CS383 Course Project - SimPL Interpreter

Yanning Chen   519021910103

June 5, 2022

Contents

1 Introduction

In this project, it's required to implement an *interpreter* for the programming language SimPL (pronounced *simple*). SimPL is a simplified dialect of ML, which can be used for both *functional* and *imperative* programming. This document presents an overview of this SimPL interpreter and describes basic algorithms and internal designs of this implementation.

2 Differences with specification

There are some differences between the specification and this implementation that need to be taken into consideration.

Evaluation strategy Lazy evaluation is the default strategy for evaluating expressions. When met with side effects, it might yield different outputs from eager evaluation.
To opt out, add `{-# Strict #-}` pragma to the top of the program.

Stack size Due to lazy evaluation, it's easier to overflow the stack.
The default stack size is set to 10240000. To set a greater value, add `{-# StackSize=<value> #-}` pragma to enlarge the stack.

GC GC is enabled by default. If memory usage is not a problem, it might be better to disable GC for better performance.
To disable GC, add `{-# GCThreshold=1 #-}` pragma to trigger GC only when heap overflow occurs.

Heap Usage The max heap usage is capped at `65536` by default.
To increase the limit, add `{-# HeapSize=<value> #-}` pragma.

3 Project Summary

3.1 Dependencies

JDK 17 Development of this project utilizes *Bellsoft Liberica JDK 17.0.3*. But any JDK compliant with *Java SE 16* is also acceptable.

Gradle 7.2 This project uses *Gradle* to manage dependencies and build tasks.

JFlex 1.8.2 *JFlex* is used to generate the lexical analyzer for SimPL.

CUP 0.11b *CUP* is used to generate the LALR parser for SimPL.

Kala Common 0.38.0 *Kala Common* provides modern container and utility classes that is similar to those provided by *Scala*'s standard library.

JUnit 5.8.2 *JUnit* is used to test the correctness of the implementation.

3.2 Structure

3.2.1 Parser

Package `simpl.parser` contains lexer and parser for SimPL. It also contains surface *AST* definitions (`Expr`) for this language.
A complete parser for SimPL has already been provided in the project template. The evaluation and typechecking of this language is *syntax-directed*, so most part of the evaluation and typechecking (without unification) algorithm is implemented in the AST module.

3.2.2 Interpreter

Package `simpl.interpreter` contains CLI, builtin functions, basic execution environments, and core `Value` types.

Once surface ASTs are typechecked, they are evaluated and converted to `Values`. `Values` are all *well-typed* terms, while AST `Exprs` may not.

3.2.3 Typing

Package `simpl.typing` contains type structures and unification algorithm.

Every type, type environment, and substitution are defined in this package. The unification algorithm is implemented in these classes. Combined with the AST module, this package provides type inference and type checking functionality of SimPL.

4 Typing

SimPL uses *Hindley-Milner* type system. It is *STLC* with *prenex polymorphism*, and its type inference is deterministic. Therefore, SimPL doesn't have explicit type annotation, and typing is completely handled by *principle type inference*.

This section will introduce the implementation of its type definition and type inference algorithm.

4.1 Basic structs

4.1.1 Type Scheme

Hindley-Milner type system has *let-polymorphism*. To achieve this, a limited ∀ quantifier is added to the type system. Different from *System-F*, ∀ is only allowed to appear in the foremost position of a type scheme, i.e. it must be in *prenex normal form*.

Here we define type schemes as follows:

$$\begin{aligned}\sigma ::= &\ \tau \\ &|(\forall\ \alpha\ \sigma)\end{aligned}$$

In the implementation, all type schemes implements the `TypeScheme` interface, and it has two kinds of implementations, `Type` interface and `Forall`.

```java
public interface TypeScheme {
    /**
     * Replace a type variable with a type.
     *
     * @param a the type variable to replace.
     * @param t the type to replace the type variable with.
     * @return the type scheme after replace.
     */
    TypeScheme replace(TypeVar a, Type t);

    /**
     * Instantiate this type scheme.
     *
     * @return the instantiated type.
     */
    Type instantiate();
}

public record Forall(TypeVar a, TypeScheme s) implements TypeScheme { … }
```

4.1.2 Type

All types implement `Type` interface, which extends `TypeScheme` interface. `Type` provides a common interface for all types, including unification, generization, collecting free variables, etc.

```java
public interface Type extends TypeScheme {
    // NIL and UNIT are moved to `NilValue.INSTANCE` and `UnitValue.INSTANCE` to avoid vm deadlock.
    // See https://bugs.openjdk.java.net/browse/JDK-6301579.

    /**
     * Determine the equality decidability.
     *
     * @return true if this type is equality type, otherwise false.
     */
    boolean isEqualityType();

    /**
     * Check occurrence of a specific type variable.
     *
     * @param tv the type variable to check.
     * @return true if this type contains the specified type variable, otherwise false.
     */
    boolean contains(TypeVar tv);

    /**
     * Unify this type with another type.
     *
     * @param t the other type to unify with.
     * @return the substitution that makes this type equal to the other type.
     * @throws TypeError if the unification fails.
     */
    Substitution unify(Type t) throws TypeError;

    /**
     * Replace a type variable with a type.
     *
     * @param a the type variable to replace.
     * @param t the type to replace the type variable with.
     * @return the type after replace.
     */
    @Override
    Type replace(TypeVar a, Type t);

    /**
     * Collect all free type variables in this type.
     *
     * @return the set of free type variables.
     */
    ImmutableSet<TypeVar> freeTypeVars();

    /**
     * Generalize all free type variables in this type.
     *
     * @param E the type environment to generalize under.
     * @return the generalized type scheme.
     */
    default TypeScheme generalizeWith(@NotNull TypeEnv E) { … }

    /**
     * Instantiate this type scheme.
     *
     * @return the instantiated type.
     */
    @Override
    default Type instantiate() { … }
}
```

4.1.3 Substitution

For convenience, the substitution info is stored like a tree. It is implemented as `Substitution` interface, and it has three implementations. Apply a substitution (on a type scheme) is generally a DFS traversal.

Identity The identity substitution.

Replace The substitution that replaces a type variable with a type.

Compose The substitution that composes two substitutions.

```java
public interface Substitution {

    Substitution IDENTITY = new Identity();

    <T extends TypeScheme> T applyOn(T t);

    default TypeEnv applyOn(final TypeEnv E) { … }
    …
    final class Identity extends Substitution {
        …
        @Override
        public <T extends TypeScheme> T applyOn(T t) {
            return t;
        }
    }

    final class Replace extends Substitution {
        private final TypeVar a;
        private final Type t;
        …
        @Override
        public <T extends TypeScheme> T applyOn(@NotNull T b) {
            return (T) b.replace(a, t);
        }
    }

    final class Compose extends Substitution {
        private final Substitution f;
        private final Substitution g;
        …
        @Override
        public <T extends TypeScheme> T applyOn(T t) {
            return f.applyOn(g.applyOn(t));
        }
    }
}
```

4.1.4 Type Environment

Type environment is a mapping from type variables to type schemes.

$$\begin{aligned}\Gamma ::= &\ \cdot \\ &|\Gamma, x : \sigma\end{aligned}$$

It's stored as a cons list.

```java
public class TypeEnv {
    public static final TypeEnv DEFAULT = withBuiltIns();
    protected final Symbol x;
    protected final TypeScheme t;
    private final TypeEnv E;

    public TypeScheme get(Symbol x) { … }

    public ImmutableSet<Symbol> typeVars() { … }
}
```

4.2 Type inference

Given the definition of type scheme, substitution and unification, we may now implement the type inference algorithm.

Notice that there are generally two styles of type inference algorithms: algorithm W/J and constraints generation. Basically, the former mixes constraint generation and solving, and the latter first collect contraints by bottom-up traversal, which will be solved independently.

In the lecture, the latter approach is taken. It makes extension to type system easier. However, in this implementation, algorithm W is used. In algorithm W, constraints are solved immediately when they are generated, and are not solved independently. Therefore, we need to modify the unification algorithm a bit to make it work.

4.2.1 Unification

$$\frac{}{\alpha \sim \alpha \rightsquigarrow \cdot}U_{same} \qquad \frac{\alpha \notin FV(\tau)}{\alpha \sim \tau \rightsquigarrow [\tau/\alpha]}U_{left} \qquad \frac{\alpha \sim \tau \rightsquigarrow S}{\tau \sim \alpha \rightsquigarrow S}U_{right}$$

$$\frac{}{int \sim int \rightsquigarrow \cdot}U_{int} \qquad \frac{}{bool \sim bool \rightsquigarrow \cdot}U_{bool}$$

$$\frac{\begin{array}{c}\tau_{11} \sim \tau_{21} \rightsquigarrow S_1 \\ S_1\tau_{12} \sim S_1\tau_{22} \rightsquigarrow S_2\end{array}}{(\tau_{11} \to \tau_{12}) \sim (\tau_{21} \to \tau_{22}) \rightsquigarrow S_2 \circ S_1}U_{arrow}$$

(Remaining cases are omitted and can be found in the source code.)

Notice how U_{arr} is different from the one given in the lecture. After unifying τ_{11} and τ_{21} producing a substitution S, S is applied to τ_{12} and τ_{22} before unifying them, in order to propagate the substitution collected. This is not needed in constraint generation based inference because if solving is deferred, substituting a variable in the whole (S, c) set is enough to propagate the information.

To elaborate, the following code implements the unification algorithm for `ArrowType`.

```java
    @Override
    public Substitution unify(Type t) throws TypeError {
        try {
            if (t instanceof ArrowType rhs) {
                var S1 = t1.unify(rhs.t1);
                var S2 = S1.applyOn(t2).unify(S1.applyOn(rhs.t2));

                return S2.compose(S1);          // U_arrow
            } else if (t instanceof TypeVar) {
                return t.unify(this);           // U_right
            }
        } catch (TypeError e) {
            e.appendTrace(this, t);             // append unify trace for debugging
            throw e;                            // rethrow the exception
        }

        throw new TypeMismatchError(this, t);
    }
```

4.2.2 Let-polymorphism

To deal with $\forall$ quantifier, there are two additional initial rules for typing that is not syntax-directed and is not mentioned in the specification.

$$\frac{\Gamma \vdash e : (\forall\ \alpha\ \sigma)}{\Gamma \vdash e : \sigma[\tau/\alpha]}T_{inst} \qquad \frac{\Gamma \vdash e : \sigma \quad \alpha \in FV(\Gamma)}{\Gamma \vdash e : (\forall\ \alpha\ \sigma)}T_{gen}$$

A type scheme can be instantiated by replacing its bound variable with a concrete type. Also, any type variable of a type scheme that is free in the type environment can be generalized with ∀ quantifier, because they are unconstrained.

The recursive type scheme instantiate method is implemented as follows. This method is referred as `inst(type)` later.

```java
public record Forall(TypeVar a, TypeScheme s) implements TypeScheme {
    @Override
    public Type instantiate() {
        return s.replace(a, new TypeVar(true)).instantiate();
    }
}
public interface Type extends TypeScheme {
    @Override
    public Type instantiate() {
        return this;
    }
}
```

And the type generalization method is implemented as follows. This method is referred as `gen(type, Env)` later.

```java
public interface Type extends TypeScheme {
    /**
     * Generalize all free type variables in this type.
     *
     * @param E the type environment to generalize under.
     * @return the generalized type scheme.
     */
    default TypeScheme generalizeWith(@NotNull TypeEnv E) {
        var tFreeVars = freeTypeVars();
        var eBoundVars = E.typeVars();

        var genVars = tFreeVars.view().filterNot(v -> eBoundVars.contains(v.name))
                               .collect(ImmutableCompactSet.factory());

        return genVars.foldLeft((TypeScheme) this, (acc, a) -> new Forall(a, acc));
    }
}
```

4.2.3 Algorithm W

Let $W(\Gamma; x) = (S; \tau)$ be type checking expression x under the type environment E, returning substitution S and type τ.

$$\frac{\tau = \mathit{inst}(\Gamma(x))}{W(\Gamma; x) = (\cdot; \tau)} W_{var}$$

$$\frac{\begin{array}{c} W(\Gamma; e_1) = (S_1; \tau_1) \\ \sigma = \mathit{gen}(\tau_1, S_1\ \Gamma) \\ W(S_1\ \Gamma, x : \sigma; e_2) = (S_2; \tau_2) \end{array}}{W(\Gamma; \mathit{let}\ x = e_1\ \mathit{in}\ e_2) = (S_2 \circ S_1; \tau_2)} W_{let}$$

$$\frac{W(\Gamma, x : \alpha; e) \vdash (S; \tau)}{W(\Gamma; (\lambda x.e)) = (S; (S\ \alpha) \to \tau)} W_{abs}$$

$$\frac{\begin{array}{c} W(\Gamma; l) = (S_1; \tau_1) \\ W(S_1 \circ \Gamma; r) = (S_2; \tau_2) \\ S_2\ \tau_1 \sim (\tau_2 \to \alpha) \rightsquigarrow S_3 \end{array}}{W(\Gamma; l\ r) = (S_3 \circ S_2 \circ S_1; S_3\ \alpha)} W_{app}$$

$$\frac{}{W(\Gamma; \mathit{true}) = (\cdot; \mathit{bool})} W_{true} \qquad \frac{}{W(\Gamma; \mathit{false}) = (\cdot; \mathit{bool})} W_{false}$$

$$\frac{\begin{array}{c} W(\Gamma; l) = (S_1; \tau_1) \\ W(S_1 \circ \Gamma; r) = (S_2; \tau_2) \\ S_2\ \tau_1 \sim \tau_2 \rightsquigarrow S_3 \\ eqType(S_3 \circ S_2\ \tau_1) \\ eqType(S_3\ \tau_2) \end{array}}{W(\Gamma; l\ r) = (S_3 \circ S_2 \circ S_1; bool)} W_{eq}$$

$$\frac{\begin{array}{c} W(\Gamma; e_1) = (S_1; \tau_1) \\ W(S_1\ \Gamma; e_2) = (S_2; \tau_2) \\ W(S_2 \circ S_1\ \Gamma; e_3) = (S_3; \tau_3) \\ S_3 \circ S_2\ \tau_1 \sim bool \rightsquigarrow S_4 \\ S_4 \circ S_3\ \tau_2 \sim S_4\ \tau_3 \rightsquigarrow S_5 \end{array}}{W(\Gamma; if\ e_1\ then\ e_2\ else\ e_3) = (S_5 \circ S_4 \circ S_3 \circ S_2 \circ S_1; S_5 \circ S_4\ \tau_3)} W_{if}$$

(Remaining cases are omitted and can be found in the source code.)

Note that to type check a variable, the type scheme must be instantiated after seeking it in the type environment. For the let binding, e_1 is generalized with respect to the type environment, then it is added to the type environment when checking e_2.

Similar to the modified unification algorithm, substitutions must be applied to any types or environments existed before that substitution was created.

To elaborate, the following code implements the type checking algorithm for let binding.

```java
public record Let(Symbol x, Expr e1, Expr e2, Boolean strict) implements Expr {
    @Override
    public TypeResult typeCheck(TypeEnv E) throws TypeError {
        var W1 = e1.typeCheck(E);
        var s = W1.ty().generalizeWith(W1.subst().applyOn(E));
        var W2 = e2.typeCheck(TypeEnv.of(W1.subst().applyOn(E), x, s));
        return TypeResult.of(W2.subst().compose(W1.subst()), W2.ty());
    }
}
```

Algorithm W is sound and complete[1].

5 Evaluation

5.1 State

A state $s \in$ **State** is a triple (E, M, c) where $E \in$ **Env** is the environment, $E \in$ **Mem** is the memory, and c is the global config.

5.1.1 Environment

Environment E is a mapping from names to values. Environments can be queried for a value by name, extended by adding new bindings, and composed with another environment. However, updating an existing binding is not allowed.

```java
public class Env implements MapView<Symbol, Value> {
    public static final Env EMPTY = …;
    private final Env E;
    private final Symbol x;
    private final Value v;
    …
    @Contract(value = "_, _, _ -> new", pure = true)
    public static @NotNull Env of(Env E, Symbol x, Value v) { … }

    public @NotNull Env extend(Symbol x, Value v) { … }

```

[1]Milner, Robin (1978). "A Theory of Type Polymorphism in Programming".

```
    @Override
    public @NotNull MapIterator<Symbol, Value> iterator() { … }

    public Value get(@NotNull Symbol y) { … }
}
```

5.1.2 Memory

Memory M is a mapping from $\mathbb{N}$ to values. However, most interfaces of a map is not exposed. A cell can be allocated from it to hold a value, which is barely a reference to the memory and an index. It can be used to read/write a memory location. Manual deallocation is not allowed. GC is dicussed later in this document.

For efficiency, memory is implemented as a continuous vector.

```
public class Mem {
    private final MutableCell<MutableList<Value>> storage = …;

    public Cell allocate(@NotNull State s, Value initialValue) throws RuntimeError { … }

    public static class Cell {
        protected final MutableCell<MutableList<Value>> storage;
        protected final MutableCell<Integer> pointer;

        public Value get() { … }

        public void set(Value v) { … }

        @Override
        public boolean equals(Object o) { … }
    }

}
```

Note that `MutableCell` is an auxiliary class for interior mutability. It's basically an `AtomicCell` without thread safety.

5.1.3 Global Config

Config c is an object that contains the configuration for this execution. It stores immutable information such as allowed stack/heap usage, default evaluation strategy, gc strategy, and so on.

```
public record Config(Boolean strict, Float gcThreshold, Integer stackSize, Integer heapSize) {…}
```

5.2 Value

`Value` corresponds to *value* in big-step semantics of the language. It has a method to determine semantics equivalence with another value.

```
public sealed interface Value permits BoolValue, ConsValue, FunValue, IntValue, NilValue,
        PairValue, RecValue, RefValue, StreamValue, ThunkValue, UnitValue {

    boolean equals(State s, Value other) throws RuntimeError;
    …
}
```

5.3 Expr

`Expr` is a parse tree of a *SimPL* program, and also a term that is not yet typechecked or evaluated. Both can be done by calling `typeCheck` or `eval` respectively.

```
public interface Expr extends EntryPoint {
    /**
     * Type check this expression.
     *
     * @param E The type environment.
     * @return Constraints and type of this expression.
     * @throws TypeError If this expression is ill-typed.
     */
    TypeResult typeCheck(TypeEnv E) throws TypeError;

    /**
     * Evaluate this expression in the given state.
     *
     * @param s The state to evaluate in.
     * @return The value of this expression.
     * @throws RuntimeError If this expression cannot be evaluated.
     */
    Value eval(State s) throws RuntimeError;
    …
}
```

5.4 Call-by-value evaluation

Call-by-value evaluation is implemented following the specification. The implementation of *add* is shown here for illustration.

```
public class Add extends ArithExpr {
    …
    @Override
    public Value eval(State s) throws RuntimeError {
        // E-Add.
        if (l.eval(s) instanceof IntValue lhs) {
            if (r.eval(s) instanceof IntValue rhs) {
                return new IntValue(lhs.n() + rhs.n());
            }
            throw new RuntimeError(r + " is not an integer");
        }
        throw new RuntimeError(l + " is not an integer");
    }
}
```

More details can be found in the source code.

5.5 Semantics equivalence

Semantics equivalence check is implemented following the specification. The implementation of *cons* is shown here for illustration.

```
public record ConsValue(@NotNull Value v1, @NotNull Value v2) implements Value {
    …
    @Override
    public boolean equals(State s, Value o) {
        if (this == o)
            return true;
        if (o == null  getClass() != o.getClass())
            return false;
        ConsValue consValue = (ConsValue) o;
        return v1.equals(consValue.v1) && v2.equals(consValue.v2);
    }
}
```

More details can be found in the source code.

6 Predefined functions

There are 7 predefined functions: `fst`, `snd`, `hd`, `tl`, `isZero`, `pred`, and `succ`.

6.1 Semantics

All functions are defined as `FunValues`. Two approaches are used to implement these functions.

For `isZero`, `pred` and `succ`, they can be defined in theory level.

Take `succ` as an example: it can be defined as $\lambda x.x + 1$. So the implementation goes by manually converting the value to *AST*.

```java
public class Succ extends FunValue {
    public static final FunValue INSTANCE = new Succ();

    private Succ() {
        super(Env.EMPTY, Symbol.of("x"), new Add(new Name(Symbol.of("x")), new IntegerLiteral(1)));
    }
}
```

For the rest functions, they can only be implemented in meta-theory level, because there's no pattern matching or eliminators in our language. To consume a *list* or *pair*, direct *AST* inspection is needed.

Take `fst` as an example:

```java
public class Fst extends FunValue {
    public static final FunValue INSTANCE = new Fst();

    private Fst() {
        super(Env.EMPTY, Symbol.of("x"), new Expr() {
            @Override
            public TypeResult typeCheck(TypeEnv E) { return null; }

            @Override
            public Value eval(State s) throws RuntimeError {
                var x = new Name(Symbol.of("x")).eval(s);
                if (x instanceof PairValue pair) {
                    return pair.v1();
                }
                throw new RuntimeError(x + " is not a pair");
            }
        });
    }
}
```

The predefined functions are inserted into the initial environment.

```java
public class State {
    public static final Env initialEnv =
            Env.EMPTY.extend(Symbol.of("fst"), Fst.INSTANCE)
                    .extend(Symbol.of("snd"), Tl.INSTANCE)
                    …
                    .extend(Symbol.of("succ"), Succ.INSTANCE);
    …
    public static @NotNull State create(Config c) {
        var E = initialEnv;
        …
        return State.of(E, M, c);
    }
}
```

6.2 Typing

The types of the predefined functions are hard-coded into the initial typing environment. For sake of let polymorphism, all arrow types are generalized with ∀ quantifier.

```java
public class TypeEnv {
    public static final TypeEnv DEFAULT = withBuiltIns();
    private static TypeEnv withBuiltIns() {
        var tyFst = new TypeVar(true);
        var tySnd = new TypeVar(true);
        …
        var defaultTypes = ImmutableSeq.of(
                // fst: t1 × t2 → t1
                Tuple.of(Symbol.of("fst"), ArrowType.of(PairType.of(tyFst, new TypeVar(true)), tyFst)),
                // snd: t1 × t2 → t2
                Tuple.of(Symbol.of("snd"), ArrowType.of(PairType.of(new TypeVar(true), tySnd), tySnd)),
                …
                // pred: int → int
                Tuple.of(Symbol.of("pred"), ArrowType.of(IntType.INSTANCE, IntType.INSTANCE))
        );

        return defaultTypes.foldLeft(new TypeEnv(),
            (E, symTy) -> TypeEnv.of(E, symTy._1, symTy._2.generalizeWith(E))
        );
    }
}
public class Interpreter {
    …
    public Type typeCheck() throws TypeError {
        return program.typeCheck(TypeEnv.DEFAULT).ty();
    }
}
```

7 Additional features

7.1 Pragma

This implementation of SimPL supports pragma. A pragma is a special instruction placed in the beginning of the source code that can be used to modify the behavior of the runtime.

For example, these are valid pragmas:

```
{-# GCThreshold=1024 #-}
{-# Strict MaxStack=10240000 #-}
```

To implement this feature, a new interface is introduced:

```
public interface EntryPoint {
    /**
     * @return pragmas for this entry point
     */
    ImmutableMap<String, String> pragmas();

    /**
     * @return body of this entry point
     */
    Expr expr();
}

public interface Expr extends EntryPoint { … }
public record Pragma(String pragma, String value, EntryPoint next) implements EntryPoint { … }
```

A new lexing state YYPRAGMA is added to the lexer to parse pragma.

```
<YYINITIAL> {
"{-#" { yybegin(YYPRAGMA); }
"#-}" {
throw new SyntaxError("Pragma mismatch, extra #-} found", yyline, yycolumn);
}
...
}
<YYPRAGMA> {
"{-#" { throw new SyntaxError("Nested pragma is not allowed", yyline, yycolumn); }
"#-}" { yybegin(YYINITIAL); }
```

```
    "="   { return token(EQ); }

    {Alphanumeric} { return token(STRING, yytext()); }
    {Whitespace}   { /* skip */ }

    [^] { throw new SyntaxError("Illegal character " + yytext(), yyline, yycolumn); }
    }
```

And a new parsing mode is added to the parser.

```
    non terminal EntryPoint p;
    non terminal Expr e;

    start with p;

    p ::= STRING:x p:e {: RESULT = new Pragma(x, null, e); :}
    | STRING:x EQ STRING:v p:e {: RESULT = new Pragma(x, v, e); :}
    | e:e {: RESULT = e; :}
    ;

    e ::= ...
```

All supported pragmas are:

Strict Enable global strict evaluation.

MaxStack Set the maximum stack size.

MaxHeap Set the maximum allowed heap objects.

GCThreshold Set the garbage collection threshold.

7.2 Lazy evaluation

According to Haskell specification, lazy evaluation is *call-by-name*[2] plus *sharing*[3], i.e. *call-by-need*. The main idea of lazy evaluation is to only evaluate an operand when needed.

Lazy evaluation is implemented using `ThunkValue`, a closure with unit argument. When let-binding an expression, or applying one to a function (during β-reduction), it doesn't need to be evaluated immediately, and is *delayed*, which means capturing the current environment and storing both in a thunk. When the value is really needed (queried from the environment, or being determined its semantics equivalence with another value), it is *forced*, which means evaluating the expression with the captured environment.

```
public record Let(Symbol x, Expr e1, Expr e2, Boolean strict) implements Expr {
    …
    @Override
    public Value eval(@NotNull State s) throws RuntimeError {
        Value v1;
        if (strict || s.config.strict()) {
            v1 = e1.eval(s);
        } else {
            v1 = ThunkValue.delay(s.E, e1);
        }

        return e2.eval(State.of(s.E.extend(x, v1), s.M, s.config));
    }
}
```

[2]Call by name is an evaluation strategy where the arguments to a function are not evaluated before the function is called.

[3]Sharing means that temporary data is physically stored, if it is used multiple times.

```
public record Name(Symbol x) implements Expr {
    …
    @Override
    public Value eval(@NotNull State s) throws RuntimeError {
        var v = s.E.get(x);
        …
        if (v instanceof ThunkValue thunk) {
            v = thunk.force(s);
        }
        …
    }
}
```

To avoid evaluating an expression multiple times, sharing is implemented by caching the result of evaluation in `ThunkValue`.

```
public final class ThunkValue implements Value {
    public final Env E;
    public final Expr e;
    public Value cachedValue = null;

    @Contract("_, _ -> new")
    public static @NotNull ThunkValue delay(Env E, Expr e) {
        return new ThunkValue(E, e);
    }

    public Value force(State s) throws RuntimeError {
        if (cachedValue == null) {
            cachedValue = e.eval(State.of(E, s.M, s.config));
        }
        return cachedValue;
    }

    @Override
    public boolean equals(State s, Value o) throws RuntimeError {
        var v = force(s);
        return v.equals(s, o);
    }
}
```

Beware that call-by-need evaluation has a different semantics from call-by-value evaluation if the language has side effects, which unfortunately is the case in *SimPL* (because of reference). The following machenism allows a user to opt out of lazy evaluation temporarily.

Function application Writing `f @x` forces the evaluation of `x` before calling `f`.

Let-binding Writing `let @x = a in e` forces the evaluation of `a` before binding `x` to `a` in `e`.

And if strict semantics is expected globally, `{-# Strict -#}` pragma can be used to disable lazy evaluation completely in this program.

The following example illustrates the difference of semantics between lazy and strict evaluation.

```
(let a = ref 0 in
    let b = a := !a + 1; a in   (* lazy *)
        !a
    end
end,
let a = ref 0 in
    let @b = a := !a + 1; a in  (* strict *)
        !a
    end
end)
(* ==> pair@0@1 *)
```

Notice that there are side effects in both let bindings. The first let-binding is lazy, so the side effect is not executed.

Another cons of lazy evaluation is that it deepens the stack when interpreting a program. This might cause stack overflow.

To let tests pass, default stack size is increased to 10240000.

7.3 Infinite Stream

An infinite stream is an infinite sequence of values, or basically a lazy list. Consider a stream of element type τ. It has one constructor: `SCons` of type $\tau \to (\tau\ stream) \to (\tau\ stream)$.

Unfortunately, it's an infinite type, and evaluating it by call-by-value strategy will not terminate. To solve this problem, `ThunkValue` is used to defer the evaluation of the second argument of `SCons`. After wrapping it in a `ThunkValue`, it's equivalent to $\tau \to (() \to \tau\ stream) \to \tau\ stream$ with call-by-need.

7.3.1 Syntax

Just like `hd::tl` for list, `hd;;tl` is the stream constructor. Lexical and syntactic definitions are updated accordingly.

```
";;" { return token(SCONS); }

terminal CONS, SCONS, UNIT, ARROW;
precedence right CONS, SCONS;

e ::= ...
| e:l SCONS e:r {: RESULT = new SCons(l, r); :}
```

7.3.2 Typing

$$\frac{\Gamma \vdash e_1 : \tau \quad \Gamma \vdash e_2 : \tau\ stream}{\Gamma \vdash e_1;;e_2 : stream\ \tau} T_{Stream}$$

7.3.3 Semantics

$$\frac{E, M; e_1 \Downarrow M'; v_1}{E, M; (e_1;;e_2) \Downarrow M'; (stream\ v_1\ (thunk\ e_2))} E_{SCons}$$

7.3.4 Predefined functions

Two predefined functions are introduced to eliminate a stream.

$$\begin{aligned} \mathbf{shd} &: \tau\ stream- > \tau \\ \mathbf{stl} &: \tau\ stream- > \tau\ stream \\ \mathbf{shd}\ (stream\ v_1\ (thunk\ e_2)) &= v_1 \\ \mathbf{stl}\ (stream\ v_1\ (thunk\ e_2)) &= v_2 \quad (e_2 \text{ evaluates to } v_2) \end{aligned}$$

7.3.5 Example

The following program generates a stream of '1's, take first 10 of them and add them up.

```
let stream_sum = rec f => fn n => fn s => if n = 0 then 0 else ((shd s) + (f (n - 1) (stl s))) in
    let ones = rec f => (1 ;; f) in
        stream_sum 10 ones
    end
end
(* => 10 *)
```

7.4 Garbage Collection

SimPL has heap-allocated *reference* values. In this implementation, all values on heap are contained in a `Cell`. A cell is considered *reachable* if it's a `RefValue`, or it's in the *sharing cache* of a forced `ThunkValue` in the environment.

A stop-the-world garbage collection machenism is implemented in this interpreter. It retrieves all cells that are not reachable from the environment of current state. This is done by *copy collection*. When allocating a new `Cell`, current heap size is checked. If heap usage is over the threshold, reachable `Cells` are copied to a new heap, and the old heap is freed.

For better performance, all `Cells` share the same storage, which is a `MutableArray`, i.e. `vector` in C++. A `Cell` contains a reference to the global storage, and an index to the specific value. When gc happens, both the storage reference and the index may be changed, so they are wrapped in `MutableCells`.

Special care is taken to ensure that garbage collection works for `ThunkValues`. When evaluating a `ThunkValue`, the captured environment is active, and the original one is not. However, we should not collect any `Cells` that are in the original environment because they might be used later. To achieve this, a stack of root environments is maintained. The environment is pushed into `root` when it is entered, and pop when exit.

```java
public class Mem {
    private final MutableCell<MutableList<Value>> storage = MutableCell.of(MutableList.create());
    private final MutableStack<Env> roots = MutableStack.create();

    public static class Cell {
        protected final MutableCell<MutableList<Value>> storage;
        protected final MutableCell<Integer> pointer;
        …
    }
    …
    /**
     * Perform copy garbage collection.
     */
    public void gc() {
        System.err.println("[GC] Starting GC. Before: " + storage.get().size() + " objects.");

        // Scan for cells that are reachable from the root.
        var reachable = roots.view()
                .flatMap(E -> E.valuesView().flatMap(CheckedFunction.of(Value::collectRefValues)))
                .collect(ImmutableCompactSet.factory())
                .toImmutableSeq();
        // Copy reachable cells to a new heap.
        var newStorage = MutableList.from(reachable.view().map(Cell::get));
        // Update the storage and index of all cells.
        reachable.view().forEachIndexed((i, v) -> {
            v.storage.set(newStorage);
            v.pointer.set(i);
        });

        System.err.println("[GC] After: " + newStorage.size() + " objects.");
        storage.set(newStorage);
    }
}
```

Two pragmas are added to configure heap and gc behavior. `MaxHeap` decides the upper limit of heap cells, and `GCThreshold` is the lowest heap usage rate that may trigger a gc. The default GC threshold is 70%.

The following program is an example of how gc works in action.

```
{-# HeapSize=4 #-}
let a = ref 0 in
    let b = ref 1 in
        a := !a + !b;
        let c = ref 2 in
            a := !a + !c;
            let d = ref 6 in
                a := !a + !d
            end
        end
    end;
    let e = ref 3 in
        a := !a + !e
    end;
    let f = ref 4 in
        a := !a + !f
    end;
    let g = ref 5 in
        a := !a + !g
    end;
    !a
end
(* => 21 *)
```

Heap debug log is as follows:

```
Allocating ref 0 = 0 got cell 0
Allocating ref 1 = 1 got cell 1
Allocating ref 2 = 2 got cell 2
[Alloc] Heap used percentage: 0.75 >= 0.70, triggering GC
[GC] Starting GC. Before: 3 objects.
[GC] After: 3 objects.
Allocating ref 6 = 6 got cell 3
[Alloc] Heap used percentage: 1.00 >= 0.70, triggering GC
[GC] Starting GC. Before: 4 objects.
[GC] After: 1 objects.
Allocating ref 3 = 3 got cell 1
Allocating ref 4 = 4 got cell 2
[Alloc] Heap used percentage: 0.75 >= 0.70, triggering GC
[GC] Starting GC. Before: 3 objects.
[GC] After: 1 objects.
Allocating ref 5 = 5 got cell 1
```

Decreasing `MaxHeap` to 3 causes an error:

```
Allocating ref 0 = 0 got cell 0
Allocating ref 1 = 1 got cell 1
Allocating ref 2 = 2 got cell 2
[Alloc] Heap used percentage: 1.00 >= 0.70, triggering GC
[GC] Starting GC. Before: 3 objects.
[GC] After: 3 objects.
Heap overflow
```

A Appendix: Expected outputs

All test programs are evaluated with default settings. Besides, this is also the expected output for the following configurations, which is enforced by unit tests:

1. `GCThreshold` set to 0 (always gc).
2. strict mode enabled. (except for `effect.simpl`, which relies on lazy evaluation)

```
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
((tv71 -> tv72) -> (tv71 list -> tv72 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/pcf.twice.spl
int
16
doc/examples/letpoly.spl
int
0
doc/examples/effect.spl
(int * int)
pair@0@1
doc/examples/stream.spl
int
10
```