CS383 Course Project Report

Name: Shi, Yu **Student ID: 5130309117**

1 Introduction

My interpreter follows the type checking algorithm strictly, generating a principle solution for type constraints. Also, it is equipped with polymorphism, which allows a polymorphic function to be applied to different type of arguments in one context. Also, I provide interpreter options like `-gc`, `-lazyval` and `-heapsize:size` allowing user to choose garbage collection, lazy evaluation and set the heap size. Finally, I expand the language to include sum type and case expressions.

2 Typing

First we should define the unification rules for each type. For example to unify a `PairType` with another type:

```
@Override
    public Substitution unify(Type t) throws TypeError {
        if(PairType.class.isInstance(t)) {
            PairType pt = (PairType)t;
            Substitution s = t1.unify(pt.t1);
            return s.apply(t2).unify(s.apply(pt.t2)).compose(s);
        }
        else if(TypeVar.class.isInstance(t))
            if(contains((TypeVar)t))
               throw new TypeCircularityError();
            else
               return Substitution.of((TypeVar)t, this);
        else
            throw new TypeError("Cannot unify " + this + " with " + t);
    }
```

If the type to be unified is also a `PairType`, then unify two subtypes. If the type to be unified is a type variable, we return a replacement to replace the type variable with the `PairType`. Otherwise a `TypeError` is thrown.

Typing is done in the `typecheck` method of class `Expr`. To implement typing for an expression precisely, we should

1. Type check every subexpression, meanwhile accumulate the genreated substitutions in the type environment.
2. Unify the types of subexpressions according to the typing rules of the expression, generating substitutions.
3. Return the type and composed substitutions to its parent expression.

To get a principle type result, there are two key points

1. Whenever a substitution is generated, integrate it into the current type environment by composing.

2. Whenever using an previously inferred type, apply the substitutions which generate after the type is inferred to the type.

For example, here's the code to check the type of an arithmetic expression.

```
1  @Override
2      public TypeResult typecheck(TypeEnv E) throws TypeError {
3          TypeResult lt = l.typecheck(E);
4          E = lt.s.compose(E);
5          TypeResult rt = r.typecheck(E);
6          E = rt.s.compose(E);
7
8          Substitution ls = rt.s.apply(lt.t).unify(Type.INT);
9          E = ls.compose(E);
10
11         Substitution rs = ls.apply(rt.t).unify(Type.INT);
12         E = rs.compose(E);
13
14         Substitution s = rs.compose(ls.compose(rt.s.compose(lt.s)));
15
16         TypeResult typeResult = TypeResult.of(s, Type.INT);
17         typeResult.mergeLocalTypeVars(lt.localTypeVars);
18         typeResult.mergeLocalTypeVars(rt.localTypeVars);
19         return typeResult;
20     }
```

In the 3rd line we infer the type of left subexpression, then in the 4th line we compose the resulting substitution into type environment immediately, according to the first key point. Similarly we can infer the type of right subexpression. Then in line 8 unify the left subexpression with `Int`. As specified by the second key point, since the substitution `rt.s` comes after the left subexpression is inferred, `lt.t` should integrate the effect of `rt.s` in line 8, as well as in later use.

In line 14 we compose all the substitution together and return it as part of the type result.

3 Semantics

In this interpreter, evaluation is the only task for semantics analysis. It is done by method `eval` of class `Expr`. Basically the process will be evaluating subexpressions and use their results to evaluate the parent expression. The `Env` should be controlled carefully. `Env` grows when entering the body of `fn, let` or `rec`, by adding new name-value binding, and shrinks when exiting. Here's the code for evaluation of function application.

```
1  @Override
2      public Value eval(State s) throws RuntimeError {
3          FunValue lf = (FunValue)l.eval(s);
4          Value rv = r.eval(s);
5          Env newE = new Env(lf.E, lf.x, rv);
6          int id = newE.id;
7          State tempState = State.of(newE, s.M, s.p, s.l.put(id, rv));
8          Value lv = lf.e.eval(tempState);
9          s.l.remove(id);
10         return lv;
11     }
```

In line 3 the left function is evaluated. Then evaluate the argument in line 4. In line 5, a new `Env` is created by binding the formal parameter of this function to the evaluated argument value. In the following lines we evaluate the body of function using this new `Env`.

For `let` and `rec` the principle is similar. Another critical point is `Ref`. This expression change the memory state directly.

```
1 @Override
2     public Value eval(State s) throws RuntimeError {
3         Value v = e.eval(s);
4         int tempP = s.M.getMemCell(v.size(), s.l, v);
5         s.M.put(tempP, v);
6         return new RefValue(tempP, v);
7     }
```

First evaluate the value to be stored into memory. In line 4 a request is sent to memory for allocating space. Because I implement garbage collection and allocate sizes for values, I need to use `getMemCell` to get the allocated location, which will explained in later part. Then store the value to allocated in line 5.

4 Polymorphic Type

Infer polymorphic type for functions, so that it can be applied to different types of arguments. However, it is not enough to infer the principle types of functions. Here's an example

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

if the inferred type of `id` should is $\alpha \to \alpha$. In `id 1`, we unify α with `int`, and the result is $\alpha =$ `int`. While in `id true`, the result is $\alpha =$ `bool`. Given this, we have to unify `int` with `bool`, which will cause a type mismatch.

What's going wrong here is that, in fact, the inferred type of `id` should not be $\alpha \to \alpha$, it should be $(\forall\alpha)\alpha \to \alpha$. (Pierce explained this in detail in his book.) And each time we apply fucntion `id`, we instantiate its type with a fresh type variable. In this case, when typing `id 1`, we instantiate $(\forall\alpha)\alpha \to \alpha$ as $\alpha_1 \to \alpha_1$, and unify α_1 with `int`. Then when typing `id true`, we instantiate $(\forall\alpha)\alpha \to \alpha$ as $\alpha_2 \to \alpha_2$ and unify α_2 with `bool`. Thus there is no conflict. With this idea in mind, again we have two key points to make a function polymorphic.

1. If the inferred type of a function `f` is $t(\alpha_1, ...\alpha_n)$, where $\alpha_1, ...\alpha_n$ are type variables. Then make all the type variables universal type, that is, make the resulting type $(\forall\alpha_1)...(\forall\alpha_n)t(\alpha_1, ...\alpha_n)$.

2. Each time when apply `f`, instantiate all the universal type variables with fresh type variables before typing the function application. That is instantiate $(\forall\alpha_1)...(\forall\alpha_n)t(\alpha_1, ...\alpha_n)$ as

$$t(\alpha_1^*, ...\alpha_n^*)$$

   where $\alpha_1^*, ..., \alpha_n^*$ are type variables haven't been used.

In this way, some originally untypable expressions can become typable, due to an enhanced flexibility of polymorphism. For example in `pcf.twice.spl`

```
let
  twice = fn f => fn x => f (f x)
in
  twice twice twice succ 0
end
```

The expression `twice twice` is untypable in the original strategy. However, after we introduce universal type, we can type it in following steps

To distinguish the `twice`'s, mark them as

$$\texttt{twice}_1\ \texttt{twice}_2\ \texttt{twice}_3\ \texttt{succ 0}$$

1. Inferred type of `twice`: $(\forall\alpha)(\alpha \to \alpha) \to \alpha \to \alpha$
2. $\texttt{twice}_1$: $(\alpha_1 \to \alpha_1) \to \alpha_1 \to \alpha_1$
3. $\texttt{twice}_2$: $(\alpha_2 \to \alpha_2) \to \alpha_2 \to \alpha_2$
4. $\texttt{twice}_1\ \texttt{twice}_2$: $(\alpha_2 \to \alpha_2) \to \alpha_2 \to \alpha_2$

   Substitution: $\alpha_1 = \alpha_2 \to \alpha_2$
5. $\texttt{twice}_3$: $(\alpha_3 \to \alpha_3) \to \alpha_3 \to \alpha_3$
6. $\texttt{twice}_1\ \texttt{twice}_2\ \texttt{twice}_3$: $(\alpha_3 \to \alpha_3) \to \alpha_3 \to \alpha_3$

   Substitution: $\alpha_2 = \alpha_3 \to \alpha_3$
7. `succ`: $\texttt{int} \to \texttt{int}$
8. $\texttt{twice}_1\ \texttt{twice}_2\ \texttt{twice}_3\ \texttt{succ}$: $\texttt{int} \to \texttt{int}$

   Substitution: $\alpha_3 = \texttt{int}$
9. $\texttt{twice}_1\ \texttt{twice}_2\ \texttt{twice}_3\ \texttt{succ 0}$: `int`

Because every time we generate fresh variable for α, every `twice` has different type variables and no type circularity occurs.

Here's another example where a polymorphic function is applied to different arguments in the same context

```
(* using polymorphic types *)
let g = rec map =>
  fn f => fn l =>
    if l=nil
    then nil
    else (f (hd l))::(map f (tl l))
in if hd ( g (fn x => x) (false::true::nil) )
        then g (fn x => x + 1) (1::2::3::nil)
        else g (fn x => x - 1) (1::2::3::nil)
end
```

Here `g = rec map` is applied to both an `int list` and a `bool list`.

5 Lazy Evaluation

The idea of lazy evaluation is simple. When an expression is bound to a name, we do not evaluate the expression immediately. Instead, we store the expression into the environment with the name, while keep the value of the name as `null` in the environment. Later, if the name is used, and we try to extract its value from the environment but only get `null`, we know that the expression hasn't been evaluated yet. Then we evaluate it at this point and store its value into the environment. Because evaluation will be done in method of `Env`, the state should also be passed into `Env`.

To do this, a new `Env` constructor is needed in addition to the original one

```
public Env(Env E, Symbol x, Expr e, State s) {
            this.E = E;
            this.x = x;
            this.v = null;
            this.e = e;
            this.s = s;
            this.id = idcnt++;
      }
```

This constructor allows us to store a name binding and its expression into environment without evaluate it. When we extract the value of a name from the environment, if the result is `null`, the expression needs to be evaluated. Thus the code of `Env.get` should be modified

```
public Value get(Symbol y) throws RuntimeError {
            if(x == y) {
                  if(v != null)
                        return v;
                  else {
                        v = e.lazyVal(s);
                        s.l.put(id, v);
                        return v;
                  }
            }
            else
                  return E.get(y);
      }
```

Also, I add a new method `lazyVal` to `Expr`. Every type of expression implements `lazyVal`. Here's the code for `Let.lazyVal`

```
1  @Override
2        public Value lazyVal(State s) throws RuntimeError {
3
4              Env newE = new Env(s.E, x, e1, s);
5              int id = newE.id;
6              State tempState = State.of(newE, s.M, s.p, s.l);
7              Value v = e2.lazyVal(tempState);
8              s.l.remove(id);
9              return v;
10       }
```

In line 4, the old `Env` is appended by `x` and its bounded expression `e1`. Then in line 7 the `lazyVal` of `e2` is called, under the appended environment.

In this way, if a name is never used, then its bounded expression is never evaluated. This approach can apply to both `let` and function application.

I make lazy evaluation as an option of interpreter, you can use it by appending `-lazyval` in the command line, for example

```
java -jar SimPL.jar lazyval3.spl -lazyval
```

We can observe lazy evaluation by inserting runtime error code into expressions. Only if the expression is evaluated, it will throw a runtime error. Here's a sample code

```
let f = fn x => fn y => fn z => if (x > y) then x else y in
        f 1 2 (tl nil)
end
```

In the body of function `f`, only `x` and `y` are used. When `f` is called, all three arguments `1, 2` and `tl nil` are passed in as expressions. However, because `z` is never used, so `tl nil` is never evaluated in `-lazyval` mode. So no runtime error occurs. Without `-lazyval`, `tl nil` is evaluated before it is passed in, and a runtime error is thrown because we try to get the tail of `nil`.

6 Garbage Collection

Before implement garbage collection, I do several modifications to the memory system to make garbage collection more interesting and meaningful.

First, I add a `size` field to `Value`. Different types of value may have different sizes. I do this because if all the sizes of values are 1, there will be no fragmentation in the heap. Then garbage collection will lose half of its sense. Here's a table for sizes of different types of values.

Table 1: Size Of Different Values

Type	Size
IntValue	1
BoolValue	1
ConsValue(v1::v2)	v1.size() + v2.size()
InlValue(v)	v.size()
InrValue(v)	v.size()
NilValue	1
PairValue(v1, v2)	v1.size() + v2.size()
RefValue(v)	1
TypeValue(t)	1
UnitValue	1
FunValue	1
RecValue	1

For `FunValue` and `RecValue`, we assume that they are stored as a pointer to the function code in memory.

I use *Copy Collection Algorithm* introduced in the class to do garbage collection. Just like *Mark and Sweep*, this algorithm requires to find out all the allocated memory locations which are reachable from variables currently alive.

So, to strictly record which variables are still alive, I add a field `liveVariables` in `State`. It is a map from `id` of variable (not using name, to distinguish variables with the same name) to its bounded value. In most cases, this map grows and shrinks as the current environment does, but there are some exceptions, that is the reason why I create a new field in `State` to record useful variables instead of using environment as the record. `id` of a variable simply marks the index of variable in the environment. It is allocated by the environment.

There are two cases where a useful variable-value binding can generate and disappear.

1. When enters a function body, the variable of formal parameter is bound to the argument value. And when exits the function, this binding disappears.

2. When enters the body of `Let`, the let variable is bound to a value. When exits the body, this binding disappears.

   Thus a variable `x` is alive only when

1. Evaluating a function application, and `x` is a formal parameter of this function.

2. Evaluating a `Let` expression, and `x` is the let variable.

So before evaluating the function body, we need to put the binding of parameter variable and argument value into `liveVariables` in `State`. And after the evaluation it should be removed. For `Let` we have similar operation. Here's the code for evaluation of `Let`

```
1  @Override
2      public Value eval(State s) throws RuntimeError {
3          Value v1 = e1.eval(s);
4          Env newE = new Env(s.E, x, v1);
5          int id = newE.id;
6          State tempState = State.of(newE, s.M, s.p, s.l.put(id, v1));
7          Value v2 = e2.eval(tempState);
8          s.l.remove(id);
9          return v2;
10     }
```

In line 6, right before the evaluation, we put the binding into `s.l`, which is the live variable list in `State`. After the evaluation, we remove it from `s.l`, which means the binding is no longer exists.

Now that we know the which variables are alive, we can use them as starting points to search memory locations reachable from those variables. These memory locations are not leak and do not need to be collected. This process corresponds to *mark*.

The goal of the search is to find out all the references reachable from those live variables. Here's something complex, because a reference, or memory location, can be reached from a variable indirectly. For example

```
let x = ref (ref (ref (ref 1))) in
        let refs = ref (ref 1, ref 2) in
        !(fst !refs) end;
        let y = ref 2 in y := !y + !!!!x; !y end
end
```

Here `x` points to a memory location, and that location stored another reference, and so on. All these references can be reached by variable `x`. Also, `refs` itself is not a reference value, but it is a pair of two references. To make a precise search, we must record exactly which memory locations can be reached from a value and its bound variable. A field `references` is added to `Value` to store this information. Here's the code of `PairValue` constructor. It copies references reachable from its two children, because references reachable from its children are also reachable from itself.

```
public PairValue(Value v1, Value v2) {
        this.v1 = v1;
        this.v2 = v2;
        references.addAll(v1.references);
        references.addAll(v2.references);
    }
```

With the preparation above, we can now start *Copy Collection* garbage collection. When a reference is created, we call `getMemCell` method shown below to get the allocated location.

```
1  public int getMemCell(int size, LiveVariables l, Value v)
2                                 throws OutOfMemoryError {
3          if(pointer + size - 1 <= fromSpace.end) {
4              pointer += size;
5              return pointer - size;
6          }
7          else if(gc) {
8              garbageCollection(l, v);
9              if(pointer + size - 1 <= fromSpace.end) {
10                 pointer += size;
11                 return pointer - size;
12             }
13             else
14                 throw new OutOfMemoryError("Heap␣space␣used␣up!");
15         }
16         else
17             throw new OutOfMemoryError("Heap␣space␣used␣up!");
18     }
```

`pointer` points to the next available memory location. Line 3 checks whether there is enough space left in the heap for this reference. If so, move the pointer forward by `size`, indicating that `size` units of heap spaces is allocated. Otherwise, in line 7 we check whether garbage collection is activated by the user. If not, `OutOfMemoryError` is thrown. If garbage collection is enabled, then we do garbage collection. If after the garbage collection, there's still no space, `OutOfMemoryError` is thrown.

Here's the code for `garbageCollection`

```
    private void garbageCollection(LiveVariables l, Value v) {
        pointer = toSpace.start;
        for(RefValue cell : v.references) {
            copy(cell);
        }
        for(int x : l.liveVariables.keySet()) {
            Value liveValue = l.liveVariables.get(x);
            if(liveValue != null) {
                for(RefValue cell : liveValue.references) {
                    copy(cell);
                }
            }
        }
        int start = fromSpace.start, end = fromSpace.end;
        fromSpace = toSpace;
        toSpace = new HalfSpace(start, end);
    }

    private void copy(RefValue cell) {
        Value liveValue = fromSpace.get(cell.p);
        if(liveValue != null) {
            if(toSpace.get(cell.p) == null) {
                toSpace.put(pointer, liveValue);
                fromSpace.remove(cell);
                cell.p = pointer;
                pointer += liveValue.size();
            }
            for(RefValue ce : liveValue.references) {
                copy(ce);
            }
        }
    }
```

Starting from the `liveVariable` list, in addition to the value requiring heap space, we search for all reachable references recursively and copy them from `fromSpace` to `toSpace`. After that, exchange `fromSpace` and `toSpace`.

7 Expand the Language: Sum Type and Case Expression

I think the most interesting thing I did in this project is expand the language to support sum type, case expression and injection. Here are the typing and evaluation rules

$$\frac{\Gamma \vdash e : t_1}{\Gamma \vdash \text{inl}[t_1 + t_2]e : t_1 + t_2}(\text{T} - \text{Inl}) \quad \frac{\Gamma \vdash e : t_2}{\Gamma \vdash \text{inl}[t_1 + t_2]e : t_1 + t_2}(\text{T} - \text{Inr})$$

$$\frac{\Gamma \vdash e : t_1 + t_2 \quad \Gamma, x_1 : t_1 \vdash e_1 : t \quad \Gamma, x_2 : t_2 \vdash e_2 : t}{\Gamma \vdash \text{case } e \text{ of inl } x_1 \Rightarrow e_1 \mid \text{inr } x_2 \Rightarrow e_2 : t}(\text{T} - \text{Case})$$

$$\frac{}{\text{case (inl } v) \text{ of inl } x_1 \Rightarrow e_1 \mid \text{inr } x_2 \Rightarrow e_2 \to e_1[v/x_1]}(\text{E} - \text{CaseInl})$$

$$\frac{}{\text{case (inr } v) \text{ of inl } x_1 \Rightarrow e_1 \mid \text{inr } x_2 \Rightarrow e_2 \to e_2[v/x_2]}(\text{E} - \text{CaseInr})$$

$$\frac{e \to e'}{\text{case } e \text{ of inl } x_1 \Rightarrow e_1 \mid \text{inr } x_2 \Rightarrow e_2 \to \text{case } e' \text{ of inl } x_1 \Rightarrow e_1 \mid \text{inr } x_2 \Rightarrow e_2}(\text{E} - \text{Case})$$

$$\frac{e \to e'}{\text{inl } e \to \text{inl } e'}(\text{E} - \text{inl}) \quad \frac{e \to e'}{\text{inr } e \to \text{inr } e'}(\text{E} - \text{inr})$$

There are a few steps to implement sum type and case expression. I divide them into subsections.

7.1 Modify the Lexical Analyzer and Parser

The lexical analyzer is generated by `Flex` automatically, so we only need to add some tokens to `simpl.lex` located in `simpl.parser`. Here are the added tokens in `simpl.lex`.

```
"case"            { return token(CASE); }
"of"              { return token(OF); }
"inl"             { return token(INL); }
"inr"             { return token(INR); }
"int"             { return token(INT); }
"bool"            { return token(BOOL); }
"list"            { return token(LIST); }
"unit"            { return token(UNIT); }
```

Next step, modify the grammar defined in `simpl.grm`. We do following changes

1. Declare the new terminals

   `terminal CASE, OF, VBAR, INL, INR, INT, BOOL, LIST, LBOX, RBOX, SUM;`

   Non terminal `SUM` is defined only to indicate the priority of injection, as you will see in the grammar of injection in step 4.

2. Because case expression requires types appears in the "[ ]" in the code explicitly. Call types in"[ ]" type expressions. We need a nonterminal for these type expressions.

   `non terminal TypeExpr t;`

3. Specify grammar for type expressions.

   ```
   t ::= INT {: RESULT = new IntTypeExpr(); :}
   | BOOL {: RESULT = new BoolTypeExpr(); :}
   | t:t1 MUL t:t2 {: RESULT = new PairTypeExpr(t1, t2); :} %prec MUL
   | t:t1 ADD t:t2 {: RESULT = new SumTypeExpr(t1, t2); :} %prec ADD
   | t:t LIST {: RESULT = new ListTypeExpr(t); :} %prec APP
   | UNIT {: RESULT = new UnitTypeExpr(); :}
   | t:t REF {: RESULT = new RefTypeExpr(t); :} %prec APP
   | LPAREN t:t RPAREN {: RESULT = t; :}
   ;
   ```

4. Add grammar for case expression and injection to non terminal `e`

   ```
   e::= ...
   | CASE e:e1 OF INL LBOX t:t1 RBOX ID:x1 ARROW e:e2 VBAR INR LBOX t:t2 RBOX ID:x2
   ARROW e:e3 {:RESULT = new Case(e1, symbol(x1), e2, symbol(x2), e3, t1, t2);:}
   | INL LBOX t:t RBOX e:e {:RESULT = new Inl(t, e);:} %prec SUM
   | INR LBOX t:t RBOX e:e {: RESULT = new Inr(t, e); :} %prec SUM
   ;
   ```

After these modifications, enter directory `parser` and `make`, new `Lexer.java` and `Parser.java` will be generated. Now our parser is capable of recognizing type expressions, case expressions and injections.

7.2 Implement Sum Type

A sum type is in the form of $\mathtt{t}_1 + \mathtt{t}_2$. Both sum type and pair type have two child types, so their implementation are quite similar. Here's the `unify` method of sum type

```java
@Override
    public Substitution unify(Type t) throws TypeError {
        if(SumType.class.isInstance(t)) {
            SumType pt = (SumType)t;
            Substitution s = t1.unify(pt.t1);
            return s.apply(t2).unify(s.apply(pt.t2)).compose(s);
        }
        else if(TypeVar.class.isInstance(t))
            if(contains((TypeVar)t))
                throw new TypeCircularityError();
            else
                return Substitution.of((TypeVar)t, this);
        else
            throw new TypeError("Cannot unify " + this + " with " + t);
    }
```

7.3 Implement TypeType

Because type expressions will appear in our code explicitly, we need a type for them. That is, a type for type, so I call it `TypeType`. All type expressions, for example `int + bool`, `int*int` have the type `TypeType`. The implementation is trivial and it doesn't contain any data field, because it only acts as a indicator for type expressions. Here's the `unify` method for `TypeType`

```java
@Override
    public Substitution unify(Type t) throws TypeError {
        if (t instanceof TypeVar) {
            return t.unify(this);
        }
        if (t instanceof TypeType) {
            return Substitution.IDENTITY;
        }
        throw new TypeMismatchError();
    }
```

7.4 Implement Type Expressions

The structures of type expressions are similar to real expressions. Real expressions are composed of int values, boolean values and operators. Type expressions are composed of lexical `int`, `bool` and operators. We can refer to the grammar of type expressions when creating classes for them. Here's the type checking and evaluation code for `PairTypeExpr`, other type expressions are similar.

```java
@Override
    public TypeResult typecheck(TypeEnv E) throws TypeError {
        TypeResult lt = l.typecheck(E);
        if(!TypeType.class.isInstance(lt.t)) {
            throw new TypeError("");
        }
        TypeResult rt = r.typecheck(E);
        if(!TypeType.class.isInstance(rt.t)) {
            throw new TypeError("");
        }
        TypeResult typeResult = TypeResult.of(Substitution.IDENTITY,
                                              Type.TYPETYPE);
        return typeResult;
    }

    @Override
    public Value eval(State s) throws RuntimeError {
        TypeValue lv = (TypeValue) l.eval(s);
        TypeValue rv = (TypeValue) r.eval(s);
        return new TypeValue(new PairType(lv.t, rv.t));
    }
```

7.5 Implement Type Values

Every expression should have a correspondent value, there is no exception for type expression. I create `TypeValue` class to represent the value of a type expression. `TypeValue` has a data field, which is a `Type` instance. In other words, it is a value storing a type.

```
public class TypeValue extends Value {

    public Type t;

    public TypeValue(Type t) {
        this.t = t;
    }

    public String toString() {
        return "type@" + t;
    }

    @Override
    public boolean equals(Object other) {
        if(other.getClass() == UnitValue.class)
            return true;
        else
            return false;
    }

    @Override
    public int size() {
        return 1;
    }
}
```

As an example, in the code fragment in **7.4**, a `PairValueExpr` is evaluated into a `TypeValue` which contains the pair type of its two children.

7.6 Implement Injections

Add classes `Inl` and `Inr` as subtypes of `Expr`. Implement `typecheck` and `eval` according to the typing and evaluation rules of injection listed in the beginning of part **7**. Here's the code for `Inl`.

```
@Override
    public TypeResult typecheck(TypeEnv E) throws TypeError {
        if(!SumTypeExpr.class.isInstance(t)) {
            throw new TypeError("");
        }
        SumTypeExpr ste = (SumTypeExpr)t;
        TypeResult te = e.typecheck(E);
        Substitution s =  te.t.unify(((SumType)ste.t).t1);
        TypeResult typeResult = TypeResult.of(s.compose(te.s), ste.t);
        typeResult.mergeLocalTypeVars(te.localTypeVars);
        typeResult.t.localTypeVars = typeResult.localTypeVars;
        return typeResult;
    }

    @Override
    public Value eval(State s) throws RuntimeError {
        Value v = e.eval(s);
        return new InlValue(v);
    }
```

Also, we need to create two `Value` classes for `Inl` and `Inr` expression. They only need to contain one value field and their implementation is quite simple.

7.7 Implement Case Expression

Finally, we can implement case expressions. Again we can refer to the typing and evaluation rules listed in the beginning of Part **7**. Note that when typing a case expression, the type environment may change. Also, when evaluating a case expression, the environment may change. We need to append the environment by either $\mathtt{x}_1$ or $\mathtt{x}_2$ before evaluating or typing the body of case expressions.

`Case` is the fourth type of expressions that can lead to growing and shrinking of `TypeEnv` and `Env`. The other three are `App`, `Let` and `Rec`. Growing and shrinking of environments means variable bindings changing. However, unlike `App` and `Let`, we cannot do lazy evaluation for `Case`, because we have to evaluate the expression `e` to either a `InlValue` or `InrValue` before we know which branch to go to.

The code for `Case` typing and evaluation is too long and it is not convenient to paste it here. Instead, I give a sample program using case expression.

```
let getfst = fn y =>
        case y
                of inl[(int list) + int*int] v1 => hd v1
                |  inr[(int list) + int*int] v2 => fst v2
        in (getfst inl[(int list) + int*int](1::2::3::4::nil),
                getfst inr[(int list) + int*int](5,6))
end
```

`getfst` is a function that accepts either a `int list` or `int` pair, and return its first element. This program takes the first element of list `(1::2::3::4::nil)` and pair `(5,6)`, then combine them into a new pair. The result of this program is `(1, 5)`.

8 Some Explanation

1. After adding some fields to `Env`, the stack size for running test cases becomes larger. So there will be stack over flow with test case `pcf.fibonacci.spl`.

2. When using type expressions, it's better to use parenthesis to indicate the associativity, like `(int list) + int*int`. Otherwise the parser can produce wrong result.

3. To enable lazy evaluation, use `-lazyval` after the file name like a compiler option:

   `java -jar SimPL.jar lazyval1.spl -lazyval`

4. To enable garbage collection, use `-gc` after the file name:

   ```
   java -jar SimPL.jar gc1.spl -gc
   ```

5. To set heapsize, use `-heapsize:size` after the file name, for example:

   ```
   java -jar SimPL.jar gc1.spl -heapsize:1024
   ```

   The default heap size is 2048.

6. All three options above can be mixed together and used simultaneously:

   ```
   java -jar SimPL.jar gc1.spl -heapsize:2048 -gc -lazyval
   ```