Featherweight Rust: An Introduction

CAS CS 392: Rust, in Theory and in Practice

March 27, 2025 (Lecture 14)

Outline

The Key Lemma

Lemma. If $\Gamma \vdash M : A$ (i.e., M is linearly well-typed), then every variable in Γ appears in M exactly once *and every bound variable appears exactly once in the body of any λ-term.*

Let's prove this

① $$\frac{}{x:A \vdash x:A}$$

② $$\frac{\Gamma, x:A, y:B, \Delta \vdash M:C}{\Gamma, y:B, x:A, \Delta \vdash M:C}$$

③ $$\frac{\Gamma, x:A \vdash M:B}{\Gamma \vdash \lambda x^A.M : A \to B}$$

④ $$\frac{\Gamma \vdash M:A \to B \qquad \Delta \vdash N:A}{\Gamma, \Delta \vdash M\,N : B}$$

Lemma. If $\Gamma \vdash M:A$ then
(i) $x \in \Gamma \Rightarrow$ x appears in M exactly once
(ii) $M = (\ldots(\lambda x.N)\ldots) \Rightarrow$ x appears exactly once in N

Proof. By ind. on deriv. $\mathcal{D}$ of $\Gamma \vdash M : A$

$$\frac{\vdots\ \mathcal{D}}{\Gamma \vdash M : A}$$

Rule ①

Base Case:

$$\frac{}{x : A \vdash x : A}$$

$\Gamma \quad M \quad A$

(i) and (ii) hold trivially

Rule ②:

$$\frac{\vdots\ \mathcal{D}'}{\Gamma, x : B, y : C, \Delta \vdash M : A}$$

$$\Gamma, y : C, x : B, \Delta \vdash M : A$$

By IH on $\mathcal{D}'$ (i) and (ii) hold of the premise

Since contexts are the same (i) holds

since subject (M) is same (ii) holds

Rule ③:

$$\frac{\vdots\ \mathcal{D}'}{\Gamma, x:A \vdash M : B} \qquad \mathcal{D}$$

$$\Gamma \vdash \lambda x^A.M : A \multimap B$$

By IH on $\mathcal{D}'$, ($y \in \Gamma \Rightarrow y$ appears exactly once in M) and (x appears exactly once in M)*, and also (ii) hold of M. By (④) (ii) holds of $\lambda x^A.M$. By (△), (i) of of $\lambda x^A M$.

Rule (4).

$$\frac{\mathcal{D}_1 :: \Gamma \vdash M : A \multimap B \qquad \mathcal{D}_2 :: \Delta \vdash N : A}{\mathcal{D} :: \Gamma, \Delta \vdash MN : B}$$

By IH on $\mathcal{D}_1$, $\mathcal{D}_2$ (ii) holds of M and N so it holds of MN, and also $x \in \Gamma$ then x appears free in M exactly once. $y \in \Delta$ implies y appears exactly once in N so if $x \in \Gamma, \Delta$ then x appears exactly once in MN.

Exercise: fix wording to use bi-implication

Outline

The Paper

A Lightweight Formalism for Reference Lifetimes and Borrowing in Rust, David J. Pearce (2021)

The Paper

A Lightweight Formalism for Reference Lifetimes and Borrowing in Rust, David J. Pearce (2021)

"Our calculus core captures many aspects of Rust, including copy- and move-semantics, mutable borrowing, reborrowing, partial moves, and lifetimes. In particular, it remains sufficiently lightweight to be easily digested and understood and, we argue, still captures the salient aspects of reference lifetimes and borrowing." (from the abstract)

The Paper

A Lightweight Formalism for Reference Lifetimes and Borrowing in Rust, David J. Pearce (2021)

"Our calculus core captures many aspects of Rust, including copy- and move-semantics, mutable borrowing, reborrowing, partial moves, and lifetimes. In particular, it remains sufficiently lightweight to be easily digested and understood and, we argue, still captures the salient aspects of reference lifetimes and borrowing." (from the abstract)

Disclaimer: This is, in no way, my work. It's just a very cool paper that works well as the basis of a PL-centric course on Rust.

Timeline

- 2006: Graydon Hoare introduces of Rust
- 2015:
  - First stable release of Rust
  - **Patina**, a formalization of surface-level Rust (earlier version)
- 2018:
  - **RustBelt**, a formalization of the Rust MIR
  - **KRust**, a formalization of the *semantics* of Rust
- 2021:
  - **FR**, a lightweight formalization of surface-level Rust
  - **Oxide**, a middleweight formalization of surface-level Rust

The Plan

1. Target a subset of Rust, the smallest subset that allows us to test basic concepts of Rust:
   - (Mutable) Variables
   - Ownership
     - boxes
     - move vs. copy semantics
   - Borrowing
     - mutable vs. immutable borrowing
     - **reborrowing**
     - freezing
     - moving out
   - Reference Lifetimes
     - scoping blocks
     - borrow lifetimes
2. Write (and implement) a type/borrow system and semantics for this subset
3. *Prove* that this subset satisfies the kinds of safety properties we want (determine what those safety properties *are*)

Ownership

```
fn main() {
    let mut x = 2;
    let mut y = Box::new(x);
    let mut z = y;
    x = x + 1;
    *z = *z - 1;
}
```

We'll support boxes so that we can put values on the heap

Ownership

```
fn main() {
    let mut x = 2;
    let mut y = Box::new(x);
    let mut z = y;
    x = x + 1;
    *z = *z - 1;
}
```

We'll support boxes so that we can put values on the heap

We'll also support move vs. copy semantics (so boxes should be have differently on reassignment than integers)

Ownership

```
fn main() {
    let mut x = 2;
    let mut y = Box::new(x);
    let mut z = y;
    x = x + 1;
    *z = *z - 1;
}
```

We'll support boxes so that we can put values on the heap

We'll also support move vs. copy semantics (so boxes should be have differently on reassignment than integers)

We will *not* support implicit dereferences, so working with boxed values will always require the * operator

Borrowing

We'll support mutable references to variables, which must be unique:

```
fn main() {
    let mut x = 2;
    let mut z = &mut x;
    *z += 1;
}
```

Borrowing

We'll support mutable references to variables, which must be unique:

```
fn main() {
    let mut x = 2;
    let mut z = &mut x;
    *z += 1;
}
```

As well as immutable references, which are not necessarily unique:

```
fn main() {
    let mut x = 2;
    let mut y = &x;
    let mut z = &x;
    let mut q = *y;
    let mut r = *z;
}
```

Scoping Blocks

We'll support scoping blocks with their own lifetimes:

```
fn main() {
    let mut x = 0;
    {x = x + 1; let mut y = x}
    x = x + 1;
}
```

Scoping Blocks

We'll support scoping blocks with their own lifetimes:

```
fn main() {
    let mut x = 0;
    {x = x + 1; let mut y = x}
    x = x + 1;
}
```

This will allow us to test some of the trickier aspects of borrowing:

```
// THIS DOES NOT COMPILE
fn main () {
    let mut x = Box::new(0);
    {
        let mut y = x;
        // let mut y = &mut x; // IT DOES WITH THIS LINE
    }
    *x = 1;
}
```

Reborrowing

```
// THIS DOES NOT COMPILE
fn main() {
    let mut n = 0;
    let mut x = &mut n;
    {
        let mut y = &mut *x;
        let mut z = &mut y;
        x = &mut **z;
    }
    *x += 1;
    assert_eq!(n, 1);
}
```

We'll support reborrowing, the ability the temporarily borrow from a mutable borrow

Reborrowing

```
// THIS DOES NOT COMPILE
fn main() {
    let mut n = 0;
    let mut x = &mut n;
    {
        let mut y = &mut *x;
        let mut z = &mut y;
        x = &mut **z;
    }
    *x += 1;
    assert_eq!(n, 1);
}
```

We'll support reborrowing, the ability the temporarily borrow from a mutable borrow

Let's do a demo

Freezing

```
// THIS DOES NOT COMPILE
fn main() {
    let mut x = 0;
    let mut y = &mut x;
    x = x + 1;
    *y = *y + 1;
}
```

We'll support *freezing*, i.e., ensuring that borrowed values cannot be mutated

We will *not* support non-lexical lifetimes (so the above code will still fail if we remove the last line in `main`)

Extending Liftimes

```
fn main() {
    let mut x = 1;
    let mut z = 2;
    let mut y = &mut z;
    {
      let mut q = &mut x;
      y = q;
    }
    *y = *y + 1;
}
```

We'll support the ability to extend the lifetime of a borrow based on it's reassignment

We will *not* support uninitialized variables.

Many More to Come...

Our game for the next several weeks will be to come up with as many examples that we can so that we can capture some of the less "obvious" behaviors of Rust

Any suggestions?

Outline

The Source-Level Grammar

```
<expr>  ::= '{' {<stmt> ';'} [<expr>] '}'  ; block
          | 'Box::new' '(' <expr> ')'      ; box
          | <lval>                         ; move/copy
          | '&' ['mut'] <lval>             ; borrow
          | <int>                          ; integers

<stmt>  ::= <expr>                         ; expression
          |'let' 'mut' <var> '=' <expr>    ; delcaration
          | <lval> '=' <expr>              ; assignment

<lval>  ::= <var>                          ; variable
          | '*' <lval>                     ; dereference
```

Some Remarks

- Our parser will also need to handle the `main` function so that we can always check our code against the actual Rust compiler:

```
fn main() {
    // ...
}
```

Some Remarks

- Our parser will also need to handle the `main` function so that we can always check our code against the actual Rust compiler:

```
fn main() {
    // ...
}
```

- We'll implement assertions so that we can more easily check our code without writing tests or print statements:

```
fn main() {
    let mut x = 0;
    let mut y = &mut x;
    *y = *y + 1;
    assert_eq!(x, 1);
}
```

Some Remarks

- The AST for FR differs slightly from that of Rust proper:
  - We want explicitly annotate blocks with lifetimes (we'll handle this at parsing)
  - We have to explicitly annotated variables with "copy" if we want it to be copied (we'll handle this at typing)
- The term `<lval>` stands for "L-value" and is now defunct, Rust now used the term **place expression**, which is an expression that represents a memory location

The Abstract Syntax Tree (AST)

```
pub enum Expr {
    Unit,
    Int(i32),
    Box(Box<Expr>),
    Lval(Lval, Copyable),
    Borrow(Lval),
    MutBorrow(Lval),
    Block(Vec<Stmt>, Box<Expr>, Lifetime),
    AssertEq(Box<Expr>, Box<Expr>),
}
```

Aside: Interpreters for Testing

Question. Why build an interpreter?

Aside: Interpreters for Testing

Question. Why build an interpreter?

- for fun
- to prototype a language construct
- **to build an oracle for testing**

Aside: Interpreters for Testing

Question. Why build an interpreter?

- for fun
- to prototype a language construct
- **to build an oracle for testing**

Fuzz Testing is (roughly) the process of testing a program against a *huge* number of cases. This is as opposed to unit testing, in which we test against a small number of well-chosen cases

Aside: Interpreters for Testing

Question. Why build an interpreter?

- for fun
- to prototype a language construct
- **to build an oracle for testing**

Fuzz Testing is (roughly) the process of testing a program against a *huge* number of cases. This is as opposed to unit testing, in which we test against a small number of well-chosen cases

The problem with fuzzing is that *we need a ground truth* (one solution is to take *not crashing* to be ground truth and generate well-formed programs)

Another fun link: Miri

Outline

Tasks

- Finish assignment 6
- Finish workshop task from Tuesday (it will be part of assignment 7)
- I'll walk around and answer any questions you have about the material and the upcoming project