
The Fearless Journey

Fearless: A minimalistic nominally typed pure OO language where there are no fields and all the state is captured by closures.

The Fearless Heart: how to encode booleans, optionals, and lists.

Braving mutability: how to add mutability to such a language without losing the reasoning advantages of functional programming.

The Treasure: support for automatic parallelisation, correct cache invalidation and representation invariants.

The Fearless Heart

First, we will sail comforting friendly waters, exploring the functional core of Fearless

Core Syntax: no inference, no sugar

```
L  ::= D[Xs] : Ts { 'x  Ms }

M  ::= sig, | sig -> e,

e  ::= x | e.m[Ts](es) | L

sig::= m[Xs](x1:T1, ..., xn:Tn):T

T  ::= X | D[Ts]
```

Overall, Fearless can be seen as 'lambda calculus' where lambdas have multiple components and we have a table of top-level declarations.

```
Core Syntax: no inference, no sugar
L  ::= D[Xs] : Ts { 'x  Ms }
M  ::= sig, | sig -> e,
e  ::= x | e m[Ts](es) | L
sig::= m[Xs](x1:T1, ..., xn:Tn):T
T  ::= X | D[Ts]
```

Syntactic sugar:
- Can omit empty parenthesis {} [] or ()
- Can omit () on 1 argument method (becomes left associative operator)
- Top level Declarations ‘x is ‘this’. Can be omitted for other declarations and is fresh

Inference:
- The trait declaration name D of a literal inside of an expression, together with the implemented types Ts is most often inferred.
- An overridden sig can omit the types
- Can omit even the method name if there is only 1 abstract method

Concrete syntax:
- Declaration overloading based on generics arity,
- Method overloading on parameter arity, method names both as .lowercase or operators

```
Person: { .age: Num, .name: Str }  //a Person with age and name
```

Not a record with two fields, but a trait with two methods taking zero arguments.
Not behave like fields: trigger (potentially non terminating) computations.
No guarantee that any storage space is used by those methods.

Example: .name captures a string, .age is the length of the same string.

```
Person{ .age -> 42, .name -> "Bob" }    //making a person directly

FPerson:{.of(age: Num, name: Str): Person ->{.age->age, .name→name}}
FPerson.of(42, "Bob")    //making a person with a factory

FPerson == Fperson[]{}      42 == 42[]{}   "Bob" ==   "Bob"[]{}
```

```
Person: { .age: Num, .name: Str }  //a Person with age and name
```

Functions can just be generic top level declarations

```
F[R]:{ #: R }          //Note: # is a valid method name just like .of
F[A, R]:{ #(a: A): R }
F[A, B, R]:{ #(a: A, b: B): R }
F[A, B, C, R]:{ #(a: A, b: B, c: C): R }
```

Now we FPerson can be a kind of function

```
FPerson:F[Num, Str, Person]{ age, name -> {.age->age, .name->name} }

FPerson#(42, "Bob")    //making a person with a function/factory
```

```
Bool: {
   .and(other: Bool): Bool,
   .or(other: Bool): Bool,
   .not: Bool,
   .if[R](m: ThenElse[R]): R
   }
ThenElse[R]:{ .then: R, .else: R }

True: Bool{
   .and(other) -> other,
   .or(other) -> this,
   .not -> False,
   .if(m) -> m.then,
   }
False:Bool{
   .and(other) -> this,
   .or(other) -> other,
   .not -> True,
   .if(m) -> m.else,
   }
```

```
//usage example
True.and(False).if({
   .then->/*code for the then case*/,
   .else->/*code for the else case*/,
   })

True.and False.if{
   .then->/*code for the then case*/,
   .else->/*code for the else case*/,
   }
```

```
Opt[T]: {
   .match[R](m: OptMatch[T,R]): R -> m.empty
   }
OptMatch[T,R]: {
   .empty: R,
   .some(t: T): R
   }
Opt: {
   #[T](t:T):Opt[T] -> { m -> m.some(t) }
   }
```

```
//usage example
Opt#bob       //Bob is here
Opt[Person]   //no one is here
```

```
List[T]: {
   .match[R](m: ListMatch[T,R]): R -> m.empty
  +(e: T): List[T] -> { m -> m.elem(this, e) },
}
ListMatch[T,R]: {
   .empty: R,
   .elem(list: List[T], e: T): R
}

//usage examples
List[Num]+1+2+3
List[Opt[Num]]+{}+{}+(Opt#3)
List[List[Num]]+{}+{}+(List[Num]+3)

Example: {
   .sum(ns: List[Num]): Num -> ns.match{
     .empty -> 0,
     .elem(list, e) -> this.sum(list) + e
   }
}
```

```
List[T]: {
   .match[R](m: ListMatch[T,R]): R -> m.empty
   +(e: T): List[T] -> { m -> m.elem(this, e) },
    .map[R](f: F[T, R]): List[R] -> this.match{
      .empty -> {},
      .elem(list, e) -> list.map(f) + (f#e)
     }
}
ListMatch[T,R]: {
   .empty: R,
   .elem(list: List[T], e: T): R
}
```

```
Html:{ .match[R](m: HtmlMatch[R]): R }
HtmlMatch[R]:{
  .h1(text: Str): R,
  .h5(text: Str): R,
  .a(link: Str, text: Str): R,
  .div(es: List[Html]): R,
}
FHtml:{
  .h1(text:Str):Html->{m->m.h1 text}
  .h5(text:Str):Html->{m->m.h5 text}
  .a(link:Str, text:Str):Html->{m->m.a(link, text)}
  .div(es:List[Html]):Html->{m->m.div es}
}
HtmlCloneVisitor:{
  .h1(text)->FHtml.h1 text,
  .h5(text)->FHtml.h5 text,
  .a(link,text)->FHtml.a(link, text),
  .div(es)->FHtml.div(es.map{ e->e.match this }),
}
CapitalizeTitles:HtmlCloneVisitor{ .h1(text) -> Fhtml.h1(text.upperCase) }
...
myHtml.match(CapitalizeTitles)//usage
myHtml.match{ .h1(text) -> FHtml.h1(text.upperCase)}//direct definition
```

```
Let:{ #[T,R](x: T, f: F[T,R]): R -> f#x }

Let#(12+foo, {x -> x*3})  //usage


//Alternative option
Let: { #[T](x: T): In[T] -> {f -> f#x } }
In[T]: { .in(f: F[T,R]): R }

Let#(12+foo) .in {x->x*3} //usage
```


Braving Mutability:
Fearless's Journey
into mutability

Mutable state obtained by inserting a magic Ref[T] implementation
Mutable state controlled with reference capabilities

```
Ref[T]: {
  read .get: read T, //two .get in overloading
  mut .get: mdf T,
  mut .swap(x: mdf T): mdf T,

  }

Ref: { #[T](x: mdf T): mut Ref[mdf T] -> Magic! }
```

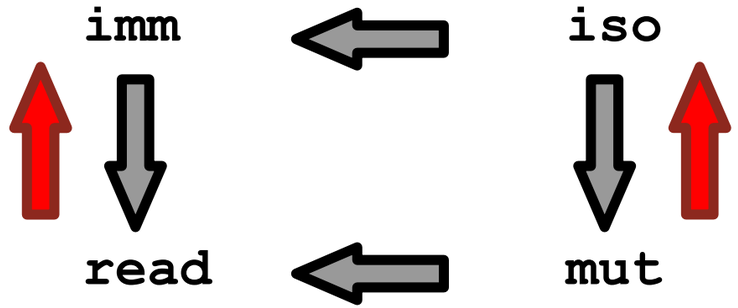
Mutable state obtained by inserting a magic Ref[T] implementation
Mutable state controlled with reference capabilities

```
Ref[T]: {
  read .get: read T, //two .get in overloading
  mut .get: mdf T,
  mut .swap(x: mdf T): mdf T,
  mut .set(x: mdf T): Void -> Do#(this.swap(x), Void),
  }

Ref: { #[T](x: mdf T): mut Ref[mdf T] -> Magic! }

Void: {}

Do: {
  #[A,B](x: mdf A, res: mdf B): mdf B -> res,
  #[A,B,C](x: mdf A,y: mdf B, res: mdf C):mdf C -> res,
  #[A,B,C,D](x: mdf A, y: mdf B, z: mdf C, res: mdf D): mdf D -> res,
  }
```

Reference modifiers and grammar

```
R ::= imm  | iso | read |  mut
```

figure:




```
L   ::= R D[Xs] : Ts { 'x Ms }
M   ::= sig, | sig -> e,
e   ::= x | e.m[Ts](es) | L
sig ::= R m[Xs](x1:T1, ..., xn:Tn):T
T   ::= R D[Ts] | R X | mdf X
R   ::= imm | iso | read |  mut
```

```
.ex01(rt: Ref[T], t: T): Void -> rt.set(t), //type error, rt is immutable

.ex02(rt: read Ref[T], t: T): Void -> rt.set(t), //type error, rt is readable

.ex03(rt: mut Ref[T], t: T):Void -> rt.set(t) //ok

.ex04(rt: mut Ref[T]): Ref[T] -> rt //type error, mut is not a subtype of imm

.ex05(rt: mut Ref[T]): read Ref[T] -> rt //ok, read is a subtype of all R

.ex06(t: T): mut Ref[T] -> Ref#t //ok, the factory returns a mut Ref[T]

.ex07(t: T): iso Ref[T] -> Ref#t //ok, iso promotion: the method takes no mut/read
                    // in input and returns an mut; this mut can be promoted to imm

.ex08(t: T):  Ref[T] -> Ref#t //ok, iso promotion + iso-> imm subtyping

.ex09(rt: Ref[T]): read T-> rt.get, //ok using the read .get

.ex10(rt: mut Ref[T]): T-> rt.get, //ok using the mut .get

.ex11(rt: Ref[T]): T-> rt.get, //ok using the read .get and imm promotion:
      //the call rt.get returns a read object, but only takes in input imm objects,
      //thus the result must be imm (or promotable to imm via iso promotion)
.ex12(rt: read Ref[T]): T -> rt.get //Oh NO!
```

Extended subtyping: The adapt rule

```
.ad01(rt: Ref[mut T]): Ref[T] -> rt,

Assume a rich List type, with get(i) and set(i), assume TallPerson<Person

.ad02(l: List[TallPerson]): List[Person] -> l,

.ad03(l: List[mut TallPerson]): List[Person] -> l,

The adapt rule


  R D[T1..Tn]<= R D[T1'..Tn']
    If all the method callable on a (promoted) R D[T1'..Tn']
    could be identically called on a (similarly promoted) R D[T1..Tn].

Consider the iconic List[T].concat(List[T]):List[T] method:
  Does it satisfy the 'adapt rule'?
A simple minded implementation/metrule
would attempt to generate an infinite proof.
```

There are two non obvious relation between fields and RC that Fearless avoids by not having explicit fields:

- If a mut reference always refers to a mut object,
  should a mut field always refer to a mut object?

Confusingly, the answer is no. Fields are (usually) declared in classes while instances of classes can be both mut and imm objects. The whole ROG of an imm object is imm, thus even the field originally declared as mut in the class, will hold an imm object when the receiver is imm.(relations between mut fields and poly)

- An iso reference is affine.
  Would an iso field be affine too?

Affine fields are not used in RC literature, instead iso fields (if allowed at all) have some different behaviour, often destructive reads plus complex language supported patterns where the field is consumed and then repristinated.

Capturing strategy for lambdas:
Assume we have a R0 Foo[]: Ts{'x M1..Mn } being typed in a gamma G
The method Mi with modifier R can capture variables using G′=(G,x:R0 Foo[])[R0,R]

```
#define G[R0,R] = G'  // where R0=receiver, R=method
  (x: T, G)[R0,R1]   = x: T[imm]  G[R0,R1]  with T = iso _ or imm _
  (x: T, G)[R0,R1]   = x: T       G[R0,R1]  with R0 and R1 in iso, mut
  (x: T, G)[R0,imm]  = x: T[imm]  G[R0,R1]  with R0 in iso,mut,read
  (x: T, G)[R0,read] = x: T[read] G[R0,R1]  with R0 in iso,mut,read
  (x: T, G)[R0,R1]   =            G[R0,R1]  otherwise
```

Where
R D[Ts][R0]=R0 D[Ts]
R X[R0]   =R0 X
mdf X[R0] =R0 X

M1..Mn must be all the abstract methods of Ts that are applicable to R0:
If the object literal is born imm or read,
mut and iso methods could never be called
→ it is an error to override a mut/iso method in an imm/read literal

Object Capabilities

Object Capabilities (OC) are not a type system feature, but a programming methodology that is greatly beneficial when it is embraced by the standard library.

The main idea is that instead of being able to make non deterministic actions like IO everywhere in the code by using static methods or public constructors, only certain specific objects have the 'capability' of doing those privileged actions, and access to those objects is kept under strict control.

In Fearless, this is done by having the main taking in input a mut System object. System is a normal trait with all methods abstract. An instance of System with magically implemented methods (like Ref[T]) is provided to the user as a parameter to the main method at the beginning of the execution.

Finally, Hello World

```
HelloW: Main{ s -> s.println("Hello World") }
```

Where Main is a trait defined as follows:

```
Main{ .main(s:mut System):Void }
```

Java execution starts from any class with a main method,
Fearless execution starts from any class transitively implementing Main.
This allows for abstraction over the Main. A unit test could look like this:

```
MyTests:UnitTest{ logger->... }
```

Where UnitTest inherits from Main and implements the main method,
forges a logger and so on, leaving a single method abstract that is called from the implemented .main.

Crucially, all non deterministic methods will be 'mut' methods.
If a capability is saved or passed around as read, it would be harmless.

Better syntax for local variables

The '= sugar' allows for local variable to be integrated in fluent APIs.
In code using fluent APIs we often have an initial receiver and then a bunch of method calls, for example in Java Streams we could have

```
myList.stream().map(x->x*2).filter(y->y>3).toList()
```

Here myList is the initial receiver of the shown sequence of method calls.
In our proposed sugar, a method call of form

```
e.m(e1,{x,self->self ... })
```

where ... is a sequence of method calls using self as the initial receiver can be written using the more compact syntax

```
e.m x=e1 . ...
```

Better syntax for local variables

We have a Block[T] trait supporting a fluent statements API.
We can obtain a Block[T] by calling Do# without parameters.
Example:

```
MyApp:Main{s->Do#
  .var[mut Fs] fs = {FIO#s}
  .var content =  {fs.read("data.txt")}
  .if {content.size > 5} .return {s.println("Big")}
  .return{s.println("Small")}
  }
```

Removing this layer of sugar the code would look as follow:

```
MyApp:Main{s->Do#
  .var[mut Fs]({FIO#s}, {fs,self1->self1
    .var({fs.read("data.txt")}, {content,self2->self2
      .if {content.size > 5} .return {s.println("Big")}
      .return{s.println("Small")}
     }))})
  }
```


safe passage


RC+OC = determinism

RC+OC = determinism

Note, with Ref[T], we need to distinguish identical by identity and structurally identical

(1)Any method taking in input only imm parameters is deterministic.
    Pass structurally identical parameters and get a structurally identical result
    Pass identity identical parameters and get a structurally identical result

(2)Any method taking in input only iso/read parameters is deterministic
 up to external mutation of its read parameter.
    Pass structurally identical parameters and get a structurally identical result

This form of determinism is weaker that functional purity:
    Non termination can happen (deterministically)
    Exceptions can happen (deterministically and not)
Capturing exceptions without breaking this determinism property is possible
but outside of the scope of this presentation


The Treasure:
Invariants, caching and automatic parallelism

Invariants and caching

```
_FRange:F[Nat,Nat,_Range]{min, max ->Do#
  .ref _min = {min}
  .ref _max = {max}
  .return{_Range{
      read .min:Nat->_min.get,
      read .max:Nat->_max.get,
      mut  .min(n:Nat):Void->_min.set(n),
      mut  .max(n:Nat):Void->_max.set(n),
    }}
  }

FRange:F[Nat,Nat,Range]{(min,max->Do#
  .var _range = {Repr#(_FRange#(min,max))}
  .do {_range.invariant {r-> r.min < r.max }}
  .var toS    = _range.cached{r-> toString}
  .return {Range{
    read .min:Nat->_range.look{r->r.min},
    read .max:Nat->_range.look{r->r.max},
    mut  .min(n:Nat):Void->_range.mutate{r->r.min(n)},
    mut  .max(n:Nat):Void->_range.mutate{r->r.max(n)},
    read .toS:Str->toS#//cached
    }}
  }
```

Invariants and caching

```
Repr{#[T](t:iso T):mut Repr[T]->/*omitted*/}
Repr[T]:{
  read .look[R](f:read RF[read T,R]):R,
  read .mutate[R](f:read RF[mut T,R]):R,
  mut .invariant(f:F[read T,Bool]):Void,
  mut .cached[R](f:F[read T,R]):mut CachedProperty[R],
  }
RF[A,R]:{ read #(a:mdf A):R }
CachedProperty[R]:{ mut #:R }


FRange:F[Nat,Nat,Range]{(min,max->Do#
  .var _range = {Repr#(_FRange#(min,max))}
  .do {_range.invariant {r-> r.min < r.max }}
  .var toS    = _range.cached{r-> toString}
  .return {Range{
    read .min:Nat->_range.look{r->r.min},
    read .max:Nat->_range.look{r->r.max},
    mut  .min(n:Nat):Void->_range.mutate{r->r.min(n)},
    mut  .max(n:Nat):Void->_range.mutate{r->r.max(n)},
    read .toS:Str->toS#//cached
    }}
  }
```

Automatic parallelism

Example of flow based code:

```
.foo(as:List[A]):List[C]->
  as.flow
    .map{a->a.bar}
    .filter{b->b.acceptable}
    .map{b->b.cab}
    .list
```

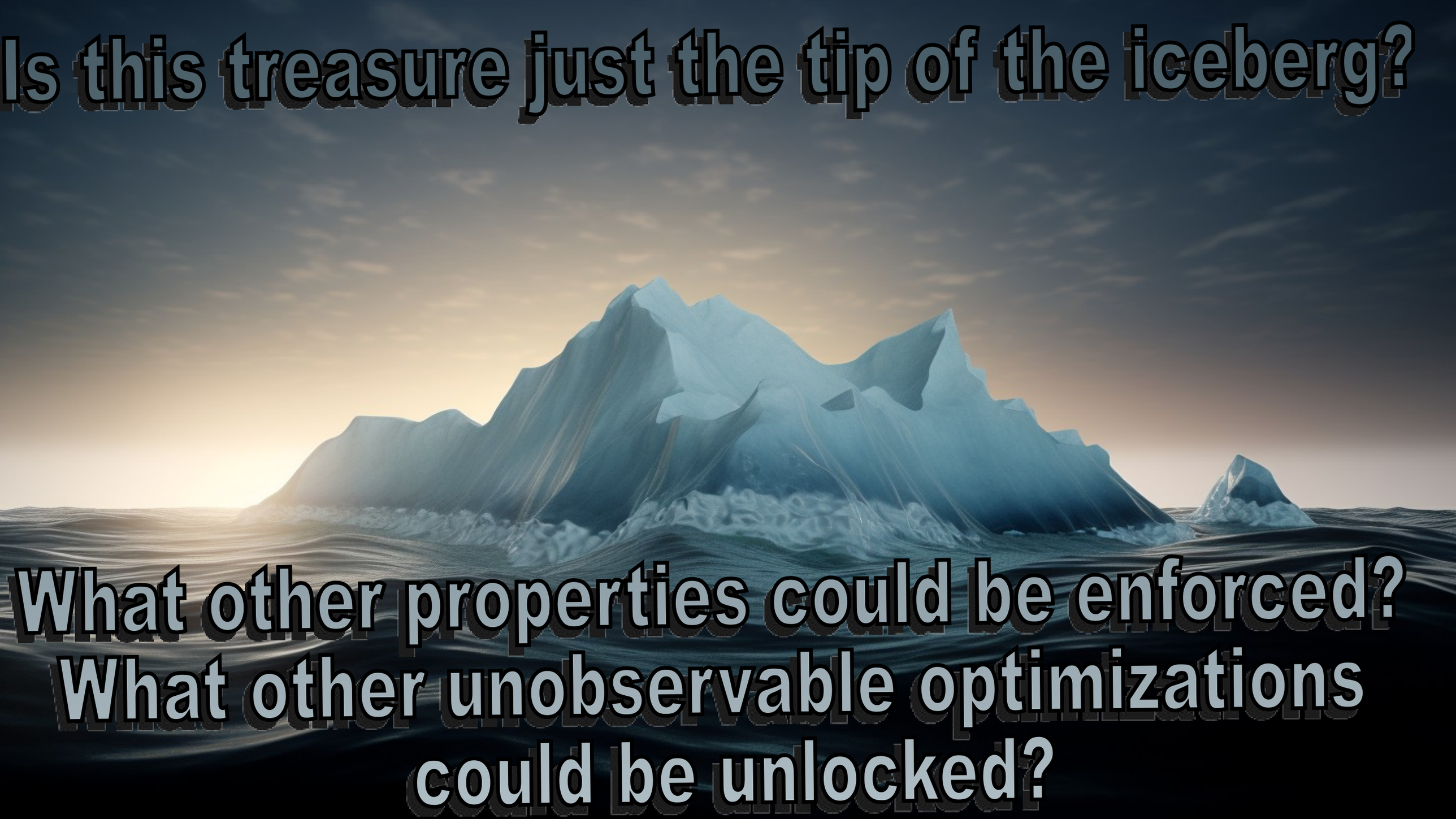
The API of Flow[T] shows what can and can not be done by those operations:

```
Flow[T]:{
  mut .map(f:F[T,R]):mut Flow[T]
  mut .filter(f:F[T,Bool]):mut Flow[T]
  mut .list:List[T]
  }
```

F[T,R] and F[T,Bool] encode deterministic computations
→ could be optimised by a smart compiler.

Could be run in parallel, and their results could be cached

Can be more! Parallel operations on a mut List[mut Repr[T]]


Is this treasure just the tip of the iceberg?
What other properties could be enforced?
What other unobservable optimizations could be unlocked?