is dealing with a let expression, he bindings a with RefValue2. Until now, no garbage is created. “x” is binding with RefValue2, which points to RefValue1, which points to 1.

Then the interpreter will deal with the assignment. The interpreter meets “a”, so he looks up the Symbol Table(Actually Env in our program) and find RefValue2. After that, he meets “ref 2”, and put 2 to address2, return RefValue3. Then he will put RefValue3 to the address RefValue2 pointing to. Now block 0 is garbage.



Address	content
0	1
1	RefValue1
2	null
3	null
4	null

Symbol	Variable
	RefValue1
a	RefValue2

(a) After Let

Address	content
0	1
1	RefValue3
2	2
3	null
4	null

Symbol	Variable
	RefValue1
a	RefValue2
	RefValue3

(b) After Assign

Figure 1: Garbage Creation

5.1.2 GC strategy

I use “Mark and Sweep” in my program. I will not introduce the principle of this strategy for fear that my report will be too long. I chose it because it is easy to apply and enough for our simple interpreter.

I add a mark in every Variable class. And whenever the interpreter meets “ref”, which means it needs to allocate a new memory block. First it will check if memory is full or reach the threshold. If is, GC starts. First, traverse the Env to find the symbols which binding to a RefValue. From this RefValue, all Values that can be reached will be marked. Finally, free all the unmarked blocks.

Since the memory uses Hash Table, I take advantage of this property and doesn’t care about the reuse of collected block. In practice, the interpreter will keep a free list and allocate new Variables to the address in this list.

5.1.3 Implementation

```
  /*garbage collection*/
       if(p.get() > 4)
       {
       /*mark*/
        System.out.println("EnterGC");
        Env env = s.E;
        while(env != Env.empty)
        {
            Value val = env.getValue();
                while(val.getClass().toString().intern()
                       == "class␣simpl.interpreter.RefValue"
                       &&val.mark ==0) //avoid endless loop
            {
                val.mark = 1;
                val = s.M.get(((RefValue)val).p);
            }
                val.mark = 1;
                env = env.getEnv();
        }
       /*sweep*/
        for (int i = 0; i< p.get(); i++)
        {
                if (s.M.get(i).mark == 0)
                s.M.put(i, null);
        }
       }
```

5.1.4 Test

For a simple program, you can set a small threshold to show the effect. I annotated a fragment of code to show the memory states before and after GC. You can use it to know how my collector works.

5.2 Polymorphic type

5.2.1 Introduction

In our type system, we use inference to do type checking. Inference in some of program may lead to a uniquely determined type. Like:

```
let v = 1 in
       v
end
```

When we let $a = 1$, the type system will believe in that v is int, a uniquely determined type. And uniquely determined types include basic types and their combination.

But type inference of other programs may result in a undetermined polymorphic type.Like

```
let f = fn x => x in
        f
end
```

First, we assume f have a type $tv1$. Then f is a function, if x have a type $tv2$, $tv1 = tv2- > tv2$. The type cannot be uniquely determined yet. In this situation, if f apply to a int value, it will be $int- > int$; if $list$, $list- > list$. This is what polymorphic means. In the above test program, the type result will be $tv2- > tv2$;

5.2.2 Implementation

The implementation actually have been finished because my type system uses Type Inference to complete type checking.

When we meets an unknown type, we will seem it as a IDENTIFY, and name it as $tv + tvcount$, in which $tvcount$ will increased when the action happen.(see in the source file $TypeVar$). This TypeValue will be unified and composed again and again during the running process. And actually it is Type Inference that allows Polymorphic Type.

5.3 Lazy Evaluation

Lazy evaluation is also known as Call-by-need strategy. The main idea is delaying the evaluation of one expression until its value is needed. There are two basic examples.

1. $let\ x\ =\ 1\ +\ 2\ in\ x + x$

we only need to evaluate x once.

2. $e1\ \ e2$

If e1 is false, we don't need to evaluate e2, and return false directly.

For the first case, I have illustrated how we take advantage of the Environment to implement the Lazy Evaluation.

For the second case, It's also easy to implement. You can view the following code to know how I make it.

```
public Value eval(State s) throws RuntimeError {
    BoolValue v1 = (BoolValue)l.eval(s);
    //System.out.format("Judging %b %n",v1.b );
    if (v1.b == false)
        return new BoolValue(false);
    BoolValue v2 = (BoolValue)r.eval(s);
    //System.out.format("Judging %b %n",v1.b );
    if (v2.b == false)
        return new BoolValue(false);
    return new BoolValue(true);
```

```
        //return null;
    }
```

6 Acknowledgement

I have experienced a hard time during the process to do this project. I finished the basic part within 2 weeks, and spent 4-5 days dealing with GC, and optimizing my program. Since I thought I would have a low Exam score. I wanted to do more in this project.

But this process is happy. Whenever at a loss, I would review the lecture material again. And every time, I would acquire more.

Thanks to professor Kenny Zhu, who keeps his criterion on what a good lesson should be and never moved. CS383 will be an unforgettable experience for me. Thanks to dear TA, Xusheng Luo, who answered my endless questions patiently. Thanks to classmates who have helped me, especially Houshuang Chen. We discussed a lot about the basic concepts.

Thanks to all again.