CS383CourseProjectDocument

Student: Chen Xuyang 5140309412

1 Introduction

In this project I implement a simPL interpreter which is able to fulfill all the typing and semantic actions described in the specification. I will illustrate the detail in this document. What’ s more, I have accomplished all the bonus in the specification and tested it with my own test programs.

2 Lexical Definition

The lexical analysis has been done in the skeleton.

3 Syntax

The syntax analysis has also been done in the skeleton.

After lexical and syntax analysis, the program will be converted to a recursive structure of expressions. Here we establish a class Expr to represent all the expression. And different expression is just different sub-class of the class Expr. Each expression has a method called typecheck(TypeEnv E), and eval(state s), the usage of them will be covered later.

4 Typing

4.1 Substitution

Substitution is used to replace one type to another type, which is useful during the type inference. Substitution is implemented as a class in our project. The method includes construction method, apply method, 2 different compose method and 3 subclass called Identity, Replace and Compose.

Here are the illustration of the method of Substitution Class:

- apply(Type b)

  Since a substitution is used to map a type scheme to another type scheme, the compose method will do the job. The function receives a type and return its type after the substitution. Different kind of substitution will have different apply function.

- compose(TypeEnv)

  This method is used to return a type environment where all the type in the original environment are applied by the substitution.

- compose(Substitution inner)
  This method returns the composed substitution (also know as the Subclass Compose) of “this” and inner, which also can be written as “this ○ inner”.
  Then we introduce the three subclass of substitution.

- Identity
  Obviously, the apply(Type b) method of this class should just return b itself.

- Replace
  Replace substitution, with private attribute Type a and Type b, has an apply method which replaces all the occurrence of type a in b and returns it.

- Compose
  Compose is just the composition of two substitutions with private attributes f and g. So its apply method just returns f.apply(g.apply(Type b)).

4.2 Type Class

We create different class to represent different kinds of types. All of them are the subclass of class Type. All class type has the following method, and I will briefly introduce its usage.

- isEqualityType()
  This function returns true or false, indicating whether this class is equality type, as illustrated in specification.

- replace(TypeVar a, Type t)
  This function returns the type of current type after replacing all the occurrence of a with t.

- contains(TypeVar tv)
  This function checks whether the expression contains type variable tv.

- unify(Type t)
  Generates the corresponding substitution of constrain {this = t}.

The implementation of the first three function is very simple and I don’ t want to talk about them in detail. The implementation of unify(Type t)

4.3 Unification

The solution of type constraint during type inference depends on unification. That is, for each type constraint, the unification gives us a substitution that solves the constraint. Since the constraint is between two types, our unification function is implemented as method function of different type class. Different type class will have different unify method, as show below: (here all unify function has the same input Type t)

- TypeVar

For type variables there are three cases we need to consider, basically following the unification rule for type variable:

```
--------------------------------------------- (a not in FV(s)) (u-var1)
(S,{a=s} U q) -> ([a=s] o S, q[s/a])


--------------------------------------------- (a not in FV(s)) (u-var2)
(S,{s=a} U q) -> ([a=s] o S, q[s/a])
```

1. T is a type variable. Then we can either return a substitution replacing t with this or this with t.
2. T is other type but contains this, which means TypeCircularityError.
3. T is other type and does not contain this. We simple return the substitution which replace this with T.

- ArrowType (with private attributes t1 and t2, meaning t1->t2)

First let' s take a look at the unification rule of function:

```
-----------------------------------------------  (u-fun)
(S, {s11 -> s12= s21 -> s22} U q) ->
(S, {s11 = s21, s12 = s22} U q)
```

It indicates that if t is an arrow type, we just need to unify t1 with t.t1 and unify t2 with t.t2 and combine the substitution generated by the second unification the the substitution of the first one. Pay attention the unifying t2 with t.t2, we should first apply them with the substitution of the first unification because of the unification rule of type variable:

```
--------------------------------------------- (a not in FV(s)) (u-var1)
(S,{a=s} U q) -> ([a=s] o S, q[s/a])


--------------------------------------------- (a not in FV(s)) (u-var2)
(S,{s=a} U q) -> ([a=s] o S, q[s/a])
```

Or if t is a type variable, we call t.unify(this) to let the unification of type variable do the job. Otherwise there will be a type error.

- BoolType

There are 3 situations:

1. T is type variable. We let t.unify(this) do the job.
2. T is BOOL. We simply return identity substitution.
3. Otherwise there is a TypeError.

- IntType

  There are 3 situations:

  1. T is type variable. We let t.unify(this) do the job.
  2. T is INT. We simply return identity substitution.
  3. Otherwise there is a TypeError.

- ListType (with private attributes t, meaning List t)

  There are 3 situations:

  1. T is type variable. We let t.unify(this) do the job.
  2. T is ListType. We return the substitution of unification between this.t and t.t.
  3. Otherwise there is a TypeError.

- PairType (with private attributes t1, t2, meaning t1×t2)

  There are 3 situations:

  1. T is type variable. We let t.unify(this) do the job.
  2. T is PairType. This situation is the same as ArrowType.
  3. Otherwise there is a TypeError.

- RefType (with private attributes t, meaning Ref t)

  There are 3 situations:

  1. T is type variable. We let t.unify(this) do the job.
  2. T is RefType. We return the substitution of unification between this.t and t.t.
  3. Otherwise there is a TypeError.

- UnityType

  There are 3 situations:

  1. T is type variable. We let t.unify(this) do the job.
  2. T is UnitType. We simply return identity substitution.
  3. Otherwise there is a TypeError.

4.4 Type Environment

Type environment includes a lot of binding between a symbol and its type. Such type environment is often enlarged when we are checking the type of condition expression or application expression. Because we have to keep record of the type of the symbol we introduce in such expression.

As the initial environment, we have to add the type of pre-defined function binding including :

1. fst: PairType->TypeVar
2. snd: PairType->TypeVar
3. hd: ListType->TypeVar
4. tl: ListType->TypeVar
5. iszero: INT->BOOL
6. succ: INT->INT
7. prec: INT->INT

4.5 Type Inference

The core of type inference is the function typecheck(TypeEnv E). This function is a method of class Expr, which is used to return the type of an expression and also the principle substitution generated by the constraint derived from the expression under environment E. Different expression class will have different typecheck method. The whole process of typecheck is recursive, which means that we will first generate the type and the principle substitution of the sub-expression, then generate the new type and the principle substitution of the expression of the current object according to the previous substitution and newly added constraints. The theoretical foundation of generating constraints is given from homework 5, some of which is shown below:

$$\frac{G \vdash u_1 \Rightarrow e_1 : t_1, q_1 \quad G \vdash u_2 \Rightarrow e_2 : t_2, q_2}{G \vdash u_1\ u_2 \Rightarrow e_1\ e_2 : a, q_1 \cup q_2 \cup \{t_1 = t_2 \to a\}} \quad \text{(CT-App)}$$

$$\frac{G \vdash u_1 \Rightarrow e_1 : t_1, q_1 \quad G \vdash u_2 \Rightarrow e_2 : t_2, q_2}{G \vdash u_1\ bop\ u_2 \Rightarrow e_1\ bop\ e_2 : a, q_1 \cup q_2 \cup \{t_1 = t_2 = a\}} \quad \text{(CT-Bop)}$$

$$\frac{G \vdash u \Rightarrow e : t, q}{G \vdash uop\ u \Rightarrow uop\ e : a, q \cup \{t = a\}} \quad \text{(CT-Uop)}$$

$$\frac{G \vdash u_1 \Rightarrow e_1 : t_1, q_1 \quad G, x : t_1 \vdash u_2 \Rightarrow e_2 : t_2, q_2}{G \vdash let\ x = u_1\ in\ u_2 \Rightarrow let\ x = e_1\ in\ e_2 : a, q_1 \cup q_2 \cup \{t_2 = a\}} \quad \text{(CT-Let)}$$

$$\frac{G \vdash u_1 \Rightarrow e_1 : t_1, q_1 \quad G \vdash u_2 \Rightarrow e_2 : t_2, q_2 \quad G \vdash u_3 \Rightarrow e_3 : t_3, q_3}{G \vdash if\ u_1\ then\ u_2\ else\ u_3 \Rightarrow\ if\ e_1\ then\ e_2\ else\ e_3 : a,} \quad \text{(CT-If)}$$
$$q_1 \cup q_2 \cup q_3 \cup \{t_1 = bool, t_2 = t_3 = a\}$$

$$\frac{G, f : a \to b, x : a \vdash u_1 \Rightarrow e_1 : t_1, q_1 \quad G, f(x) : b \vdash u_2 \Rightarrow e_2 : t_2, q_2}{G \vdash letfun\ f(x) = u_1\ in\ u_2 \Rightarrow letfun\ f(x : a) : b = e_1\ in\ e_2 : c,} \quad \text{(CT-Letfun)}$$
$$q_1 \cup q_2 \cup q_3\ \{t_1 = b, t_2 = c\}$$

$$\frac{G \vdash u_1 \Rightarrow e_1 : t_1, q_1 \quad G \vdash u_2 \Rightarrow e_2 : t_2, q_2}{G \vdash \{u_1, u_2\} \Rightarrow \{e_1, e_2\} : a * b, q_1 \cup q_2 \cup \{t_1 = a, t_2 = b\}} \quad \text{(CT-Pair)}$$

Also, the process of generating the substitution Snew of the current object from the substitution of sub-expression Sold is just by composing Sold and the substitution generated by the additional constraint of the current object. I will illustrate the implementation of each expression:

· A bop B, bop∈{+,-,*,/,%}, with environment E

We first generate the substitution S1 and type T1 of A under the environment E by call A.typecheck(E). Then before generating the substitution S2 of B, we need to pay attention to the process of unification, as shown below:

```
-------------------------------------------- (a not in FV(s)) (u-var1)
(S,{a=s} U q) -> ([a=s] o S, q[s/a])


-------------------------------------------- (a not in FV(s)) (u-var2)
(S,{s=a} U q) -> ([a=s] o S, q[s/a])
```

The process of unification indicates that after we have unified all the constraint of A, the constraint of B need to be applied by the substitution S1 of A. So instead of calling B.typecheck(E),we call B.typecheck(S1.compose(E)) and get the substitution S2 and type T2. S1.compose(E) applies all the type and type variables in E by S1. And according to the type constraint rule, we need to handle two more constraints, which is {T1 = int} and {T2 = int}, as is shown below:

```
G |- u1 ==> e1 : t1, q1 G |- u2 ==> e2 : t2, q2
---------------------------------------------------------------------
G |- u1 + u2 ==> e1 + e2 : int, q1 U q2 U {t1 = int, t2 = int}
```

For each type class of type T, we have a method T.unify(U) to generate the substitution corresponding to the constraint {T = U}. So here we use (S2oS1(T1)).unity(INT) to generate the first substitution St1 ('o' means compose). Then we move on to the next constraint, using: (St1oS2oS1(T1)).unity(INT) to generate the second constraint St2 . So the final substitution of A op B should be St2oSt1oS2oS1, and the type should be INT.

- ref E, with environment E

  The substitution of ref e is exactly the substitution of e, and the type is ref type with the type of e.

- e1:=e2, with environment E

  To get the substitution of constraint generated by e1 and e2, we call e1.typecheck(E) and e2.typecheck(S1.compose(E)), which is exactly the same as A bop B. And since the additional constraint of e1:=e2 is {T1 = ref T2}, we then call:
  S2oS1(T1).unify(new Ref (S2oS1(T2))) to get the substitution S3 of that constraint. So the final substitution of e1:=e2 should be S3oS2oS1, with type UNIT.

- A bop B, bop∈{<,<=,>,>=}, with environment E

The procedure is almost the same as A bop B, bop∈{+,-,*,/,%}, except that the type of the expression should be BOOL. The type constraint rule is shown below:

- !e, with environment E

  We first call e.typecheck(E) to get the type T1 of e and the substitution S1 of constraint generated by e. Then according to the type rule, we know that the additional constraint should be {T1 = Ref T2}. So we create a new type variable T2 and a new Ref type using T2. Then we call S1(T1).unify(new Ref(T2)) to get the substitution S2 of the additional constraint. The final substitution of !e should be S2oS1, with type T2.

- A bop B, bop∈{=,<>}, with environment E

  The procedure is almost the same as A bop B, bop∈{+,-,*,/,%}, except that the additional constraint should be {T1 = T2}, which creates substitution S3. The final substitution of !e should be S3oS2oS1, with type BOOL. Pay attention that the procedure can go correctly only if both T1 and T2 are equality type, otherwise we will throw an error.

- (e1,e2) , with environment E

  The procedure is almost the same as A bop B, bop∈{+,-,*,/,%}, except that there is no additional constraint. So the final substitution of !e should be S2oS1, with type Pair(S2oS1(T1),S2oS1(T2)

- e1 andalso e2, with environment E

  This procedure is the same as A bop B, bop∈{+,-,*,/,%}, except the additional constraint should be {T1=BOOL}, {T2=BOOL}, which creates substitution S3 and S4. So the final substitution of e1 andalso e2 should be S4oS3oS2oS1, with type BOOL.

- e1 orelse e2, with environment E

  This procedure is the same as e1 andalso e2.

- e1::e2, with environment E

  The procedure is almost the same as A bop B, bop∈{+,-,*,/,%}, except that the additional constraint should be {T2 = List T1}, which generates substitution S3. The final result should be S3oS2oS1, and the final type should be T2.

- e1 e2, with environment E

  The procedure is almost the same as A bop B, bop∈{+,-,*,/,%}, except that the additional constraint should be {T1 = T2->Tv}, where Tv is a newly generated type variable. Let the substitution of the unification of the additional constraint be S3. The final substitution should be S3oS2oS1 and the type should be S3oS2oS1(Tv).

- if e1 then e2 else e3, with environment E
  We first get the type T1 and substitution S1 of e1 with environment E. Then we get type T2 and the substitution S2 of e2 with environment S1.apply(E). Finally we get type T3 and the substitution S3 of e3 with environment (S2oS1).apply(E). And the additional constraint of condition expression is {T1 = BOOL} and {T2 = T3}, the unification of which will generate substitution S4 and S5. Notice that the type on both side of the unification should be applied by all composition of the previous substitution.

- let x =e1 in e2 end, with environment E
  We first get the type T1 and substitution S1 of e1 with environment E. Then we get the type T2 and substitution S2 of e2. Here the environment should be E applied by S1 and added by a new symbol x with type T1. The final substitution should be S2oS1 and and type should be S2oS1(T2).

- while e1 do e2, with environment E
  The procedure is almost the same as A bop B, bop∈{+,-,*,/,%}, except that the additional constraint should be {T1 = BOOL}. The final substitution should be S2oS1 and the final type should be UNIT.

5 Semantics

5.1 State

In our implementation, state is a class with 3 attributes: Environment E, Memory M and Int P. We will introduce each piece in detail.

· Environment

This is different from the previous type environment. Here the environment records the binding between symbol and values.

The environment is implemented like a stack, in a recursive way. Each environment records the binding at the top of the stack using attributes x and v. The stack below the top is stored in another Environment instance, and the current environment has an attribute E pointing to it.

The method get(Symbol y) of Environment checks whether the current environment has the bounding of y at the top, if not, then it goes to the environment below top, which is stored in this.E, to check the existence of y recursively.

· Memory

Class Mem is implemented using a hash table in our program. The index of the has table indicates the memory address while the value in certain cell indicated by the index mimics the storage of values in memory. We can use the default method of has table: get and put to get or update the element of memory.

· Int

Int is a class indicating an interger. The use of Int p in Class state is to show the next available place for storing value in memory, which means the next empty space we can use to store a value in memory.

5.2 Classes of Values

we use different classes to represent different kinds of values, as proposed in specification. All of them are the subclass of class Value.

Each subclass of Class Value has the following functions: the construction function, the toString function (because we need to print these value) and the equals function. The equals(Object other) function returns whether the current instance is equal to the Value class other according to the equality comparison in specification. The implementation of equals(Object other) is easy so I won' t illustrate them in detail. I will briefly introduce each value class.

- BoolValue

This class has a Boolean attribute b, representing the value of this BoolValue.

- ConsValue

This class has attributes v1 and v2, meaning that the value is v1::v2. It also has an attribute len, used to record the length of the list.

·FunValue

This class has attributes E, x, e, representing the environment, the symbol of the input parameter, and the body of the function.

- IntValue

This class has attribute n, representing the actual number.

- NilValue

This class has no attribute. But the equal(Object other) function will only return true if other is an instance of NilValue.

- PairValue

This class has attributes v1 and v2, meaning that the value is (v1, v2).

- RecValue

This class has attributes E, x, e, representing the environment, the name of the recursion function, and the body of the recursion.

- RefValue

  This class has int attributes p, representing the memory address.
- UnitValue

  This class has no attribute. But the equal(Object other) function will only return true if other is an instance of NilValue.

5.3 Evaluation according to Rules

eval(state s) is a method of class Expr, it evaluates the instance of its class and returns a value, which is a subclass of Class Value (we also create classes for different kinds of value).

Generally, the process of evaluation is also recursive. We first evaluate the value of its sub-expression, then get the value according to the evaluation rule.

Different expression has different eval(state s) method. I will illustrate each of them in detail.

- Add/Sub/Mul/Div/Mod (l +/-/*///% r)

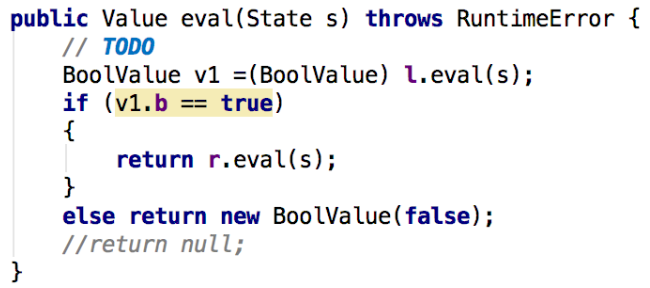
  According to rule E-Add, E-Sub, E-Mul, E-Div, E-Mod, we just need to evaluate the value of its sub-expression l and r by l.eval(s) and r.eval(s), check whether the return value is IntValue, add/sub/mul/div/mod the attribute of the two IntValue and return an IntValue with the calculated attribute. If anything goes wrong during the process, we throw runtime error. Notice that if the divisor is zero during div or mod, we also throw runtime error.

- AndAlso (l andalso r)

$$\frac{E,M,p;e_1 \Downarrow M',p';\mathbf{tt} \quad E,M',p';e_2 \Downarrow M'',p'';v}{E,M,p;e_1\ \texttt{andalso}\ e_2 \Downarrow M'',p'';v} \qquad \text{(E-AndAlso1)}$$

$$\frac{E,M,p;e_1 \Downarrow M',p';\mathbf{ff}}{E,M,p;e_1\ \texttt{andalso}\ e_2 \Downarrow M',p';\mathbf{ff}} \qquad \text{(E-AndAlso2)}$$

According to the rule of AndAlso, we need to first evaluate the value of v1. Then we make sure that v1 is BoolValue. Finally return the evaluation of r if v1 is true, otherwise return false. If anything goes wrong during the process, we will throw run time error.



```
public Value eval(State s) throws RuntimeError {
    // TODO
    BoolValue v1 =(BoolValue) l.eval(s);
    if (v1.b == true)
    {
        return r.eval(s);
    }
    else return new BoolValue(false);
    //return null;
}
```

- App (l r)

According to the rule of E-App, we first need to evaluate the value V of sub-expression r. Then the sub-expression l should be evaluated into a FunValue. Finally we evaluate and return the body(the attribute e) of that FunValue with the environment enlarged by the symbol-value binding between f.x and V. If anything goes wrong during the process, we will throw run time error.

- Assign (l := r)

According to the rule of E-Assign, we first evaluate the value V1 of sub-expression l and value V2 of sub-expression r. V1 should have type RefValue and we set the memory cell with index of V1.p to be V2. And the function returns a UNIT value. If anything goes wrong during the process, we will throw run time error.

- Condition (if e1 then e2 else e3)

According to the rule of E-Cond, we first evaluate the value of e1. If e1 is BoolValue and e1 is true, then we evaluate e2 and return its value, else we evaluate e3 and return its value. If anything goes wrong during the process, we will throw run time error.

- Cons (l :: r)

According to the rule of E-Cons, we simply return the ConsValue(l, r).

- Deref (!e)

According to the rule E-Deref, we first evaluate the value V of sub-expression and make sure it is of type RefValue. The we return the value stored in the memory cell with index V.p.

- Eq (l = r)

According to the rule E-Eq, we just need to get the value V1 after evaluating l and value V2 after evaluating r. Then call V1.equals(V2). If anything goes wrong during the process, we will throw run time error.

- Fn (fn x=> e)

According to the rule E-Fn, we just need to return the FunValue with (current environment s.E, x and e). If anything goes wrong during the process, we will throw run time error.

- Greater/GreaterEq/Less/LessEq (l >/>=/</<=/ r)

According to the rule E-Greater/ E-GreaterEq/ E-Greater, we first get the value V1 of l and V2 of r by calling eval method. Then we check whether V1 and V2 are IntValue, if so, we compare their attribute n and return the comparison result using BoolValue. If anything goes wrong during the process, we will throw run time error.

• Group ((e))

According to the rule E-Group, we simply return the value of e after evaluating e. If anything goes wrong during the process, we will throw run time error.

• Let (let x = e1 in e2 end)

According to the rule E-Let, we first evaluate the value V1 of sub-expression e1. Then return the value after evaluating e2 with the enlarged environment which contains the additional symbol-value binding between x and V1. If anything goes wrong during the process, we will throw run time error.

• Loop (while e1 do e2)

According to the rule E-Loop, we first evaluate the value V1 of e1 and make sure V1 is a BoolValue. Then if V1 is true then we further evaluate and return the value of a sequence: e2; while e1 do e2. It can be achieved by forming a new Seq expression with e2 and Loop. Else if V1 is false, we simply return UnitValue.

• Name (x)

According to the rule E-Name, there are 2 cases:

1. The value V of x in the current environment in state s is RecValue, then we form a recursion expression (Class Rec) with the symbol V.x and body V.e, evaluate the expression and return its value.
2. Otherwise we simply return the value V of x in the current environment in state s.

If the binding for x is not found in current environment, we throw runtime error.

• Neg (-e)

According to the rule E-Neg, we first evaluate the value V of e. Make sure it is of type IntValue and return a new IntValue which has the negative attribute –n of the attribute n of V.

• Neq (e1 != e2)

The process is almost the same as the eq expression, except the return should be negative of the result of eq.

• Nil (nil)

According to the rule E-Nil, we simply return NilValue

• Not (!e)

The process is almost the same as the Neg expression, except that the value V of e should be of type BoolValue and the return value is the negative of V.

- OrElse (l orelse r)

The process is almost the same as the AndAlso expression, except there are some slight difference in the control flow.

- Pair ((e1, e2))

According to the rule E-Pair, we simply return the PairValue after evaluating e1 and e2.

- Rec (rec x=> e)

According to the rule E-Rec, we should evaluate the body e within the enlarged environment with additional binding between symbol x and RecValue of the previous environment, x and e), and return its value.

- Ref (ref e)

According to the rule E-ref, we add the pointer p of the state s by one (meaning the current pointer is not pointed to empty memory cell) and store the original pointer in variable tmp. We then evaluate e and get its result V. After that we set the memory cell with V at the index tmp. Finally we return UnitValue.

- Seq (l; r)

According to the rule E-Seq, we simply evaluate the value V1 of l and the value V2 of r. And return V2.

- Unit

Simply return a UnitValue. Here for the sake of saving space, we use a static variable Value.Unit to represent all the UnitValue so that we don’ t need to create a new instance of UnitVale each time as we return.

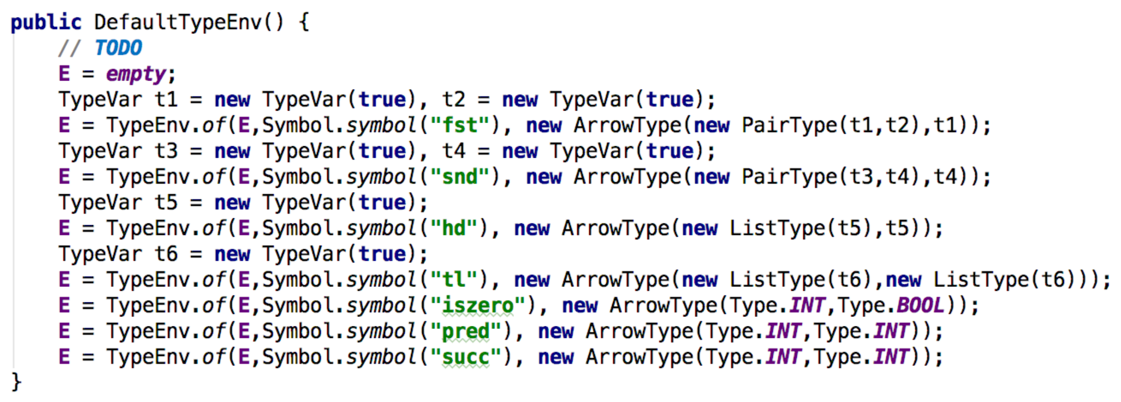
6 Implementation

6.1 Command-line Interface

To receive the file from the command-line, we simply call interpret(args[0]); in the main method in interpreter.

6.2 Predefined Functions

All the types of predefined functions are defined in default type environment as illustrated in the specification.



```
public DefaultTypeEnv() {
    // TODO
    E = empty;
    TypeVar t1 = new TypeVar(true), t2 = new TypeVar(true);
    E = TypeEnv.of(E,Symbol.symbol("fst"), new ArrowType(new PairType(t1,t2),t1));
    TypeVar t3 = new TypeVar(true), t4 = new TypeVar(true);
    E = TypeEnv.of(E,Symbol.symbol("snd"), new ArrowType(new PairType(t3,t4),t4));
    TypeVar t5 = new TypeVar(true);
    E = TypeEnv.of(E,Symbol.symbol("hd"), new ArrowType(new ListType(t5),t5));
    TypeVar t6 = new TypeVar(true);
    E = TypeEnv.of(E,Symbol.symbol("tl"), new ArrowType(new ListType(t6),new ListType(t6)));
    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));
}
```

Then we show the implementation of such functions.

- fst

  fst is defined as a FunValue which stores its environment, argument name, and body.

  For environment, since the function is pre-defined, so the emironment is just empty environment.

  For argument name, we can define it as “fstx”.

  For the body, we create a new subclass of Expr, the eval(state s) of which is to get the value the input argument (“fstx”) represents, transform it into PairValue and return its first value.

- snd

  snd is almost the same as fst except that it returns the second value of PairValue.

- hd

  hd is almost the same as fst except that it transforms the value into ConsValue and return its first value.

- tl

  tl is almost the same as fst except that it transforms the value into ConsValue and return its second value.

- iszero

  iszero is defined as a FunValue which stores its environment, argument name, and body.

  For environment, since the function is pre-defined, so the emironment is just empty environment.

  For argument name, we can define it as “iszx”.

For the body, we can create it by creating a condition expression which checks whether "iszx" is zero and return true if it is, return false if not.

- pred

  The body of pred is simply a Sub expression with argument and 1.

- succ

  The body of pred is simply an Add expression with argument and 1.

6.3 Other implementation

I have fully illustrated all other implementation of the primitive interpreter in chapter 4 and 5.

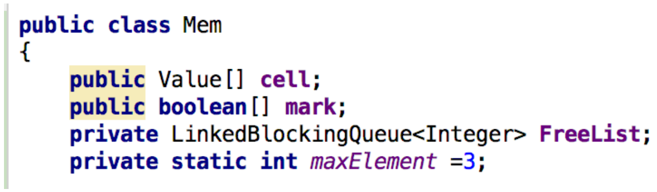
7 Bonus

7.1 Garbage collection (of ref cells)

1. implementation

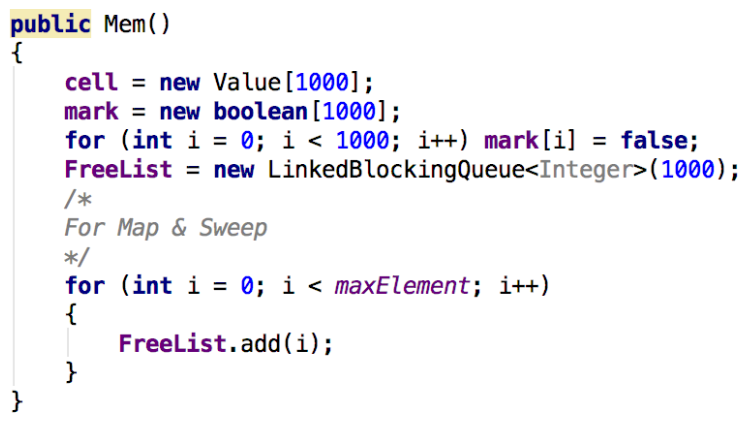
Nowadays the most popular ways used for garbage collection in functional programming language is mark-and-sweep. Generally, mark-and-sweep is divided into two passes:
Pass I: Mark all nodes that are (directly or indirectly) accessible from the memory by setting their MB=1.
Pass II: Sweep through the entire memory and return all unmarked (MB=0) nodes to the free list.

The implementation of Memory
To make mark-and-sweep possible I rewrite the entire Mem class. Instead of a hash table, Mem class has an Value array called cell, a Boolean array called mark, an integer queue called FreeList and a static int variable maxElement. Cell array is used to store the value according to the memory address(index). Mark array is used to record the MB of the memory at certain index. FreeList stores all the free place of the memory. MaxElement is the maximum number of values the memory can store. The code-level implementation is shown below:



```
public class Mem
{
    public Value[] cell;
    public boolean[] mark;
    private LinkedBlockingQueue<Integer> FreeList;
    private static int maxElement =3;
```

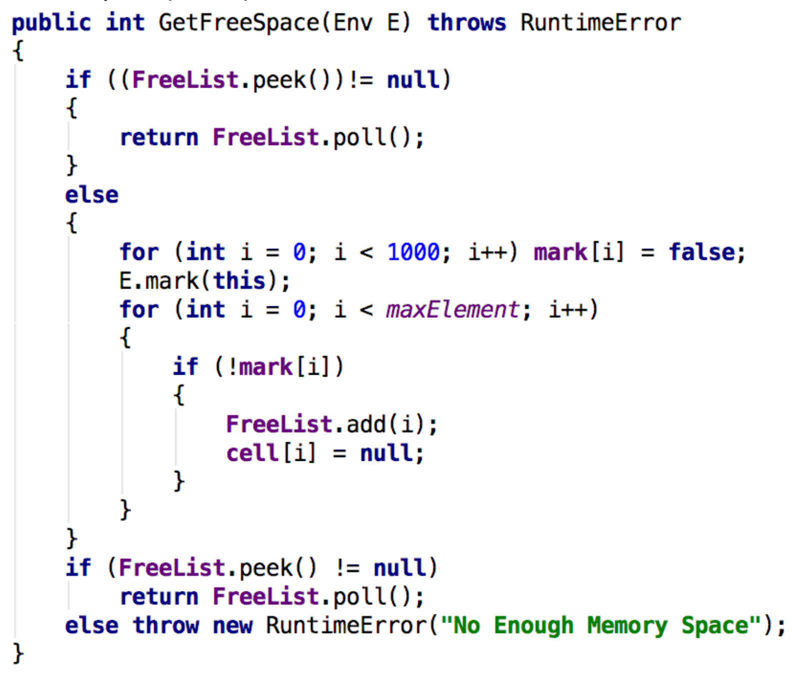
Initialization



```
public Mem()
{
    cell = new Value[1000];
    mark = new boolean[1000];
    for (int i = 0; i < 1000; i++) mark[i] = false;
    FreeList = new LinkedBlockingQueue<Integer>(1000);
    /*
    For Map & Sweep
    */
    for (int i = 0; i < maxElement; i++)
    {
        FreeList.add(i);
    }
}
```

As we can see, mark array is initialized as false according to the mark-and-sweep policy. And FreeList has all the index of the memory at first.

GetFreeSpace(Env E)



```
public int GetFreeSpace(Env E) throws RuntimeError
{
    if ((FreeList.peek())!= null)
    {
        return FreeList.poll();
    }
    else
    {
        for (int i = 0; i < 1000; i++) mark[i] = false;
        E.mark(this);
        for (int i = 0; i < maxElement; i++)
        {
            if (!mark[i])
            {
                FreeList.add(i);
                cell[i] = null;
            }
        }
    }
    if (FreeList.peek() != null)
        return FreeList.poll();
    else throw new RuntimeError("No Enough Memory Space");
}
```

This method is used to return the next empty place in memory for storing value. If the FreeList is not empty, we simple dequeue one index from FreeList and regard it as the index for the empty space. Else if the FreeList is empty then we should trigger the mark-and-sweep procedure. We first initialize all the element in mark array as false. Then mark the mark array using a method mark in E. E.mark is also a method I add to implement the mark-and-sweep. Its implementation is shown as below:

```
public void mark(Mem M)
{
    v.mark(M);
    E.mark(M);
}
```

Basically the idea is to mark all the place indicated by the RefVaalue. But since we need to detect all the direct or indirect reachability, we must look further into each to check the existence of RefValue. So we add a method mark to **each subclass of Value**. For example, the mark method for RefValue is:

```
@Override
public void mark(Mem M)
{
    M.mark[p] = true;
    v.mark(M);
}
```

While the mark method for value with recursive structure, like Pairs, is:

```
@Override
public void mark(Mem M)
{
    v1.mark(M);
    v2.mark(M);
}
```

After marking all the value that is reachable from environment, we simply add the index of cell marked false into FreeList and clear the cell. If the FreeList is still empty, we throw runtime error.

2. Example

I run the following program and set the maxElement, that is, **the maximum number of values a memory can store** as 5.

```
let x = ref ref 1999 in
    ref 12; ref 1777; ref 9999; ref 20000; (ref 99)::(ref 100)::nil
end
```

The memory state of each Ref evaluation is as follow:

```
 NULL 1999 NULL NULL NULL
 ref@1999 1999 NULL NULL NULL
 ref@1999 1999 12 NULL NULL
 ref@1999 1999 12 1777 NULL
 ref@1999 1999 12 1777 9999
 ref@1999 1999 20000 NULL NULL
 ref@1999 1999 20000 99 NULL
 ref@1999 1999 20000 99 100
```

We can see that when we add the 6th element 20000 to the memory, the memory is overflow. The FreeList is empty. So the mark-and-sweep is triggered and since 12, 1777 and 9999 are not reachable from the environment, they are cleared. However, ref 1999 (directly reachable) and 1999 (indirectly reachable) should be marked and not be cleared.

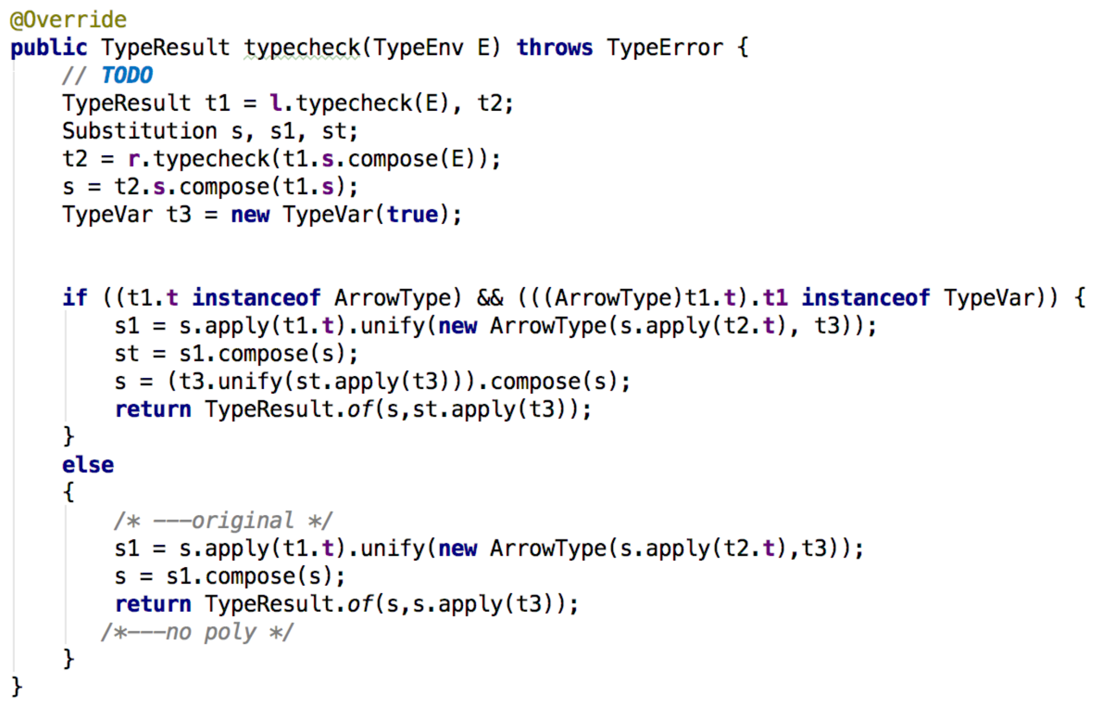
7.2 Polymorphic type

1. implementation

Here the polymorphic type means "Infer polymorphic type for functions", which means a function can accept different types of arguments. To implement polymorphic type, we only need to modify the type inference.

Instead of creating a new type to represent any type, we simply use typeVar. Because these two are actually the same when considering polymorphism.

For the type inference part, we only modify the typecheck(TypeEnv E) for Application expression. As shown below:



```java
@Override
public TypeResult typecheck(TypeEnv E) throws TypeError {
    // TODO
    TypeResult t1 = l.typecheck(E), t2;
    Substitution s, s1, st;
    t2 = r.typecheck(t1.s.compose(E));
    s = t2.s.compose(t1.s);
    TypeVar t3 = new TypeVar(true);


    if ((t1.t instanceof ArrowType) && (((ArrowType)t1.t).t1 instanceof TypeVar)) {
        s1 = s.apply(t1.t).unify(new ArrowType(s.apply(t2.t), t3));
        st = s1.compose(s);
        s = (t3.unify(st.apply(t3))).compose(s);
        return TypeResult.of(s,st.apply(t3));
    }
    else
    {
        /* ---original */
        s1 = s.apply(t1.t).unify(new ArrowType(s.apply(t2.t),t3));
        s = s1.compose(s);
        return TypeResult.of(s,s.apply(t3));
      /*---no poly */
    }
}
```

In the original version, the application will generate a constraint:

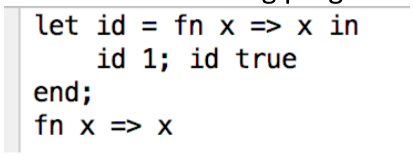
$$\frac{G \vdash u_1 \Rightarrow e_1 : t_1, q_1 \quad G \vdash u_2 \Rightarrow e_2 : t_2, q_2}{G \vdash u_1\ u_2 \Rightarrow e_1\ e_2 : a, q_1 \cup q_2 \cup \{t_1 = t_2 \rightarrow a\}}$$

Actually it is **this constraint** that fixes the type of a function. So we simply do not compose the substitution generated by that constraint with the final substitution. But we need the composed substitution to get the correct type of the result of application. So we use an additional substitution st to record the composed substitution, which is the final substitution in the original version, to get the correct type of application. Since in the code we use a new

type variable t3 to present the type of the application and also in order to make the final substitution able to tell the type of t3, we manually compose the final substitution with:
[t3 = st(t3)] in the end.

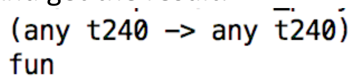
2. Example

We run the following program:



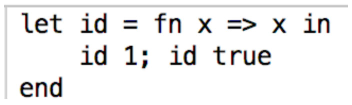
```
let id = fn x => x in
    id 1; id true
end;
fn x => x
```

And get the result:



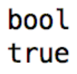
```
(any t240 -> any t240)
fun
```

We can see that the type of the function is polymorphic. To know the result of the let expression, we run:



```
let id = fn x => x in
    id 1; id true
end
```

The result is:



```
bool
true
```

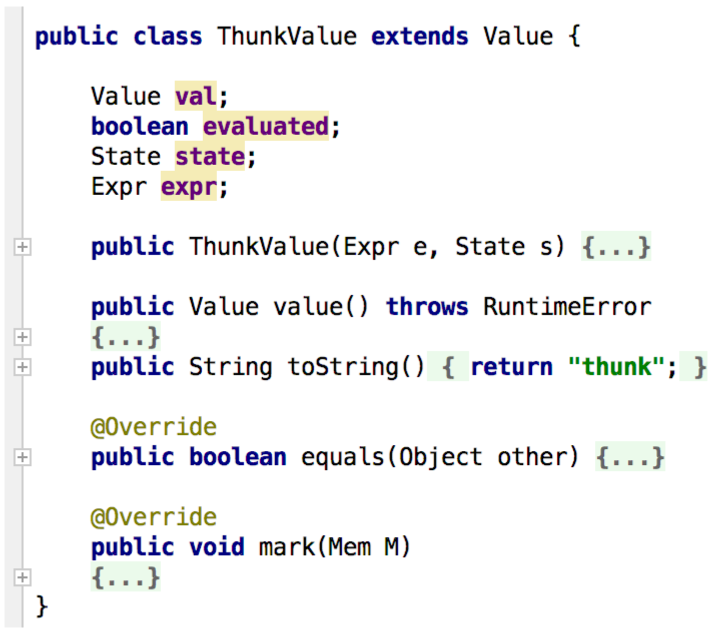
Everything goes smoothly.

7.3 Lazy evaluation

1. Implementation

Lazy evaluation is a technique to delay evaluation until it is needed. The implementation of lazy evaluation is carried out by modifying the eval(state s) method, adding a new method called forceEval(state s), and adding a new value type called Thunk.

The implementation of class Thunk is shown below:

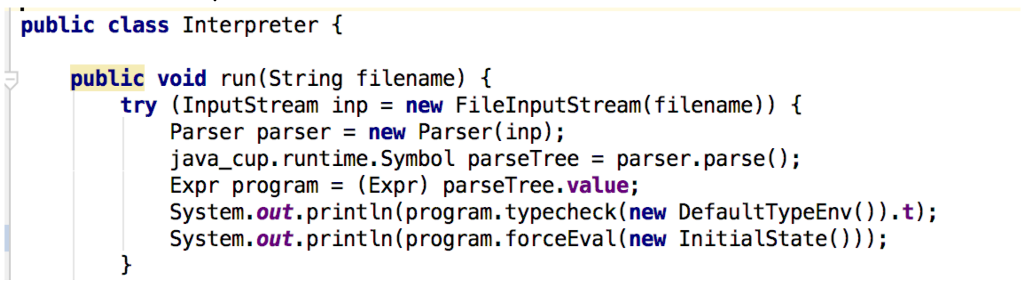


```
public class ThunkValue extends Value {

    Value val;
    boolean evaluated;
    State state;
    Expr expr;

    public ThunkValue(Expr e, State s) {...}

    public Value value() throws RuntimeError
    {...}
    public String toString() { return "thunk"; }

    @Override
    public boolean equals(Object other) {...}

    @Override
    public void mark(Mem M)
    {...}
}
```

Thunk is used to store the expression whose evaluation is delayed. And attribute expr is used to store that evaluation. Since we only need to evaluate the Thunk once, we need an attribute called evaluated to indicate whether this Thunk is evaluated. The value of the Thunk is stored in attribute val.

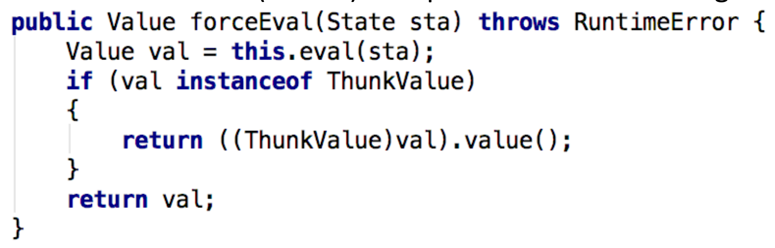
To implement lazy evaluation, we add a method called forceEval(state s) to Expr class. The forceEval is used to force a Thunk value to be further evaluated to other values other than Thunk while the previous evaluation method eval(state s) will regard Thunk as an ordinary value that cannot be further evaluated.

Method forceEval(state s) is first called at the top of the evaluation, which is in the main method of Interpreter.



```
public class Interpreter {

    public void run(String filename) {
        try (InputStream inp = new FileInputStream(filename)) {
            Parser parser = new Parser(inp);
            java_cup.runtime.Symbol parseTree = parser.parse();
            Expr program = (Expr) parseTree.value;
            System.out.println(program.typecheck(new DefaultTypeEnv()).t);
            System.out.println(program.forceEval(new InitialState()));
        }
```

The method forceEval(state s) is implemented as following:



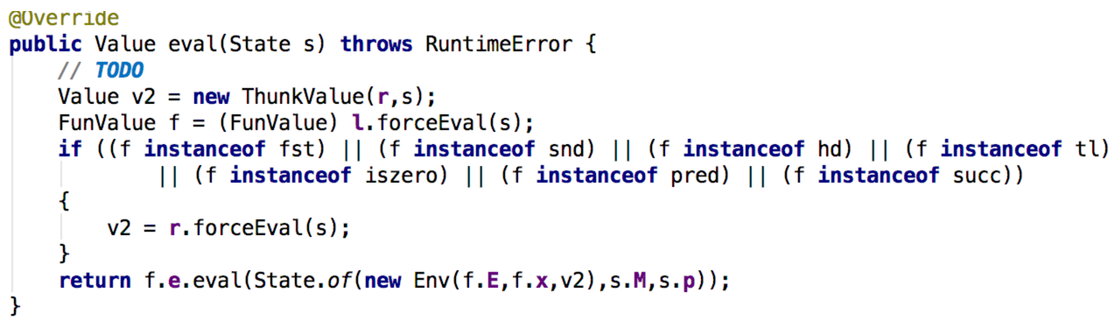
```
public Value forceEval(State sta) throws RuntimeError {
    Value val = this.eval(sta);
    if (val instanceof ThunkValue)
    {
        return ((ThunkValue)val).value();
    }
    return val;
}
```

In forceEval(state s), we first do the tradition evaluation, and if we find that the returned value is a Thunk, then it force evaluate the expression in this Thunk by calling value() method.

The crucial part of lazy evaluation lies in the evaluation function of application expression. Instead of evaluating both sub-expressions l and r. The forceEval(state s) method **only force evaluate the l sub-expression**. For the r sub-expression, we **create a Thunk value to store it**. The reason why we do this can be illustrated with the following example:

Suppose we have a constant function f, which is fn x=>1, and we want to know the result of the application f (1000!). The primitive interpreter will evaluate the value of 1000! before application. But obviously we do not need to evaluate the 1000! Because f is not related to the input argument. In our interpreter with lazy-evaluation, 1000! will be transformed into a Thunk value. And and we can still get the result 1 because the actual value of x is not used during the evaluation of the body of the function.

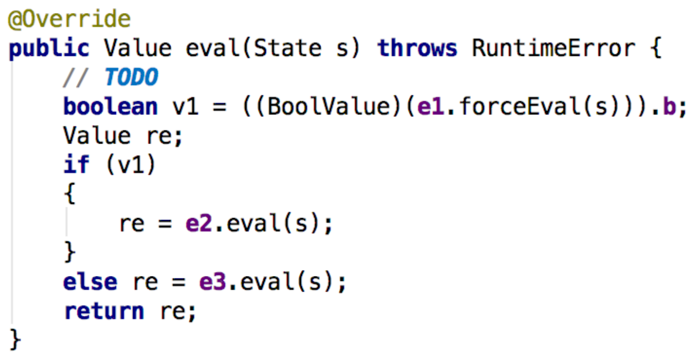
Based on the previous idea, we can easily modify the evaluation of application expression as follows:



```
@Override
public Value eval(State s) throws RuntimeError {
    // TODO
    Value v2 = new ThunkValue(r,s);
    FunValue f = (FunValue) l.forceEval(s);
    if ((f instanceof fst) || (f instanceof snd) || (f instanceof hd) || (f instanceof tl)
            || (f instanceof iszero) || (f instanceof pred) || (f instanceof succ))
    {
        v2 = r.forceEval(s);
    }
    return f.e.eval(State.of(new Env(f.E,f.x,v2),s.M,s.p));
}
```

Except for the application, there are still the evaluation method of some other expression that we need to modify.

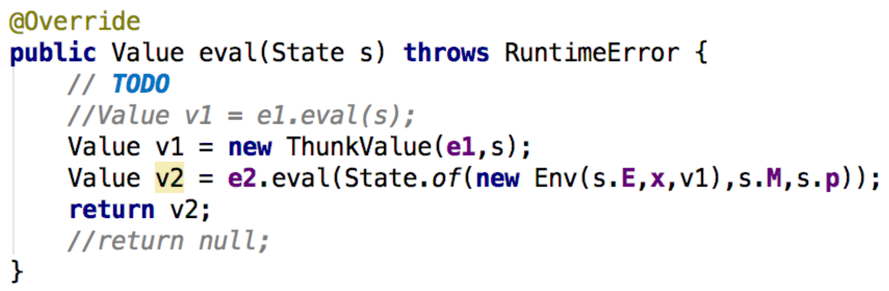
· Condition (if e1 then e2 else e3)



```
@Override
public Value eval(State s) throws RuntimeError {
    // TODO
    boolean v1 = ((BoolValue)(e1.forceEval(s))).b;
    Value re;
    if (v1)
    {
        re = e2.eval(s);
    }
    else re = e3.eval(s);
    return re;
}
```

We have to force evaluate the e1 because we need the actual value of e1 to decide whether to evaluate e2 or e3.

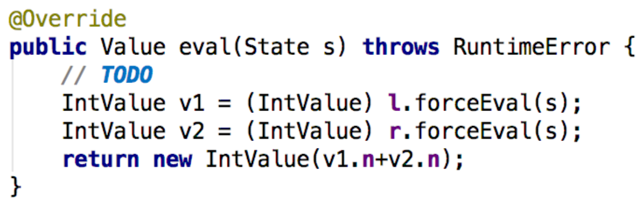
· let (let x = e1 in e2)



```
@Override
public Value eval(State s) throws RuntimeError {
    // TODO
    //Value v1 = e1.eval(s);
    Value v1 = new ThunkValue(e1,s);
    Value v2 = e2.eval(State.of(new Env(s.E,x,v1),s.M,s.p));
    return v2;
    //return null;
}
```

We can store transform e1 into a Thunk then directly evaluate e2 because we do not need the value of e1 right now. The evaluation of e1 can be delayed until the evaluation of e2.

· Other expression

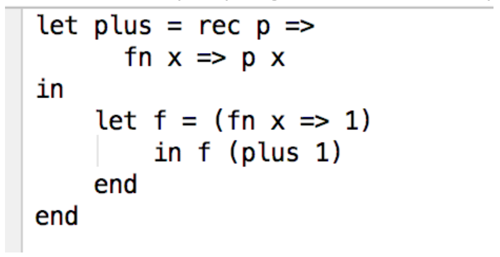


```
@Override
public Value eval(State s) throws RuntimeError {
    // TODO
    IntValue v1 = (IntValue) l.forceEval(s);
    IntValue v2 = (IntValue) r.forceEval(s);
    return new IntValue(v1.n+v2.n);
}
```

Above is the example of Add expresson. Since we cannot do arithmetic or Boolean operation on two Thunk together, we'll have to force evaluate the l and r expression. And all other expression also have their own properties such that we cannot delay their evaluation. So we simply replace all eval(state s) to forceEval(state s) in all those expressions.

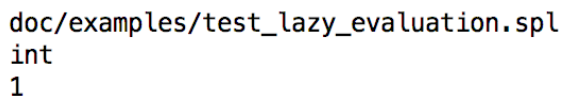
2. Example

We use a simple program as an example:



```
let plus = rec p =>
        fn x => p x
in
    let f = (fn x => 1)
        in f (plus 1)
    end
end
```

Obviously plus is a function with infinite recursion. So the primitive interpreter cannot give the value of this program. But in our lazy evaluation, the plus is regarded as a Thunk so that its value will not be evaluated in this program because function f is a constant function. The result is shown below:



```
doc/examples/test_lazy_evaluation.spl
int
1
```