SimPL Interpreter

王梓涵　**517021911179**

SimPL Interpreter

王梓涵　517021911179

- Introduction
  - Objective
  - Project Structure
    - Parser
    - Interpreter
    - Typing
- Typing
  - Type Inference
    - Substitution
    - Unification
  - Type Checking
  - Let-polymorphism
    - Algorithm
    - Implementation
- Semantics
  - State
    - Environment
    - Memory
  - Values
  - Evaluation
    - Name-Value Binding
    - Reference Cells
- Predefined Functions
  - Semantics
  - Environment Definition
  - Type Definition
- Bonus Features
  - Garbage Collection
    - Implementation
    - Result
  - Lazy Evaluation
    - Implementation
    - Result
  - Mutually Recursive Combinator
    - Design
    - Implementation
    - Result
  - Infinite Stream
    - Implementation
    - Result

Appendix
Notice
Output

Introduction

Objective

In this project I am asked to implement an interpreter for the programming language SimPL. SimPL is a simplified dialect of ML, which can be used for both functional and imperative programming.

Project Structure

Parser

In package `parser`, infrastructures for lexical and syntactical analysis are already provided. The parser parses the source program into AST representation. My task is to implement type checking and evaluation methods for all the AST nodes.

Interpreter

In package `interpreter`, class `Interpreter` serves as the program entry for the whole project. Besides, there are classes representing runtime environment and values in this packages. Library functions should also be defined in this package.

Typing

In package `typing` resides classes representing types and typing environment. Substitution and unification methods should be implemented for all types, which form the basis of type inference.

Typing

Type Inference

Procedures required to perform type inference are listed in abstract class `Type`. These methods are shown in the listing:

```
1  public abstract class Type {
2      public abstract boolean isEqualityType();
3      public abstract boolean contains(TypeVar tv);
4      public abstract Type replace(TypeVar a, Type t);
5      public abstract Substitution unify(Type other) throws TypeError;
6      /* Other members ... */
7  }
```

Method `isEqualityType` is mainly used for type checking in equality expression, so I will skip it here. In the following, implementation of the rest three methods will be discussed.

The discussion may be divided with respect to different categories of types: 'Primitive type' refers to unit, boolean and integer. 'Compound type` refers to arrow, pair, list, reference, etc. Type variable may also appear as a separate category.

Substitution

`contains` and `replace` are related to type substitution. `contains` tests whether a certian type variable ever appears in this type, and `replace` replaces a type variable with another type if that type variable appears in this type.

Primitive types cannot contain any type variable, and they cannot be replaced. Compound types may contain type variables, depending on whether its components contains that. And it can call `replace` on its components, and then combine the resulting substitutions. Type variable contains another type variable if it shares the same name with that, and replacement result is that type.

Unification

Type unification finds a substitution that can make a certain type uniform with another type. This algorithm is implemented in method `unify`.

For any type, if one but not both of the types is a type variable, and that type variable appears on the RHS, the two variables are swapped. Primitive type can only be unified with the same type, yielding an identity substitution. Compound type can only be unified with another compound type of the same kind, and then unifies its component pairwise. For other cases, report a type mismatch error. The substitution returns by the components should be composed.

For type variables, if another type is also a type variable, do nothing if they share the same name, or substitute that for this type if they don't. If that type still contains this type variable, there is a type circularity.

Type Checking

Type checking algorithm is implemeted in `typeCheck` methods of AST nodes. The procedure is supported by type environment `TypeEnv`, which records mapping from names to types. When a `Name` node is visited, the algorithm look the name up in `TypeEnv`.

In the interpreter, unification is performed *along with* derivation of constraints, insteading of *after* that. Generally, the implementation can follow a certain pattern. First, call `typeCheck` on sub-expressions and get their respective `TypeResults`. Then, perform unification on the types of sub-expressions. Finally, return type of this expression. Once a new substitution is derived, it is immediately composed with previous one. The implementation of `typeCheck` in `ArithExpr` class can serve as a good example for this pattern.

```
@Override public TypeResult typeCheck(TypeEnv E) throws TypeError {
    // Check types of both operands
    var lhsTr = l.typeCheck(E);
    var rhsTr = r.typeCheck(E);
    var subst = lhsTr.s.compose(rhsTr.s);

    // Unify both types to `int`
    subst = subst.compose(lhsTr.t.unify(Type.INT));
    subst = subst.compose(rhsTr.t.unify(Type.INT));

    // Return typing result
    return TypeResult.of(subst, Type.INT);
}
```

Let-polymorphism

Algorithm

Here I adopt the algorithm introduced in Section 22.7 of *Types and Programming Languages*, instead of strictly following the rule in the specification. When a type t is bound to a name s, all type variables, except those already mentioned in the typing environment, $x_1, x_2, \ldots, x_n$, are generalized as $\forall x_i : t$. When that name is accessed during type checking, all generalized variables are instantiated with new ones $y_1, y_2, \ldots, y_n$ and type scheme $[y_1/x_1, y_2/x_2, \ldots, y_n/x_n]\ t$ is returned as result. By this algorithm can s takes on different forms.

Implementation

Several things have to be modified to support this feature. First, two additional methods should be implemented for `Type`:

```
public abstract Set<TypeVar> collect();
public abstract Type clone();
```

`collect` recursively collects all type variables and return them as a set. `clone` makes a copy of the original type so that the result and original `Type` object is not reference equal.

Second thing is to reimplement `TypeEnv` as explicit name-type pairs instead of the implicit function definition provided in the original implementation. This makes it possible to iterate through each entry in the typing environment.

```
public class TypeEnv {
    public TypeEnv E;
    public final Symbol x;
    public final Type t;

    /* Other methods ... */
}
```

The rest of reimplemented `TypeEnv` looks very likely to `Env`, so I will not show it here.

Then I implement let-polymorphism as a method `ofGeneralized` of `TypeEnv`. Its implementation follows the algorithm stated above. The generalized variables are stored in `generalized`. The instantiation step is implemented in overidden `get` method, which replaces all generalized variables.

```
public static TypeEnv ofGeneralized(final TypeEnv E, final Symbol x, final Type t) {
    // Collect all type variables from given type
    var generalized = t.collect();

    // Prune variables that are already mentioned in typing environment
    // The rest are the ones to be generalized
    generalized.removeIf((TypeVar tv) -> {
        var curE = E;
        while (!curE.isEmpty()) {
            if (curE.t.contains(tv))
                return true;
            curE = curE.E;
        }
        return false;
    });

```

```
17      // Override get method of this entry
18      return new TypeEnv(E, x, t) {
19          @Override public Type get(Symbol x) {
20              // Search inner entries if name does not match
21              if (this.x != x) {
22                  if (E != null)
23                      return E.get(x);
24                  else
25                      return null;
26              }
27
28              // Instantiate all generalized type variables
29              var ret = t.clone();
30              for (var tv : generalized){
31                  var inst = new TypeVar(tv.isEqualityType());
32                  ret = ret.replace(tv, inst);
33              }
34              return ret;
35          }
36      };
37  }
```

The rest work is simple, just replace `TypeEnv.of` with `TypeEnv.ofGeneralized` when necessary. Of course, when checking e_2 in `let` expression, we need this to bind e_1 to a given name. Besides, for generic library functions, we also need polymorphism, since their contexts are quite similar to e_1 in `let` expressions. This work is done in constructor of `DefaultTypeEnv`.

Semantics

State

From the specification, it can be known that the machine state is composed of environment E, memory M, and memory pointer p.

Environment

Environment E stores mapping from names to values. It can be composed with another mapping, if we want to create a new binding. We can also query the environment to find value bound to a certian name.

Memory

Memory M stores mappings from integer addresses to values. Memory pointer p stores the address for next reference cell. It can also be understood as the number of total reference cells at that time. This pointer helps allocation of new reference cells.

Values

Package `interpreter` contains several classes representing values. Since the language specification adopts big-step semantics, the evaluation result of any expression must be a value. For `Value` subclasses, only the following method need to be implemented:

```
@Override public boolean equals(Object other) { /* Implementation ... */ }
```

This is actually the method inherited from `Object`. Only values that are of equality type can be compared. Otherwise, it always returns `false`. Implementation of this method for values is trivial, so I skip it here.

Evaluation

`eval` method of AST nodes need to be implemented to support evaluation. Like in type checking, I just discuss the common pattern and list cases that need special care.

The pattern is quite simple. Just call `eval` on sub-expressions according to the order specified by the corresponding evaluation rule. Usually we need to check if the values returned match a certain pattern. Finally, perform operation on the values and return as result. `eval` method in `Add` could serve as a good example:

```
@Override public Value eval(State s) throws RuntimeError {
    var v1 = l.eval(s);
    if (!(v1 instanceof IntValue)) {
        throw new RuntimeError("lhs is not an integer");
    }
    var v2 = r.eval(s);
    if (!(v2 instanceof IntValue)) {
        throw new RuntimeError("rhs is not an integer");
    }
    return new IntValue(((IntValue) v1).n + ((IntValue) v2).n);
}
```

Name-Value Binding

Environment E stores all name-value bindings at runtime. It is queried when evaluating `Name` expressions. A new environment can be created by composing a new binding with a previous environment. There are three places where new bindings are created: `App`, `Let` and `Rec`. Name bindings created in `eval` method of `App` and `Let` are straightforward. `eval` method of `Rec` also create new binding, but the binding is stored in environment of the closure, instead of altering machine state.

Reference Cells

`Ref`, `Deref` and `Assign` nodes have something to do with reference cells in memory. `Ref` creates new reference cell and assign value to it. `Deref` reads value from a reference cell. `Assign` assigns another value to an exisiting reference cell, overwriting previous value. In my implementation, I abstract basic operations of memory as methods in `Mem` for better readability and extendability:

```java
public class Mem extends HashMap<Integer, Value> {

    public Value read(int ptr) { return get(ptr); }

    public void write(int ptr, Value val) { put(ptr, val); }

    public int alloc(State s) {
        var ptr = p.get();
        p.set(ptr + 1);
        return ptr;
    }
}
```

Predefined Functions

There are seven predefined functions that I have to implement: four library functions `fst`, `snd`, `hd` and `tl`; three PCF functions `iszero`, `pred` and `succ`. They follow similar implementation patterns. Here I choose `fst` as the example and go through all the work that have to be done.

Semantics

Semantics for predefined functions are defined by calling the constructor of its super class `FunValue`. Top level functions are in empty environment. Its parameter name can be arbitrary, as long as it is consistent with the one in function body. The body expresson is an anonymous subclass of `Expr`. There is nothing to do with `typeCheck` method because it will never be called. `eval` method is our concern. The implementation is quite straightforward. Finally, the implementation looks like this:

```java
super(Env.empty, Symbol.symbol("x"), new Expr() {
    Symbol x = Symbol.symbol("x");

    @Override public TypeResult typeCheck(TypeEnv E) {
        return null;
    }

    @Override public Value eval(State s) throws RuntimeError {
        var pairVal = s.E.get(x, s);
        if (!(pairVal instanceof PairValue)) {
            throw new RuntimeError("not a pair");
        }
        return ((PairValue) pairVal).v1;
    }
});
```

Environment Definition

Predefined functions should be added to environment to make it detectable in later evaluation. This work should be done in `initialEnv`, which is a static method of `InitialState`.

```java
E = Env.of(E, Symbol.symbol("fst"), new Fst());
```

Type Definition

Type definition of predefined functions should be added to type environment to ensure the type checking for its call will not fail. This work is done in constructor of `DefaultTypeEnv`. We start by an empty type environment, and successively add type definitions of functions to it. Types of the four library functions are all parameterized types. One thing to be notice is that, if two type parameters are definitely the same, they should be assigned the same type variable.

```
public DefaultTypeEnv() {
    // Create empty type environment as the starting point
    E = new TypeEnv();

    // Add type declarations for predefined functions
    var fstLhs = new TypeVar(true);
    var fstRhs = new TypeVar(true);
    E = ofGeneralized(E, Symbol.symbol("fst"), new ArrowType(new PairType(fstLhs, fstRhs),
fstLhs));

    /* Other predefined functions ... */
}
```

Bonus Features

Garbage Collection

Implementation

In SimPL interpreter, garbage collection means freeing reference cells that can no longer be used, allowing these locations to be allocated again in later evaluation. The key problem is how to know a reference cell is used. In SimPL, a reference is in use means it is bound to one or more names. The most convenient way to know this is to check the environment. If the environment contains mapping to this reference cell, it is used. Otherwise, it is not, and we can free this cell for a later allocation.

Thanks to encapsulation work done before, only `alloc` method in `Mem` needs to be modified. Mark-sweep algorithm is implemented here. The method iterates through all the name-value bindings in the environment, and marks all the cells in use. If all cells are in use, it increments the address counter and return address for the new cell. Otherwise, it returns the first cell not in use.

Result

Consider program `gc.spl`:

```
let a = ref 0 in
    let b = ref 1 in
        b := 2
    end;
    let d =
        let c = ref 3 in
            ref 4;
            ref 5;
            c
        end in
        !d
    end
end
```

This example demonstrates several possible cases where a reference cell could be used. The first cell is in use throughout the whole program. The second is bound to `b` but later lives out its scope. The third is bound to `c` first, and then `d`. The fourth and fifth are never bound to any name. Without GC, five difference cells are created. But with GC, only three are actually created.

Let's do a little hack on the `eval` method of `Ref` and see what happens:

```
@Override public Value eval(State s) throws RuntimeError {
    var cellVal = e.eval(s);
    var ptr = s.M.alloc(s);
    s.M.write(ptr, cellVal);
    System.out.println("ref@" + ptr); // print address of allocated cell
    return new RefValue(ptr);
}
```

Interesting part of output:

```
ref@0
ref@1
ref@1
ref@2
ref@2
```

It can be seen that only three different cells are allocated. GC works properly.

Lazy Evaluation

Implementation

In eager (call-by-value) evaluation strategy, E stores mappings from name to value. However, in lazy evaluation, E could also store mappings from name to expression *and* environment. For any expression whose value should be bound to a name in eager evaluation, we directly create a mapping from that name to the expression, plus the environment required to evaluate this expression. When a name is being evaluated, the interpreter should first evaluate that expression with corresponding environment. To avoid reevaluating that expression later, the interpreter caches the value in the environment.

The implementation is divided to two parts: creation of the mappings and evaluation of mapped expressions. To support this feature, `Env` class should be firstly modified as follows:

```java
public class Env {
    public final Env E;
    private final Symbol x;
    public Value v;
    private final Expr e;
    private final Env Ee;

    // Name-expression-environment mapping
    public Env(Env E, Symbol x, Expr e, Env Ee) {
        this.E = E;
        this.x = x;
        this.v = null;
        this.e = e;
        this.Ee = Ee;
    }

    public static Env of(Env E, Symbol x, Expr e, Env Ee) {
        return new Env(E, x, e, Ee);
    }

    /* Other members ... */
}
```

Only two places should create this kind of mapping: `App` and `Let`. `Rec` also creates new mapping, but it actually maps to a `RecValue`, and no expression or environment is involved. Evaluation of mapped expression is done in `get` method of `Env`. The algorithm is already stated, so I just show the code here:

```java
public Value get(Symbol y, State s) throws RuntimeError {
    if (x != y) { // symbol not found at this level
        if (E == null)
            return null;
        else
            return E.get(y, s);
    }
    if (v == null) { // not evaluated yet
        assert e != null;
        v = e.eval(State.of(Ee, s.M, s.p));
    }
    return v;
}
```

Result

Run all the provided test cases with lazy evaluation and GC enabled.

```
doc/examples/plus.spl
int
3
doc/examples/factorial.spl
int
24
```

```
doc/examples/gcd1.spl
int
1029
doc/examples/gcd2.spl
int
0
doc/examples/max.spl
int
2
doc/examples/sum.spl
int
6
doc/examples/map.spl
((tv59 -> tv66) -> (tv59 list -> tv66 list))
fun
doc/examples/pcf.sum.spl
(int -> (int -> int))
fun
doc/examples/pcf.even.spl
(int -> bool)
fun
doc/examples/pcf.minus.spl
int
46
doc/examples/pcf.factorial.spl
int
720
doc/examples/pcf.fibonacci.spl
int
6765
doc/examples/letpoly.spl
int
0
```

The output of programs are the same as in eager evaluation, except for `gcd2.spl` (see Appendix for reference). It is expected to output `1029`, but we got `0` instead. Let's consider this program:

```
let gcd = fn x => fn y =>
    let a = ref x in
        let b = ref y in
            let c = ref 0 in
                (while !b <> 0 do c := !a; a := !b; b := !c % !b);
                !a
            end
        end
    end
in  gcd 34986 3087
end
```

Since it is an imperative program, it is likely that the use of reference cells causes this problem. Let's take a look at address of each allocation, just like in the previous section:

```
ref@0
ref@1
ref@0
```

In call-by-value evaluation, three different cells should be created, but here we only got two. The first cell is bound to `b` in when evaluating `!b <> 0`, the second bound to `c` and the third to `a` when evaluating `c := !a`. The reason why `a` and `b` share the same cell is that the environment for `ref x` contains nothing. When evaluating `!a`, the allocation procedure with GC takes it for granted that all cells are not in use, so it assigns `ref@0` to `a`. If we follow the code, it is easy to find that value stored in `ref@0` is zero in the first iteration of `while` loop, and it exits the loop because `!b` is also zero. By `!a`, we get result `0` for this program.

We can draw a conclusion that GC could lead to surprising result for a program that is not purely functional in lazy evaluation. If we disable GC, and still run `gcd2.spl` in lazy mode, the result is correct:

```
doc/examples/gcd2.spl
int
1029
```

Mutually Recursive Combinator

Design

I have to extend the original SimPL definition to support this feature. When designing this syntax feature, I refer to OCaml language. In OCaml, the syntax for MRC looks like this:

```
let rec function1-nameparameter-list =
function1-body
and function2-nameparameter-list =
function2-body
```

Combining this and characteristics of SimPL, I design syntax for MRC as follows:

$$
\begin{array}{rll}
e ::= & \ldots & \textit{expressions} \\
\mid & \texttt{let}\ x = e\ \texttt{and}\ x = e\ \texttt{in}\ e\ \texttt{end} & \textit{mutually recursive combinator}
\end{array}
$$

The evaluation rule is similar to E-Let, but much trickier than that. It seems that when evaluating e_1, E have to contain mapping from y, and it requires e_2 to be evaluated first. But evaluation of e_2 also needs x from environment, which requires e_1 to be evaluated first. Circularity is intrinsic for MRC here. There's a way to solve this. Since e_1 must be a function definition, it could evaluates to a closure, whose environment just contains neither x or y. After evaluating e_1 and e_2 to two closures, add two mappings $x \mapsto (\texttt{fun}, E', s, e_1')$ and $y \mapsto (\texttt{fun}, E', t, e_2')$ to the environment, with the environment of both closures the newly created environment. That's why E' appears on both sides of the third premise.

$$
\frac{\begin{array}{l}E, M, p; e_1 \Downarrow M', p'; (\texttt{fun}, E_1, s, e_1') \quad E, M', p'; e_2 \Downarrow M'', p''; (\texttt{fun}, E_2, t, e_2') \\ E' = E[x \mapsto (\texttt{fun}, E', s, e_1')][y \mapsto (\texttt{fun}, E', t, e_2')] \quad E', M'', p''; e_3 \Downarrow M''', p'''; v\end{array}}{E, M, p; \texttt{let}\ x = e_1\ \texttt{and}\ y = e_2\ \texttt{in}\ e_3 \Downarrow M''', p'''; v} \ (\text{E-LetAnd})
$$

Similar circularity problem arises in terms of typing rule. We can solve this by assuming the type form of e_1 and e_2. Since e_1 and e_2 are all functions, they must have form $t \to t$. We can assume e_1 to be of type $t_1 \to t_2$ and e_2 of type $t_3 \to t_4$, where $t_i, i \in 1..4$ are type variables, instead of concrete types. Types can be checked for e_1 and e_2, whose results are $t_1' \to t_2'$ and $t_3' \to t_4'$ respectively. Then we check type of e_3 with mapping $x : t_1' \to t_2'$ and $y : t_3' \to t_4'$, resulting t, which is the type for the whole expression.

$$\frac{\Gamma[y : t_3 \to t_4] \vdash e_1 : t_1' \to t_2' \quad \Gamma[x : t_1 \to t_2] \vdash e_2 : t_3' \to t_4' \quad \Gamma[x : t_1' \to t_2'][y : t_3' \to t_4'] \vdash e_3 : t}{\Gamma \vdash \texttt{let}\ x = e_1\ \texttt{and}\ y = e_2\ \texttt{in}\ e_3 : t} \text{ (T-LetAnd)}$$

Implementation

Since the syntax is defined by myself, I have to modify the grammar file and regenerate lexer and parser classes for the new grammar. In file `simpl.lex`, I add `and` keyword in `<YYINITIAL>` block. This enables lexer to recognize this keyword as a token.

```
<YYINITIAL> {
    ...
    "and"     { return token(AND); }
    ...
}
```

In `simpl.grm`, I first add `AND` as a terminal:

```
terminal LET, AND, IN, END;
```

Then specify the syntax of MRC:

```
e ::= ...
    | LET ID:x EQ e:e1 AND ID:y EQ e:e2 IN e:e3 END {: RESULT = new LetAnd(symbol(x), e1, symbol(y), e2, e3); :}
    ;
```

In package `parser.ast`, create a new class `LetAnd`:

```
public class LetAnd extends Expr {
    public Symbol x, y;
    public Expr e1, e2, e3;

    public LetAnd(Symbol x, Expr e1, Symbol y, Expr e2, Expr e3) {
        this.x = x;
        this.y = y;
        this.e1 = e1;
        this.e2 = e2;
        this.e3 = e3;
    }

    /* Other methods */
}
```

Then run the `Makefile` script in `parser` directory, `Lexer` and `Parser` are automatically generated in this directory. The rest work is to implement `typeCheck` and `eval` methods. The implementation just follows the rules stated before.

Result

Consider program `mrc.even.spl` which decides whether a number is even or odd:

```
let iseven = fn n =>
    if iszero n then true else isodd (pred n)
and isodd = fn n =>
    if iszero n then false else iseven (pred n)
in iseven 3
end
```

This program contains two mutually recursive functions: `iseven` and `isodd`. I pass `3` to `iseven` so that either function will be called at least twice. This could test whether the environment is correct. Run the interpreter on this program and it outputs:

```
doc/examples/mrc.even.spl
bool
false
```

The typing and evaluation results are all correct. This feature is properly implemented.

Infinite Stream

A stream is an infinite list. Like a list, a stream value contains two members, and the first is current element. The main difference is that its second field is a function $\lambda x.\, e$ of type $unit \to 'a\ stream$ which specifies how the following elements could be produced by the stream. The expression e will only be evaluated when the next element is actually needed, producing a new stream starting at the next position.

Implementation

As preparation, I create class `StreamType` for typing and class `StreamValue` for runtime value representation. Their definitions are quite similar to their list counterparts so I omit them here. Then comes the critical part: implementation of basic operations on streams. I decide to implement them as library functions, so no new syntax is introduced. In this project, three operations are supported: `stream` for stream creation, `take` for building finite list from stream, `drop` for dropping elements from stream.

`stream` creates a stream from an element and a function. Its type is $'a \to (unit \to 'a\ stream) \to 'a\ stream$. As a library function, we only need to override its evaluation method. However, unlike other library functions implemented before, this one has two arguments, so there is a nested function inside it. When `streams` takes one argument, it returns another function with first argument in environment. The actual construction process is done in the body of inner function. The code is shown in the following.

```
public class Stream extends FunValue {
    public Stream() {
        super(Env.empty, Symbol.symbol("x"), new Expr() {
            Symbol x = Symbol.symbol("x");

            @Override public TypeResult typeCheck(TypeEnv E) {
                return null;
            }

```

```
            @Override public Value eval(State s) throws RuntimeError {
                var elemVal = s.E.get(x, s);
                return new FunValue(s.E, Symbol.symbol("f"), new Expr() {
                    Symbol f = Symbol.symbol("f");

                    @Override public TypeResult typeCheck(TypeEnv E) {
                        return null;
                    }

                    @Override public Value eval(State s) throws RuntimeError {
                        var funVal = s.E.get(f, s);
                        if (!(funVal instanceof FunValue)) {
                            throw new RuntimeError("not a function");
                        }
                        return new StreamValue(elemVal, (FunValue) funVal);
                    }
                });
            }
        });
    }
}
```

`take` takes an integer `n` and a stream `s` and returns a list containing `n` elements from the beginning of `s`. Its type is $int \to 'a\ stream \to 'a\ list$. The mechanism of this function is simple. It just need to continously take elements from stream and evaluate the function in the stream to get next stream, until there are enough elements required by `n`. One thing that need to be taken care is that, in SimPL, the tail of a list lies in the deepest position in AST. When we collect elements in forward direction, we have to construct the list value in *reverse* direction. The `eval` method for inner expression is listed below:

```
@Override public Value eval(State state) throws RuntimeError {
    // Check stream argument
    var streamVal = state.E.get(s, state);
    if (!(streamVal instanceof StreamValue)) {
        throw new RuntimeError("not a stream");
    }

    // Store elements in an array
    var array = new ArrayList<Value>();
    var curStream = (StreamValue) streamVal;
    for (var i = 0; i < ((IntValue) numVal).n; i++) {
        array.add(curStream.x);
        var funcVal = curStream.f;
        var nextStream = funcVal.e.eval(State.of(funcVal.E, state.M, state.p));
        if (!(nextStream instanceof StreamValue))
            throw new RuntimeError("not a stream");
        curStream = (StreamValue) nextStream;
    }

    // Assemble list using cons in reverse direction
    Value list = new NilValue();
```

```
    for (var i = array.size() - 1; i >= 0; i--)
        list = new ConsValue(array.get(i), list);
    return list;
}
```

`drop` takes an integer `n` and a stream `s` and returns another stream with first `n` elements dropped from `s`. Its type is $int \to 'a\ stream \to 'a\ stream$. The implementation is similar to `take` except that no elements should be collected, and the final stream should be returned. The code is omitted here.

Result

Consider program `stream.spl`.

```
let fib =
    let gen = rec f => fn a => fn b =>
        stream a (fn u => f b (a + b))
    in gen 1 1 end
in take 20 (drop 4 fib) end
```

`fib` generate a stream of Fibonacci sequence. `take 20 (drop 4 fib)` drops the first four numbers in this stream and then take 20 numbers from remaining stream. To show content of this list, I modified `toString` method of `ConsValue`.

```
public String toString() { return v1 + "::" + v2; }
```

Run interpreter on this program, we can get the following output.

```
doc/examples/stream.spl
int list
5::8::13::21::34::55::89::144::233::377::610::987::1597::2584::4181::6765::10946::17711::286
57::46368::nil
```

The output is correct. Actually we can take far more elements from the stream. Without stream we can still get this sequence by defining a recursive function, but that could easily cause a stack overflow of interpreter, even when the required length is not so large. But with stream we don't need to worry about that. The only limit is the size of heap memory.

Appendix

Notice

- If there are too many levels of recursion in input program, JVM could possibly throws `StackOverflow` exception. If you are sure that the program will not cause infinite recursion, try set stack size of JVM larger through `-Xss` argument, for example `-Xss8m` if a stack of 8MB is desired.
- Two bonus features: GC, lazy evaluation, can be enabled and disabled by setting constants in `Feature` class in package `simpl.interpreter`. Whether to enable these features should be decided before compilation. Mutually recursive combinator only works in eager evaluation.

```java
public class Feature {
    // Whether to enable garbage collection
    public static final boolean GC = true;
    // Whether to enable lazy evaluation
    public static final boolean LAZY = false;
}
```

Output

The following is the output of all provided test cases, excluding those written by myself. The configuration of the interpreter is: (1) eager (call-by-value) evaluation strategy (2) GC enabled. The exported `jar` also uses this configuration.

```
doc/examples/plus.spl
int
3
doc/examples/factorial.spl
int
24
doc/examples/gcd1.spl
int
1029
doc/examples/gcd2.spl
int
1029
doc/examples/max.spl
int
2
doc/examples/sum.spl
int
6
doc/examples/map.spl
((tv77 -> tv78) -> (tv77 list -> tv78 list))
fun
doc/examples/pcf.sum.spl
(int -> (int -> int))
fun
doc/examples/pcf.even.spl
(int -> bool)
fun
doc/examples/pcf.minus.spl
int
46
doc/examples/pcf.factorial.spl
int
720
doc/examples/pcf.fibonacci.spl
int
6765
doc/examples/letpoly.spl
```

```
int
0
```