Linear Lambda Calculus

and the search for an embeddable interpreter github.com/srijs

@periping

λ

Agenda

- Motivation
- What is Lambda Calculus?
  - Example: Addition
  - Restrictions of the Linear Lambda Calculus
- Implementing Lamba Calculus
  - Recursion and Stack-Safety
  - Suspending and Resuming Traversals
- Summary

Motivation

- trying to embed a scripting language
  - in a safe way, will deal with user-supplied input
  - with a nice to use API

Motivation

- trying to embed a scripting language
  - don’t want to something implemented in C
  - Rust’s ownership translates badly into GC’d guest languages

What is Lambda Calculus?

Lambda Calculus

```
Term ::= x          (variable)
       | (λx.t)     (abstraction)
       | (t s)      (application)
```

Lambda Calculus

```
Term ::= x          (variable)

       | |x| t      (abstraction)

       | t(s)       (application)
```

Lambda Calculus

```
enum Term {
    Var(String),
    Abs(String, Box<Term>),
    App(Box<Term>, Box<Term>)
}
```

Beta Reduction

```
(λx.t)s -> t[x := s]

(|x| t)(s) -> { let x = s; t }
```

Substitution

```
x[x := r] = r

y[x := r] = y if x != y


(t s)[x := r] = (t[x := r])(s[x := r])


(λx.t)[x := r] = λx.t

(λy.t)[x := r] = λy.(t[x := r]) if x != y
```

Example: Beta-Reducing an Addition

Reducing an Addition

```
λx. λy. x + y
```

Reducing an Addition

```
(λx. λy. x + y) 1 2
```

```
???
```

Reducing an Addition

```
(λx. λy. x + y) 1 2

((λx. λy. x + y) 1) 2

???
```

Reducing an Addition

```
(λx. λy. x + y) 1 2

((λx. λy. x + y) 1) 2

((λy. x + y)[x := 1]) 2

???
```

Reducing an Addition

```
(λx. λy. x + y) 1 2

((λx. λy. x + y) 1) 2

((λy. x + y)[x := 1]) 2

(λy. 1 + y) 2

???
```

Reducing an Addition

```
(λx. λy. x + y) 1 2

((λx. λy. x + y) 1) 2

((λy. x + y)[x := 1]) 2

(λy. 1 + y) 2

(1 + y)[y := 2]

???
```

Reducing an Addition

```
(λx. λy. x + y) 1 2

((λx. λy. x + y) 1) 2

((λy. x + y)[x := 1]) 2

(λy. 1 + y) 2

(1 + y)[y := 2]

(1 + 2)

???
```

Reducing an Addition

```
(λx. λy. x + y) 1 2

((λx. λy. x + y) 1) 2

((λy. x + y)[x := 1]) 2

(λy. 1 + y) 2

(1 + y)[y := 2]

(1 + 2)

3
```

Restrictions of the Linear Lambda Calculus

Addition

```
add = λx. λy. x + y
```

Double

```
double = λx. x * 2
```

Double

```
double = λx. x + x
```

Double

```
double = λx. x + x
```

```
error: use of moved value: `x`

fn double(x: T) { x + x }
                      ^
```

Implementing Lambda Calculus

Substitution

```
x[x := r] = r
```

```
fn sub(t: Term, x: String, r: Term) -> Term {
  match t {
    Var(y) if x == y => r
  }
}
```

Substitution

```
y[x := r] = y if x != y
```

```
fn sub(t: Term, x: String, r: Term) -> Term {
    match t {
        Var(y) if x != y => Var(y)
    }
}
```

Substitution

```
(t s)[x := r] = (t[x := r])(s[x := r])
```

```
fn sub(t: Term, x: String, r: Term) -> Term {
    match t {
        App(t, s) => App(sub(t, x, r), sub(s, x, r))
    }
}
```

Substitution

```
(λx.t)[x := r] = λx.t
```

```
fn sub(t: Term, x: String, r: Term) -> Term {
   match t {
      Abs(y, t) => if x == y => Abs(y, t)
   }
}
```

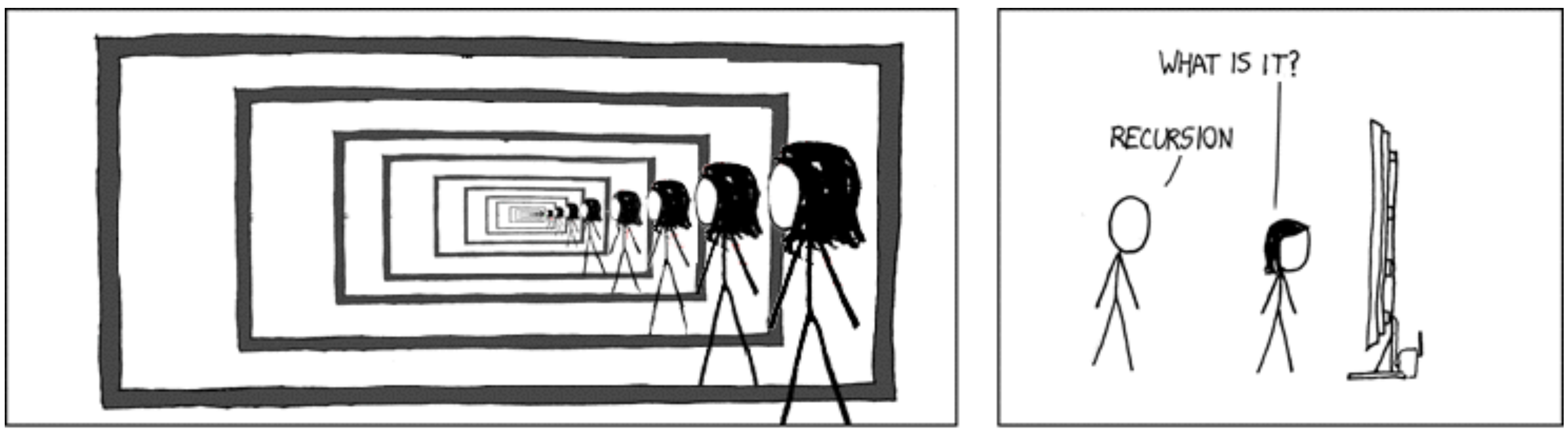
Substitution

```
(λy.t)[x := r] = λy.(t[x := r]) if x != y
```

```
fn sub(t: Term, x: String, r: Term) -> Term {
    match t {
        Abs(y, t) => if x != y => Abs(y, sub(t, x, r))
    }
}
```

Recursion and Stack-Safety

Recursion



Recursion

```
fn factorial(n: u64) -> u64 {
    match n {
        0 => 1,
        n => n * factorial(n - 1)
    }
}
```

Recursion

```
_factorial:
    …
    call factorial
    …
    ret
```

Tail Recursion

Tail Recursion

```
fn fact_iter(p: u64, n: u64) -> u64 {
    match n {
        0 => p,
        n => fact_iter(n * p, n - 1)
    }
}

fn factorial(n: u64) -> u64 {
    fact_iter(1, n)
}
```

Tail Recursion

```
_fact_iter:
    …
    jnz fact_iter
    …
    ret

_factorial:
    …
    jmp fact_iter
```

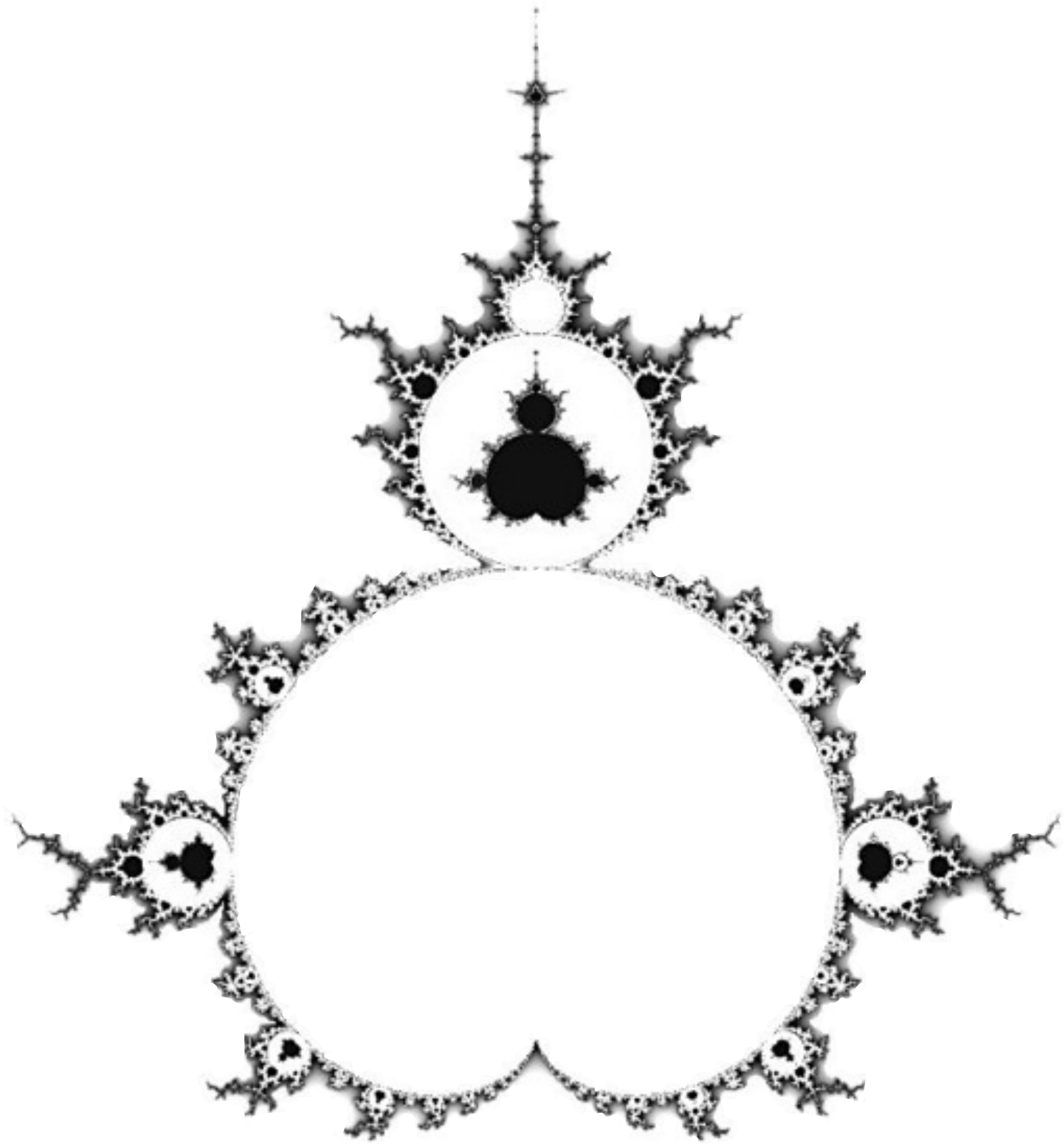
Tail Recursion

```
fn fact_iter(p: u64, n: u64) -> u64 {
    match n {
        0 => p,
        n => become fact_iter(n * p, n - 1)
    }
}

fn factorial(n: u64) -> u64 {
    become fact_iter(1, n)
}
```

??? Recursion

```
fn fact_iter(s: (u64, u64)) -> (u64, u64) {
    match s {
        (p, 0) => (p, 0),
        (p, n) => (n * p, n - 1)
    }
}

fn factorial(n: u64) -> u64 {
    ???
}
```

Fixed Point



Fixed Point Recursion

```
fn fact_iter(s: (u64, u64)) -> (u64, u64) {
  match s {
    (p, 0) => (p, 0),
    (p, n) => (n * p, n - 1)
  }
}

fn factorial(n: u64) -> u64 {
  fix(fact_iter, (1, n)) // output == input
}
```

Fixed Point Functions

```
enum Fix<T> {
  Pro(T),
  Fix(T)
}

trait FixFn<T>

impl<T> FixFn<T> for Fn(T) -> Fix<T> {}
```

Fixed Point Functions

```
fn fact_iter(s: (u64, u64)) -> Fix<(u64, u64)> {
    match s {
        (p, 0) => Fix::Fix((p, 0)),
        (p, n) => Fix::Pro((n * p, n - 1))
    }
}

fn factorial(n: u64) -> u64 {
    fix(fact_iter, (1, n)) // output == Fix::Fix(_)
}
```

Fixed Point Composition

```
fn identity<X>(x: X) -> Fix<X> {
   Fix::Fix(x)
}

fn compose<F, G, X>(f: F, g: G, x: X) -> Fix<X>
   where F: FixFn<X>, G: FixFn<X> {
   match f(x) {
      Fix::Pro(y) => Fix::Pro(y),
      Fix::Fix(y) => g(y)
   }
}
```

Fixed Point Composition

```
// ∀ X : ∀ f g h ∈ FixFn<X> : (f * g) * h = f * (g * h)
assert_eq!(
    compose(|x| compose(f, g, x), h, x),
    compose(f, |x| compose(g, h, x), x)
);
```

```
// ∀ X : ∃ e ∈ FixFn<X> : ∀ f ∈ FixFn<X> : f = e * f = f * e
assert_eq!(f(x), compose(identity, f, x));
assert_eq!(f(x), compose(f, identity, x));
```

Fixed Point Recursion

- stack-safe tail recursion
- efficient specialisation of trampolines
- lawful model for composing recursions
- allows for multi-tasking (suspending and resuming computations)

Suspending and Resuming Traversals

Term

```
enum Term {
  Var(String),
  Abs(String, Box<Term>),
  App(Box<Term>, Box<Term>)
}
```

Iterators

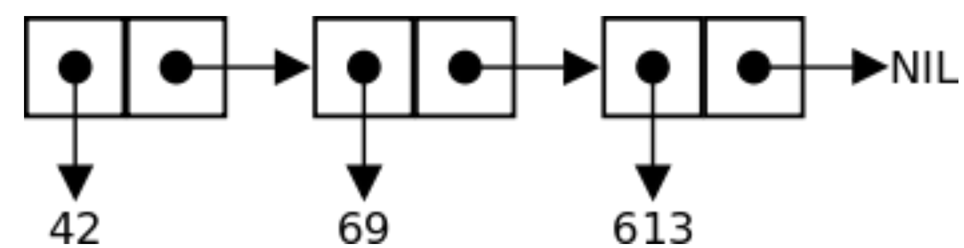
```
struct Iter<‘a> {
  …
}

impl<‘a> Iterator for Iter<‘a> {
  type Item = ???;
  fn next(&mut self) -> ???;
}
```

Iterators

- 1-dimensional traversal
- bound to lifetime of data structure (can’t easily be paused and resumed)
- mutable interface

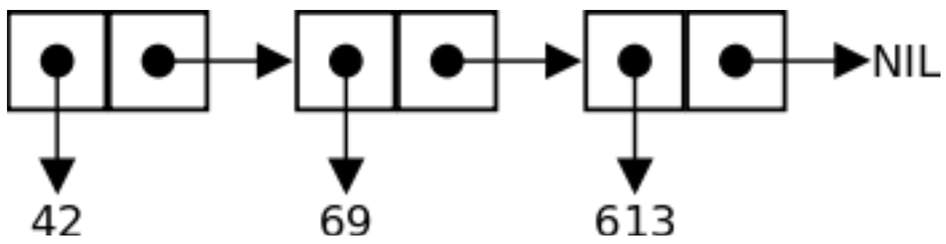
Cons List

```
enum List<A> {
  Nil,
  Cons(A, Box<List<A>>)
}
```



View

```
struct View<A> {
  left: List<A>,
  right: List<A>
}
```



View

```
impl<A> View<A> {
  pub fn from_list(x: List<A>) {
    View{left: List::Nil, right: x}
  }
}
```

Traversal

```
impl<A> View<A> {
    pub fn move_right(self) -> Option<Self> {
        match self.right {
            List::Nil => None,
            List::Cons(center, right) => Some(View{left:
List::Cons(center, self.left), right: right})
    }
}
```

Traversal

```
impl<A> View<A> {
   pub fn move_left(self) -> Option<Self> {
      match self.left {
         List::Nil => None,
         List::Cons(center, left) => Some(View{left:
left, right: List::Cons(center, self.right)})
   }
}
```

Introspection

```
impl<A> View<A> {
    pub fn get_current(&self) -> Option<&A> {
        match self.right {
            &List::Nil => None,
            &List::Cons(ref center, _) => Some(center)
        }
    }
}
```

Modification

```
impl<A> View<A> {

   pub fn set_current(self, other: A) -> Self {

      match self.right {

         List::Nil => self,

         List::Cons(center, right) => View{left:
self.left, right: List::Cons(other, right)}

   }

}
```

Fixed Point Traversal

```
impl<A> View<A> {

   pub fn move_right(self) -> Fix<Self> {

      match self.right {

         List::Nil => Fix::Fix(self),

         List::Cons(center, right) =>
Fix::Pro(View{left: List::Cons(center, self.left),
right: right})

   }

}
```

Fixed Point Traversal

```
impl<A> View<A> {
  pub fn move_end(self) -> Self {
    fix(View::move_right, self)
  }
}
```

Zippers

- constant-space and constant-time moves
- completely pure, immutable machinery
- traversal can be paused and resumed
- can be generalised for every algebraic data type!

Term

```
enum Term {
   Var(String),
   Abs(String, Box<Term>),
   App(Box<Term>, Box<Term>)
}
```

Context

```
enum Ctx {
  Top,
  Abs(String, Box<Ctx>),
  AppL(Box<Ctx>, Term),
  AppR(Term, Box<Ctx>)
}
```

View

```
struct View {
  ctx: Ctx,
  term: Term
}
```

View

```
impl View {
  pub fn down(self) -> Fix<Self>;
  pub fn up(self) -> Fix<Self>;
  pub fn left(self) -> Fix<Self>;
  pub fn right(self) -> Fix<Self>;
}
```

Environmentally-aware Zippers

Term

```
enum Term {
   Var(String),
   Abs(String, Box<Term>),
   App(Box<Term>, Box<Term>)
}
```

Context

```
enum Ctx {
  Top,
  Abs(String, Box<Ctx>),
  AppL(Box<Ctx>, Term),
  AppR(Term, Box<Ctx>)
}
```

Environment

```
struct Env<T> {
  map: HashMap<String, Vec<Option<T>>>
}
```

Environment

```
fn down(self) -> Fix<Self> {
  match self.ctx {
      Abs(s, t) => {
self.env.entry(s).or_insert(Vec::new()).push(());
      …
      }
   }
}
```

Environment

```
fn up(self) -> Fix<Self> {
  match self.ctx {
      Abs(s, t) => {
          self.env.entry(s).or_insert(Vec::new()).pop();
      …
      }
    }
}
```

View

```
struct View {
  ctx: Ctx,
  env: Env,
  term: Term
}
```

Environmentally-aware Zippers

- central implementation for scoping rules, can be reused by every traversal
- constant-time lookup of bindings
- still (amortised) constant-time moves

Summary

Great Journey

- learned a lot about how functional concepts can benefit Rust
- published several useful crates in the process
- found a bug in the borrow checker

Summary

- a safe, functional implementation of LLC
- pause-able and resumable, even serialisable
- an API that is very usable from Rust
- the restrictions of the host language map directly to the restrictions of the guest language — many benefits retained