# SimPL Report

*Yesheng Ma*
`kimi.ysma@gmail.com`

December 19, 2016

**Abstract**

SimPL is a simplified version of the Standard ML programming language. It is a functional programming language with impure features. In this report, we will discuss the implementation of a SimPL interpreter and some advanced features.

## 1 Introduction

First let's shed a light on what we should do to implement a SimPL interpreter. Just like building a compiler, we need a lexical analyzer, a parser, a semantic analyzer, and a evaluation system. In this report, I will not put much emphasis on the lexer and parser part since they are already written by TA. The type checking and evaluation are what we are interested in.

## 2 Lexer and Parser

This part is already written by TA. However, I think it is also worthwhile for us to look into the lexer and parser to get a more concrete understanding of SimPL.

The implementation of lexer and parser uses automatic generator: JFlex and CUP, which are variants of the lex and yacc of the UNIX operating system. To generate the source code for a lexer/parser, what we need to do is simply write down lexical specification and grammar rules and those lexer/parser generator will generate related Java code for us.

We can see from the `.lex` and `.grm` files that we specify lexical rules by regular expression and grammar rules by Barkus-Naur form. Then JFlex/CUP make our life easier by generating lexer and parser automatically.

## 3 Type System

When building a type system, we need to assign a unique or generic type to every syntactic structure in a program. In this part, we will implement a type checker and type inferencer for our SimPL. It may sound difficult at first thought, but actually it is not that hard if we can think *recursively*.

## 3.1 Types and Operations on It

SimPL has a rather simple type system: no user-defined types, no higher-kined types. This makes the job of building a type system much easier.

In our type system, all types inherit the base class `Type` and we define some particular types such as `ListType` by extending the abstract `Type`. Note that types are defined *recursively*, e.g. the pair type is defined by two other types and the list type is defined on the type of each member. Some important types are `ArrowType`, `UnitType`, `BoolType`, and `IntType`. For an `ArrowType`, it has a type as its type of argument and another type as its type of return value. For the `UnitType`, `BoolType`, and `IntType`, these are basic types and they don't rely on any other type.

The operations defined on types are:

1. `replace`: replaces all occurences of type variable `a` with type `t` and returns the replaced type.
2. `contains`: returns whether this type contains a type variable `tv`.
3. `unify`: takes two types and returns the result of unification of these two types, i.e. a subsitution.

## 3.2 Typing Environment, Substitution, and Type Errors

A typing environment is actually a set of bindings of names and types. For example, in the expression `let x = 1 in x + 1 end`, there should be a binding of `x -> Int`.

A substitution is result of a unification. There are two kinds of substitutions: identity and replace. A identity substitution returns a type as it is and a replace substitution returns a type as it is except that it is specified otherwise to be replaced by another type. We can do composition on substitutions just like function composition.

There are two typical kinds of type errors:

1. `TypeMismatchError`: say you want an `Int` but a `Bool` is given, it is a type mismatch error.
2. `TypeCirculationError`: you cannot define a type on itself, e.g. if you define a `List` whose member is also of the same type, it makes no sense to define such an object.

## 3.3 Implementation Details of Type System

On each node of abstract syntax tree, we define a semantic action called `typecheck`. We call the `typecheck` method recursively on its children and itself. I will take several most representative AST nodes as an example.

### 3.3.1 Typechecking of Fn

When doing type checking on a syntactic structure, the essential part is the *typing rules*. If we keep the typing rules in mind, type checking can be handled easily.

As regard to the case of `Fn`, we can first write down its constraint generation rule:

$$\frac{\Gamma, x : a \vdash u \Rightarrow e : t, q}{\Gamma \vdash \lambda x.u \Rightarrow \lambda x : a.e : a \to b, \{t = b\} \cup q}$$

What this rule tell us is that we assign a type variable to $x$ and the untyped term $u$ should be of type $t$. Thus the type of this function should be $a \to b$.

```java
public TypeResult typecheck(TypeEnv E) throws TypeError {
    Type xTpe = new TypeVar(true);
    TypeResult predRes = e.typecheck(TypeEnv.of(E, x, xTpe));
    Substitution sub = predRes.s;
    return TypeResult.of(sub,
        new ArrowType(sub.apply(xTpe), sub.apply(predRes.t)));
}
```

Listing 1: Type Checking of Fn

### 3.3.2 Typechecking of Cond

The type checking of `cond` expression is similar to other language constructs like unary and binary operators. However the `cond` expression is a bit more complex than others since it is a ternary operator where some werid dependency problems may arise.

I will not write down the constraint generation rules here but the key idea is simple: the predicate of a `cond` should be of type `bool` and the two clauses should be of the same type. Though the idea is simple, there are some detail to note when implementing it:

1. We first type check `e1`, `e2`, and `e3`.
2. Next we do substitution on the type results of the three expressions. What is tricky here is that we need to apply the composition of other two substitutions to get the desired type of each expression.
3. Next, we unify the types of `e2` and `e3` and get a substitution and we compose it with the other 3 substitution to get the final substitution.
4. Finally, we apply the type of `e2` to the above generated substituion and get the type of the whole `cond` expression.

```java
public TypeResult typecheck(TypeEnv E) throws TypeError {
    TypeResult e1Res = e1.typecheck(E);
    TypeResult e2Res = e2.typecheck(E);
    TypeResult e3Res = e3.typecheck(E);
    Type e1Tpe = e2Res.s.compose(e3Res.s).apply(e1Res.t);
    Type e2Tpe = e3Res.s.compose(e1Res.s).apply(e2Res.t);
    Type e3Tpe = e1Res.s.compose(e2Res.s).apply(e3Res.t);
    Substitution sub = e1Res.s.compose(
        e2Res.s.compose(
            e3Res.s.compose(
                e1Tpe.unify(Type.BOOL).compose(
                    e2Tpe.unify(e3Tpe)))));
    return TypeResult.of(sub, sub.apply(e2Tpe));
}
```

Listing 2: Type Checking of Cond

## 4 Evaluation System

Since we have already built a sound type system, we need to implement a evaluation system to make the interpreter really working.

### 4.1 Overview of the Evaluation System

#### 4.1.1 Values and Related Operations

In SimPL, we evaluate a program in a bottom-up manner. For each node in abstract syntax tree, we call the `eval` method recursively and we will get the final evaluation result at topmost node.

All values in SimPL inherit the abstract class `Value`. And for each specific type of value, it has related attribtes. For example, for the `RefValue` part, the related the value is a pointer to the referenced value; for the `FunValue`, we treat a function as a value too. In a `FunValue`, we have three attributes, namely environment `E`, argument symbol `x`, and expression `e`. As we all know, each function forms a closure and we need to evaluate a function under a certain environment.

### 4.2 Evaluation Environment

In SimPL interpreter, we use `State` to define a machine state and a machine state contains three parts:

1. Environment `E`: similar to typing environment, but in an evaluation environment, we have name-value bindings rather than name-type bindings. I will explain this later.

2. Memory `M`: since SimPL is a functional programming language with impure features, we need a memory to mimic the behaviour of the imperative parts. Memory is actually a hash map of address-value bindings.

3. Pointer `p`: this is a pointer to current free memory address.

An evaluation environment is a stack where name-value bindings are stored. For example, when evaluating a function application like `(fn x => x + 1) 2`, we are acutally using call-by-value semantic by substitution. In other words, we are evaluating `x + 1` under an environment where `x = 1`.

## 4.3 Implementation Details of Evaluation System

Most parts of evaluation implementation is rather trivial. I will mainly focus on the impure features and function evaluation.

### 4.3.1 Eval of Memory Reference

As we have analyzed above, memory model of SimPL is just a hash map of address-value bindings. To evaluate the memory creation expression `ref e`, we first evaluate the value `v` of `e`. Then we query evaluation state for a pointer to next empty memory space.

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

Listing 3: Evaluation of Ref

### 4.3.2 Eval of Recursion Functions

I think the implementation of recursion is the most interesting part of SimPL evaluation system. When implementing recursion, we get a deeper understanding of what role *environment* plays during evaluation.

On the first though, a recursion structure may seem infinity. However, if we apply the method of delayed evaluation, we can avoid the infinity expansion of a recursive body. The source code of recursion eval is shown as follows:

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

Listing 4: Evaluation of Rec

Here we create a new environment with the recursion as a `RecValue`. Say the result of evaluating the recursion is a `FunValue` and in the function application part the recursion value is no longer called and in this case the recursion body won't infinitely expand.

## 5 Bonus Part

### 5.1 Garbage Collection

We do garbage collection for heap management. In SimPL runtime, we define heap as a hash map of name-value bindings, which makes our task of implementing garbage collection much easier with the help of Java runtime reflection.

I implement the mark-and-sweep GC in SimPL and only if user turns on GC and the memory exceeds a threshold of 1000 will the interpreter start a GC. Every time the program ask for a new memory location, if the total size of memory exceeds a limit, a mark-and-sweep GC procedure will be invoked which marks all available names in current environment and all their descendants. If a value is an instance of `RefValue`, we will continue to trace the value the reference points to until there is no `RefValue` in the chain. The code for GC is shown as follows:

```
// mark all variables and related values
for (Env curEnv = s.E; curEnv != Env.empty;
        curEnv = curEnv.getPrevEnv()) {
    for (Value v = curEnv.getValue(); v instanceof RefValue && !v.mark;
            v = s.M.get(((RefValue) v).p)) {
        v.mark = true;
    }
    curEnv.getValue().mark = true;
}
// sweep through memory refs
for (Iterator<Mem.Entry<Integer, Value>> it = s.M.entrySet().iterator();
        it.hasNext(); ) {
    Mem.Entry<Integer, Value> entry = it.next();
    if (!entry.getValue().mark) {
    it.remove();
    }
}
```

Listing 5: Garbage Collection Implementation

### 5.2 Polymorphic Type

The polymorphic type is actually the simplest among all these bonuses. The key idea is that you should introduce type variables whenever you can. All the constraints will be resolved during the process of unification. If the whole program is unified and there is still type variables remaining, then these type variables will result in polymorphism.

For example, in the function `fn x => if x = nil then 0 else 1`, we assign type variable `tv1` to `x` and assign type variable `List[tv2]` to nil. If you have properly implemented type unification, you will get the substitution that `tv1 = tv2`. Finally, we will get a generic type of this function, i.e. `tv1->Int`. Actually for the sample `map.spl` program, which contains a typical polymorphic function `map`, my interpreter generates a porlymorphic type `((tv121 -> tv128) -> (tv122 list -> tv128 list))` and this type is consistent with the semantic of the high order function `map`.

Therefore, the polymorphic types are already included in the implementation of type system.

### 5.3 Lazy Evaluation

I think the lazy evaluation part is the most difficult one among all these bonuses. I also investigate into another functional programming language Haskell to get some idea of lazy evaluation. However, it is really hard to implement a lazy evaluation system like Haskell in our SimPL since SimPL use call-by-value semantic and SimPL is impure (which can actually lead to semantic difference in a lazy evaluation manner).

I also tried to cache the result of function application in order to speed up the execution (this is also naive way to mimic the technique called graph reduction in Haskell). Since functions in SimPL is impure, I define a method called `isPure` and a function is called pure iff every part of it is pure. Then when a "pure" function is called, I cache the result of the function application. If later function calls happen to have been computed before, the interpreter will use the former computed result. Though this method seems fantastic, when I benchmarked it and find it isn't more efficient than the original "eager evaluation" version.

I finally implement lazy evaluation in two language constructs:

1. In a let expression `let x = e1 in e2 end`, we only evaluate `e1` once and substitute all occurences of `x` in `e2`.

2. When evaluating a boolean expression, we apply the "short-circuit" evaluation strategy, e.g. in `x andalso y` if we find that `x` evaluates to false then we will no longer evaluate `y` and directly return false as the final result.

## 6 Conclusion

I implement the SimPL interpreter in about 10 days. In these days, I devoted my whole time to this project and referred to many materials to complete this project, including StandardML doc, lecture notes, Wikipedia, and many other online blog posts.

I've really learned a lot from this project:

1. I use the IntelliJ Idea IDE to develop this project and I find it a really good tool for productivity. Also, I program a lot in Scala and used to dislike Java a little. However, in this project I find Java is not so bad and I actually program happily with Java.

2. I have a better understanding of those abstract PL concepts by implementing a interpreter on my own, including lambda calculus, call-by-value semantics, type inference and type unification.

## Acknowledgement

Thanks Prof. Kenny Zhu for guidance on programming language theory and TA Xusheng for holding discussion sessions and kindly answering questions from us.