

CS383 Programming Language

Course Project

simPL interpreter

Yifeng CHEN
5140309458

December 21, 2016

1 Overview

In this project, an interpreter for SimPL is implemented. The interpreter has mainly 4 stages: 1. lexical analysis, 2. Syntactic analysis, 3. Type Checking, 4. Evaluating Result. The first two steps have already been done in the skeleton, so I only need to complete type checking and result evaluating. Meanwhile, polymorphism, garbage collection and lazy evaluation are also realized in the interpreter.

2 Type Checking

In this part, I will introduce how the interpreter does type checking. The type inference rules are given in the spec_4.4.pdf. The core idea is that whenever the current expr brings us more information about type, we introduce a new typevar as an intermediate type, and use unify to bind this new type to former type to create new substitution. And notice that we are not applying the method introduced in the class to solve for principal solution; instead, we choose to create substitution and solve it simultaneously here. The details will be discussed later.

2.1 Simpl.typing.Type

This is the fundamental package which stores type information of an ast node. This package consists of multiple classes, such as ArrowType, PairType, TypeVar..., etc, which are all implementations of abstract class 'simple.typing.type'. TypeVar works as a meta-type-variable here; each time a new typeVar is created, it will be given a unique name as 'tvxx'. Meanwhile, each of these subclasses has to implement 'isEqualityType', 'unify', 'replace', 'contains' functions relevantly. 'isEqualityType' is used to tell whether this type is allowed for comparison. Notice that ArrowType and UnitType are not allowed for equality test in simPL, so they simply return false. 'Unify' is a function to solve for the substitution. It binds two types (generally speaking, a typevar and a type) together and uses this bind to create a new substitution. 'Replace' is a method to do type replacement. For example, we can call a.replace(b,c). And this function will replace all type b's occurrences in type a with c. Function 'contains' is used to tell whether type b occurs in type a.

2.2 Simpl.typing.Substitution

Substitution is a vital part in type checking part; it is used to record down the type binding info. It contains 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 substitutions into one. The key function in substitution is apply. This function uses the binding info in substitution to do replacement in a type:

```

public Type Compose.apply(Type t) {
    return f.apply(g.apply(t));
}
public Type Replace.apply(Type b) {
    return b.replace(a, t);
}

```

Obviously, this function will track down the substitution until it meets a substitution of Replace type, and then it will leave the replace work to `Simpl.Typing.replace` (which is introduced in former part) according to its binding info.

2.3 `Simpl.ast.typecheck`

With `Simpl.typing.Type` and `Simpl.typing.Substitution`, we are able to use two types to create new substitution and use substitution to do type replacement. Now we have to look deep into the details of how type checking is completed in different ast nodes. After the input program has been analyzed by the parser, the syntactic analysis result will be stored in the constitution of different ast nodes, which are different implementations of abstract class `'Simpl.ast.Expr'`. The type checking job will be sent to these specific classes, which implement the abstract function `'typecheck'` diversely. Function `'typecheck'` takes in an ast node and returns its `TypeResult` value, which is a composition of its type and its substitution. I've listed some key ast nodes's `typecheck` as follows.

`ArithExpr.typecheck`

```

public TypeResult typecheck(TypeEnv E) throws TypeError {
    TypeResult type_result_left = l.typecheck(E);
    TypeResult type_result_right = r.typecheck(E);

    Substitution s = type_result_right.s
        .compose(type_result_left.s);

    Type type_left = type_result_left.t;
    Type type_right = type_result_right.t;

    type_left = s.apply(type_left);
    type_right = s.apply(type_right);

    s = type_right.unify(Type.INT)
        .compose(type_left.unify(Type.INT))
        .compose(s);

    return TypeResult.of(s, Type.INT);
}

```

In `typecheck` of `ArithExpr`, we will first recursively do `typecheck` of its left hand operand and right operand. Then we combine the two side's substitution into one and apply it on left and right hand's type. Since arithmetic's two operands must be 'INT' type, so we use `'unify'` to bind their type with `Type.INT`.

Notice that doing this will create new substitution, and we have to compose this new substitution with the former to gain a new one. Finally, the result of arithmetic is still `INT`. One thing we have to take care is the order of `compose`. If we write `type_result_left.s.compose(type_result_right.s)` instead of `type_result_right.s.compose(type_result_left.s)`, this will lead to a problem. When doing `apply`, we will first search `type_result_right.s`

to do type replacement. However, according to the type inference order, we have to first do type replacement in `type_result_left.s!`

Let.typecheck

```
TypeResult type_result_e1 = e1.typecheck(E);
TypeResult type_result_e2 = e2.typecheck(TypeEnv
    .of(E, x, type_result_e1.t));
Substitution s = type_result_e2.s
    .compose(type_result_e1.s);
return TypeResult.of(s, type_result_e2.t);
```

According to T-LET: $\frac{\Gamma \vdash e_1 : t_1 \quad \Gamma[x : t_1] \vdash e_2 : t_2}{\Gamma \vdash \text{let } x = e_1 \text{ in } e_2 \text{ end} : t_2}$ We check e_1 's type and add its binding with x in current typeEnv. Then we evaluate e_2 under this new typeEnv. Finally, the Let expr's type is the same as e_2 .

2.4 Type Errors

Two kinds of errors are defined in `Simpl.Typing.TypeError`: `TypeCircularityError` and `TypeMismatchError`. They both occur during the process of 'unify'. If we try to unify `typevar a` with `typevar a`, this will cause a `TypeCircularityError`. If we try to bind a `unify int` with `bool`, this will cause a `TypeMismatchError`;

3 Evaluating Result

Type check and result evaluation are not carried out simultaneously. Actually, we have to traverse the AST twice, one for type check and one for value evaluation.

3.1 Simpl.interpreter.Value

Like `simpl.typing.type.package Value` stores value info for different kinds of ast nodes. `Simpl.interpreter.Value` includes: `IntValue`, `PairValue`, `RecValue`, etc. And the abstract class contains `Value.NIL` and `Value.UNIT` as static members. `FunValue` is the one which is slightly different from others:

FunValue

```
public class FunValue extends Value {
    public final Env E;
    public final Symbol x;
    public final Expr e;
}
```

x is function's parameter symbol, e is its body. What's more, `FunValue` also contains the `Env` where it can find the former parameters' values in.

3.2 State

As explained in the `spec.4.4.pdf`, $State = Env \times Mem \times N$. `Env` is a composition of bindings of symbol and its value. `Mem` is a binding with pointer and its stuff, which can be realized in a hashmap. `N` is the current pointer. According to the evaluation rules given in `spec.4.4.pdf`, we can see that state will keep changing and send the necessary info to the evaluated expr to calculate its value. Notice here that we have to the pre-defined functions (`fst`, `snd`, `hd`, `tl`, `succ`, `pred`, `iszero`) into `Env` before we start evaluating.

3.3 Simpl.ast.eval

Just like the way we do type check, we still assign the evaluation job to relevant ast nodes, which implement 'eval' function diversely. Let's take a look at some of them:

AndAlso.eval

```
public Value eval(State s) throws RuntimeError {
    Value value_left = l.eval(s);
    // if left hand is not boolean already
    if (!(value_left instanceof BoolValue))
        throw new RuntimeError("not a bool value");
    //if left hand is false
    if( ! ((BoolValue) value_left).b ){
        return new BoolValue(false);
    }
    Value value_right = r.eval(s);
    //if right hand is not a boolean
    if (!(value_right instanceof BoolValue))
        throw new RuntimeError("not a bool value");
    //ow. it depends on the right hand's value
    return new BoolValue( ((BoolValue) value_right).b );
}
```

AndAlso first evaluate its left operand's value. If it is already false, it will return false directly. Otherwise, it will return the right hand's operand value.

Ref.eval

```
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 is actually a well-wrapped pointer in Mem. It is the only ast that will ask the STATE's Mem for new spaces. The Ref first gets a pointer from STATE and then stores the corresponding value in the Mem. And if a deref is applied on ref later, he can use his pointer to get the stuff he stored in Mem. Assign is different from Ref; it merely modifies the content that Ref is pointing to without applying for new spaces.

4 Polymorphism

In programming languages and type theory, polymorphism is the provision of a single interface to entities of different types. A polymorphic type is one whose operations can also be applied to values of some other type, or types. In our type system, the giving inference rules actually support Polymorphism already. We can actually view `TypeVar` as type scheme. For example, we have the ast below: `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`. When we are applying it to different parameters, `tv1` can be replaced with any specific types. This kind of Polymorphism is also called as Parametric polymorphism.

5 Garbage Collection

5.1 Implementation

In this case,i use mark and sweep strategy to do garbage collection.So we have to first define a new class to include both value and its mark info:

memCont

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

Since ref is the only ast that will apply for new spaces in Mem and can have access to Mem,so it becomes quite easy to monitor the memory usage.The state's Env stores all symbols we've evaluate value already.If x's value is a ref,then it can point to some space in the mem.We can use this to do mark step:

State.mark

```
public void mark(){
    mark(this.E);
}

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);
    }
}
}
```

As for sweep,we have to rewrite mem to record which spaces have been allocated.Meanwhile,to make use of the garbage area,i use a stack freeList to store the collected free spaces's pointers.

Mem

```
public class Mem {
    public HashMap<Integer ,memCont> memMap = new HashMap<Integer ,memCont>();
    private static final long serialVersionUID = 1517654669000677591L;};
    public Stack<Integer> freeList = new Stack<Integer>();
    public HashSet<Integer> alloList = new HashSet<Integer>();
    private static final long serialVersionUID = 3032335706122811691L;};

    public void sweep(){
        for(Integer p:alloList){
            memCont m = memMap.get(p);
        }
    }
}
```

```

        if (m.mark) {
            demark(p);
        } else {
            tmp.push(p);
        }
    }
    while (!tmp.isEmpty()) {
        Integer p = tmp.pop();
        delete(p);
    }
}
}

```

I also rewrite the `get_pointer` function to get pointer from `M's free_list` first, only when `free_list` is empty, it will ask `mem` to apply new spaces. To increase flexibility, I use `gc_enable()` function to set when gc occurs.

5.2 Test

To test the GC part, I use the `simPL` code below:

`memtest.spl`

```

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

```

For clearness, I call GC when handling `x's` value, and the result is as follow:
Obviously, `Mem[2]` is marked and the space of `Mem[0]` and `Mem[1]` is collected.

6 Lazy Evaluation

In programming language theory, lazy evaluation is an evaluation strategy which delays the evaluation of an expression until its value is needed and also avoids repeated evaluations. I merely focus on avoiding repeated evaluations here. For example, when we call `fibonacci 20`, it will recursively call `fibonacci 19` and `fibonacci 18`. When we calculate `fibonacci 18`, its value actually has already been calculated by `fibonacci 19`, but we have to calculate it again, which wastes a lot of time and space. So the idea is to use a table to store the calculated value in it.

6.1 Implementation

One important point here is to identify a function uniquely. For example, how do we distinguish `fibonacci 20` from `fibonacci 19`? To realize this, I define a class "function_entry", and use function 'equal' to judge whether two `function_entry` is the same one:

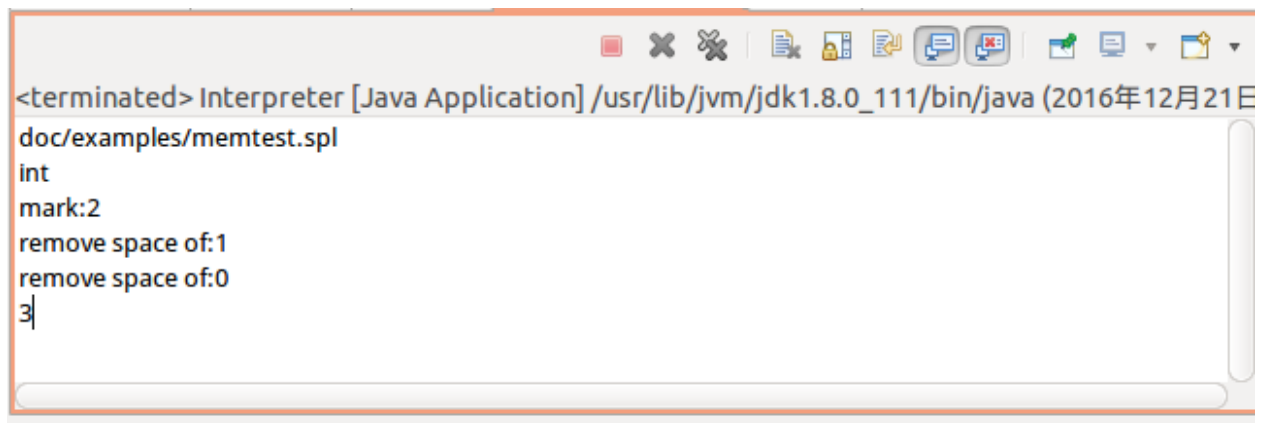


Figure 1: GC test

function_entry

```
public class function_entry {
    Value fun;
    Value para;
    Value result;

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

In this class, fun is this function, and para is its value. Now we can use this to set up a table, and include this table into State.

loopUpTable

```
public class loopUpTable {
    private Stack<function_entry> table ;
}
public class State {
    public final Env E;
    public final Mem M;
    public final loopUpTable LUT;
    public final Int p;
}
```

When we are evaluating App's value, we can first check LUT for result:

App

```
function_entry fe = new function_entry(value_e1, value_e2);
Value result = s.LUT.get_result(fe);
if(result == null){
    State state_new = State.of( new Env(fun.E, fun.x, value_e2), s.M, s.p, s.LUT);
    result = fun.e.eval(state_new);
}
```

```

    s.LUT.put(fe,result);
}
return result;

```

Since i use stack to implement the table,this means that this table cannot be too large.Otherwise,it will cost a lot of time on searching for the existence of some function_entry.So my strategy here is to only record recursive function's value done.

6.2 Test

I use the following code to do test.For clearness,i only record the rec f's value down:

APP

```

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 19
  end
end

```

The result are as follows and we can see that LUT can speed up the execution greatly.In this case,the one with LUT is 3 times faster!

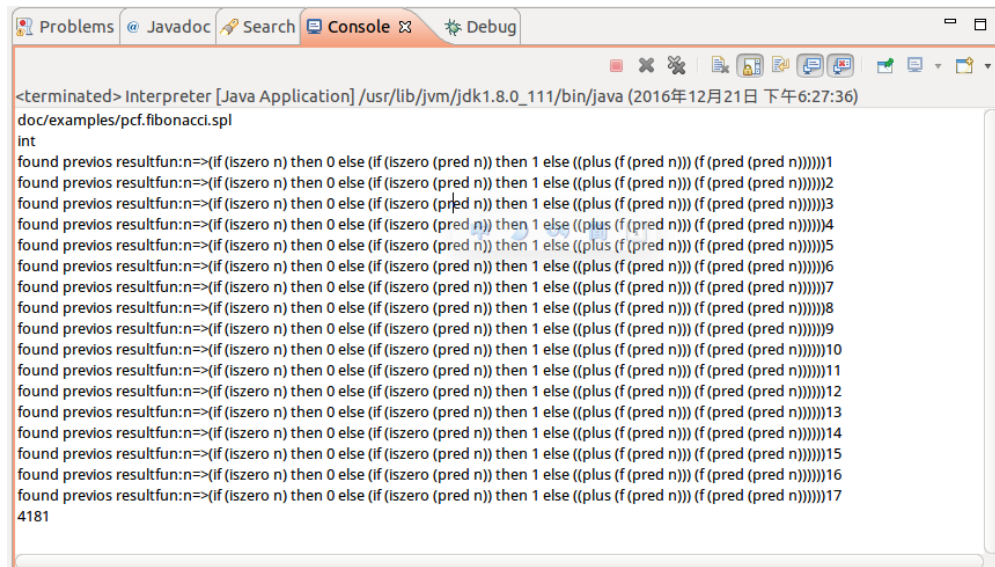
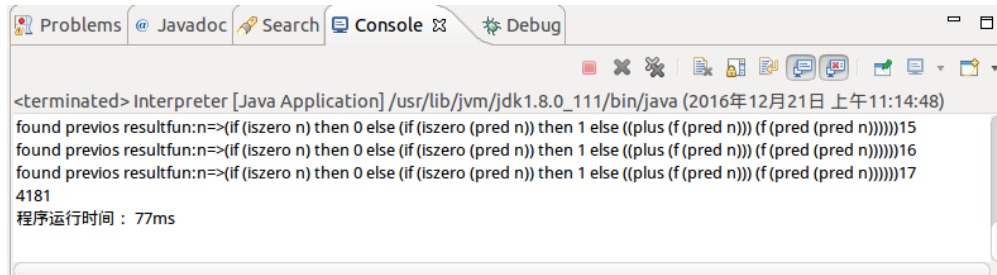
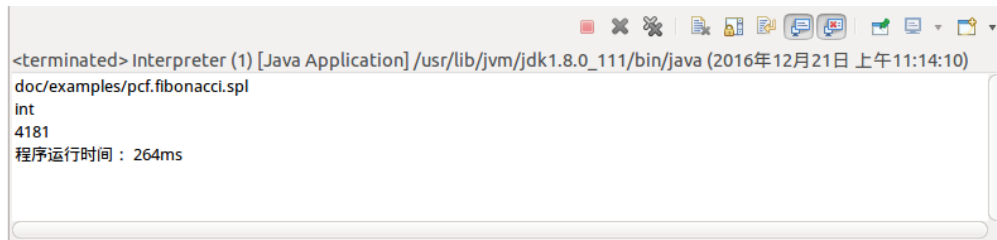


Figure 2: LUT test



```
<terminated> Interpreter [Java Application] /usr/lib/jvm/jdk1.8.0_111/bin/java (2016年12月21日 上午11:14:48)
found previos resultfun:n=>(if (iszero n) then 0 else (if (iszero (pred n)) then 1 else ((plus (f (pred n))) (f (pred (pred n))))))15
found previos resultfun:n=>(if (iszero n) then 0 else (if (iszero (pred n)) then 1 else ((plus (f (pred n))) (f (pred (pred n))))))16
found previos resultfun:n=>(if (iszero n) then 0 else (if (iszero (pred n)) then 1 else ((plus (f (pred n))) (f (pred (pred n))))))17
4181
程序运行时间 : 77ms
```

Figure 3: using LUT exec-time of fibo 19



```
<terminated> Interpreter (1) [Java Application] /usr/lib/jvm/jdk1.8.0_111/bin/java (2016年12月21日 上午11:14:10)
doc/examples/pcf.fibonacci.spl
int
4181
程序运行时间 : 264ms
```

Figure 4: not using LUT exec-time of fibo 19

7 Summary

After finishing this project, not only do I have a better understanding of λ expression and type inference, but I see the necessity of memory management and special treatment to recursive functions. Finally, I want to thank Prof. Kenny Zhu for his excellent teaching skills and T.A Xusheng Luo for his dedication in tutorial classes. It is really a pleasure for me to take this course!