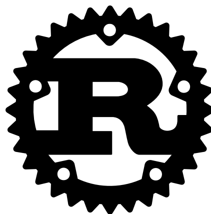
Rust and functional programming

Isaac Elliott

19 September, 2023

What is Rust?

https://www.rust-lang.org/



- Created at Mozilla in late 2000s and early 2010s
- "Systems programming" language
- Prioritises safety and performance
- Built on modern (30 years or younger) programming language research

Hello world

```
fn main() {
    println!("Hello world!");
}
```


Rust is commonly described as a "systems programming" language. To me it's a programming language for people who want to pay more attention to machine-level details, such as memory layout, instruction counts, and amount of branching.

It differs from C in its approach to safety. Whereas C favours simplicity and performance at the expense of safety, Rust favours safety and performance at the expense of simplicity. Rust is a more complex programming language than C, and Rust compilers are harder to implement than C compilers, because the Rust language and compiler does more for the programmer.

What is functional programming?

What *is* functional programming?

Is it programming with functions? With higher-order functions? With immutable data structures? Does functional programming just mean, "doing your best to embed a Haskell dialect in your language of choice"?

I'm not going to try to answer this question. I think the question of whether it *is* anything in particular might be misguided. People have spent a lot of energy debating what should and shouldn't be considered functional programming, and which languages should be considered "functional". Right now I don't see much value in an algorithm that decides what is and isn't "functional".

Here's why:

What is functional programming?

- Programming with functions?

What *is* functional programming?

Is it programming with functions? With higher-order functions? With immutable data structures? Does functional programming just mean, "doing your best to embed a Haskell dialect in your language of choice"?

I'm not going to try to answer this question. I think the question of whether it *is* anything in particular might be misguided. People have spent a lot of energy debating what should and shouldn't be considered functional programming, and which languages should be considered "functional". Right now I don't see much value in an algorithm that decides what is and isn't "functional".

Here's why:

What is functional programming?

- Programming with functions?
- ...with higher-order functions?

What *is* functional programming?

Is it programming with functions? With higher-order functions? With immutable data structures? Does functional programming just mean, "doing your best to embed a Haskell dialect in your language of choice"?

I'm not going to try to answer this question. I think the question of whether it *is* anything in particular might be misguided. People have spent a lot of energy debating what should and shouldn't be considered functional programming, and which languages should be considered "functional". Right now I don't see much value in an algorithm that decides what is and isn't "functional".

Here's why:

What is functional programming?

- Programming with functions?
- ...with higher-order functions?
- ...with immutable data structures?

What *is* functional programming?

Is it programming with functions? With higher-order functions? With immutable data structures? Does functional programming just mean, "doing your best to embed a Haskell dialect in your language of choice"?

I'm not going to try to answer this question. I think the question of whether it *is* anything in particular might be misguided. People have spent a lot of energy debating what should and shouldn't be considered functional programming, and which languages should be considered "functional". Right now I don't see much value in an algorithm that decides what is and isn't "functional".

Here's why:

What is functional programming?

- Programming with functions?
- ...with higher-order functions?
- ...with immutable data structures?
- Writing Haskell in disguise?

What *is* functional programming?

Is it programming with functions? With higher-order functions? With immutable data structures? Does functional programming just mean, "doing your best to embed a Haskell dialect in your language of choice"?

I'm not going to try to answer this question. I think the question of whether it *is* anything in particular might be misguided. People have spent a lot of energy debating what should and shouldn't be considered functional programming, and which languages should be considered "functional". Right now I don't see much value in an algorithm that decides what is and isn't "functional".

Here's why:

Personal identity and functional programming

Before I get into the technical details, I want to talk about a use (misuse?) of the term "functional programming": as a sort of "tribal identity marker". Here's something true of my experience that might also be true for you:

I enjoy spending time with people who have similar interests to me. I like talking about programming with other programmers (that's why I'm here tonight!), and listening to music with people who have similar tastes. In situations like these, in a community of sorts, I feel a kind of satisfaction that I don't feel when I'm alone or with people who are extremely different to me.

I've noticed that if I don't pay enough attention, I have a tendency to act as if *my* interests / tastes/ preferences are somehow objectively better than the alternatives.

Personal identity and functional programming

I've also experienced something kind of toxic that happens when this (usually incorrect) tendency to think that my choice is Better spreads to the group and becomes part of the way we bond. We start to celebrate the supposed fact that we are the small group who made the "right" choice, and start to find meaning in pointing out the ways that everyone else is "wrong".

This is kind of how I related to "functional programming" when I was younger. I enjoyed a particular style of programming or flavour of programming language, and thought I was being attracted to some kind of objective superiority. Sometimes I even had a feeling of *ethical* superiority. Under those perspectives, at the worst of times, the question of "is X functional programming?" could be less like asking, "in which section of the library do we put the book on X?" and more like, "do I approve of X or is it yucky / weird / other?"

I failed to see the ways in which my attraction to "functional programming" was due to my personal history and my taste in writing software. And I failed to acknowledge the nebulousity of this "paradigm" which at the time seemed so objective.

Common functional programming concepts

So as an effort of humility, instead of presenting a definition of functional programming and measuring Rust against it, I'm going cover a few topics that seem most closely associated to "functional programming" (whatever *that* is), and talk about the extent to which they show up in Rust.

Before I reveal them, I want to do a survey. What are some of the ideas *you* would include?

Common functional programming concepts

- Algebraic datatypes and higher-order functions

So as an effort of humility, instead of presenting a definition of functional programming and measuring Rust against it, I'm going cover a few topics that seem most closely associated to "functional programming" (whatever *that* is), and talk about the extent to which they show up in Rust.

Before I reveal them, I want to do a survey. What are some of the ideas *you* would include?

Common functional programming concepts

- Algebraic datatypes and higher-order functions
- Immutability

So as an effort of humility, instead of presenting a definition of functional programming and measuring Rust against it, I'm going cover a few topics that seem most closely associated to "functional programming" (whatever *that* is), and talk about the extent to which they show up in Rust.

Before I reveal them, I want to do a survey. What are some of the ideas *you* would include?

Common functional programming concepts

- Algebraic datatypes and higher-order functions
- Immutability
- Static types

So as an effort of humility, instead of presenting a definition of functional programming and measuring Rust against it, I'm going cover a few topics that seem most closely associated to "functional programming" (whatever *that* is), and talk about the extent to which they show up in Rust.

Before I reveal them, I want to do a survey. What are some of the ideas *you* would include?

Common functional programming concepts

- Algebraic datatypes and higher-order functions
- Immutability
- Static types
- Equational reasoning

So as an effort of humility, instead of presenting a definition of functional programming and measuring Rust against it, I'm going cover a few topics that seem most closely associated to "functional programming" (whatever *that* is), and talk about the extent to which they show up in Rust.

Before I reveal them, I want to do a survey. What are some of the ideas *you* would include?

Algebraic datatypes and higher-order functions

Algebraic datatypes and higher-order functions — ADTs

Definition

```
enum Sum<A, B>{
    Left(A),
    Right(B)
}

struct Product<A, B>{
    first: A,
    second: B
}
```


Rust has sums, which it calls "enums", and products, which it calls "structs".

Enums are examined by pattern matching, and structs by field accessors.

Algebraic datatypes and higher-order functions — ADTs

Definition

```
enum Sum<A, B>{
    Left(A),
    Right(B)
}

struct Product<A, B>{
    first: A,
    second: B
}
```

Creation

```
let x = Sum::Left(1);
let y = Sum::Right(true);

let z = Product{
    first: "x",
    second: "y"
};
```


Rust has sums, which it calls "enums", and products, which it calls "structs".

Enums are examined by pattern matching, and structs by field accessors.

Algebraic datatypes and higher-order functions — ADTs

Definition

```
enum Sum<A, B>{
    Left(A),
    Right(B)
}

struct Product<A, B>{
    first: A,
    second: B
}
```

Creation

```
let x = Sum::Left(1);
let y = Sum::Right(true);

let z = Product{
    first: "x",
    second: "y"
};
```

Use

```
let x: Sum<A, B> = ...;
let y = match x {
    Sum::Left(a) => ...,
    Sum::Right(b) => ...
};
```

```
let x: Product<A, B> = ...;
let y = x.first;
let z = y.second;
```


Rust has sums, which it calls "enums", and products, which it calls "structs".

Enums are examined by pattern matching, and structs by field accessors.

Algebraic datatypes and higher-order functions — ADTs

- Extremely easy to define, create, and use
- Has a number of shorthands that make ADTs even more enjoyable to work with

Algebraic datatypes and higher-order functions — ADTs

- Extremely easy to define, create, and use
- Has a number of shorthands that make ADTs even more enjoyable to work with
  - Named enum fields

```
enum Sum<A, B>{
    Left{left_arg: A},
    Right{right_arg: B}
}
```

Algebraic datatypes and higher-order functions — ADTs

- Extremely easy to define, create, and use
- Has a number of shorthands that make ADTs even more enjoyable to work with
  - Named enum fields
    ```
    enum Sum<A, B>{
        Left{left_arg: A},
        Right{right_arg: B}
    }
    ```
  - Named field punning
    ```
    let first = ..; let second = ..;
    let z = Pair{first, second};
    ```

Algebraic datatypes and higher-order functions — ADTs

- Extremely easy to define, create, and use
- Has a number of shorthands that make ADTs even more enjoyable to work with
  - Named enum fields
    ```
    enum Sum<A, B>{
        Left{left_arg: A},
        Right{right_arg: B}
    }
    ```
  - Named field punning
    ```
    let first = ..; let second = ..;
    let z = Pair{first, second};
    ```
  - Disjunctive patterns / "or-patterns"
    ```
    match x {
        0 => ..,
        1 | 2 | 3 => ..,
        4 => ..,
        _ => ..
    }
    ```

Algebraic datatypes and higher-order functions — HOFs

Rust has anonymous functions, known as *closures*.

Creation

```
let f = |x| x + 1;
```

Use

```
let y = f(x);
```

Algebraic datatypes and higher-order functions — HOFs

Rust has anonymous functions, known as *closures*.

Creation

```
let f = |x| x + 1;
```

Use

```
let y = f(x);
```

Higher-order functions are fairly common in the standard library.

Example: `map`

```
let xs: Vec<u32> = vec![1, 2, 3, 4];
let ys: Vec<u32> = xs.iter().map(|x| x + 1).collect();
```

Algebraic datatypes and higher-order functions — HOFs

Working with closures / HOFs in Rust is more complex than in other languages.

Haskell

```haskell
map :: (a -> b) -> Maybe a -> Maybe b
maybe :: b -> (a -> b) -> Maybe a -> b
```

Rust

```rust
fn map<A, B, F: FnOnce(A) -> B>(
    f: F,
    value: Option<A>
) -> Option<B>

fn map_or<A, B, F: FnOnce(A) -> B>(
    default: B,
    f: F,
    value: Option<A>
) -> B
```

Algebraic datatypes and higher-order functions — HOFs

Most languages have a single function type.

Rust has 3 kinds of closure:

- `FnOnce`
- `Fn`
- `FnMut`

Most languages have a single function type. Rust has more. There are 3 kinds of closure: `FnOnce`, `Fn`, and `FnMut`. I won't explain what they mean right now. Let me just say: they are all justified in the context of Rust's goals of safety and performance. The price of achieving these goals for closures in Rust is simplicity: when you write a higher-order function, you need to figure out which kind of higher-order function is most appropriate.

Algebraic datatypes and higher-order functions

Higher-order functions (in some form or another) seem like an essential component of "functional programming".

It seems reasonable to me to claim that higher-order functions are an essential part of functional programming.

I've put algebraic datatypes and higher-order functions in the same section because they're actually interdefinable. If you have one, then you have the other (assuming you have first-order function definitions).

Using higher-order functions to define algebraic datatypes is called Church encoding.

Going the other way, algebraic datatypes (together with first-order functions) to can be used to define higher-order functions through a method called Defunctionalisation, which was the subject of Jack's recent talk, "Everything looks like a function".

Algebraic datatypes and higher-order functions

Higher-order functions (in some form or another) seem like an essential component of "functional programming".

Algebraic datatypes and higher-order functions are interdefinable:

It seems reasonable to me to claim that higher-order functions are an essential part of functional programming.

I've put algebraic datatypes and higher-order functions in the same section because they're actually interdefinable. If you have one, then you have the other (assuming you have first-order function definitions).

Using higher-order functions to define algebraic datatypes is called Church encoding.

Going the other way, algebraic datatypes (together with first-order functions) to can be used to define higher-order functions through a method called Defunctionalisation, which was the subject of Jack's recent talk, "Everything looks like a function".

Algebraic datatypes and higher-order functions

Higher-order functions (in some form or another) seem like an essential component of "functional programming".

Algebraic datatypes and higher-order functions are interdefinable:

- ADTs via HOFs — Church encoding

It seems reasonable to me to claim that higher-order functions are an essential part of functional programming.

I've put algebraic datatypes and higher-order functions in the same section because they're actually interdefinable. If you have one, then you have the other (assuming you have first-order function definitions).

Using higher-order functions to define algebraic datatypes is called Church encoding.

Going the other way, algebraic datatypes (together with first-order functions) to can be used to define higher-order functions through a method called Defunctionalisation, which was the subject of Jack's recent talk, "Everything looks like a function".

Algebraic datatypes and higher-order functions

Higher-order functions (in some form or another) seem like an essential component of "functional programming".

Algebraic datatypes and higher-order functions are interdefinable:

- ADTs via HOFs — Church encoding
- HOFs via ADTs — Defunctionalization

It seems reasonable to me to claim that higher-order functions are an essential part of functional programming.

I've put algebraic datatypes and higher-order functions in the same section because they're actually interdefinable. If you have one, then you have the other (assuming you have first-order function definitions).

Using higher-order functions to define algebraic datatypes is called Church encoding.

Going the other way, algebraic datatypes (together with first-order functions) to can be used to define higher-order functions through a method called Defunctionalisation, which was the subject of Jack's recent talk, "Everything looks like a function".

Immutability

Immutability

Variable bindings in Rust are immutable by default:

Immutable by default

```
let x = 1;
x = x + 1; // error: cannot assign twice to immutable variable `x`
```

Immutability

Variable bindings in Rust are immutable by default:

Immutable by default

```rust
let x = 1;
x = x + 1; // error: cannot assign twice to immutable variable `x`
```

Rust allows mutability, which comes in 3 flavours:

Immutability

Variable bindings in Rust are immutable by default:

Immutable by default

```rust
let x = 1;
x = x + 1; // error: cannot assign twice to immutable variable `x`
```

Rust allows mutability, which comes in 3 flavours:

- Local mutability — mild

Immutability

Variable bindings in Rust are immutable by default:

Immutable by default

```
let x = 1;
x = x + 1; // error: cannot assign twice to immutable variable `x`
```

Rust allows mutability, which comes in 3 flavours:

- Local mutability — mild
- Mutable references — medium

Immutability

Variable bindings in Rust are immutable by default:

Immutable by default

```
let x = 1;
x = x + 1; // error: cannot assign twice to immutable variable `x`
```

Rust allows mutability, which comes in 3 flavours:

- Local mutability — mild
- Mutable references — medium
- Interior (hidden) mutability — spicy

~~Im~~Mutability — Local

Local mutability

```
fn two() -> u32 {
    let mut x: u32 = 0;
    x += 1;
    x += 1;
    x
}
```

This is the most benign form of mutability because it is compositional. When I call a function, that function's use of local mutability is not a concern to me.

~~Im~~Mutability — References

Definition

```
fn increment_immutable(x: &u32) {
    *x = x + 1; // error: cannot assign to `*x`, which is
                // behind a `&` reference
}

fn increment_mutable(x: &mut u32) {
    *x = x + 1; // ok
}
```


I've lost some compositionality: if I'm not careful with my design then I could create a history-sensitive function. A function that only works properly when I call it in the right place at the right time.

The upside is that because the mutability is in the type signature I'm reminded of this possibility, so I can more consciously opting in to (potential) mutation when I call such functions.

~~Im~~Mutability — References

Use

```
fn increment_mutable(x: &mut u32) {
    *x = x + 1;
}

let x = 1;
increment_mutable(&x) // error: types differ in mutability

let x = 1;
increment_mutable(&mut x) // error: cannot borrow `x` as mutable,
                          // as it is not declared as mutable
let mut x = 1;
increment_mutable(&mut x) // ok
```

You can only take a mutable reference to a mutable variable. I find adding the `mut` keyword to a binding to be a helpful "confirmation" step for mindful mutability.

~~Im~~Mutability — Interior (hidden)

Some types allow mutation through an "immutable" reference:

- `Cell`
- `RefCell`
- `Mutex`
- `RwLock`
- atomics

Some types allow mutation through an "immutable" reference. That's a hint that the term "immutable reference" is a bit of a misnomer. Instead of talking about "immutable" and "mutable" references, sometimes it's better to use the terms "shared" and "exclusive", respectively.

Rust prevents data races by only allowing writes via an exclusive reference.

~~Im~~Mutability — Interior (hidden)

Some types allow mutation through an "immutable" reference:

- `Cell`
- `RefCell`
- `Mutex`
- `RwLock`
- atomics

Type	Better Name	Multiplicity	Writable (`*x = ..`)

Some types allow mutation through an "immutable" reference. That's a hint that the term "immutable reference" is a bit of a misnomer. Instead of talking about "immutable" and "mutable" references, sometimes it's better to use the terms "shared" and "exclusive", respectively.

Rust prevents data races by only allowing writes via an exclusive reference.

~~Im~~Mutability — Interior (hidden)

Some types allow mutation through an "immutable" reference:

- `Cell`
- `RefCell`
- `Mutex`
- `RwLock`
- atomics

Type	Better Name	Multiplicity	Writable (`*x = ..`)
&T	Shared reference	Many	No

Some types allow mutation through an "immutable" reference. That's a hint that the term "immutable reference" is a bit of a misnomer. Instead of talking about "immutable" and "mutable" references, sometimes it's better to use the terms "shared" and "exclusive", respectively.

Rust prevents data races by only allowing writes via an exclusive reference.

~~Im~~Mutability — Interior (hidden)

Some types allow mutation through an "immutable" reference:

- `Cell`
- `RefCell`
- `Mutex`
- `RwLock`
- atomics

Type	Better Name	Multiplicity	Writable (`*x = ..`)
`&T`	Shared reference	Many	No
`&mut T`	Exclusive reference	One	Yes

Some types allow mutation through an "immutable" reference. That's a hint that the term "immutable reference" is a bit of a misnomer. Instead of talking about "immutable" and "mutable" references, sometimes it's better to use the terms "shared" and "exclusive", respectively.

Rust prevents data races by only allowing writes via an exclusive reference.

~~Im~~Mutability — Interior (hidden)

Definition

```
fn increment_interior(x: &Cell<u64>) -> u64 {
    let result = x.get() + x.get();
    x.set(x.get(x) + 1);
    result
}
```

Mutability is no longer in the type signature.

Example

```
struct S{
    x: Cell<u64>
}

// Does `f_S` mutate its argument?
fn f_S(s: &S) -> { .. }
```


When is interior mutability useful?

It's necessary for thread-shared mutable state, and that's the only time I've used it. I've never used ‘Cell‘ or ‘RefCell‘ and I haven't thought about when they'd be justified.

Static types

Static types — common features

Algebraic datatypes — structs and enums

```
struct TypeName{
  field_1: FieldType1,
  field_2: FieldType2,
  ...
}

enum TypeName{
  Constructor1(FieldType11, FieldType12, ...),
  Constructor2,
  ...,
}
```

Static types — common features

Parametric polymorphism — generics

```
struct TypeName<Param1, Param2>{ ... }
enum TypeName<Param1, Param2>{ ... }

fn id<A>(x: A) -> A { x }
```

Static types — common features

Ad-hoc polymorphism — traits

```
trait Monoid {
  fn empty() -> Self;
  fn combine(self, other: Self) -> Self
}

fn fold<M: Monoid>(xs: Vec<M>) -> M { .. }
```

Static types — uncommon features

- Mutability tracking — `&T` vs. `&mut T`
- Reference lifetime analysis (borrow checking)
- Uniqueness types

Static types — reference lifetime analysis

Every reference points to valid memory.

Rust

```rust
struct IntAndBool{
  first: i32,
  second: bool
}

fn first<'a>(input: &'a IntAndBool) -> &'a i32 {
  &input.first
}

fn bad<'a>() -> &'a i32 {
    let x = IntAndBool{first: 99, second: true};
    let y = first(&x);
    y
//  ^ error: returns a value referencing data
//    owned by the current function
}
```

C

```c
typedef struct {
  long first;
  bool second;
} IntAndBool;

long* first(IntAndBool* input) {
  return &input->first;
}

long* bad() {
  IntAndBool x = { .first = 99, .second = true };
  long* y = first(&x);
  return y; // ok...?
}
```


Reference lifetime analysis, generally known as "borrow checking", determines at compile time whether a program references data in a safe way.

For example, you’re allowed to get a refrence to a struct’s field, pass it to and return it from functions, store it in another datatype, etc. and the compiler will prevent you from doing anything that could cause the reference to outlive the struct. In other words, no dangling references / use-after-frees.

Static types — uniqueness types

&T — shared

Static types — uniqueness types

&T — shared

Concurrent use

```
fn f(x: &T, y: &T) { .. }

..

let x: T = ..;
f(&x, &x); // ok
```

Static types — uniqueness types

&T — shared

Concurrent use

```
fn f(x: &T, y: &T) { .. }

..

let x: T = ..;
f(&x, &x); // ok
```

Sequential use

```
fn g(x: &T) { .. }
fn h(x: &T) { .. }

..

let x: T = ..;
f(&x);
g(&x); // ok
```

Static types — uniqueness types

`&mut T` — exclusive

Static types — uniqueness types

`&mut T` — exclusive

Concurrent use

```
fn f(x: &mut T, y: &mut T) { .. }

..

let mut x: T = ..;
// error: cannot borrow `x` as
// mutable more than once at
// a time
f(&mut x, &mut x);
//         ^^^^^^ second mutable
//         borrow occurs here
```

Static types — uniqueness types

`&mut T` — exclusive

Concurrent use

```
fn f(x: &mut T, y: &mut T) { .. }

..

let mut x: T = ..;
// error: cannot borrow `x` as
// mutable more than once at
// a time
f(&mut x, &mut x);
//        ^^^^^^ second mutable
//        borrow occurs here
```

Sequential use

```
fn g(x: &mut T) { .. }
fn h(x: &mut T) { .. }

..

let mut x: T = ..;
f(&mut x);
g(&mut x); // ok
```

Static types — uniqueness types

T — unique

You can think of a variable with of a unique type as being a "resource" that can be consumed by a function. After the resource has been consumed, it is no longer available for use. Rust calls this "transferring ownership".
Following the "ownership" metaphor, taking a reference is called "borrowing". (Because ownership is returned once the reference is deleted)
I find unique types extremely interesting because of how they interact with equational reasoning and mutability, which I will cover later.

Static types — uniqueness types

T — unique

Concurrent use

```
fn f(x: T, y: T) { .. }

..

let x: T = ..;
// error: use of moved value `x`
f(x, x);
//   ^ value used here after move
```

Sequential use

```
fn g(x: T) { .. }
fn h(x: T) { .. }

..

let x: T = ..;
f(x);
// error: use of moved value `x`
g(x);
//^ value used here after move
```


You can think of a variable with of a unique type as being a "resource" that can be consumed by a function. After the resource has been consumed, it is no longer available for use. Rust calls this "transferring ownership".
Following the "ownership" metaphor, taking a reference is called "borrowing". (Because ownership is returned once the reference is deleted)
I find unique types extremely interesting because of how they interact with equational reasoning and mutability, which I will cover later.

Static types and functional programming

There's a deep theoretical reason that types are associated with functional programming. The lambda calculus is a turing-complete model of computation which consists *only* of functions. The definitive functional programming language. There are ways of defining type systems for the lambda calculus that are equivalent to mathematical logics. In this setting, a type is equivalent to a proposition, and a well-typed lambda calculus program is a proof of that proposition. This is known as the Curry-Howard correspondence and I think Donovan will be covering it in more detail next month.

Given this, some people might suggest that a functional programming language is one that corresponds to a logic via Curry-Howard.

The validity of "types as propositions and proofs as programs" for a programming language requires type checking of that language to be decidable.

Rust's type checker is known to be undecidable. It's possible to write Rust programs that would take an infinite amount of time to type check. I think it's nearly impossible in a typical software engineering context.

Static types and functional programming

- Come to Donovan's talk next month!

There's a deep theoretical reason that types are associated with functional programming. The lambda calculus is a turing-complete model of computation which consists *only* of functions. The definitive functional programming language. There are ways of defining type systems for the lambda calculus that are equivalent to mathematical logics. In this setting, a type is equivalent to a proposition, and a well-typed lambda calculus program is a proof of that proposition. This is known as the Curry-Howard correspondence and I think Donovan will be covering it in more detail next month.

Given this, some people might suggest that a functional programming language is one that corresponds to a logic via Curry-Howard.

The validity of "types as propositions and proofs as programs" for a programming language requires type checking of that language to be decidable.

Rust's type checker is known to be undecidable. It's possible to write Rust programs that would take an infinite amount of time to type check. I think it's nearly impossible in a typical software engineering context.

Static types and functional programming

- Come to Donovan's talk next month!
- Curry-Howard correspondence: types are propositions and (lambda calculus) programs are proofs

There's a deep theoretical reason that types are associated with functional programming. The lambda calculus is a turing-complete model of computation which consists *only* of functions. The definitive functional programming language. There are ways of defining type systems for the lambda calculus that are equivalent to mathematical logics. In this setting, a type is equivalent to a proposition, and a well-typed lambda calculus program is a proof of that proposition. This is known as the Curry-Howard correspondence and I think Donovan will be covering it in more detail next month.

Given this, some people might suggest that a functional programming language is one that corresponds to a logic via Curry-Howard.

The validity of "types as propositions and proofs as programs" for a programming language requires type checking of that language to be decidable.

Rust's type checker is known to be undecidable. It's possible to write Rust programs that would take an infinite amount of time to type check. I think it's nearly impossible in a typical software engineering context.

Static types and functional programming

- Come to Donovan's talk next month!
- Curry-Howard correspondence: types are propositions and (lambda calculus) programs are proofs
- Only holds in general when a type system is decidable

There's a deep theoretical reason that types are associated with functional programming. The lambda calculus is a turing-complete model of computation which consists *only* of functions. The definitive functional programming language. There are ways of defining type systems for the lambda calculus that are equivalent to mathematical logics. In this setting, a type is equivalent to a proposition, and a well-typed lambda calculus program is a proof of that proposition. This is known as the Curry-Howard correspondence and I think Donovan will be covering it in more detail next month.

Given this, some people might suggest that a functional programming language is one that corresponds to a logic via Curry-Howard.

The validity of "types as propositions and proofs as programs" for a programming language requires type checking of that language to be decidable.

Rust's type checker is known to be undecidable. It's possible to write Rust programs that would take an infinite amount of time to type check. I think it's nearly impossible in a typical software engineering context.

Static types and functional programming

- Come to Donovan's talk next month!
- Curry-Howard correspondence: types are propositions and (lambda calculus) programs are proofs
- Only holds in general when a type system is decidable
- Rust's type system is (technically) undecidable
  - https://sdleffler.github.io/RustTypeSystemTuringComplete/
  - https://github.com/Ashymad/fortraith

There's a deep theoretical reason that types are associated with functional programming. The lambda calculus is a turing-complete model of computation which consists *only* of functions. The definitive functional programming language. There are ways of defining type systems for the lambda calculus that are equivalent to mathematical logics. In this setting, a type is equivalent to a proposition, and a well-typed lambda calculus program is a proof of that proposition. This is known as the Curry-Howard correspondence and I think Donovan will be covering it in more detail next month.

Given this, some people might suggest that a functional programming language is one that corresponds to a logic via Curry-Howard.

The validity of "types as propositions and proofs as programs" for a programming language requires type checking of that language to be decidable.

Rust's type checker is known to be undecidable. It's possible to write Rust programs that would take an infinite amount of time to type check. I think it's nearly impossible in a typical software engineering context.

Static types and functional programming

- Come to Donovan's talk next month!
- Curry-Howard correspondence: types are propositions and (lambda calculus) programs are proofs
- Only holds in general when a type system is decidable
- Rust's type system is (technically) undecidable
  - `https://sdleffler.github.io/RustTypeSystemTuringComplete/`
  - `https://github.com/Ashymad/fortraith`
- But you probably don't need to worry about that ;)

There's a deep theoretical reason that types are associated with functional programming. The lambda calculus is a turing-complete model of computation which consists *only* of functions. The definitive functional programming language. There are ways of defining type systems for the lambda calculus that are equivalent to mathematical logics. In this setting, a type is equivalent to a proposition, and a well-typed lambda calculus program is a proof of that proposition. This is known as the Curry-Howard correspondence and I think Donovan will be covering it in more detail next month.

Given this, some people might suggest that a functional programming language is one that corresponds to a logic via Curry-Howard.

The validity of "types as propositions and proofs as programs" for a programming language requires type checking of that language to be decidable.

Rust's type checker is known to be undecidable. It's possible to write Rust programs that would take an infinite amount of time to type check. I think it's nearly impossible in a typical software engineering context.

Equational reasoning

Equational reasoning

Reasoning about / manipulating programs as if they are a set of equations.

You use equational reasoning whenever you factor out an expression or function, and when you go the other way by inlining.

Equational reasoning also comes up whenever you simpilify or optimise code. Optimisations are equations between programs where one side of the equation has a lower "cost" than the other side.
It's a really important concept!

Equational reasoning

Reasoning about / manipulating programs as if they are a set of equations.

Factor out

```
(y + 1) * (y + 1)
```

⇝

```
let x = y + 1;
x * x
```


You use equational reasoning whenever you factor out an expression or function, and when you go the other way by inlining.

Equational reasoning also comes up whenever you simpilify or optimise code. Optimisations are equations between programs where one side of the equation has a lower "cost" than the other side.
It’s a really important concept!

Equational reasoning

Reasoning about / manipulating programs as if they are a set of equations.

Factor out

```
(y + 1) * (y + 1)
```

⇝

```
let x = y + 1;
x * x
```

Inline

```
let x = y + 1;
x * x
```

⇝

```
(y + 1) * (y + 1)
```


You use equational reasoning whenever you factor out an expression or function, and when you go the other way by inlining.

Equational reasoning also comes up whenever you simpilify or optimise code. Optimisations are equations between programs where one side of the equation has a lower "cost" than the other side.
It's a really important concept!

Equational reasoning

Reasoning about / manipulating programs as if they are a set of equations.

Factor out

```
(y + 1) * (y + 1)
```

$\rightsquigarrow$

```
let x = y + 1;
x * x
```

Inline

```
let x = y + 1;
x * x
```

$\rightsquigarrow$

```
(y + 1) * (y + 1)
```

Simplify

```
let y = x * 0 + 2
```

$\rightsquigarrow \{ \forall x.\ x * 0 = 0 \}$

```
let y = 2
```


You use equational reasoning whenever you factor out an expression or function, and when you go the other way by inlining.

Equational reasoning also comes up whenever you simpilify or optimise code. Optimisations are equations between programs where one side of the equation has a lower "cost" than the other side.
It's a really important concept!

Equational reasoning in Rust

Not always possible in Rust.

Example

```
let mut x: u32 = 2;

let mut f = || -> u32 {
    x += 1;
    x
};

let y: u32 = f();
y + y
```

Execution

```
let y = f();
y + y
```

Equational reasoning in Rust

Not always possible in Rust.

Example

```
let mut x: u32 = 2;

let mut f = || -> u32 {
    x += 1;
    x
};

let y: u32 = f();
y + y
```

Execution

```
let y = f();
y + y
```

$\rightsquigarrow$

```
let y = 3;
y + y
```

Equational reasoning in Rust

Not always possible in Rust.

Example

```
let mut x: u32 = 2;

let mut f = || -> u32 {
    x += 1;
    x
};

let y: u32 = f();
y + y
```

Execution

```
let y = f();
y + y
```

$\rightsquigarrow$

```
let y = 3;
y + y
```

$\rightsquigarrow$

```
3 + 3
```

Equational reasoning in Rust

Not always possible in Rust.

Example

```
let mut x: u32 = 2;

let mut f = || -> u32 {
    x += 1;
    x
};

let y: u32 = f();
y + y
```

Execution

```
let y = f();
y + y
⇝
let y = 3;
y + y
⇝
3 + 3
⇝
6
```

Equational reasoning in Rust

Not always possible in Rust.

Example (inlined)

```rust
let mut x: u32 = 2;

let mut f = || -> u32 {
    x += 1;
    x
};

// let y: u32 = f();
f() + f()
```

Execution (inlined)

```rust
f() + f()
```

Equational reasoning in Rust

Not always possible in Rust.

Example (inlined)

```rust
let mut x: u32 = 2;

let mut f = || -> u32 {
    x += 1;
    x
};

// let y: u32 = f();
f() + f()
```

Execution (inlined)

```
f() + f()
```

$\rightsquigarrow$

```
3 + f()
```

Equational reasoning in Rust

Not always possible in Rust.

Example (inlined)

```rust
let mut x: u32 = 2;

let mut f = || -> u32 {
    x += 1;
    x
};

// let y: u32 = f();
f() + f()
```

Execution (inlined)

```
f() + f()

⇝

3 + f()

⇝

3 + 4
```

Equational reasoning in Rust

Not always possible in Rust.

Example (inlined)

```rust
let mut x: u32 = 2;

let mut f = || -> u32 {
    x += 1;
    x
};

// let y: u32 = f();
f() + f()
```

Execution (inlined)

```
f() + f()
⇝
3 + f()
⇝
3 + 4
⇝
7
```

Equational reasoning in Rust

Equational reasoning fails when expressions depend on an implicit context.

Example 1 — read

```
fn f(x: &mut u32) -> u32 { *x + 1 }

let x: &mut u32 = ..;
f(x)
```

Example 2 — read-write

```
fn f(x: &mut u32) -> u32 { *x += 1; 0 }

let x: &mut u32 = ..;
f(x)
```


An expression that depends on an implicit context can evaluate to different results at different points in the program.

In example 1, the expression `f(x)`, applying the same function to the same reference, could return different values at different points in the program. It depends on an implicit mutable store.

An expression that *changes* an implicit context is not safe to inline, even if it returns the same result every time — see example 2. Inlining such expressions can change the meaning of the program, because the program's meaning depends on how many times the expression is evaluated.

Equational reasoning in Rust

Widespread equational reasoning is not a design goal for Rust.

Many standard library functions are non-pure / have side-effects, e.g.:

- Standard input / output
- Environment variables
- File system access
- Networking

Terminology

- An expression whose evaluation does not depend on an implicit context is described as *pure*.
- A function that doesn't depend on an implicit context is a *pure function*.
- Changes to an implicit context are called *side-effects*.

Mutable memory isn't the only implicit context in Rust programs. There is also the program input / output, environment variables, file system, networking, among others.

In Rust, the level of equational reasoning in a codebase depends on the programmers' judgement. When writing new code, the author decides whether or not equational reasoning is relevant. When refactoring, the programmer has to understand whether they are working with pure or non-pure functions.

Equational reasoning and mutability

Mutability doesn't always break equational reasoning.

A pure function that uses mutability

```
fn add(x: u32, y: u32) -> u32 {
    let mut counter = y;
    let mut result = x;

    while counter > 0 {
        counter -= 1;
        result += 1;
    }

    result
}
```


I've showed a bunch of examples where mutability, in particular mutable references, breaks equational reasoning. But mutability isn't *intrinsically* opposed to equational reasoning.

Here's an example of a pure function that uses mutability internally. Equational reasoning still holds.

Equational reasoning and mutability

Rust's *uniqueness types* offer another way to use mutation while preserving equational reasoning.

Example

Equational reasoning and mutability

Rust's *uniqueness types* offer another way to use mutation while preserving equational reasoning.

Example

```
fn push<T>(mut xs: Vec<T>, x: T) -> Vec<T> {
    xs.push(x);
    xs
}
```

Equational reasoning and mutability

Rust's *uniqueness types* offer another way to use mutation while preserving equational reasoning.

Example

```
fn push<T>(mut xs: Vec<T>, x: T) -> Vec<T> {
    xs.push(x);
    xs
}
fn length<T>(xs: &Vec<T>) -> usize {
    xs.len()
}
```

Equational reasoning and mutability

Rust's *uniqueness types* offer another way to use mutation while preserving equational reasoning.

Example

```
fn push<T>(mut xs: Vec<T>, x: T) -> Vec<T> {
    xs.push(x);
    xs
}
fn length<T>(xs: &Vec<T>) -> usize {
    xs.len()
}
let xs = vec![1, 2, 3];
let y = length(&xs);
let new_xs = push(xs, 4);
y + y
```

Equational reasoning and mutability

Rust's *uniqueness types* offer another way to use mutation while preserving equational reasoning.

Execution

```
let xs = vec![1, 2, 3];
let y = length(&xs);
let new_xs = push(xs, 4);
y + y
```

Equational reasoning and mutability

Rust's *uniqueness types* offer another way to use mutation while preserving equational reasoning.

Execution

```
let xs = vec![1, 2, 3];
let y = length(&xs);
let new_xs = push(xs, 4);
y + y
```

⇝

```
let xs = /* pointer `p` to [1, 2, 3] */;
let y = length(&xs);
let new_xs = push(xs, 4);
y + y
```

⇝

...

Equational reasoning and mutability

Rust's *uniqueness types* offer another way to use mutation while preserving equational reasoning.

Execution

...

$\rightsquigarrow$

```
let y = 3; /* length(&xs) */
let new_xs = push(xs, 4);
y + y
```

Equational reasoning and mutability

Rust's *uniqueness types* offer another way to use mutation while preserving equational reasoning.

Execution

...

⇝

```
let y = 3; /* length(&xs) */
let new_xs = push(xs, 4);
y + y
```

⇝

```
let new_xs = /* pointer `p` to [1, 2, 3, 4] */;
y + y
```

Equational reasoning and mutability

Rust's *uniqueness types* offer another way to use mutation while preserving equational reasoning.

Execution

...

⇝

```
let y = 3; /* length(&xs) */
let new_xs = push(xs, 4);
y + y
```

⇝

```
let new_xs = /* pointer `p` to [1, 2, 3, 4] */;
y + y
```

⇝

```
3 + 3
```

Equational reasoning and mutability

Rust's *uniqueness types* offer another way to use mutation while preserving equational reasoning.

Execution

...

⇝

```
let y = 3; /* length(&xs) */
let new_xs = push(xs, 4);
y + y
```

⇝

```
let new_xs = /* pointer `p` to [1, 2, 3, 4] */;
y + y
```

⇝

```
3 + 3
```

⇝

```
6
```

Equational reasoning and mutability

Rust's *uniqueness types* offer another way to use mutation while preserving equational reasoning.

Example

```
fn push<T>(mut xs: Vec<T>, x: T) -> Vec<T> { .. }

fn length<T>(xs: &Vec<T>) -> usize { .. }

let xs = vec![1, 2, 3]
// let y = length(&xs);
let new_xs = push(xs, 4);

length(&xs) + length(&xs)
```

Equational reasoning and mutability

Rust's *uniqueness types* offer another way to use mutation while preserving equational reasoning.

Example

```
fn push<T>(mut xs: Vec<T>, x: T) -> Vec<T> { .. }

fn length<T>(xs: &Vec<T>) -> usize { .. }

let xs = vec![1, 2, 3]
// let y = length(&xs);
let new_xs = push(xs, 4);
// error: borrow of moved value `xs`
length(&xs) + length(&xs)
//     ^^^ value borrowed here after move
```

Equational reasoning and mutability

A unique variable is only in scope while it's guaranteed to remain unchanged.

Example

```
fn push<T>(mut xs: Vec<T>, x: T) -> Vec<T> { .. }

fn length<T>(xs: &Vec<T>) -> usize { .. }

let xs = vec![1, 2, 3]
// `xs` is immutable...
let new_xs = push(xs, 4); // ...until we actually change it
.. // after which we can no longer refer to `xs`
```


I find this intriguing. push is a non-pure function that doesn't break equational reasoning. When you have a static type system, purity is sufficient, but not necessary for equational reasoning.

Closing remarks

Summary

How Rust relates to:

- Algebraic datatypes and higher-order functions
- Immutability
- Static types
- Equational reasoning

Functional programming?

- Focused on how we write and reason about programs

What is functional programming? I'm still going to refrain from a definition.

But I have a final observation to share. These four points I've discussed include specific tools we use to write programs (language features), styles of code (preference for immutability or mutability), and ways of reasoning about code.

I think this isn't a coincidence: it seems to me that functional programming is largely concerned with programs themselves, and the process of programming, as objects of study. It is less concerned with the specific behaviours of any particular program.

Another way to put it is that FP seems to focus on the relationship between programmers and programs, rather than the relationship between programs and the world.

I think this is super important and is also inherently risky. It means that the subject of FP, like other reflective / meta-level subjects, is especially prone to "navel-gazing".

Functional programming?

- Focused on how we write and reason about programs
  - Independent of specific program behaviours

If functional programming is less concerned with specific program behaviours, then it's a necessarily incomplete answer to the question of how to create good software. I don't think FP is going to tell me about the appropriate memory / CPU usage bounds for a program and how to achieve them. Or how to create intuitive user interfaces or render photorealistic 3D graphics or make sure a drone stays level when it flies.

I think that in the past my definition of "good software" was missing a few dimensions, which lead to me overestimating the important of "functional programming" in "good software". Now I know that FP isn't a panacea and it doesn't have to be. Whatever it is, it's now one of many tools of thought for pushing my software along a few of many dimensions of "good". And that's awesome, because learning's fun and there's even more to learn!

Functional programming?

- Focused on how we write and reason about programs
  - Independent of specific program behaviours
- There are many dimensions of software quality that functional programming can't address

Functional programming?

- Focused on how we write and reason about programs
  - Independent of specific program behaviours
- There are many dimensions of software quality that functional programming can’t address
  - That’s okay and even good!