Functional Programming: More than just a coding style

Matthew Watt
Software Engineer

Full disclosure

- Functional programming is amazing
- F# is amazing - you should learn and use it

...are you convinced?

Reverse psychology!

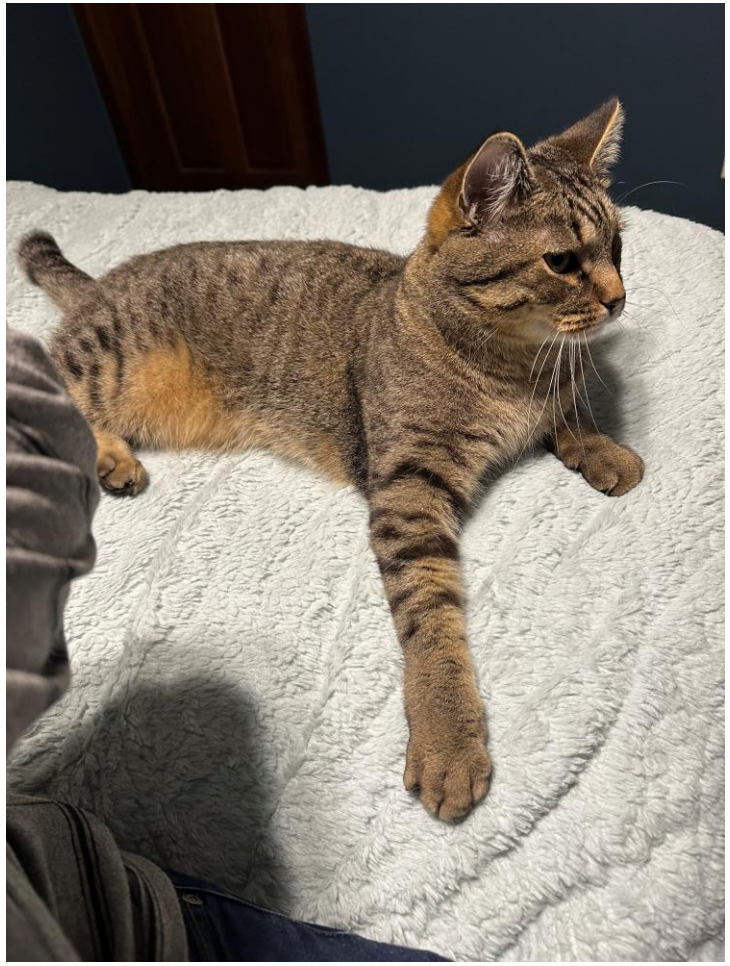
- Functional programming is awful and full of scary math and symbols. You should *definitely never learn it, it won't help you be a better programmer*
- F# is awful, it's offensive to musicians everywhere, it's built on Microsoft Java which is basically also awful and you should also never use it, you'd have more fun writing assembly

...now we're getting somewhere

About me

- 29 years old
- Missing fingers since birth
- Married
- (Technically) professional trombone player
- Novice pickleball player
- Enjoyer of:
  - Scotch
  - Bourbon
  - Beer
  - Cigars
- Cat owner

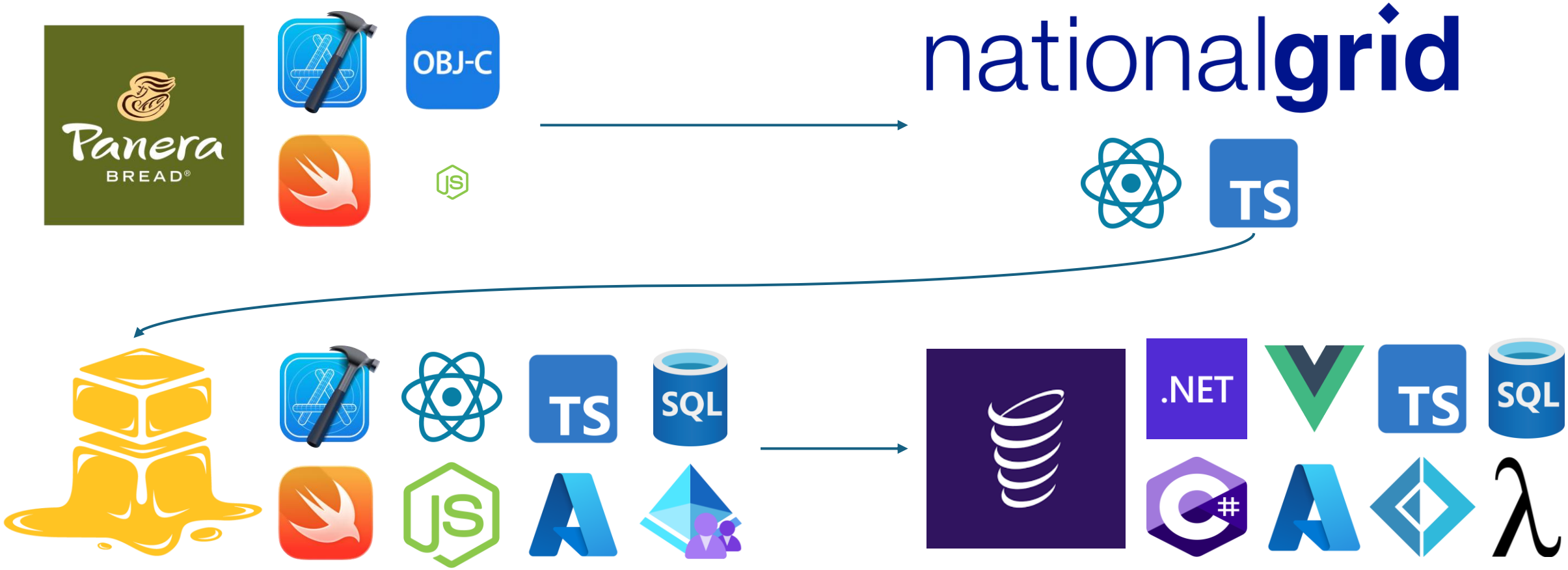
Obligatory cat pics



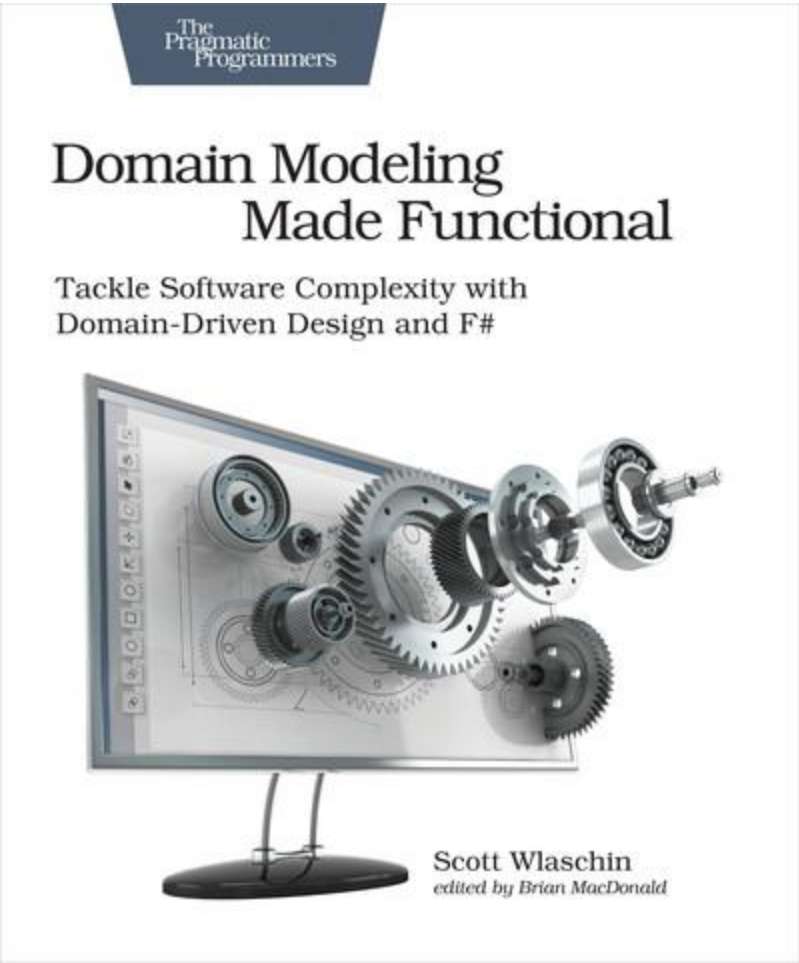
Obligatory cat pics

My journey

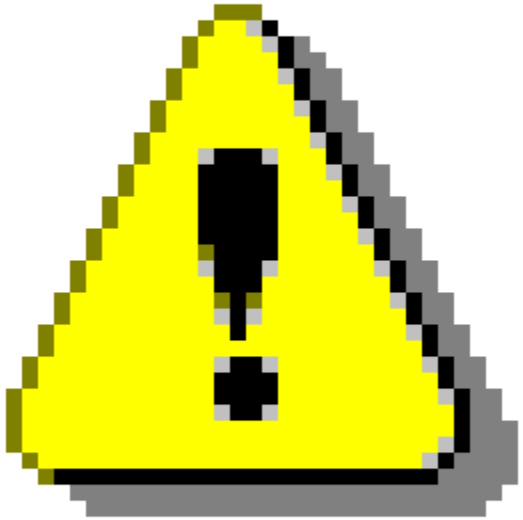
- 8 years in industry



My “aha!” moment



cool new branding! → Omnitech



Denominational

v

~~Cult~~ affiliation disclaimer

Two denominations

Dynamic typing

- Common Lisp
- Clojure
- Scheme
- Racket

Static typing

- Standard ML
- F#
- OCaml
- Haskell

Why choose one over the other?

Dynamic typing	Static typing
Flexibility	Correct by construction

A case study: the Galileo Jupiter Orbiter

“Also in 1993 I used MCL to help generate a code patch for the Gallileo magnetometer. The magnetometer had an RCA1802 processor, 2k each of RAM and ROM, and was programmed in Forth using a development system that ran on a long-since-decommissioned Apple II. The instrument had developed a bad memory byte right in the middle of the code. The code needed to be patched to not use this bad byte. The magnetometer team had originally estimated that resurrecting the development environment and generating the code patch would take so long that they were not even going to attempt it. Using Lisp I wrote from scratch a Forth development environment for the instrument (including a simulator for the hardware) and used it to generate the patch. The whole project took just under 3 months of part-time work.” - Lisping at JPL, Ron Garret

A case study: Deep Space I

“The Remote Agent software, running on a custom port of Harlequin Common Lisp, flew aboard Deep Space 1 (DS1), the first mission of NASA's New Millennium program. Remote Agent controlled DS1 for two days in May of 1999. During that time we were able to debug and fix a race condition that had not shown up during ground testing. (Debugging a program running on a $100M piece of hardware that is 100 million miles away is an interesting experience. Having a read-eval-print loop running on the spacecraft proved invaluable in finding and fixing the problem. The story of the Remote Agent bug is an interesting one in and of itself.)” – Lisping at JPL, Ron Garret

Why choose one over the other?

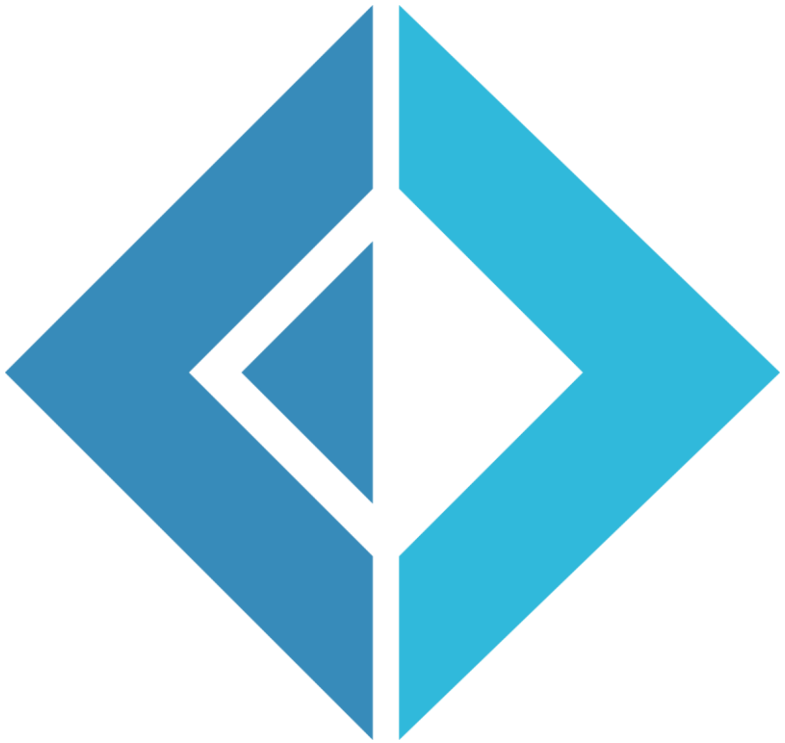
Dynamic typing	Static typing
Flexibility	Correct by construction
	“If it builds, it works”

A case study: Cardano blockchain

- Written in Haskell
- Used for:
  - Decentralized finance (DeFi)
  - Digital identity management
  - Supply chain management
  - Data storage
  - Voting systems
  - Healthcare

A case study:

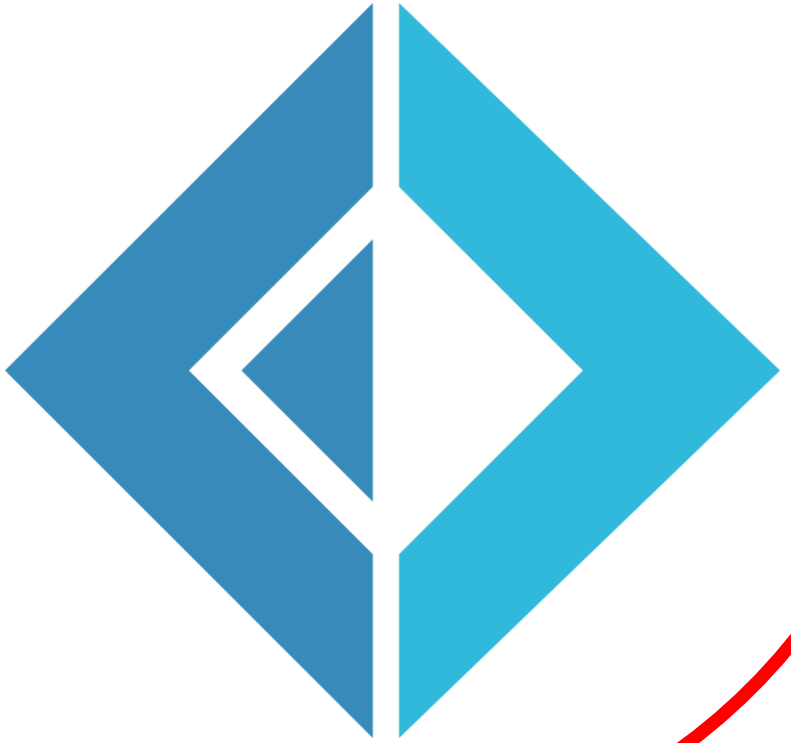
- My current side project!
  - Backend API
  - Web app
  - Mobile app



Two denominations

Dynamic typing

Static typing



Two denominations

Key Concept #1

Immutability is the foundational simplicity of functional programming

Key Concept #1

Immutability is the foundational simplicity of functional programming

Key Concept #1: Immutability is simplicity

Immutable = Unable to be changed without exception

Key Concept #1: Immutability is simplicity

A variable, once defined,
cannot change

Key Concept #1: Immutability is simplicity

value

A ~~variable~~, once defined, cannot change

Key Concept #1: Immutability is simplicity

A <u>structure</u>, once defined, cannot change

Key Concept #1: Immutability is simplicity

An _____, once defined, cannot change

Key Concept #1: Immutability is simplicity

A __list__, once defined, cannot change

Key Concept #1: Immutability is simplicity

A _____, once defined, cannot change

Key Concept #1: Immutability is simplicity

A __tree__, once defined, cannot change

Key Concept #1: Immutability is simplicity

A __graph__, once defined, cannot change

Key Concept #1: Immutability is simplicity

A <u>structure</u>, once defined, cannot change

Key Concept #1: Immutability is simplicity

Value-oriented programming

Key Concept #1: Immutability is simplicity

Immutability is the foundational simplicity of functional programming

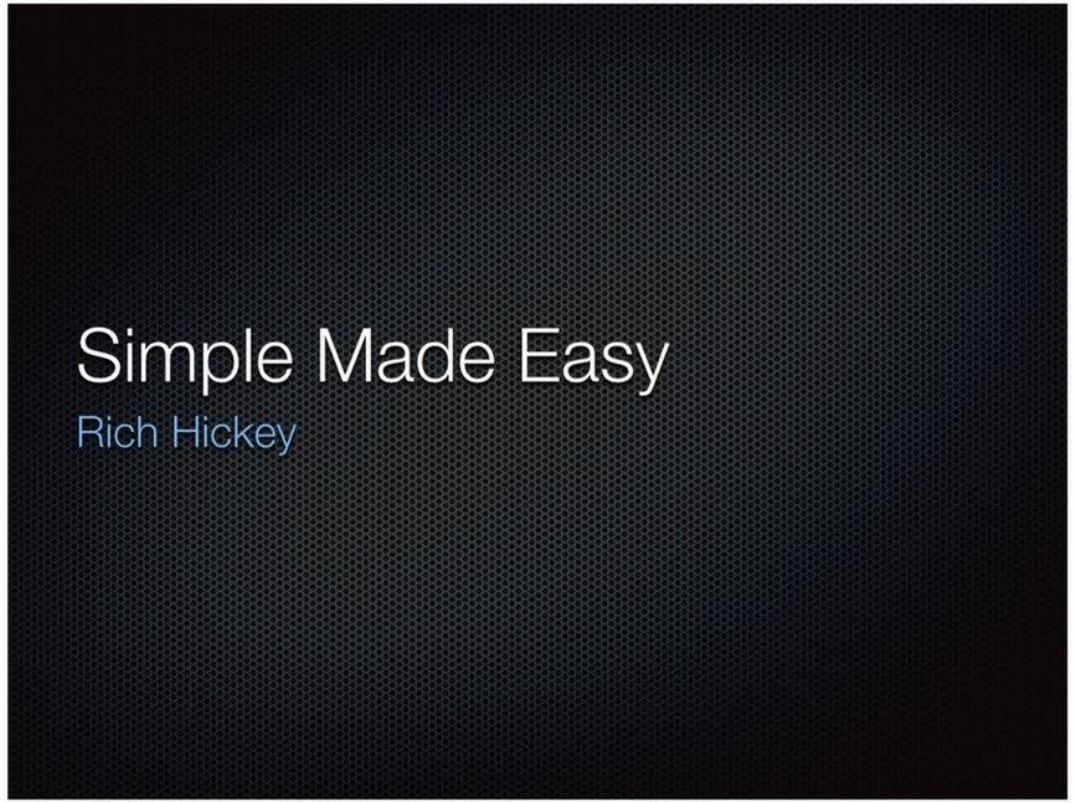
Key Concept #1: Immutability is simplicity

Immutability is the foundational <u>simplicity</u> of functional programming

Key Concept #1: Immutability is simplicity

strangeloop

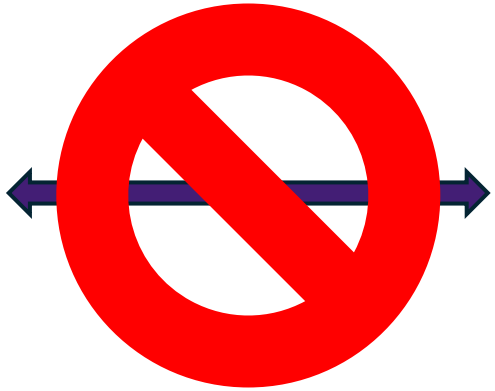
Strange Loop
Sept 19-20, 2011
https://thestrangeloop.com



https://www.youtube.com/watch?v=SxdOUGdseq4

Key Concept #1: Immutability is simplicity

Simple



Easy

Key Concept #1: Immutability is simplicity

Simple = one fold/braid

= not interleaved

Key Concept #1: Immutability is simplicity

Complex = braided/interleaved

Key Concept #1: Immutability is simplicity

Simple and complex are objective notions

Key Concept #1: Immutability is simplicity

Easy = near, at hand

Key Concept #1: Immutability is simplicity

Easy and hard are subjective notions

Key Concept #1: Immutability is simplicity

Easy = near, at hand

Key Concept #1: Immutability is simplicity

"If you want everything to be familiar [easy], you will never learn anything new, because it can't be significantly different from what you already know and not drift away from the familiarity" – Rich Hickey

Key Concept #1

Immutability is the foundational simplicity of functional programming

Key Concept #1

Mutability is complex?

Yes.

Key Concept #1: Immutability is simplicity

A value, once defined, cannot change

Key Concept #1: Immutability is simplicity

What is a value that can change?

A variable

Key Concept #1: Immutability is simplicity

What is a *value* that can change *over time?*

A variable

Value and time are interleaved

Key Concept #1: Immutability is simplicity

What is another name for a value that changes over time?

State

Key Concept #1: Immutability is simplicity

State is complex *by definition*

Invalid states?

Data races?

Asynchrony?

Concurrency?

Threading?

Key Concept #1: Immutability is simplicity

Immutability is the foundational simplicity of functional programming

Key Concept #1: Immutability is simplicity

How do we write programs that process data without mutating it?

Key Concept #1: Immutability is simplicity

```
string -> int
```

```
string -> Uri option
```

Functions!

```
int -> string
```

```
int -> int -> int
```

```
PaymentCard -> CardLast4
```

```
Registration -> Result<Success, RegistrationError>
```

Key Concept #2

Avoid implicit behavior

Key Concept #2: Avoid implicit behavior

```
string -> int
```

```
string -> Uri option
```

Functions!

```
int -> string
```

```
int -> int -> int
```

```
PaymentCard -> CardLast4
```

```
Registration -> Result<Success, RegistrationError>
```

Key Concept #2: Avoid implicit behavior

Implicit behavior		Made explicit
Object mappers	→	Explicit mapping
IoC Containers	→	Direct injection

Key Concept #2: Avoid implicit behavior

```
string -> int
```

```
string -> Uri option
```

Functions!

```
int -> string
```

```
int -> int -> int
```

```
PaymentCard -> CardLast4
```

```
Registration -> Result<Success, RegistrationError>
```

Key Concept #2: Avoid implicit behavior

Functions should be *total*

Every input produces a valid output

Key Concept #2: Avoid implicit behavior

```
parseInt

string -> int
```

Key Concept #2: Avoid implicit behavior

parseInt

st PARTIAL nt

Key Concept #2: Avoid implicit behavior

What if this can’t be converted to an int?

parseInt

string -> int

Key Concept #2: Avoid implicit behavior

Constrain input

IntegralString -> int

EmailGuaranteedToBeInDatabase -> Account

Extend output

string -> int option

EmailAddress -> Result<Account option, Error>

Key Concept #3

Model your domain with composable types

Key Concept #3: Use composable types

Composable types = non-scary, non-mathy way of saying “algebraic data types”

Key Concept #3: Use composable types

Only two kinds

You already use them

Key Concept #3: Use composable types

AND types

Classes, structs, records, tuples

Aggregates

CreditCard = Name AND Last4 AND ExpirationDate

OR types

enums, unions

Choices

Brand = Visa OR Mast OR Disc OR Amex

Key Concept #3: Use composable types

They are OR types!

```
type Option<T> =
  | None
  | Some of T
```

```
type Result<Ok, Error> =
  | Ok of Ok
  | Error of Error
```

Key Concept #4

Make invalid states unrepresentable

Key Concept #4: No invalid states

Probably shouldn’t be empty

```
type PaymentCardInfo =
  { CardholderName : string
    CardBrand : string
    CardLast4 : string
    CardExpiration : string }
```

Key Concept #4: No invalid states

```
type PaymentCardInfo =
  { CardholderName : NonEmptyString ✓
    CardBrand : string ←———— Probably should be a fixed set of choices
    CardLast4 : string
    CardExpiration : string }
```

Key Concept #4: No invalid states

```
type CardBrand =
  | Visa
  | MasterCard
  | Discover
  | AmericanExpress
```

```
type PaymentCardInfo =
  { CardholderName : NonEmptyString
    CardBrand : CardBrand
    CardLast4 : string
    CardExpiration : string }
```

Probably should only allow 0000-9999

Key Concept #4: No invalid states

```
type CardBrand =
  | Visa
  | MasterCard
  | Discover
  | AmericanExpress
```

```
type PaymentCardInfo =
  { CardholderName : NonEmptyString
    CardBrand : CardBrand
    CardLast4 : CardLast4
    CardExpiration : string }
```

Probably should only allow "MM/YY"

Key Concept #4: No invalid states

```
type CardBrand =
  | Visa
  | MasterCard
  | Discover
  | AmericanExpress
```

```
type PaymentCardInfo =
  { CardholderName : NonEmptyString ✓
    CardBrand : CardBrand ✓
    CardLast4 : CardLast4 ✓
    CardExpiration : CardExpiration } ✓
```

```
type CardExpiration = private CardExpiration of string

module CardExpiration =
  let create str = validation { ... }
```

Key Concept #4: No invalid states

```
type PaymentCardInfo =
  { CardholderName : string
    CardBrand : string
    CardLast4 : string
    CardExpiration : string }
```

How many possible values?

Key Concept #4: No invalid states

```
type PaymentCardInfo =
  { CardholderName : string
    CardBrand : string
    CardLast4 : string
    CardExpiration : string }
```

How many possible values?

How many possible values?

Key Concept #4: No invalid states

```
type PaymentCardInfo =
  { CardholderName : string
    CardBrand : string
    CardLast4 : string
    CardExpiration : string }
```

How many possible values?

How many possible values?

Frighteningly many

Key Concept #4: No invalid states

How many of those possible values are *valid*?

```
type PaymentCardInfo =
  { CardholderName : string
    CardBrand : string
    CardLast4 : string
    CardExpiration : string }
```

How many possible values?

How many possible values?

# of PaymentCardInfo values =
*Frighteningly many * Frighteningly many * Frighteningly many * Frighteningly many*

Key Concept #4: No invalid states

```
type PaymentCardInfo =
  { CardholderName : NonEmptyString
    CardBrand : CardBrand
    CardLast4 : CardLast4
    CardExpiration : CardExpiration }
```

How many possible values?

Key Concept #4: No invalid states

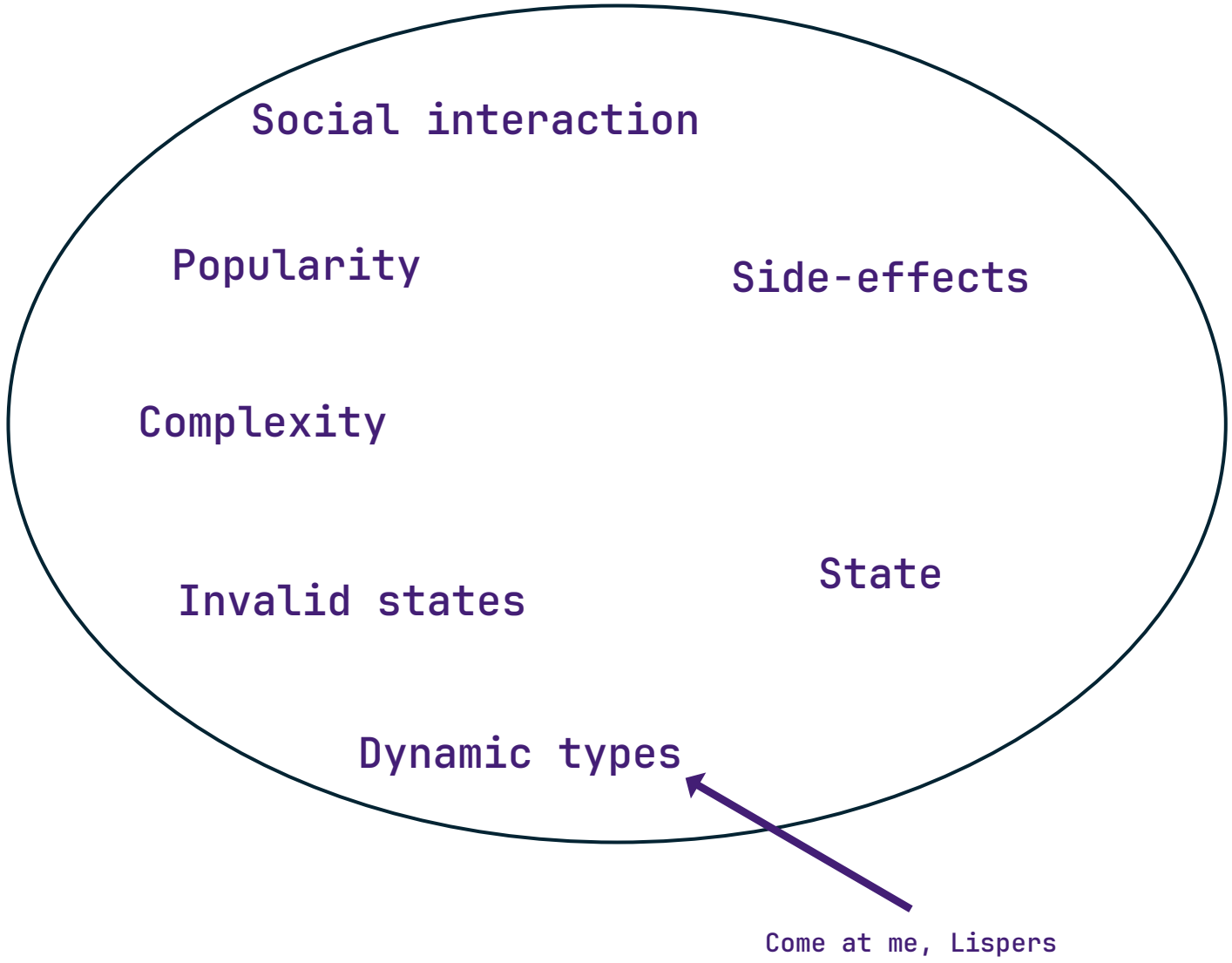
```
type PaymentCardInfo =
  { CardholderName : NonEmptyString
    CardBrand : CardBrand
    CardLast4 : CardLast4
    CardExpiration : CardExpiration }
```

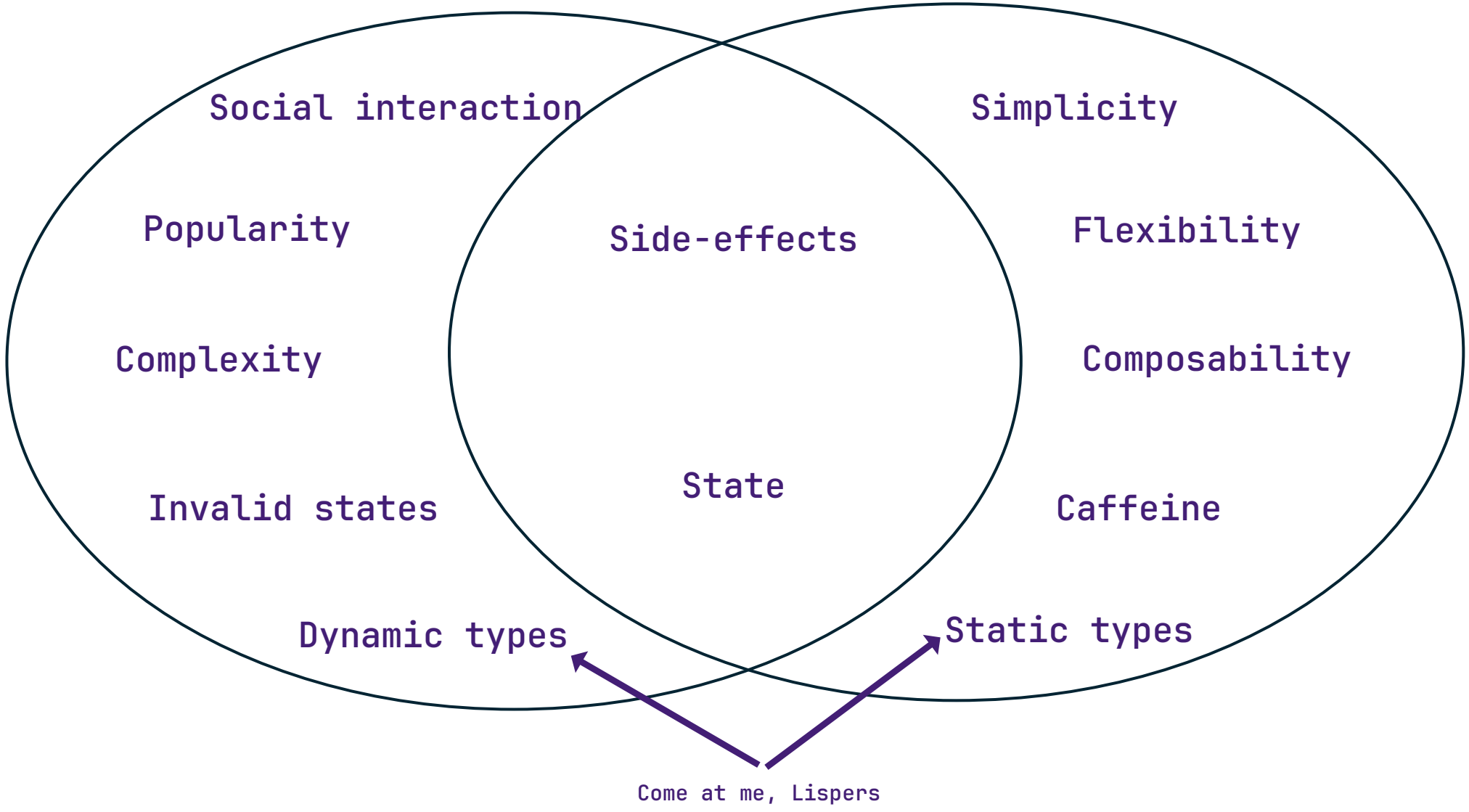
How many possible values?

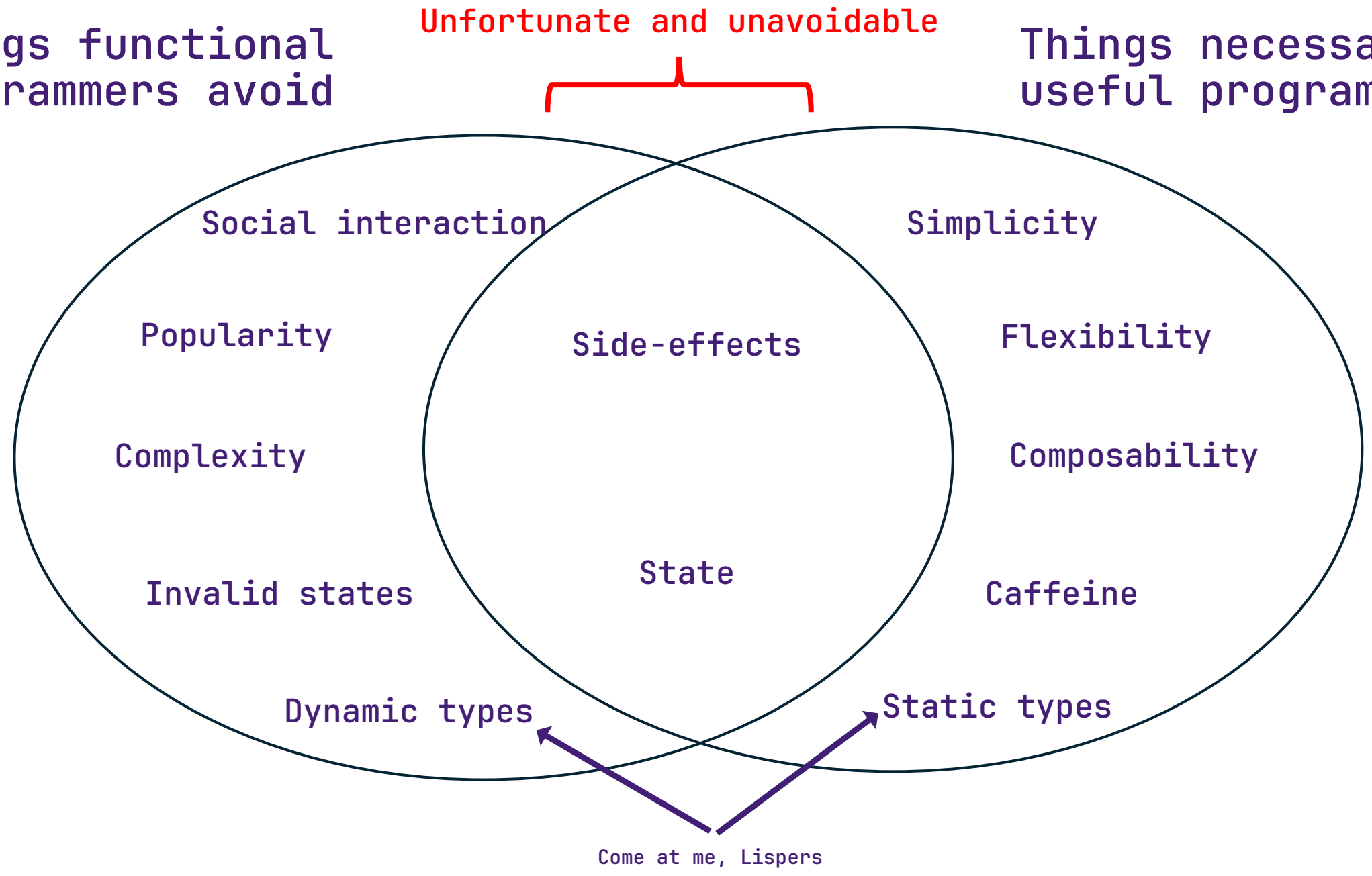
Still a lot. But *orders of magnitude fewer.*

[Slide intentionally blank. Take a breather, you’ve earned it]

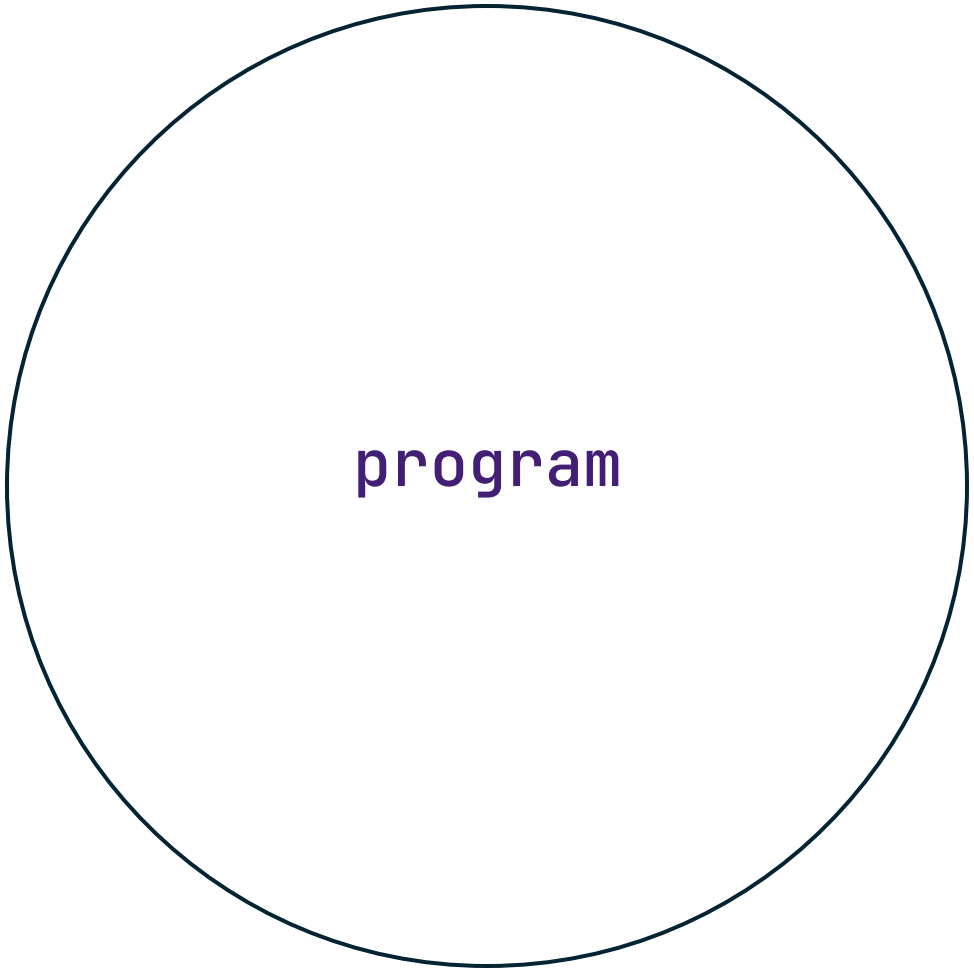
Things functional programmers avoid



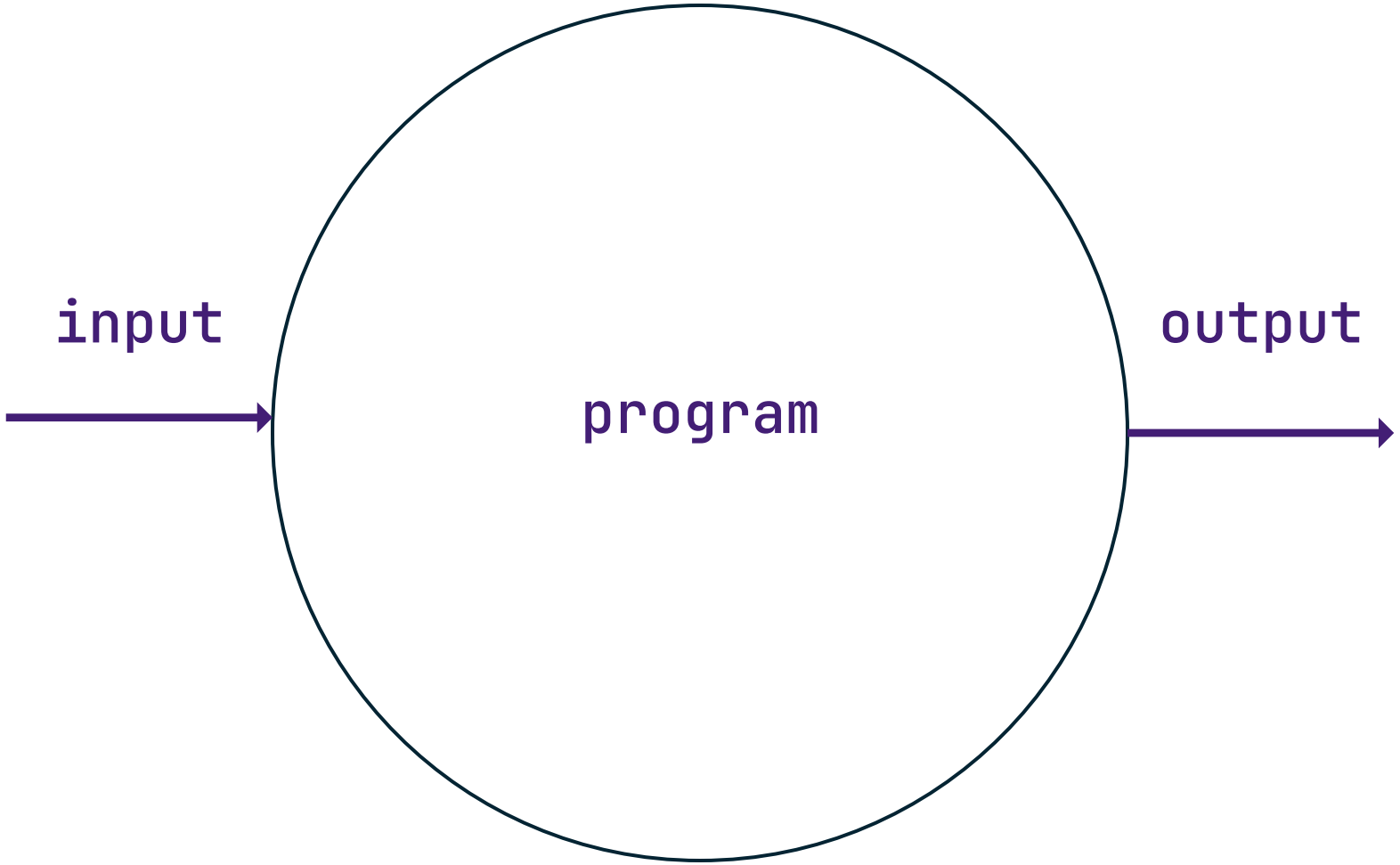

Things functional programmers avoid
Things necessary for useful programs
Social interaction
Popularity
Complexity
Invalid states
Dynamic types
Side-effects
State
Simplicity
Flexibility
Composability
Caffeine
Static types
Come at me, Lispers


Things functional programmers avoid
Unfortunate and unavoidable
Things necessary for useful programs
Social interaction
Popularity
Complexity
Invalid states
Dynamic types
Side-effects
State
Simplicity
Flexibility
Composability
Caffeine
Static types
Come at me, Lispers

All programs are impure...



All programs are impure...at the boundary



Key Concept #5

Functional core, imperative shell

Key Concept #5: Keep I/O at the edge

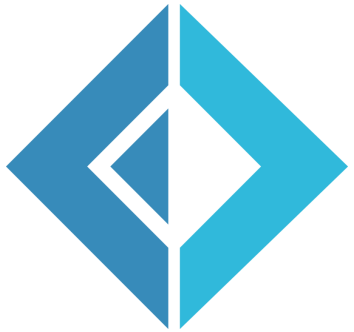
Use case (workflow, endpoint, process, feature)

1. Fetch what you need to make a decision
2. Make the decision
3. Act based on the decision

Demo time

Questions?

matt@twopoint.dev



twopoint

https://twopoint.dev/posts/why-the-fsharp-would-i-write-real-code-like-this