CS383 Project: An Interpreter for SimPL

Kangwei Ling 5140219295

December 21, 2016

Contents

1 Introduction

SimPL is a simplified dialect of ML, which can be used for both functional and imperative programming.

In this project, on the basis of lexical analysis and syntactic analysis, I implemented the **Type System(Type Inference, Type Checking)** of **SimPL** and the **Evaluator** using the **Environment Model**.

2 Type System

Implementing Type System is the most fun part of this project, because it requires me to use the type inference knowledge learned in class to implement the inference algorithm and a real working type checker.

I must say that the provided **code skeleton** is very awesome, the structure is so great that implement the **Unification** algorithm is straightforward.

2.1 Unification

Unification is the process of solving type constraints. Each **Type Class** must implement a `unify` method, which take `Type t` as an argument and return a substitution that satisfy the type constraint,

$$\text{this type} = \text{t}$$

where `t` may be a `TypeVar`, etc.

2.1.1 Basic Types (Bool, Int, Unit)

BoolType, IntType, UnitType are basic types, the unification step for these types are pretty simple. If the argument is a TypeVar, then we call the argument's unify method to solve this constraint (A substitution that substitute that TypeVar with the current type). If not, the argument must have the same type as the current type, then we return the IDENTITY substitution, otherwise we throw a TypeMismatchError. Below is a code snippet for UnitType.

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

2.1.2 Complex Types (List, Pair, Ref)

I call these complex types because they are built on basic types and complex types themselves. If we are unifying them with a TypeVar, then we do same thing we did for **Basic Types**. If not, they must have the same main type and inner type. Otherwise, TypeMismatchError occurs. Below is the code for PairType as an example.

```
public Substitution unify(Type t) throws TypeError {
    if (t instanceof TypeVar) {
        return t.unify(this);
    }
    if (t instanceof PairType) {
        Substitution s1 = this.t1.unify(((PairType) t).t1);
        Substitution s2 = s1.apply(this.t2).unify(s1.apply(((PairType) t).t2));

        return s2.compose(s1);
    }
    throw new TypeMismatchError();
}
```

2.1.3 TypeVar

A `TypeVar` is created during the `typecheck` procedure, it acts like a placeholder for expression whose type is unknown yet. If we are unifying two TypeVars, then just return a substitution between these two TypeVars. Otherwise, we must check `TypeCircularityError` here. If the current `TypeVar` is also contained in the argument type **`t`**, which is not a `TypeVar`, then we throw `TypeCircularityError`. If there is no error, we can replace the current **TypeVar** with argument Type t.

```
public Substitution unify(Type t) throws TypeCircularityError {
    if (t instanceof TypeVar) {
        return Substitution.of(this, t);
    }
    if (t.contains(this)) {
        throw new TypeCircularityError();
    } else {
        return Substitution.of(this, t);
    }
}
```

2.2 Type Inference and Type Checking

The process of type inference and type checking are done simultaneously when we are traversing the **Abstract Syntax Tree** during the recursive call of `typecheck`.

2.2.1 Approach

Note that, the `typecheck` method for each **Ast Node** takes a ***TypeEnv*** as argument and return a ***TypeResult***.

TypeEnv a TypeEnv is a linked list of type bindings, which binds symbols to their type or a **TypeVar**.

TypeResult a TypeResult contains two component. One is a **Substitution** that satisfy all type constraints of expressions rooted at the current node. The other is the **Type** of the current **ast node**.

The whole process is a **bottom-up** strategy:

1. we first do type checking on subexpressions,

2. then, based on the typing rule, we generate type constraints and solving them by **unification**.

3. compose all the substitutions we get as the final substitution, and according to the type rule, give the current node a proper **Type**.

4. return the result.

It is very **important** to see that if there are **multiple subexpressions** need to be type checked, we must also pass the substitutions we have derived for previous subexpressions. This is done by compose the substitution with the type env. See below the `typecheck` for `arithmetic` expression an example:

```
public TypeResult typecheck(TypeEnv E) throws TypeError {
    // type check on left expression
    TypeResult lRes = l.typecheck(E);
    Substitution s0 = lRes.s;

    // must satisfy this constraint
    Substitution s1 = lRes.t.unify(Type.INT);

    // right expression
    TypeResult rRes = r.typecheck(s1.compose(s0.compose(E)));
    Substitution s2 = rRes.s;

    // same constraint
    Substitution s3 = rRes.t.unify(Type.INT);
    return TypeResult.of(s3.compose(s2.compose(s1.compose(s0))), Type.INT);
}
```

2.2.2 Function and Recursion

When doing type inference and type, before doing type check on subexpression, we first need to create a **TypeVar** and binds the `Symbol` of the func and rec to the **TypeVar**, since they may appear in the subexpression. The code for function and recursion is given below:

- function

```
public TypeResult typecheck(TypeEnv E) throws TypeError {
    TypeVar tv = new TypeVar(false);
    TypeResult Res = e.typecheck(TypeEnv.of(E, x, tv));

    return TypeResult.of(Res.s, new ArrowType(Res.s.apply(tv), Res.t));
}
```

- recursion

```
public TypeResult typecheck(TypeEnv E) throws TypeError {
    TypeVar tv = new TypeVar(false);

    TypeResult Res = e.typecheck(TypeEnv.of(E, x, tv));
    Substitution s1 = Res.t.unify(Res.s.apply(tv));

    s1 = s1.compose(Res.s);
    return TypeResult.of(s1, s1.apply(tv));
}
```

3 Evaluator

The evaluation part is where all the real computation goes. Just like the **type checking** process, evaluation is done recursively during a traversal of the **Abstract Syntax Tree** via the `eval` method. During the traversal, a **State** s is passed all the down the tree to pass necessary information for evaluation.

3.1 Evaluation State

The **State** is composed of three part:

Env the environment, a linked list of Variable-Value bindings (You can also view this as the evaluation stack).

Mem a hashmap **Memory Model** for **ref cells**.

p the memory pointer for the memory model.

3.1.1 Environment

The environment is essential to evaluation. For example,

- when we are evaluating a **Name**, a.k.a. identifier, we will retrieve its value from the environment in the state.

- when evaluating a function value, we will capture the current environment and create a `FunValue`. (**Closure**)
- When dealing with **Let Binding**, we will put the binding of Symbol-Value into the environment, and evaluate the **in** part with the extended environment.
- Function application, after we've retrieve the `FunValue` from the environment, we'll evaluate the right expression to value. Then evaluate the expression of `FunValue` in the **closure** (captured environment) extended with the binding of Symbol x with the right expression's value.

3.1.2 Memory Model

The hashmap-simulated Memory Model is a mapping: $\mathbb{N} \to$ `Value`, and **p** is the number points to the next free slot.

Since only `ref`, `deref` and `assign` access the heap memory, it's very easy to implement the evaluation part for them via the hashmap.

- For `ref`,

```
public Value eval(State s) throws RuntimeError {
    Value v = e.eval(s);
    int p = s.p.get();
    s.M.put(p, v);
    s.p.set(p + 1);
    return new RefValue(p);
}
```

- For `deref`,

```
public Value eval(State s) throws RuntimeError {
    RefValue rv = (RefValue) e.eval(s);
    return s.M.get(rv.p);
}
```

- For `assign`,

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

3.2 Implementation

After figuring out how the **environment model** and **memory model** work, it's really not that difficult to implement the `eval` method for all **ast node** according to the *evaluation rules*, which are well documented in the specification.

Most part of evaluation are rather simple and straightforward, I'll focus on the implementation of harder parts.

3.2.1 App - Function Application

App has the form: $e_1\ e_2$. First we need to evaluate e_1 to value, which must be a `FunValue`, then we evaluate e_2 to value v_2, then evaluate the body of the `FunValue`,

$$e_1\ e_2 \rightarrow (fun, E_1, x, e)\ e_2 \rightarrow (fun, E_1, x, e)\ v_2 \xrightarrow{E_1[x \mapsto v_2]} v$$

```
public Value eval(State s) throws RuntimeError {
    FunValue lval = (FunValue) l.eval(s);
    Value rval = r.eval(s);
    return lval.e.eval(State.of(new Env(lval.E, lval.x, rval), s.M, s.p));
}
```

3.2.2 Fn & Rec

When creating a `FunValue`, we need to capture the current **Environment**, so that when apply the function it is evaluated with the right scope.

```
public Value eval(State s) throws RuntimeError {
    return new FunValue(s.E, x, e);
}
```

For `Recursion`, the captured environment must also capture the binding of itself to the `RecValue`, then we must evaluate the subexpression of the **rec** node, to return the real value under rec. The code is more clear:

```
public Value eval(State s) throws RuntimeError {
    RecValue rv = new RecValue(s.E, x, e);
    Env env = new Env(s.E, x, rv);
    return e.eval(State.of(env, s.M, s.p));
}
```

3.2.3 Let Binding

During let binding: $let\ x = e_1\ in\ e_2$, we need bind x to e_1's value to the environment then evaluate e_3.

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

3.2.4 Name - Retrieve Value from Environment

The value of a `Name` node must be retrieved from the environment. If the name is not found in the environment, we must throw an exception, otherwise we must return the value.

Note that, if the value is a `RecValue`, we need to evaluate it to return the value underlying.

```
public Value eval(State s) throws RuntimeError {
    Value v = s.E.get(x);
    if (v == null)
        throw new NameError(x);
    if (v instanceof RecValue) {
        return ((RecValue) v).e.eval(s);
    }
    return v;
}
```

3.3 Exception Handling

When evaluating, there are several exceptions (runtime error) we must deal with. I covered **NameError** and **ZeroDivisionError**.

NameError a name can't be find in the environment

ZeroDivisionError trying to divide a number by zero or trying to calculate `n mod 0`.

The code for **NameError** is already listed when I talked about `Name`. **ZeroDivisionError** is used in `Div` and `Mod`.

- Div

```
public Value eval(State s) throws RuntimeError {
    IntValue iv1 = (IntValue) l.eval(s);
    IntValue iv2 = (IntValue) r.eval(s);

    if (iv2.equals(Value.ZERO))
        throw new ZeroDivisionError();
    return new IntValue(iv1.n / iv2.n);
}
```

- Mod

```
public Value eval(State s) throws RuntimeError {
    IntValue iv1 = (IntValue) l.eval(s);
    IntValue iv2 = (IntValue) r.eval(s);

    if (iv2.equals(Value.ZERO))
        throw new ZeroDivisionError();
    return new IntValue(iv1.n % iv2.n);
}
```

4 Built-in Functions

Types and values of built-in functions must be defined in `DefaultTypeEnv` and `InitialState` respectively. For example, for `hd`, which return the head element of a list, its type must be defined:

```
public DefaultTypeEnv() {
    ...
    Symbol hd = Symbol.symbol("hd");
    t = new TypeVar(false);
    ArrowType hd_t = new ArrowType(new ListType(t), t);
    E = TypeEnv.of(E, hd, hd_t);
    ...
}
```

its value can define as:

```
public hd() {
        super(Env.empty, Symbol.symbol("__list__hd"), new Expr() {
            @Override
            public TypeResult typecheck(TypeEnv E) throws TypeError {
                return null;
            }
            @Override
            public Value eval(State s) throws RuntimeError {
                Value v = s.E.get(Symbol.symbol("__list__hd"));
                if (v instanceof ConsValue) {
                    return ((ConsValue) v).v1;
                }
                throw new RuntimeError("hd can only be applied on lists.");
            }
        });
    }
```

In `InitialState`, I added the bindings of built-in functions (their Symbol and their corresponding **value**. For example,

```
public InitialState() {
    super(initialEnv(Env.empty), new Mem(), new Int(0));
}
private static Env initialEnv(Env E) {
    ...
    E = new Env(E, symbol("hd"), new hd());
    ...
    return E;
}
```

5 Garbage Collection

Garbage Collection is a very important feature for modern programming languages, so I decide to implement it for this toy language.

At first I was thinking of using *reference counting* to implement a simple garbage collector, but after considering it for a very long time, that method was discarded because I may have to change too much existing project structure. Then *Mark-and-Sweep* came to my mind, which is very easy to understand and implement.

5.1 Reachability and Problems

If a `ref cell` in memory(hashmap) can never be reached from the current place we are evaluating and later on, then that cell is a garbage and we need to reclaim that chunk of memory.

It is intuitive that we search through the environment and mark all `ref cells` that we see, then collect those aren't marked.

However, there are some exceptions. Consider we are applying a function which create `ref cells` in its body but the function is defined very earlier, which means all `ref cells` that defined after that are not shown in the environment of that function's closure. So if we are doing garbage collection in a closure, many **reachable cells** will be incorrectly collected.

To solve this problem, every time the evaluator enters a closure, I'll push the current outer environment to a stack, and after the evaluation, I pop the stack. When garbage collector begins its job, it not only mark all `ref cells` reachable from the current environment but also the environment that just before we enter the closure (which will be the environment when we get out, so there are no mistakes).

5.2 Solution

I made some modifications to the memory model to support the garbage collector.

```
// in Mem class
private LinkedList<Integer> freeList;
private Stack<Env> envStack;
private static final int heapSize = 2;
private HashMap<Integer, Boolean> markBit;
```

freeList a linked list of free cells, initially, it is a list of 0,1,2..,heapSize-1.

envStack the stack that stores all reachable outer environments.

heapSize total number of memory cell available.

markBit a hashmap records whether a cell is marked (reachable).

I add the following methods to `Mem` class:

```
public int nextFree();
public void gc(Env env);
private void markEnv(Env env);
```

```java
private void markVal(Value val);
public void pushEnv(Env env);
public Env popEnv();
```

nextFree method just return first element of the `freeList`, which is a number that denotes a free cell.

gc start garbage collection. the `Env` in parameter is the current environment.

```java
public void gc(Env env) {
    System.out.println("Start GC");
    markBit.clear();
    markEnv(env);
    for (Env e : envStack) {
        markEnv(e);
    }
    sweep();
}
```

markEnv do marking for all value bindings in the environment.

```java
private void markEnv(Env env) {
    for (; env != Env.empty && env != null; env = env.getEnv()) {
        Value v = env.getValue();
        markVal(v);
    }
}
```

markVal do marking for a val, if it is one of the following types: `RefValue`, `ConsValue`, `PairValue`, `FunValue`, `RecValue`. (Actually we will only do a real marking for `RefValue`.)

```java
private void markVal(Value val) {
    if (val instanceof RefValue) {
        markBit.put(((RefValue) val).p, Boolean.TRUE);
        markVal(this.get(((RefValue) val).p));
    }
    else if (val instanceof ConsValue) {
        markVal(((ConsValue) val).getV1());
        markVal(((ConsValue) val).getV2());
    }
    else if (val instanceof PairValue) {
        markVal(((PairValue) val).getV1());
        markVal(((PairValue) val).getV2());
    }
    else if (val instanceof FunValue) {
        markEnv(((FunValue) val).getE());
    }
    else if (val instanceof RecValue) {
        // for rec value, skip over itself to prevent infinite loop
        markEnv(((RecValue) val).getE().getEnv());
    }
}
```

sweep go through all memory cells and collect garbage with the help of `markBit`.

```
private void sweep() {
    freeList.clear();
    for (int i = 0; i < heapSize; ++i) {
        if (markBit.getOrDefault(i, Boolean.FALSE) == Boolean.FALSE) {
            System.out.println("collect " + String.valueOf(i));
            this.put(i, null);
            freeList.addLast(i);
        }
    }
}
```

Besides these change to `Mem` class, I also add some `getter` method for several classes, I won't list them here.

With garbage collector, now the `ref` ast node need to be modified as follow (actually `p` is useless now):

```
public Value eval(State s) throws RuntimeError {
    Value v = e.eval(s);
    int p = s.M.nextFree();
    if (p == -1) {
        s.M.gc(s.E);
        p = s.M.nextFree();
    }
    if (p == -1) {
        throw new RuntimeError("HeapOverFlowError: No enough space.");
    }
    s.M.put(p, v);
    return new RefValue(p);
}
```

5.3 Test

The following code will be correctly interpreted when `heapSize` is only 2:

```
let f = fn x => ref x in
    let y = ref 2 in
        !(f 1) + !(f 1) + !y
    end
end
```

result:

```
int
Start GC
collect 1
4
```

And the following will throw `HeapOverFlowError` correctly:

```
let f = fn x => ref x in
    let y = ref 2 in
        let x1 = f 3 in
            let x2 = f 4 in
             !y
            end
            end
    end
end
```

result:

```
int
Start GC
runtime error
```

6 Polymorphism

In our previous **Type System**, the following code will not pass the type checking:

```
let id = fn x => x in
    ((id 1),(id false))
end
```

Indeed, the type of `id` must be `int -> int` and also `bool -> bool`, which causes type conflict.

In this part, I'll implement **Let Polymorphism** for this language, which make id's type as : $\forall X.X \to X$. I implemented the algorithm provided in the book *Types and Programming Languages* (Piece 2002), **Chapter 22**, which indeed provides enough information for implementation.

So, how does it work?

Basically, when we are dealing with a `let` expression `let x = e1 in e2`, once we get `x`' type, we don't bind `x` to `e1`'s type, but rather create a **Type Scheme** for it, which may or may not has some **TypeVar** in it. Each time we encounter `x` in `e2`, we'll instantiate a new type from the **type scheme** (replace all unsolved **TypeVar** in scheme with new vars). In this way, we implement the **Let Polymorphism**, it's not that difficult at all.

6.1 Implementation

For each `Type`, I added a new method called `unsolved`, which will return a set of `TypeVar` that occurs in that type.

For `TypeEnv`, I added a new method called `getEnvUnsolved()`, which will return a set of `TypeVar` that binds in that environment.

Note that, only **TypeVars** created after we start type check `e1` need to be generalized. The `TypeScheme` class is basically as follows:

```
public class TypeScheme extends Type {
    private Type t;
    private Set<TypeVar> unsolved;

    public TypeScheme(Type t, Set<TypeVar> un) {
        this.t = t;
        this.unsolved = un;
    }
    public static TypeScheme generateScheme(Type t, TypeEnv env) {
        Set<TypeVar> envUnsolved = env.getEnvUnsolved();
        Set<TypeVar> unsolved = t.unsolved();

        unsolved.removeAll(envUnsolved);
        return new TypeScheme(t, unsolved);
    }
    ...
    public Type instantiate() {
        Type newType = this.t;
        for(TypeVar tv : unsolved) {
            TypeVar nTv = new TypeVar(tv.isEqualityType());
            newType = newType.replace(tv, nTv);
        }
        return newType;
    }
}
```

When we retrieve type from `TypeEnv`, we'll check if it's a `TypeScheme`, if it is, a newly instantiated type will be returned. For example:

```
public Type get(Symbol x1) {
    if (x == x1) {
        if (t instanceof TypeScheme)
            return ((TypeScheme) t).instantiate();
        else
            return t;
    }
    return E.get(x1);
}
```

With this extension to our **Type System**, the `let` part is modified as follows:

```
public TypeResult typecheck(TypeEnv E) throws TypeError {
    TypeResult e1Res = e1.typecheck(E);

    // let polymorphism
    TypeScheme scm = TypeScheme.generateScheme(e1Res.t, E);

    TypeResult e2Res = e2.typecheck(TypeEnv.of(e1Res.s.compose(E), x, scm));

    return TypeResult.of(e2Res.s.compose(e1Res.s), e2Res.t);
}
```

6.2 Test

Run the same code we mentioned at the start:

```
(int * bool)
pair@1@false
```

7 Lazy Evaluation

Lazy evaluation, or call-by-need is an evaluation strategy which delays the evaluation of an expression until its value is needed (non-strict evaluation) and which also avoids repeated evaluations (sharing).

For example, the following code would cause runtime error

```
let ite = fn e => fn e1 => fn e2 => if e then e1 else e2 in
    ite true (2 / 1) (1 / 0)
end
```

However, by using **lazy evaluation**, we can delay the real computation until we really need it. In this case, the value of `(1 / 0)` is never used.

7.1 Implementation

My implementation is a very basic one of **Lazy Evaluation**, as I' don't have enough time.

7.1.1 Lazy keyword

I added a `lazy` keyword to this language, for example, `lazy (1 + 1)` will delay the calculation of `1 + 1` until we need the value.

In the **simpl.grm** file, a terminal `LAZY` is added, with precedence the same as `APP` and had left associativity. The grammer rule is:

$$\begin{aligned} e ::=& \ldots \\ & \text{LAZY } e : e\{: \ \text{RESULT} \ = \ \text{new Lazy}(e); :\} \end{aligned}$$

7.1.2 Lazy node

An ast node for `Lazy` is created, which is the fundamental part of delaying computation.

```
@Override
public Value eval(State s) throws RuntimeError {
    return new LazyValue(s.E, e);
}
```

7.1.3 LazyValue

`LazyValue` is a new subclass of `Value`, which is just like a `FunValue`, it has a captured environment for evaluating the delayed computation. A `force` method is used to compute the ultimate value of the expression.

```
private final Env E;
private final Expr e;

private boolean isVal;
private Value val;

public LazyValue(Env E, Expr e) {
    this.E = E;
    this.e = e;
}

public boolean isVal() {
    return this.isVal;
}

public Value force(State s) throws RuntimeError {
    if (this.isVal)
        return this.val;
    Value v = e.eval(State.of(this.E, s.M, s.p));
    this.isVal = true;
    if (v instanceof LazyValue)
        this.val = ((LazyValue) v).force(s);
    else
        this.val = v;
    return this.val;
}
```

To avoid repeated computations, as long as it has been calculated once, the value will be saved and `isVal` flag set.

7.1.4 force

I defined `force` as a predefined `FunValue` just like `fst`, which will force the evaluation of lazy values.

```
public class force extends FunValue{
    public force() {
        super(Env.empty, Symbol.symbol("force_val"), new Expr() {
            @Override
            public TypeResult typecheck(TypeEnv E) throws TypeError {
                return null;
            }

            @Override
            public Value eval(State s) throws RuntimeError {
                Value v = s.E.get(Symbol.symbol("force_val"));
                if (v instanceof LazyValue) {
                    return ((LazyValue) v).force(s);
                }
                throw new RuntimeError("force can only be applied on lazy.");
            }
        });
    }
}
```

7.2 Test

Surely there are other parts (like type of `force`) I didn't listed, they are just easy to code.

With the above modifications to the project, actually lazy evaluation works, just like it did in `OCaml` language.

```
let x = lazy (1 / 0) in
    let y = lazy (2 / 1) in
        let ite = fn e => fn e1 => fn e2 => if e then (force e1) else (force e2) in
            ite true y x
        end
    end
end
```

No error comes out!

```
int
2
```

8 Acknowledgement

I want to thank Prof. Zhu and TA for providing such a great course and project. In CS383, I really learned a lot of stuff about programming language.

They are very difficult of course but also very interesting to learn. The most fun part was doing the project, it's really wonderful to realize that I've implemented such a beast, :)! Thank you all!