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

Programming Languages 2024/25

May 2025

1 Introduction

This report describes the big step evaluation rules for the 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 lazy lists.

2 Values in L1++

In addition to the base values of L1 (integers, booleans, closures, references), L1++ introduces support for lazy lists through a new value type:

$$\mathit{value}\ \mathcal{E} \quad \mathit{AST}\ H \quad \mathit{AST}\ T \quad \mathit{value}\ \mathit{lcons}(\mathcal{E}, H, T) \tag{1}$$

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

Note that for pattern matching and other operations, L1++ still uses regular list values (nil and cons). Lazy lists are converted to regular lists when they are evaluated (forced).

3 Big Step Semantics for Lazy Lists

3.1 Lazy List Constructors

3.1.1 lcons Constructor

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

$$\frac{}{\mathcal{E}; \mathcal{S}; \mathit{lcons}(M, N) \Downarrow \mathit{lcons}(\mathcal{E}, M, N); \mathcal{S}} \tag{2}$$

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

3.2 Lazy List Pattern Matching

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

3.2.1 Match with lcons

$$\frac{\begin{array}{c}\mathcal{E};\mathcal{S};M \Downarrow \mathit{lcons}(\mathcal{E}',H,T);\mathcal{S}' \\ \mathcal{E}';\mathcal{S}';H \Downarrow V;\mathcal{S}'' \\ \mathcal{E}';\mathcal{S}'';T \Downarrow L;\mathcal{S}''' \\ \mathcal{E}[x \mapsto V][l \mapsto L];\mathcal{S}''';R \Downarrow U;\mathcal{S}''''\end{array}}{\mathcal{E};\mathcal{S};\mathit{match}\ M\{\mathit{nil} \to N \mid \mathit{cons}(x,l) \to R\} \Downarrow U;\mathcal{S}''''} \tag{3}$$

This rule shows a critical aspect of lazy evaluation: when a lazy list is used in a pattern match against regular list patterns, it forces evaluation. Specifically:

1. The expression M evaluates to a lazy list value $\mathit{lcons}(\mathcal{E}', H, T)$ 2. The head expression H is evaluated in the captured environment $\mathcal{E}'$ (not the current environment) 3. The tail expression T is also evaluated in the captured environment $\mathcal{E}'$ 4. The results V and L are bound to variables x and l in the current environment $\mathcal{E}$ 5. The cons branch R of the match expression is evaluated with these bindings

This demonstrates the essential mechanism of lazy evaluation: expressions are only evaluated when their values are actually needed, and evaluation happens in the environment that was captured when the lazy list was created.

4 Implementation

4.1 Lazy Lists

The lazy list implementation consists of the following components:

4.1.1 VLazyList Class

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

```
1  public class VLazyList implements IValue {
2      private final Environment<IValue> env;
3      private final ASTNode headExpr;
4      private final ASTNode tailExpr;
5      private boolean evaluated;
6      private IValue head;
7      private IValue tail;
8
9      public VLazyList(Environment<IValue> env, ASTNode headExpr,
       ASTNode tailExpr) {
10         this.env = env;
11         this.headExpr = headExpr;
12         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() {
        try {
            evaluate();
            return "<lazy " + this.head.toStr() + " :: " + this.
    tail.toStr() + ">";
        } catch (InterpreterError e) {
            return "<unevaluated lazy list>";
        }
    }
}
```

4.1.2 ASTLCons Class

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

```
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);
    }
}
```

4.1.3 Lazy List Pattern Matching in ASTMatch

The ASTMatch class handles lazy lists by forcing their evaluation during pattern matching:

```
// 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;
    }
}
```

5 Example

5.1 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.

6 Conclusion

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

Lazy lists capture the environment and store 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.

A key feature of lazy lists is that they can represent potentially infinite sequences, since elements are only computed when they are actually needed. This enables a more declarative programming style where infinite structures can be defined recursively, and only the needed portions are evaluated.