Interpreter for LTLC

CAS CS 392: Rust, in Theory and in Practice

March 25, 2025 (Lecture 14)

Outline

Recap

Workshop

Structural Rules

$$\frac{\Gamma \vdash M : A}{\Gamma, x : B \vdash M : A}\ x \notin \Gamma \qquad \text{(Weakening)}$$

$$\frac{\Gamma, x : B, y : B \vdash M : A}{\Gamma, x : B \vdash M[x/y] : A} \qquad \text{(Contraction)}$$

$$\frac{\Gamma, x : B, y : C, \Delta \vdash M : A}{\Gamma, y : C, y : B, \Delta \vdash M : A} \qquad \text{(Exchange)}$$

Substructural Type Systems

Once we write are system to have structural rules, we have a degree of freedom to define *new systems*

Substructural logics/type systems disallow certain structural rules

System	Weakening	Contraction	Variable use
Unrestricted	yes	yes	any number of times
Affine	yes	no	at most once
Relevant	no	yes	at least once
Linear	no	no	exactly once

Linearity in Rust

We cannot use a variable more than once (without references):

```
// This does not compile
fn dup<T>(t: T) -> (T, T) {
  (t, t)
}
```

Rust without references is *linear*

Linear Typed λ-Calculus (LTLC)

Syntax:

$$\begin{aligned} V_{Ty} &::= a \mid b \mid c \ldots \\ Ty &::= V_{Ty} \mid \bot \mid Ty \multimap Ty \\ V_T &::= x \mid y \mid z \ldots \\ T &::= V_T \mid \lambda V.T \mid TT \end{aligned}$$

Type system:

$$\frac{}{x : A \vdash x : A} \qquad \frac{\Gamma \vdash M : A}{\pi(\Gamma) \vdash M : A}$$

$$\frac{\Gamma \vdash M : A \multimap B \qquad \Delta \vdash N : A}{\Gamma, \Delta \vdash MN : B} \qquad \frac{\Gamma, x : A \vdash M : B}{\Gamma \vdash \lambda x.M : A \multimap B}$$

The Key Lemma

Lemma. If $\Gamma \vdash M : A$ then the variable x appears free in M *exactly once* if and only if x appears in Γ

This means no cloning in a substitution-based model

Outline

Recap

Workshop

Goals

- Make sure we understand the structure of an interpreter
- Take a look at some lexer/recursive descent parser code
- Implement type checking and evaluation for *both* STLC and LTLC

Review: Interpretation Pipeline

Interpretation is done in 4 stages:

1. **Lexical Analysis:** Group characters in the input `char` stream into units called **tokens**, eliminating whitespace and comments
2. **Syntactic Analysis:** Convert our stream of tokens into an **abstract syntax tree (AST)**, giving the program its heirarchical structure
3. **Static Analysis:** Make sure the AST is well-formed, doing any scope/type/borrow-checking, potentially building an intermediate representation
4. **Evaluation:** Determine the value associated with the AST based on the given semantics

Review: Lexing

```rust
#[derive(Debug, Clone)]
pub enum Token {
    Lparen,
    Rparen,
    Lambda,
    FunTy,
    EmptyTy,
    Var(String),
}
```

A **lexer** is, in essence, a *peekable* iterator where `next()` gives you the next token represented in the character stream

More complex lexers need to deal with backtracking, we're not going to worry about that

Review: Recursive Descent Parsing

Recursive decent parsing is an ad hoc parsing method which "mini-parsers" which mirror the our AST

```
pub enum Expr {
    Var(Ident),
    App(Box<Expr>, Box<Expr>),
    Lam(Ident, Type, Box<Expr>),
}
fn parse_expr(&mut self) -> Option<Expr> {
    match self.lexer.next()? {
        Token::Var(s) => {
            Some(Expr::Var(s))
        }
        Token::Lparen => {
            match self.lexer.peek_keyword() {
                Some(Token::Lambda) => {
                    // ...
```

Tasks

- Form a group of 2-3
- Download the starter code from the course webpage
- Implement the method `Expr::ty` in the file `type.rs`
- Implement the method `Expr::eval` in the file `eval.rs`
- Look through `lexer.rs` and `parser.rs` for "inspiration" for the final project