*Acharya, Sneh*

*SimPL Interpreter: Implementation Report*

**Table of Content:**

[**SimPL Interpreter: Implementation Report** 2](#_Toc182486278)

[**Introduction** 2](#_Toc182486279)

[**Project Structure and Execution** 2](#_Toc182486280)

[**2. Lexical Definition, Syntax, and Typing Rules** 3](#_Toc182486281)

[**3. Semantic Structure: Environment, Memory, and Pointer** 3](#_Toc182486282)

[**4. Key Interpreter Components** 4](#_Toc182486283)

[**5. Values** 6](#_Toc182486284)

[**6. Expressions** 6](#_Toc182486285)

[**7. Types in SimPL** 7](#_Toc182486286)

[**8. Testing and Sample Programs** 8](#_Toc182486287)

[**Conclusion** 8](#_Toc182486288)

**SimPL Interpreter: Implementation Report**

**Author:** Sneh Acharya  
**Project URL:** [GitHub Repository](https://github.com/YourRepository/SimPL-Interpreter)

**Introduction**

This document details the design and functionality of the SimPL interpreter, a Java-based implementation for a simplified, ML-inspired programming language. SimPL supports functional and imperative programming constructs, including type checking, expression evaluation, and environment management. The interpreter uses a robust type system, including polymorphic typing, pairs, lists, and support for lambda calculus expressions.

**Project Structure and Execution**

**1.1 Source Code Structure**

The SimPL interpreter is structured into three main packages:

* **src/simpl/parser/**: Defines syntax and parsing rules, transforming code into an abstract syntax tree (AST).
* **src/simpl/interpreter/**: Manages runtime evaluation, including Value classes, the environment, and built-in functions.
* **src/simpl/typing/**: Implements SimPL’s type system, performing static type checking and inference.

**1.2 Running the Interpreter**

To run SimPL programs, compile the source code into a .jar file, such as SimPL.jar, and execute it with a program file. For example:

java -jar SimPL.jar ./factorial.spl

Expected output:

int

24

**2. Lexical Definition, Syntax, and Typing Rules**

**2.1 Lexical Definition and Syntax**

SimPL’s lexical rules define atom types, operators, and keywords:

* **Atoms**:
  + **Integer Literals**: Whole numbers in decimal form.
  + **Identifiers**: Alphanumeric names beginning with a letter.
* **Keywords**: if, then, else, let, in, end, fn, rec, ref, true, false, etc.
* **Operators**: Includes +, -, \*, /, :=, ::, and logical operators like andalso and orelse.

SimPL’s parser processes code to build an abstract syntax tree (AST), which the interpreter uses to evaluate expressions and enforce type safety.

**2.2 Typing Rules**

SimPL’s typing rules ensure that expressions are well-typed. Each type is bound by constraints to prevent errors and enforce type safety:

* **Arithmetic Types**: Operands of arithmetic expressions like Add and Sub must be integers.
* **Boolean Types**: Logical expressions like AndAlso require boolean operands.
* **Function Types**: Lambda expressions (Fn) require parameter and return types, specified by an ArrowType.
* **Polymorphic Types**: TypeVar enables polymorphic typing, useful in general expressions like pairs.
* **Equality Types**: Eq allows only types that support equality checks.

Each rule is enforced through SimPL’s type checker to ensure only valid expressions are evaluated.

**3. Semantic Structure: Environment, Memory, and Pointer**

SimPL’s interpreter relies on an environment and memory model to manage variables and expressions at runtime.

**3.1 Environment, Memory, and Pointer**

* **Environment (Env)**: Maps variable names (symbols) to values during evaluation, supporting scope-based variable resolution.
* **Memory (Mem)**: Emulates storage, enabling imperative operations such as assignments and references.
* **Pointer (p)**: Keeps track of available memory locations.

The interpreter evaluates expressions by updating these structures based on each expression's semantics.

**3.2 Semantic Rules for Expressions**

Different expressions have specific rules for type-checking and evaluation:

* **Assignments**: For Assign expressions, the interpreter evaluates the right-hand value and updates the memory at a specified location.
* **Lambda Applications**: The App expression applies functions to arguments by binding the argument to the function’s parameter in a new environment scope.
* **Conditionals**: If statements (Cond) evaluate boolean expressions and choose branches based on the condition.

**4. Key Interpreter Components**

**4.1 Library Function: fst**

The fst function returns the first element of a pair. It is implemented as a FunValue with a type-safe, anonymous Exprclass.

package simpl.interpreter.lib;

import simpl.interpreter.Env;

import simpl.interpreter.FunValue;

import simpl.interpreter.PairValue;

import simpl.interpreter.State;

import simpl.parser.Symbol;

import simpl.parser.ast.Expr;

import simpl.typing.TypeEnv;

import simpl.typing.TypeResult;

import simpl.typing.TypeVar;

public class fst extends FunValue {

public fst() {

super(Env.empty, Symbol.symbol("fst"), getExpr());

}

private static Expr getExpr() {

return new Expr() {

@Override

public TypeResult typecheck(TypeEnv E) {

return TypeResult.of(new TypeVar(true));

}

@Override

public Value eval(State s) {

PairValue v = (PairValue) s.E.get(Symbol.symbol("fst"));

return v.v1;

}

};

}

}

* **Type Checking**: Returns a type variable (TypeVar) to support various input types.
* **Evaluation**: Retrieves the first element (v1) of a PairValue.

**4.2 Primitive Computable Function: iszero**

The iszero function checks if an integer is zero and returns a boolean result.

package simpl.interpreter.pcf;

import simpl.interpreter.Env;

import simpl.interpreter.FunValue;

import simpl.interpreter.IntValue;

import simpl.interpreter.State;

import simpl.parser.Symbol;

import simpl.parser.ast.Expr;

import simpl.typing.IntType;

import simpl.typing.TypeEnv;

import simpl.typing.TypeResult;

public class iszero extends FunValue {

public iszero() {

super(Env.empty, Symbol.symbol("iszero"), getExpr());

}

private static Expr getExpr() {

return new Expr() {

@Override

public TypeResult typecheck(TypeEnv E) {

return TypeResult.of(new IntType());

}

@Override

public Value eval(State s) {

IntValue v = (IntValue) s.E.get(Symbol.symbol("iszero"));

return v.n == 0 ? new BoolValue(true) : new BoolValue(false);

}

};

}

}

* **Type Checking**: Confirms that the function operates on integers by returning an IntType.
* **Evaluation**: Returns a BoolValue depending on whether the integer value is zero.

**5. Values**

Values represent the evaluated results of expressions.

**5.1 IntValue**

Represents integers in SimPL.

public class IntValue extends Value {

public final int n;

public IntValue(int n) { this.n = n; }

}

**5.2 PairValue**

Stores pairs of values, supporting functions like fst.

public class PairValue extends Value {

public final Value v1, v2;

public PairValue(Value v1, Value v2) { this.v1 = v1; this.v2 = v2; }

}

**5.3 UnitValue**

Represents unit types for expressions with no meaningful return.

public class UnitValue extends Value {

public static final UnitValue INSTANCE = new UnitValue();

}

**6. Expressions**

Expressions define computations, including arithmetic, lambda calculus, and variable binding.

**6.1 Fn (Lambda Expression)**

Defines lambda functions.

public class Fn extends Expr {

public Symbol x;

public Expr e;

}

**6.2 App (Application Expression)**

Represents function application.

public class App extends Expr {

public Expr e1, e2;

}

**6.3 Add (Arithmetic Expression)**

Implements addition for integers.

public class Add extends ArithExpr {

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

}

**6.4 Let (Let Binding Expression)**

Defines variable binding within an expression’s scope.

public class Let extends Expr {

public Symbol x;

public Expr e1, e2;

}

**7. Types in SimPL**

SimPL’s type system includes integer, boolean, function, and polymorphic types.

**7.1 IntType**

Defines the integer type.

public class IntType extends Type { }

**7.2 BoolType**

Defines the boolean type.

public class BoolType extends Type { }

**7.3 ArrowType (Function Type)**

Represents function types like a -> b.

public class ArrowType extends Type {

public final Type from, to;

}

**7.4 TypeVar (Type Variable)**

Supports polymorphic type inference.

public class TypeVar extends Type {

private int id;

}

**8. Testing and Sample Programs**

**Factorial Function**

A recursive function demonstrating SimPL’s type safety and recursion handling:

let fact = rec f => fn x => if x=1 then 1 else x \* (f (x-1)) in fact 4 end

**Conditional Expressions**

Tests conditional functionality:

if iszero(0) then 1 else 2

Each example demonstrates specific features of SimPL, such as recursion, conditional logic, and lambda calculus.

**Conclusion**

This report details the SimPL interpreter’s implementation, covering every major aspect, from lexical and syntax rules to type inference and evaluation. Each component demonstrates how Java’s object-oriented features can build a reliable, type-safe interpreter for a functional language.