SimpL Interpreter

Overview

This project completed the interpreter for a new language called SimpL, which is defined in the project description file.

Project Structure

```
├──bin
|   └──simpl
|       ├──interpreter
|       |   ├──lib
|       |   └──pcf
|       ├──parser
|       |   └──ast
|       └──typing
├──doc
|   └──examples
├──examples
├──lib
└──src
    └──simpl
        ├──interpreter
        |   ├──lib
        |   └──pcf
        ├──parser
        |   └──ast
        └──typing
```

Folder `src` contains all the source code. Within this file, codes are divided into three parts: `interpreter`, `parser`, and `typing`.

Folder `lib` contains `java` runtime library.

Folder `example` and `doc/example` contains several test cases.

Folder `bin` is the class file of the source code, which is generated by `javac`.

The entry `main` function is in `src/simpl/interpreter/Interpreter.java`

Value

The definition of the values in SimpL is in `src/interpreter`.

```
├──interpreter
        |   |   BoolValue.java
        |   |   ConsValue.java
        |   |   Env.java
        |   |   FunValue.java
        |   |   InitialState.java
        |   |   Int.java
        |   |   Interpreter.java
        |   |   IntValue.java
```

```
        |   |   Mem.java
        |   |   NilValue.java
        |   |   PairValue.java
        |   |   RecValue.java
        |   |   RefValue.java
        |   |   RuntimeError.java
        |   |   State.java
        |   |   UnitValue.java
        |   |   Value.java
```

In the value class definition, we define the value composition, the string representation, and the equality of the value.

For example,

```
public class FunValue extends Value {

    public final Env E;
    public final Symbol x;
    public final Expr e;

    public FunValue(Env E, Symbol x, Expr e) {
        this.E = E;
        this.x = x;
        this.e = e;
    } // FunValue contains the environment, the symbol, and the expression
    //fn x.e

    public String toString() {
        return "fun"; // FunValue is shown as "fun"
    }

    @Override
    public boolean equals(Object other) {
        return false; // FunValue has no equality.
    }
}
```

Type

The definition of the types in SimpL is in `src/typeing`.

```
——typing
                ArrowType.java
                BoolType.java
                DefaultTypeEnv.java
                IntType.java
                ListType.java
                PairType.java
                RefType.java
                Substitution.java
                Type.java
                TypeCircularityError.java
                TypeEnv.java
                TypeError.java
                TypeMismatchError.java
```

```
                TypeResult.java
                TypeVar.java
                UnitType.java
```

In the type value definition, we define the type composition, the type equality, the unify rule, the containing judgement, the replacing rule, and the string representation of this type.

For example,

```java
public final class ArrowType extends Type {

    public Type t1, t2;

    public ArrowType(Type t1, Type t2) {
        this.t1 = t1;
        this.t2 = t2;
    }// ArrowType is composed of t1 -> t2

    @Override
    public boolean isEqualityType() {
        return false;
    }// ArrowType has no equality

    @Override
    public Substitution unify(Type t) throws TypeError {
        if(t instanceof TypeVar)
            return Substitution.of(((TypeVar) t), this);
        else if(t instanceof ArrowType){
            Substitution s1 = ((ArrowType) t).t1.unify(this.t1);
            Substitution s2 = s1.apply(((ArrowType)
t).t2).unify(s1.apply(this.t2));
            return s1.compose(s2);
        }
        else throw new TypeMismatchError();
    }// The unification rule of ArrowType:
    /**
         * 1. t1->t2 = tv => [tv/t1->t2]
         * 2. t1->t2 = tv1->tv2 => t1=tv1, t2=tv2
         * 3. t1->t2 = others => error
         */

    @Override
    public boolean contains(TypeVar tv) {
        return this.t1.contains(tv) || this.t2.contains(tv);
    }// Judge whether this ArrowType contains a certain type variable.

    @Override
    public Type replace(TypeVar a, Type t) {
        return new ArrowType(this.t1.replace(a, t), this.t2.replace(a, t));
    }// replace a type variable with a type

    public String toString() {
        return "(" + t1 + " -> " + t2 + ")";
    }// ArrowType is represented as "t1 -> t2"
}
```

Expression

The definition of expressions in SimpL is in `src/interpreter/ast`.

```
├──interpreter
│   └──ast
        |           Add.java
        |           AndAlso.java
        |           App.java
        |           ArithExpr.java
        |           Assign.java
        |           BinaryExpr.java
        |           BooleanLiteral.java
        |           Cond.java
        |           Cons.java
        |           Deref.java
        |           Div.java
        |           Eq.java
        |           EqExpr.java
        |           Expr.java
        |           Fn.java
        |           Greater.java
        |           GreaterEq.java
        |           Group.java
        |           IntegerLiteral.java
        |           Less.java
        |           LessEq.java
        |           Let.java
        |           Loop.java
        |           Mod.java
        |           Mul.java
        |           Name.java
        |           Neg.java
        |           Neq.java
        |           Nil.java
        |           Not.java
        |           OrElse.java
        |           Pair.java
        |           Rec.java
        |           Ref.java
        |           RelExpr.java
        |           Seq.java
        |           Sub.java
        |           UnaryExpr.java
        |           Unit.java
```

The expressions are divided into several parts, including binary expression, conditional expression, boolean literal, unary expression, functional expression, integer literal, let expression, loop expression, name expression, nil, recursive expression, and unit value.

In each definition, the composition, the string representation, the evaluation rule, the typing rule, and a replace method is included. The replace method method is used to implement the let polymorphism. This kind of realization of let polymorhism is learned from Youngzt998/SimPL-Interpreter: An interpreter for a exercise programming language named SimPL (github.com).

For example,

```
public class AndAlso extends BinaryExpr {

    public AndAlso(Expr l, Expr r) {
        super(l, r);
    }

    public String toString() {
        return "(" + l + " andalso " + r + ")";
    }// AndAlso contains two sub-expressions

    @Override
    public AndAlso replace (Symbol x, Expr e) {
        return new AndAlso(l.replace(x, e), r.replace(x, e));
    }// replace each sub-expression

    @Override
    public TypeResult typecheck(TypeEnv E) throws TypeError {
        // TODO
        /**
         * CT-ANDALSO
         * G|-e1:t2, q1    e2:t2, q2
         * -------------------------
         * G|-e1 andalso e2:bool, q1 U q2 U {t1 = bool} U {t2 = bool}
         */
        TypeResult t1 = l.typecheck(E);
        TypeEnv E2 = t1.s.compose(E);
        TypeResult t2 = r.typecheck(E2);
        Substitution s_all = t1.s.compose(t2.s);
        s_all = s_all.compose(s_all.apply(t1.t).unify(Type.BOOL));
        s_all = s_all.compose(s_all.apply(t2.t).unify(Type.BOOL));
        return TypeResult.of(s_all, Type.BOOL);
    }// just an implementation of the contrain generation rule of AndAlso

    @Override
    public Value eval(State s) throws RuntimeError {
        // TODO
        /**
         * tt andalso v -> v
         * ff andalso e -> ff
         */
        if (((BoolValue) l.eval(s)).b) {
            return r.eval(s);
        }
        return (BoolValue) l.eval(s);
    }// just an implementation of the evaluation rule of AndAlso
}
```

Pre-defined Functions

Pre-defined functions are defined in `src/simpl/interpreter/lib` and `src/simpl/interpreter/pcf`.

```
├──interpreter
|   ├──lib
|   |       fst.java
|   |       hd.java
|   |       snd.java
|   |       tl.java
|   |
|   └──pcf
|           iszero.java
|           pred.java
|           succ.java
```

Pre-defined functions in SimpL are considered as new expressions. They should contains any methods that are defined in other expressions. They should also declared in the default type environment.

For example,

```
public class iszero extends FunValue {

    public iszero() {
        super(Env.empty, Symbol.symbol("iszero"), new Expr() {
            @Override
            public Expr replace (Symbol x, Expr e) {
                return this;
            }// composition of a iszero function

            @Override
            public TypeResult typecheck (TypeEnv E) throws TypeError {
                return TypeResult.of(new TypeVar(true));
            }// check the type validity of iszero

            @Override
            public Value eval(State s) throws RuntimeError{
                /**
                 * E,M,p;e => M',p';v    v == 0
                 * ----------------------------
                 * E,M,p;iszero e => M',p';tt
                 */
                IntValue v = (IntValue) s.E.get(Symbol.symbol("iszero"));
                if (v.n == 0) {
                    return new BoolValue(true);
                }
                return new BoolValue(false);
            }// evaluate the iszero according to the evaluation rule
        });
    }
}
```

Tests

Test cases are included in `doc/examples` and `examples`.

```
├──doc
```

```
|   └──examples
|           factorial.spl
|           gcd1.spl
|           gcd2.spl
|           map.spl
|           max.spl
|           pcf.even.spl
|           pcf.factorial.spl
|           pcf.fibonacci.spl
|           pcf.lists.spl
|           pcf.minus.spl
|           pcf.sum.spl
|           pcf.twice.spl
|           plus.spl
|           sum.spl
|           true.spl
|
├──examples
|       factorial.spl
|       gcd1.spl
|       gcd2.spl
|       letpoly.spl
|       map.spl
|       max.spl
|       pcf.even.spl
|       pcf.factorial.spl
|       pcf.fibonacci.spl
|       pcf.minus.spl
|       pcf.sum.spl
|       plus.spl
|       sum.spl
```

They are all written in plain SimpL syntax.

For example,

```
let gcd = rec g => fn a => fn b =>
            if b=0 then a else g b (a % b)
in  gcd 34986 3087
end
(* ==> 1029 *)
```