L1++ Language: Big Step Evaluation Rules for Lists and Lazy Lists

Programming Languages 2024/25

May 2025

1 Introduction

This report describes the big step evaluation rules for the list and lazy list primitives in the L1++ language. L1++ is an extension of the L1 functional-imperative language studied in the course, with additional support for immutable lists and lazy lists.

2 Values in L1++

In addition to the base values of L1 (integers, booleans, closures, references), L1++ introduces two new value types:

$$\textit{value nil} \tag{1}$$

$$\textit{value } V \quad \textit{value } L \quad \textit{value cons}(V, L) \tag{2}$$

For lazy lists, a new value type is introduced:

$$\textit{value } \mathcal{E} \quad \textit{AST } H \quad \textit{AST } T \quad \textit{value lcons}(\mathcal{E}, H, T) \tag{3}$$

Where $\mathcal{E}$ is the environment, and H and T are unevaluated AST nodes for the head and tail expressions.

3 Big Step Semantics for Regular Lists

3.1 List Constructors

3.1.1 nil Constructor

The nil constructor creates an empty list:

$$\frac{}{\mathcal{E}; \mathcal{S}; \textit{nil} \Downarrow \textit{nil}; \mathcal{S}} \tag{4}$$

3.1.2 cons Constructor

The cons constructor evaluates both the head and tail expressions before creating a new list node:

$$\frac{\mathcal{E}; \mathcal{S}; M \Downarrow V; \mathcal{S}' \quad \mathcal{E}; \mathcal{S}'; N \Downarrow L; \mathcal{S}'' \quad U = V :: L}{\mathcal{E}; \mathcal{S}; \mathit{cons}(M, N) \Downarrow U; \mathcal{S}''} \tag{5}$$

3.2 List Pattern Matching

Pattern matching on lists has two cases - matching against an empty list and matching against a cons cell:

3.2.1 Match with nil

$$\frac{\mathcal{E}; \mathcal{S}; M \Downarrow \mathit{nil}; \mathcal{S}' \quad \mathcal{E}; \mathcal{S}'; N \Downarrow U; \mathcal{S}''}{\mathcal{E}; \mathcal{S}; \mathit{match}\ M\{\mathit{nil} \to N \mid \mathit{cons}(x, l) \to R\} \Downarrow U; \mathcal{S}''} \tag{6}$$

3.2.2 Match with cons

$$\frac{\mathcal{E}; \mathcal{S}; M \Downarrow \mathit{cons}(V, L); \mathcal{S}' \quad \mathcal{E}[x \mapsto V][l \mapsto L]; \mathcal{S}'; R \Downarrow U; \mathcal{S}''}{\mathcal{E}; \mathcal{S}; \mathit{match}\ M\{\mathit{nil} \to N \mid \mathit{cons}(x, l) \to R\} \Downarrow U; \mathcal{S}''} \tag{7}$$

4 Big Step Semantics for Lazy Lists

4.1 Lazy List Constructors

4.1.1 lcons Constructor

The lcons constructor does not evaluate its arguments immediately, but instead creates a suspended computation:

$$\frac{}{\mathcal{E}; \mathcal{S}; \mathit{lcons}(M, N) \Downarrow \mathit{lcons}(\mathcal{E}, M, N); \mathcal{S}} \tag{8}$$

Where $\mathcal{E}$ is the current environment, which is captured along with the unevaluated expressions M and N.

4.2 Lazy List Pattern Matching

Pattern matching against a lazy list forces the evaluation of the head and tail expressions:

4.2.1 Match with lcons

$$\frac{\begin{array}{l}\mathcal{E}; \mathcal{S}; M \Downarrow \mathit{lcons}(\mathcal{E}', H, T); \mathcal{S}' \\ \mathcal{E}'; \mathcal{S}'; H \Downarrow V; \mathcal{S}'' \\ \mathcal{E}'; \mathcal{S}''; T \Downarrow L; \mathcal{S}''' \\ \mathcal{E}[x \mapsto V][l \mapsto L]; \mathcal{S}'''; R \Downarrow U; \mathcal{S}''''\end{array}}{\mathcal{E}; \mathcal{S}; \mathit{match}\ M\{\mathit{nil} \to N \mid \mathit{cons}(x, l) \to R\} \Downarrow U; \mathcal{S}''''} \tag{9}$$

This rule shows that when a lazy list is matched, the head and tail expressions are evaluated in the captured environment $\mathcal{E}'$, and then the results are bound to the variables x and l in the current environment $\mathcal{E}$.

5 Implementation

5.1 Regular Lists

The regular list implementation consists of the following key components:

5.1.1 VList Class

The VList class implements the IValue interface and represents both nil and cons list values:

```java
public class VList implements IValue {
    private final IValue head;
    private final IValue tail;
    private final boolean isNil;

    // Constructor for nil
    public VList() {
        this.head = null;
        this.tail = null;
        this.isNil = true;
    }

    // Constructor for cons
    public VList(IValue head, IValue tail) {
        this.head = head;
        this.tail = tail;
        this.isNil = false;
    }

    // Methods to access head and tail
    public final boolean isNil() {
        return this.isNil;
    }

    public final IValue getHead() {
        if (this.isNil) throw new RuntimeException("Cannot get head
     of nil list");
        return this.head;
    }

    public final IValue getTail() {
        if (this.isNil) throw new RuntimeException("Cannot get tail
     of nil list");
        return this.tail;
    }

    // String representation
    @Override
    public final String toStr() { /* ... */ }
}
```

5.1.2 ASTNil Class

The ASTNil class implements the ASTNode interface and evaluates to a nil value:

```java
public class ASTNil implements ASTNode {
    @Override
    public IValue eval(Environment<IValue> e) throws
    InterpreterError {
        return new VList();
    }
}
```

5.1.3 ASTCons Class

The ASTCons class implements the ASTNode interface and evaluates both the head and tail expressions before creating a new list node:

```java
public class ASTCons implements ASTNode {
    private final ASTNode head;
    private final ASTNode tail;

    public ASTCons(ASTNode head, ASTNode tail) {
        this.head = head;
        this.tail = tail;
    }

    @Override
    public IValue eval(Environment<IValue> e) throws
    InterpreterError {
        final IValue headValue = head.eval(e);
        final IValue tailValue = tail.eval(e);

        if (!(tailValue instanceof VList)) {
            throw new InterpreterError("Cons tail must be a list");
        }

        return new VList(headValue, tailValue);
    }
}
```

5.1.4 ASTMatch Class

The ASTMatch class implements pattern matching on lists:

```java
public class ASTMatch implements ASTNode {
    private final ASTNode expr;
    private final ASTNode nilCase;
    private final String headVar;
    private final String tailVar;
    private final ASTNode consCase;

    // Constructor...

    @Override
    public IValue eval(Environment<IValue> e) throws InterpreterError {
        final IValue value = this.expr.eval(e);

        // Case 1: Regular list
        switch (value) {
            case VList vList -> {
                if (vList.isNil()) {
                    return nilCase.eval(e);
                } else {
                    final Environment<IValue> newEnv = e.beginScope();
                    newEnv.assoc(headVar, vList.getHead());
                    newEnv.assoc(tailVar, vList.getTail());
                    return consCase.eval(newEnv);
                }
            }
            // Lazy list case...
            default -> throw new InterpreterError("Match expression must evaluate to a list or lazy list");
        }
    }
}
```

5.2 Lazy Lists

The lazy list implementation adds the following components:

5.2.1 VLazyList Class

The VLazyList class implements the IValue interface and represents a list with delayed evaluation:

```java
public class VLazyList implements IValue {
    private final Environment<IValue> env;
    private final ASTNode headExpr;
    private final ASTNode tailExpr;
    private boolean evaluated;
    private IValue head;
    private IValue tail;

    public VLazyList(Environment<IValue> env, ASTNode headExpr,
    ASTNode tailExpr) {
        this.env = env;
        this.headExpr = headExpr;
        this.tailExpr = tailExpr;
        this.evaluated = false;
    }

    public void evaluate() throws InterpreterError {
        if (!this.evaluated) {
            // Evaluate head and tail expressions in the captured
    environment
            this.head = headExpr.eval(env);
            this.tail = tailExpr.eval(env);
            this.evaluated = true;
        }
    }

    // Methods to access head and tail (with evaluation)
    public boolean isEvaluated() {
        return this.evaluated;
    }

    public boolean isNil() throws InterpreterError {
        evaluate();
        if (this.tail instanceof VList vList) {
            return vList.isNil() && this.head == null;
        }
        return false;
    }

    public IValue getHead() throws InterpreterError {
        evaluate();
        return this.head;
    }

    public IValue getTail() throws InterpreterError {
        evaluate();
        return this.tail;
    }

    // String representation
    @Override
    public String toStr() { /* ... */ }
}
```

5.2.2 ASTLCons Class

The ASTLCons class implements the ASTNode interface and creates a lazy list node without evaluating its arguments:

```java
public class ASTLCons implements ASTNode {
    private final ASTNode head;
    private final ASTNode tail;

    public ASTLCons(ASTNode head, ASTNode tail) {
        this.head = head;
        this.tail = tail;
    }

    @Override
    public IValue eval(Environment<IValue> e) throws
    InterpreterError {
        // For lazy cons, we don't evaluate the expressions yet
        // Instead, we store the unevaluated expressions and the
    environment
        // We make a copy of the environment to capture the current
     bindings
        Environment<IValue> capturedEnv = e.copy();
        return new VLazyList(capturedEnv, head, tail);
    }
}
```

5.2.3 Lazy List Pattern Matching in ASTMatch

The ASTMatch class is extended to handle lazy lists:

```
// Inside the eval method of ASTMatch
case VLazyList lazyList -> {
    // Force evaluation of the lazy list when matched
    try {
        // Get head and tail - this will force evaluation if needed
        IValue head = lazyList.getHead();
        IValue tail = lazyList.getTail();

        // Create new environment with bindings for head and tail
        final Environment<IValue> newEnv = e.beginScope();
        newEnv.assoc(headVar, head);
        newEnv.assoc(tailVar, tail);

        return consCase.eval(newEnv);
    } catch (InterpreterError ex) {
        // If there's an error evaluating the lazy list, it might
    be nil
        try {
            if (lazyList.isNil()) {
                return nilCase.eval(e);
            }
        } catch (InterpreterError inner) {
            // Re-throw the original error
            throw ex;
        }
        throw ex;
    }
}
```

6 Examples

6.1 Regular Lists Example

The following example demonstrates the use of regular lists by creating a list of integers and computing their sum:

```
let emptyList = nil;
let singletonList = 1 :: nil;
let myList = 1 :: 2 :: 3 :: nil;

let sum = fn list => {
  match list {
    nil -> 0
    | head :: tail -> head + sum(tail)
  }
};

sum(myList);
```

This evaluates to 6, demonstrating that the list operations are working correctly.

6.2 Lazy Lists Example

The following example demonstrates the use of lazy lists by creating an infinite stream of natural numbers and taking the first 5 elements:

```
let naturals = fn from => {
  lcons(from, naturals(from + 1))
};

let take = fn n, lazyList => {
  if (n <= 0) {
    nil
  } else {
    match lazyList {
      nil -> nil
      | head :: tail -> head :: take(n - 1, tail)
    }
  }
};

let natStream = naturals(1);
let firstFive = take(5, natStream);

let sum = fn list => {
  match list {
    nil -> 0
    | head :: tail -> head + sum(tail)
  }
};

sum(firstFive);
```

This evaluates to 15 $(1 + 2 + 3 + 4 + 5)$, demonstrating that lazy evaluation is working correctly.

7 Conclusion

The L1++ language successfully extends the L1 language with support for both regular and lazy lists. The implementation follows the big step evaluation semantics defined for these primitives.

Regular lists are evaluated eagerly, with both the head and tail expressions evaluated when a cons node is created. Lazy lists, on the other hand, capture the environment and store the unevaluated expressions for the head and tail, only evaluating them when needed (typically during pattern matching).

This approach allows for powerful programming patterns, including the creation of infinite data structures like the stream of natural numbers shown in the example.