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 omitted ‘x = ‘this’. Omitted ‘x for declarations in methods is fresh
- Omitted declaration name D of a literal inside of an expression is fresh

Concrete syntax:

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

Inference:

- The implemented types Ts of a declaration are inferred when the target type is known.
- An overridden sig can omit the types
- Can omit even the method name if there is only 1 abstract method

```
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
```

What are 42 and “Bob”? Are they in the shown syntax? Yes!
- **42 *desugared as* FreshName:42[]{}**
- **"Bob" *desugared as* FreshName:"Bob"[]{}**
- **FPerson == FPerson[]{} == FreshName:FPerson[]{}**

The singleton instance of Any top level declaration with no abstract method can be ‘summoned’ by just writing the name

```
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
```

```
~~Person: { .age: Num, .name: Str }~~  //not declared at top level
```

Person as an internally declared concept instead

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

FPerson#(42, "Bob")    //same instantiation syntax as before,
                       //but now this is guaranteed to
                       //be the only way to make a Person.
//A declaration name introduced inside a method body is ‘final’ and
//can not be inherited. Thus writing ‘Person{…}’ anywhere else would
//be a type error.
```

```
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
```

```
Opt[T]: {
  .match[R](m: OptMatch[T,R]): R -> m.empty
  }
OptMatch[T,R]: {
  .empty: R,
  .some(t: T): R
  }
Opt: {
  #[T](t:T):Some[T] -> Some[T]:Opt[T]{ m -> m.some(t) }
  }
//We could give an explicit name to the Some Opt, but we
//avoid defining more declaration names than strictly needed

//Note the generics: Some[T]{...} is ‘funneling’ the generic arguments
//used inside of it. All the generic parameters
//captured from the environment must be repeated in the declaration
```

```
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]: {//Ideally
  read .get: T,
  mut  .swap(x: T): T,
  }

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

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

```
Ref[T]: {   //three .get in overloading
  read[imm T] .get: imm T,
  read .get: read T, //== read[T] .get: read T
  mut .get: T,
  mut .swap(x: T): T,
  mut .set(x: T): Void -> Block#(this.swap(x), Void),
  }

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

Void: {}

Block: {
  #[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:

$$\begin{tikzcd}
\text{imm} \arrow[d, shift left] & \text{iso} \arrow[l] \arrow[d, shift left] \\
\text{read} \arrow[u, shift left] & \text{mut} \arrow[l] \arrow[u, shift left]
\end{tikzcd}$$

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

Note the method syntax: we specify the receiver for this, and a specialized version of the generic parameters

Concrete syntax: D[Ts] desugared as imm D[Ts]

Multiple method typing:

A method has an additional valid signature, where all the mut parameters are turned in iso and all the read parameters are turned in imm; then a mut result can be turned into iso and a read result into imm.

This also applies for whole method bodies: if a method body does not use any mut/read parameters, then it can return a mut value as an iso or imm, and a read value as an imm.

Extended subtyping: The adapt rule

```
.ad01(r: Ref[mut Person]): Ref[Person] -> r,    //OK

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

.ad02(l: List[TallPerson]): List[Person] -> l,     //OK
.ad03(l: List[mut TallPerson]): List[Person] -> l, //OK
.ad04(l: mut List[TallPerson]): mut List[Person] -> l, //Correctly rejected

The adapt rule

  R D[T1..Tn]<= R D[T1′..Tn′]
    If all the methods 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.
```

```
.ex01(r: Ref[Person], p: Person): Void -> r.set(p), //type error, r is immutable

.ex02(r: read Ref[Person], p: Person): Void -> r.set(p), //type error, r is readable

.ex03(r: mut Ref[Person], p: Person):Void -> r.set(p) //ok

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

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

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

.ex07(p: Person): iso Ref[Person] -> 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(p: Person):  Ref[Person] -> Ref#p //ok, iso promotion + iso-> imm subtyping

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

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

.ex11(r: Ref[mut Person]): Person-> r.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(r: read Ref[Person]): Person -> r.get //ok using the read[imm T] .get
.ex13(r: Ref[Person]): Person -> r.get //ok with either the read[imm T] .oget or the
      //read .get and imm promotion
```

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 object literals:
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) 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 Block# without parameters.
Example:

```
MyApp:Main{s->Block#
  .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->Block#
  .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")}
     })})
  }
```

a 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->Block#
  .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: the Repr API

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


FRange:F[Nat,Nat,Range]{(min,max->Block#
  .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, cache clear possible because of Magic above
    }}
  }
```

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:

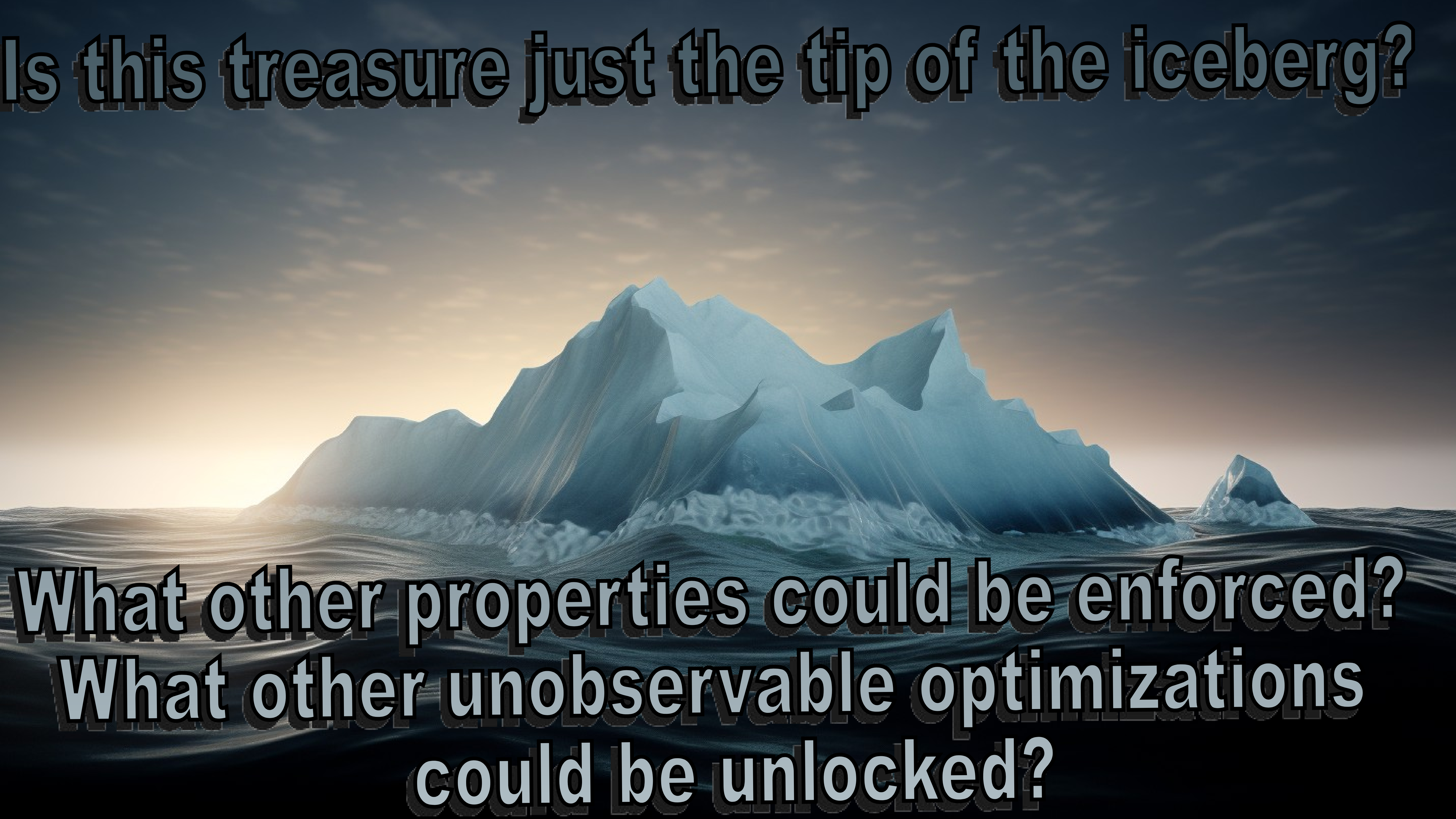
```
Flow[T]:{
  mut .map(f:F[T,R]):mut Flow[T]
  mut .filter(f:F[T,Bool]):mut Flow[T]
  mut .list:List[T]
  }
```

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?

More: Fork Join

- We could have a magic forkJoin method

```
.forkJoin[A,B,Ra,Rb,R](
  a: mut Repr[A], b: mut Repr[B],
  fa: F[mut A,imm Ra], fb: F[mut B,imm RB],
  f: mut F[imm RA,imm RB,R])→
  )-> f#(a.mutate(fa), b.mutate(fb))//but in parallel
```

Same for more then 2 Repr if needed.
Note how an object Node capturing two Repr[Node] children to represent a tree can now encode a parallel tree computaition.

More: Normalization

- We could have a magic norm method

  ```
  .norm[A](a: imm A)-> imm A
  ```

  Returning the a single ‘normalized/interning representation for any immutable object for structurally equivalent objects.
- Now the cache of .cached methods can be recover. For example fibonacci could rely on normed computational objects with a cached # method, and get the memoized performance of fibonacci automatically

More: Eager cache

- In addition to .cached we could have other variants.

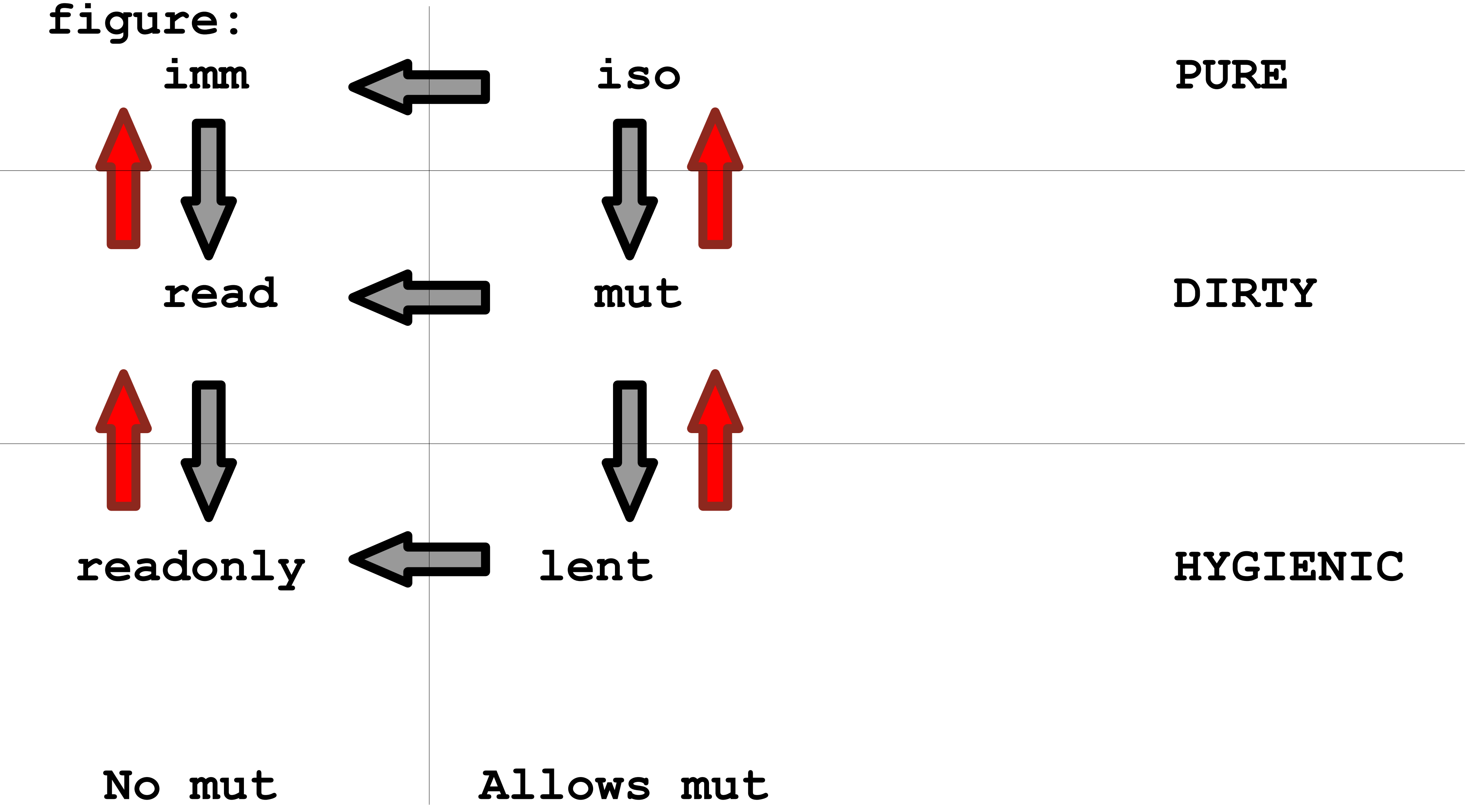
```
mut .cached[R](f:F[read T,imm R]):_
mut .eagerCached[R](f:F[read T,imm R]):_
mut .property[R](f:F[read T,imm R]):_
```

  cached makes a lazy cache,
  eagerCached starts the computation immediately on a parallel worker,
  property runs the computation immediately and re run it immediately any time the cache is cleared. Inveriants are actually a kind of properties

More Reference modifiers and grammar

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

figure:



readonly and lent can be in input for imm/iso promotions, making them more flexible.
Mostly only for references: List[lent Person] or List[readonly Person] are problematic

Multiple method typing:

```
Additional valid signatures:
//Repeated to show lent/readonly parameters stay untouched
All the mut parameters are turned in iso and all the read
parameters are turned in imm; then a mut result can be turned
into iso and a read result into imm.
Compact:
sig[mut=iso, read=imm]
//New1:
sig[mut=iso, read=imm, readonly=imm]
//New2:
sig[result=hygienic][mut=iso, read=readonly]
//New3:
sig[result=hygenic][1_mut=lent, other_muts=iso]
//1 mut parameter goes to lent, the other muts to iso

sig[result=hygienic] ==> mut,iso=lent, read=readonly
```

Added Flexibility:

-(1) A method taking no mut/read can easily produce an iso.
  Can still take lent/readonly to edit/access mutable data.
  (eg parser)

-(2) A mut→mut method now works
  both as an imm→imm and as a lent→lent

-(3) A read*→read method works as a readonly*→readonly

-(4) A lent/readonly object can capture
    other readonly objects using (3)

-(5) A lent object can capture zero or one lents using (2)