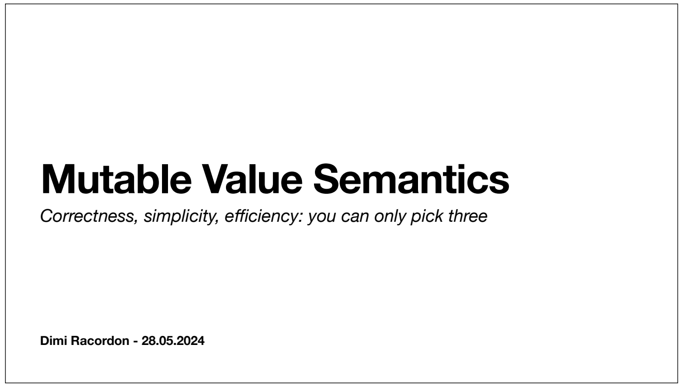

Mutable Value Semantics

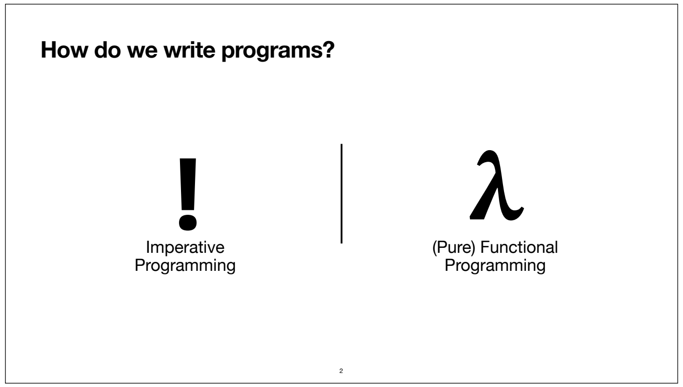
Correctness, simplicity, efficiency: you can only pick three

Dimi Racordon - 28.05.2024

Hello everyone. My name is Dimi Racordon. I'm a postdoc researcher at EPFL, working on programming language design and implementation, with a particular focus on safety and generic programming.

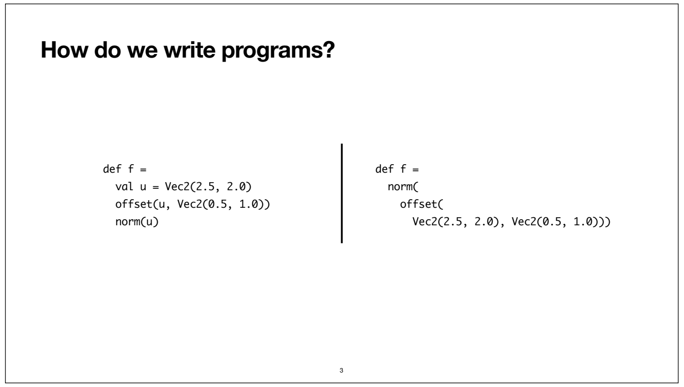
I'm truly delighted to be here today, and I'd like to take this opportunity to talk about MVS, which is a framework central to my research. That is a rather broad topic, however, so I we won't be able to cover everything. Instead we'll focus on three things today. I'll first setup some context to understand what MVS is and what problems it solves, then I'll show some of its formal foundations, and finally I'll give a teaser of a programming language entirely based on this framework.

So without any further ado, let's get started.


How do we write programs?

!
Imperative Programming
λ
(Pure) Functional Programming

To set the stage, I'd like to talk little about the rivalry between imperative and functional programming. It's often that people fall into one "camp" when discussing these two paradigms, fighting over what approach is best for performance and/or correctness. Let's look at an example to understand the core arguments.



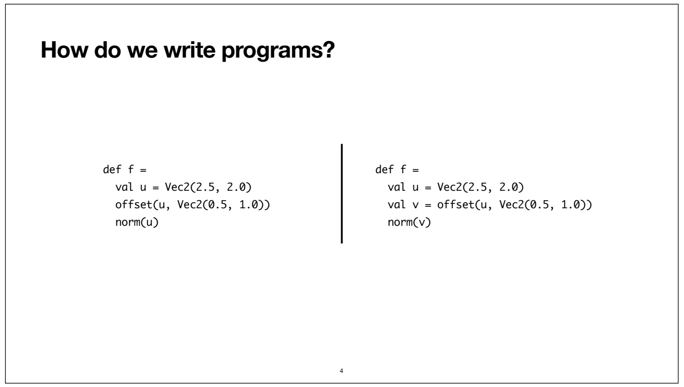
How do we write programs?

```
def f =
  val u = Vec2(2.5, 2.0)
  offset(u, Vec2(0.5, 1.0))
  norm(u)
```

```
def f =
  norm(
    offset(
      Vec2(2.5, 2.0), Vec2(0.5, 1.0)))
```

Here is a program written in an imperative style on the left and in a pure functional style on the right. As we can see, the distinguishing feature of imperative programming is that we express computation in terms of *state transitions*. We go from one statement to the next and in doing so we cause *side effects* on the state of the program. In contrast, functional programming uses composition as a sequencing mechanism. So the state of the program is now part of the data flow.

To improve legibility, and I hope I won't offend any Lisp enthusiast here, we can still have bindings in functional programming to avoid large expression trees and so-called "pyramids of doom". For instance, we can rewrite the program on the right as follows.



How do we write programs?

```
def f =
  val u = Vec2(2.5, 2.0)
  offset(u, Vec2(0.5, 1.0))
  norm(u)
```

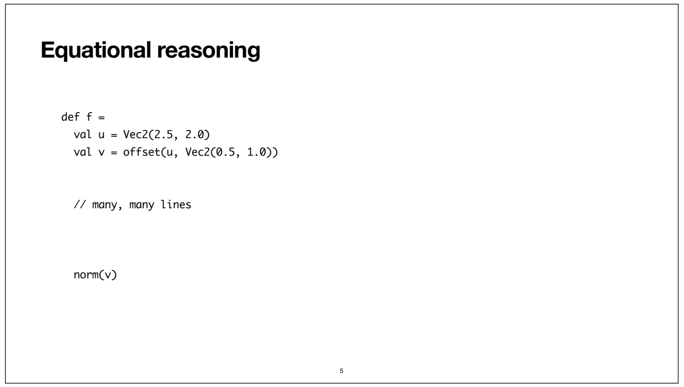
```
def f =
  val u = Vec2(2.5, 2.0)
  val v = offset(u, Vec2(0.5, 1.0))
  norm(v)
```

It's important to note that the two paradigms have the same theoretical expressiveness. That means there's no program written in an imperative style that we cannot also write in a functional style, and vice versa. In fact if, there's even a general translation pattern.

>> Everything that is being mutated by an expression in the imperative world can be simply returned as a new value in the functional world. This idea is often to as a *functional update*.

Despite this equivalence, in practice, the choice of a particular paradigm has important implications. For instance, advocates of imperative programming will often mention efficiency while advocates of functional programming will often mention referential transparency and equational reasoning.

I propose we focus on our functional program a little bit to explore these concepts.



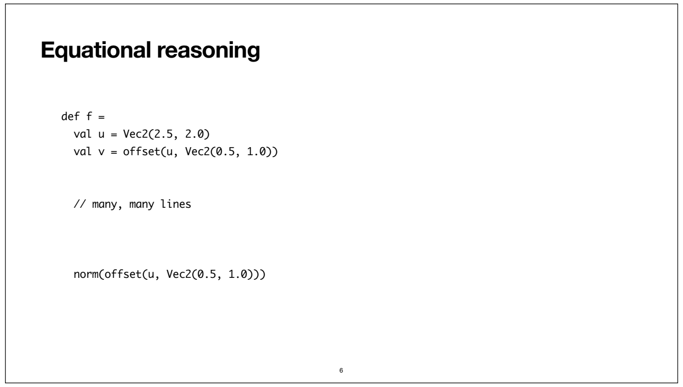
Equational reasoning

```
def f =
  val u = Vec2(2.5, 2.0)
  val v = offset(u, Vec2(0.5, 1.0))


  // many, many lines



  norm(v)
```

5

If we stick to a functional style and refrain from mutating anything, then we get *referential transparency*, that is the ability to replace any expression by another expression computing the same value. That's a nice property because it tells us that there's nothing in the many, many lines we've added that can change the value of `v`.

So even more generally, we can simply replace occurrences of `v` by the value to which it was assigned.



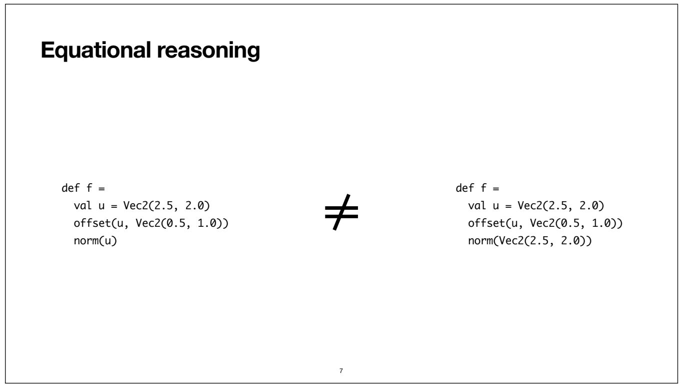
Equational reasoning

```
def f =
  val u = Vec2(2.5, 2.0)
  val v = offset(u, Vec2(0.5, 1.0))


  // many, many lines


  norm(offset(u, Vec2(0.5, 1.0)))
```

6

And obviously we can do the same for `u` as well. If we do that, we'll get to the original program, without the addition of the local bindings.

From the notion of referential transparency we get *equational reasoning*, that is the ability to interpret code by substituting expressions known to be equal. And we can do that in any order we like, so an optimizer could come here and decide to run two arbitrary expressions in parallel to make our program faster.



Equational reasoning

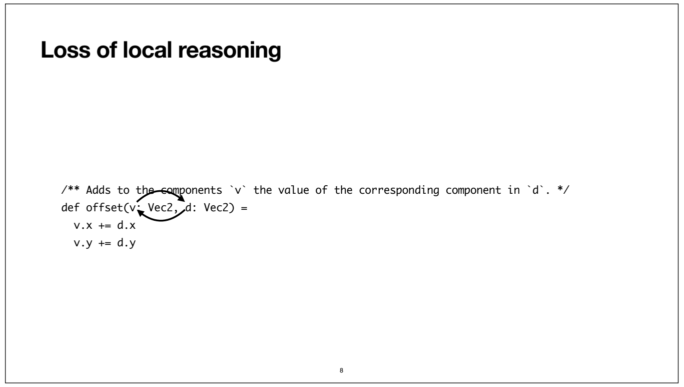
```
def f =
  val u = Vec2(2.5, 2.0)
  offset(u, Vec2(0.5, 1.0))
  norm(u)
```

$\neq$

```
def f =
  val u = Vec2(2.5, 2.0)
  offset(u, Vec2(0.5, 1.0))
  norm(Vec2(2.5, 2.0))
```

7

We clearly *don't* have equational reasoning in our imperative program since evaluation order is now significant. So obviously we have lost referential transparency and all the nice benefits we got from a pure functional style have disappeared.



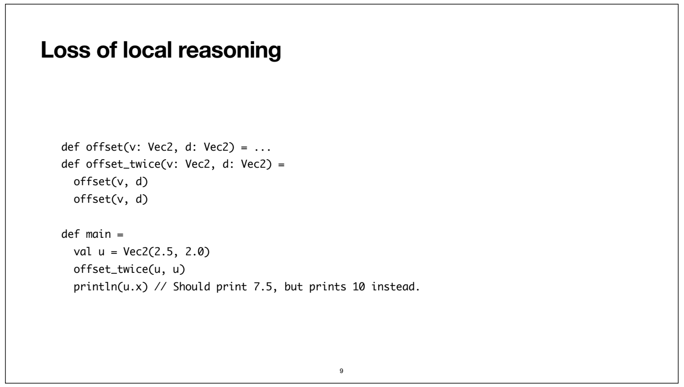
Loss of local reasoning

```
/** Adds to the components `v` the value of the corresponding component in `d`. */
def offset(v: Vec2, d: Vec2) =
  v.x += d.x
  v.y += d.y
```

Personally, I think the biggest loss is actually local reasoning. The is most apparent if we examine the imperative implementation of the offset method.

In Scala, like in many other languages, the arguments we get in a function are references to some shared objects. The result is that there's no way to reason locally about the consequences of modifying a shared object.

>> In this particular example, it may be that `v` and `d` alias each other. When that's the case, both `v` and `d` will be modified, which is an undocumented behavior. And that's an obvious threat to correctness.

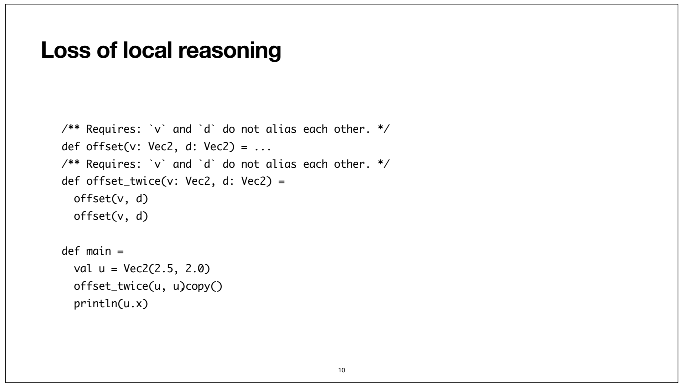


Loss of local reasoning

```
def offset(v: Vec2, d: Vec2) = ...
def offset_twice(v: Vec2, d: Vec2) =
  offset(v, d)
  offset(v, d)

def main =
  val u = Vec2(2.5, 2.0)
  offset_twice(u, u)
  println(u.x) // Should print 7.5, but prints 10 instead.
```

Here, for example, we'll get surprising results if the arguments to `offset_twice` alias each others. And I would even claim that the bug isn't exactly easy to spot if you don't know where to look, despite this program being about 10 lines long. So we can easily imagine the scale of the problem in an actual codebase.

In reference-oriented languages, like Scala, Java, Python, JavaScript and many others, there are only two ways to address this problem.



Loss of local reasoning

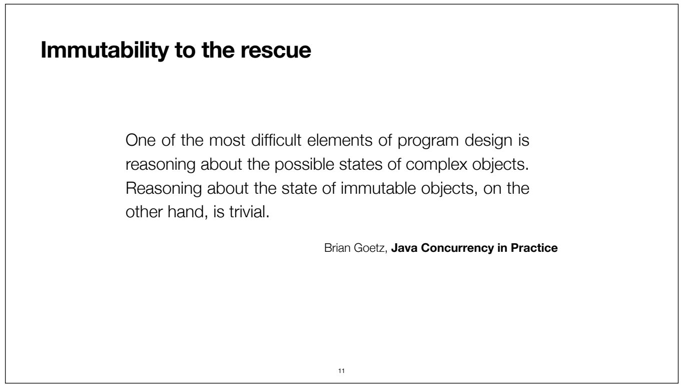
```
/** Requires: `v` and `d` do not alias each other. */
def offset(v: Vec2, d: Vec2) = ...
/** Requires: `v` and `d` do not alias each other. */
def offset_twice(v: Vec2, d: Vec2) =
  offset(v, d)
  offset(v, d)

def main =
  val u = Vec2(2.5, 2.0)
  offset_twice(u, u)copy()
  println(u.x)
```

10

The first is to properly document our precondition and expect users to satisfy them.

>> Unfortunately compilers typically don't enforce comments so it falls on us insert defensive copying everywhere. That's not a great deal because forgetting a single copy can create nasty bugs, while copying too much is inefficient.

The other solution is to give up mutation.



Immutability to the rescue

One of the most difficult elements of program design is reasoning about the possible states of complex objects. Reasoning about the state of immutable objects, on the other hand, is trivial.

Brian Goetz, **Java Concurrency in Practice**

11

And in fact this idea that immutability is the answer to the problems caused by reference semantics is pretty wide spread. And given the situation we have discussed so far it's easy to understand why.

But before we conclude that mutation is evil, I'd like to talk another example.

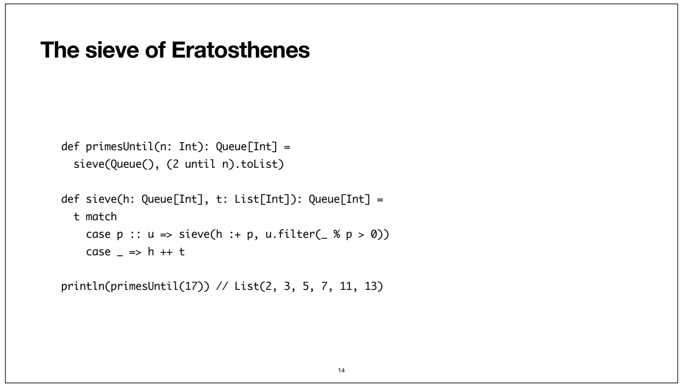
The sieve of Eratosthenes

And for that I'll need the help of Eratosthenes of Cyrene, a Greek mathematician who's credited for having discovered a pretty neat algorithm to compute lists of prime numbers.

The sieve of Eratosthenes

~~1~~	2	3	~~4~~	5	~~6~~	7	~~8~~	~~9~~	~~10~~
11	~~12~~	13	~~14~~	~~15~~	~~16~~	17	~~18~~	19	~~20~~

The spirit of the algorithm is to start with a sequence of numbers and to cross off those that are multiple of known primes, starting with 2. It's a very simple algorithm and we can find many examples of implementations online, in pretty much any language you like.



The sieve of Eratosthenes

```
def primesUntil(n: Int): Queue[Int] =
  sieve(Queue(), (2 until n).toList)

def sieve(h: Queue[Int], t: List[Int]): Queue[Int] =
  t match
    case p :: u => sieve(h :+ p, u.filter(_ % p > 0))
    case _ => h ++ t

println(primesUntil(17)) // List(2, 3, 5, 7, 11, 13)
```

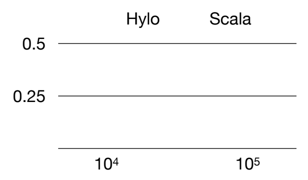
14

And we might stumble on something like that for Scala, which is a functional implementation of the sieve.

>> The bulk of the work is done by this function, which takes two ordered sequence. The first contains the prime numbers we have found and the second is a sequence of candidates.
>> At each iteration, we identify the first element of `t` to be prime, since it hasn't been excluded for being multiple of any member in `h`. We then removes all multiple of that new prime before starting a new iteration.

This implementation is concise and elegant. I would even go as far as to say that it's the kind of code we could be tempted to use as evidence for the beauty of pure functional programming. But that would be a mistake, because this program is hopelessly inefficient.

The sieve of Eratosthenes



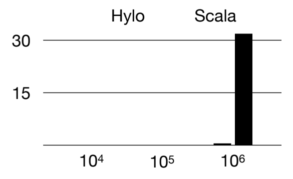
15

Here are the results of a benchmark I ran to compare the Scala implementation we just saw and a competing imperative version written in Hylo, which is the MVS-based language I mentioned earlier. The x-axis shows the length of the candidate list the y-axis is the time it took identify prime numbers.

I compiled both programs using the same compiler backend, specifically LLVM, to make the comparison as fair as possible and so far so good. The difference in performance is explained by the fact that Scala is a little harder to optimize than Hylo but I think most Scala developers would be satisfied with those results.

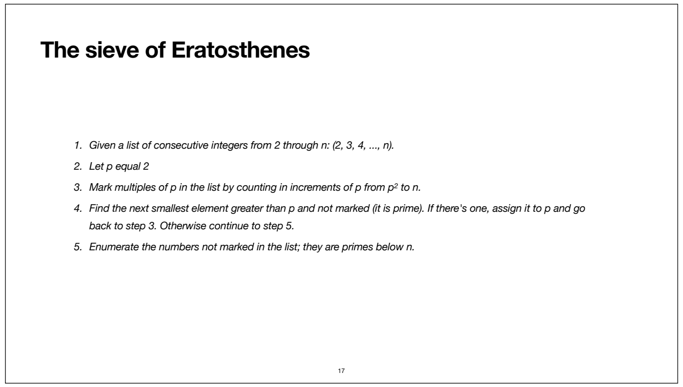
Now let's see how things scale.

The sieve of Eratosthenes



Ouch!

It took my machine half a minute to compute prime numbers up to 1 million in Scala and only a few hundreds of milliseconds in Hylo. Clearly we're not in the same ballpark anymore. So surely there's a bug somewhere. So maybe we should take a closer look at the algorithm we're trying implement.



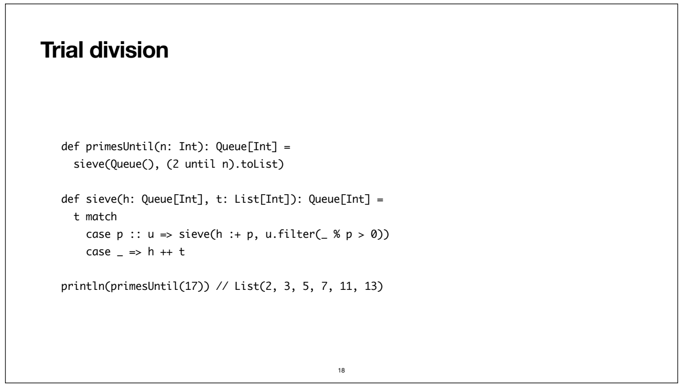
The sieve of Eratosthenes

1. *Given a list of consecutive integers from 2 through n: (2, 3, 4, ..., n).*
2. *Let p equal 2*
3. *Mark multiples of p in the list by counting in increments of p from p^2 to n.*
4. *Find the next smallest element greater than p and not marked (it is prime). If there's one, assign it to p and go back to step 3. Otherwise continue to step 5.*
5. *Enumerate the numbers not marked in the list; they are primes below n.*

Here's a full description the algorithm.

read the algorithm

>> Notice that in step 3 we're supposed to enumerate multiples of the newly identified prime *by increments of itself starting from its square*. That is *not* what my Scala implementation does.

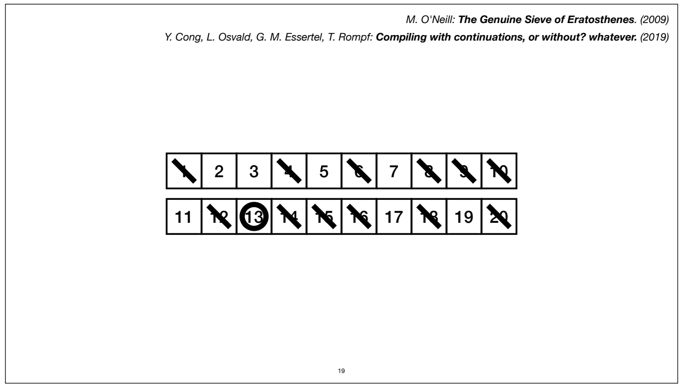


Trial division

```
def primesUntil(n: Int): Queue[Int] =
  sieve(Queue(), (2 until n).toList)

def sieve(h: Queue[Int], t: List[Int]): Queue[Int] =
  t match
    case p :: u => sieve(h :+ p, u.filter(_ % p > 0))
    case _ => h ++ t

println(primesUntil(17)) // List(2, 3, 5, 7, 11, 13)
```

18

Here, the filter operates on all elements that are not divisible by the primes smaller than `p`. So my implementation needs to consider many more numbers than the actual sieve. That's because I implemented another algorithm, called trial division.



M. O'Neill: ***The Genuine Sieve of Eratosthenes****. (2009)*

Y. Cong, L. Osvald, G. M. Essertel, T. Rompf: ***Compiling with continuations, or without? whatever.*** *(2019)*

19

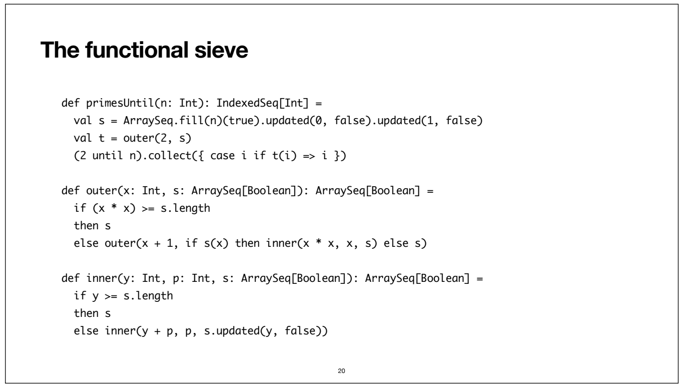
So for example, let's imagine we've discovered that `13` is a prime number. In the actual sieve, we'd have no more work to do because 13 squared is out of bounds already. But in the trial division we still have to consider 2, 3, 5, 7 and 11.

>> Melissa O'Neill's wrote a beautiful paper about this example that you should definitely read fore mode details. In fact, my slides are just a shameless ripoff. But the reason why I brought up this example is that focusing on "concise" and "pure" functional implementations is sometimes a recipe to forget about the efficiency of the original algorithm, or even worse, to write a different one. Sure, concision makes reasoning simpler, but if it comes at the cost of efficiency, it's actually a loss.

>> The claim is typically that optimizers will fix everything, leaving us to only

think about correctness, which sadly isn't true. There are still programs that escape our most sophisticated optimization techniques and the whole process is based on heuristics that can't give predictable results. But even if optimizers could always fix inefficient functional updates, it's unlikely that an optimizer will figure out the actual algorithm we wanted to write. And it turns out that many algorithms are actually described in an imperative way, like the sieve, Dijkstra's shortest path, Quicksort, etc. Translating them into a pure functional world can be quite challenging.

So, what if we write the actual sieve, but in a functional way.



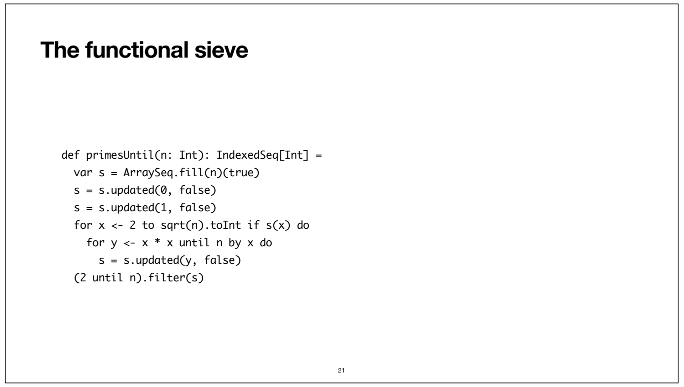
The functional sieve

```
def primesUntil(n: Int): IndexedSeq[Int] =
  val s = ArraySeq.fill(n)(true).updated(0, false).updated(1, false)
  val t = outer(2, s)
  (2 until n).collect({ case i if t(i) => i })

def outer(x: Int, s: ArraySeq[Boolean]): ArraySeq[Boolean] =
  if (x * x) >= s.length
  then s
  else outer(x + 1, if s(x) then inner(x * x, x, s) else s)

def inner(y: Int, p: Int, s: ArraySeq[Boolean]): ArraySeq[Boolean] =
  if y >= s.length
  then s
  else inner(y + p, p, s.updated(y, false))
```

20

Well, while I appreciate the beauty of tail recursion as much as the next person, I think the claims of conciseness pretty much flew out of the window here. I would even go as far as to say that this program is now quite hard to read, so maybe we can sprinkle some local variables to make our life easier.

To be fair to Scala developers, that's what they would tell me to do anyway.



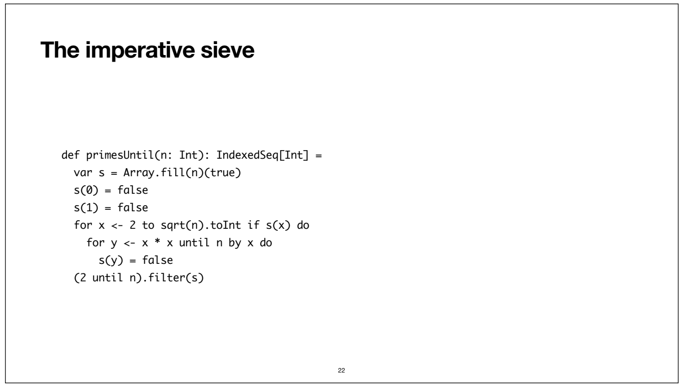
The functional sieve

```
def primesUntil(n: Int): IndexedSeq[Int] =
  var s = ArraySeq.fill(n)(true)
  s = s.updated(0, false)
  s = s.updated(1, false)
  for x <- 2 to sqrt(n).toInt if s(x) do
    for y <- x * x until n by x do
      s = s.updated(y, false)
  (2 until n).filter(s)
```

That's already much better if you ask me. And don't worry, we'll see in a minute why the local variables are actually innocuous.

The interesting question for now is to know whether this program is as fast, or at least in the same ballpark as its imperative counterpart. Well, sadly it's even worse than the one we had before. The culprit is here >>

This line implements a functional update modeling the in-place mutation of an array. Unfortunately, while it accurately reproduces the semantics of an imperative assignment, it does not have the same performance. In a nutshell, the problem is that we're creating an entire new collection only to change one element to replace the one we had. So not only we're wasting compute, we're also putting a lot of pressure on the heap allocator. Meanwhile, an imperative version can just re-use the same array.

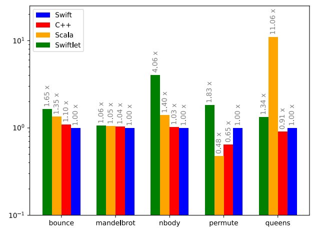


The imperative sieve

```
def primesUntil(n: Int): IndexedSeq[Int] =
  var s = Array.fill(n)(true)
  s(0) = false
  s(1) = false
  for x <- 2 to sqrt(n).toInt if s(x) do
    for y <- x * x until n by x do
      s(y) = false
  (2 until n).filter(s)
```

In fact, if we bite the bullet and write an imperative sieve, we finally get as fast as the Hylo program. The most significant change we have made is to replace the functional update by an actual in-place mutation.

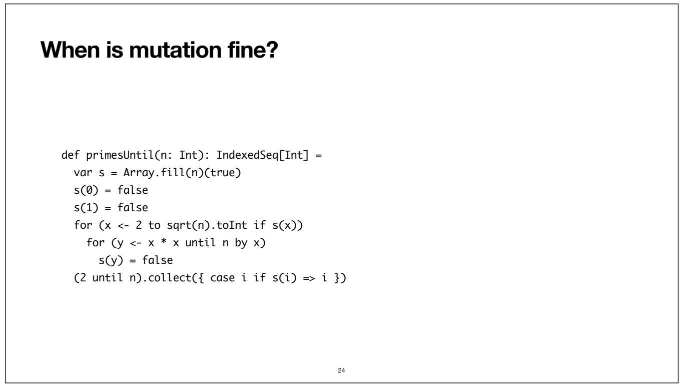
D. Racordon, D. Shabalin, D. Zheng, D. Abrahams, B. Saeta: ***Implementation Strategies for Mutable Value Semantics.*** *(2022)*



You can look at this paper that I co-wrote with people from google if you want more compelling results about the inefficiency of functional update.

We may be disappointed that we had to sacrifice functional purity but this function is actually not that bad. In fact, for all intents and purposes, callers of `primesUntil` can consider the function to be as pure as its inefficient counterpart. And to be fair, that's exactly what Scala developers will tell you. What's a little sad in my opinion is that the compiler won't help us making sure we're not using mutation in a way that may break local reasoning. In a world where immutability is your only safety net, mutating is a very dangerous escape hatch.

So while `primesUntil` is a fine function, the broader question asks *when* we can use mutation fearlessly.

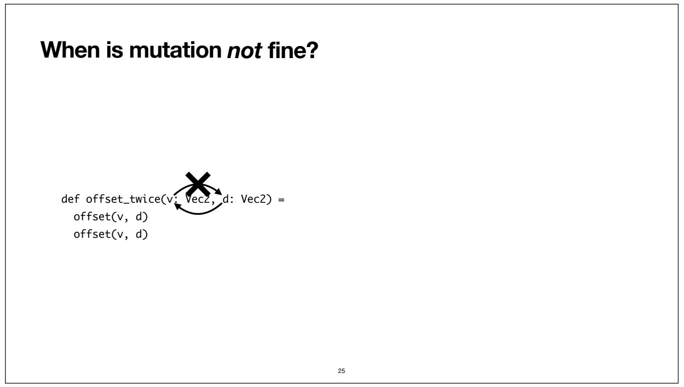


When is mutation fine?

```
def primesUntil(n: Int): IndexedSeq[Int] =
  var s = Array.fill(n)(true)
  s(0) = false
  s(1) = false
  for (x <- 2 to sqrt(n).toInt if s(x))
    for (y <- x * x until n by x)
      s(y) = false
  (2 until n).collect({ case i if s(i) => i })
```

For example, using a local variable is fine because callers of the function obviously can't observe its mutation. So it becomes just an implementation detail.

>> Likewise, modifying an unaliased local variable is fine because there is nothing this mutation can do behind our backs. What we see is what we get.

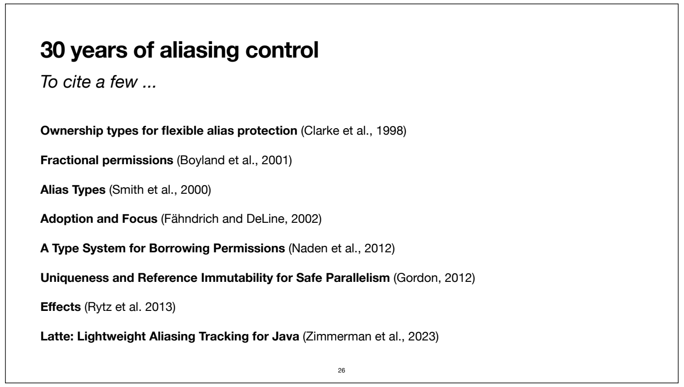


When is mutation *not* fine?

```
def offset_twice(v: Vec2, d: Vec2) =
  offset(v, d)
  offset(v, d)
```

In contrast, mutation is harmful when it applies to potentially shared data structures because it breaks local reasoning.

This observation explains the essence of MVS. It's a programming discipline that eliminates shared mutable state. >> Hence, of all intents an purposes, all values can be considered an independent piece of data whose mutation cannot possibly have an impact on anything else. In other words, it's a discipline that identifies aliasing of mutable data as the root of all evil, not mutation itself.



30 years of aliasing control

To cite a few ...

Ownership types for flexible alias protection (Clarke et al., 1998)

Fractional permissions (Boyland et al., 2001)

Alias Types (Smith et al., 2000)

Adoption and Focus (Fähndrich and DeLine, 2002)

A Type System for Borrowing Permissions (Naden et al., 2012)

Uniqueness and Reference Immutability for Safe Parallelism (Gordon, 2012)

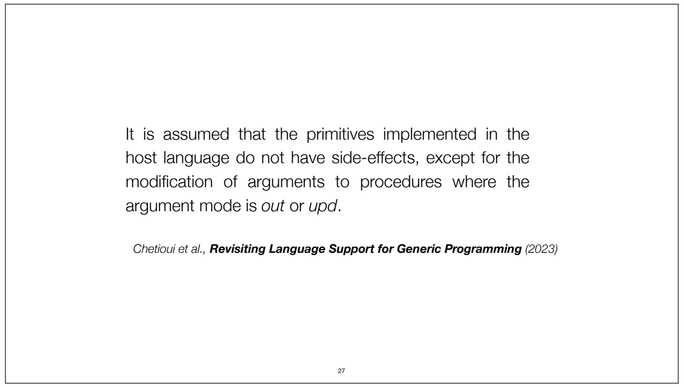
Effects (Rytz et al. 2013)

Latte: Lightweight Aliasing Tracking for Java (Zimmerman et al., 2023)

26

Of course we're not the first ones to make a similar observation. There are more than 3 decades of research about taming aliasing, yielding a wealth of interesting techniques, all with their pros and cons. What's interesting is that uniqueness is a recurring theme in this body work when it comes down to mutation.

One way or another, all of these systems provide a mechanism to uphold local reasoning.



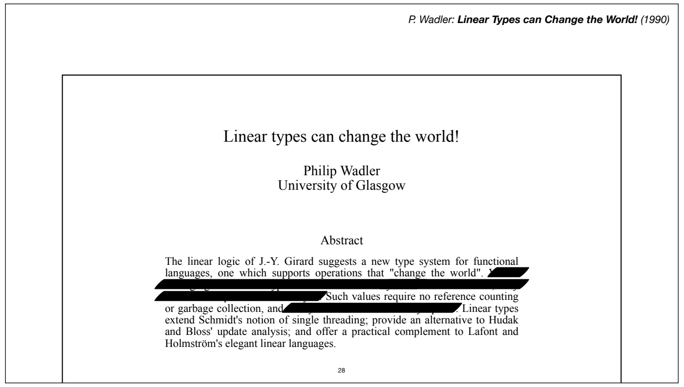
It is assumed that the primitives implemented in the host language do not have side-effects, except for the modification of arguments to procedures where the argument mode is *out* or *upd*.

Chetioui et al., ***Revisiting Language Support for Generic Programming*** *(2023)*

27

In fact, uniqueness is so important that it shows up as a core assumption in other areas of programming language research. If you read enough PL papers, you will eventually stumble on some form of value independence. For instance, here's a quote from a paper about generic programming.

So hopefully that should convince us that uniqueness is important. But thinking about the constraints of MVS, one question arises. If we boot references out of the user model, then how exactly are we supposed to implement mutation across function boundaries, without using inefficient functional updates?



P. Wadler: **Linear Types can Change the World!** (1990)

Linear types can change the world!

Philip Wadler
University of Glasgow

Abstract

The linear logic of J.-Y. Girard suggests a new type system for functional languages, one which supports operations that "change the world". [redacted] Such values require no reference counting or garbage collection, and [redacted]. Linear types extend Schmidt's notion of single threading; provide an alternative to Hudak and Bloss' update analysis; and offer a practical complement to Lafont and Holmström's elegant linear languages.

28

Well there's one very important paper in the business of taming aliasing that I haven't mentioned yet. In 1990, Wadler wrote about *linear types*, casting Girard's linear logic into the realm of type systems. In his abstract he wrote:

>> Values belonging to a linear type must be used exactly once.

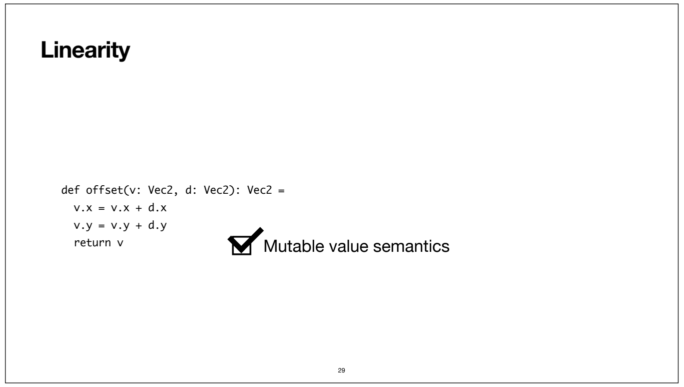
That's nice because if values behave linearly, then they obviously become independent. But that's not the only interesting bit of the abstract. Later we read:

>> [Linear values] safely admit destructive array updates.

This sentence suggests that we can exploit linear types to perform functional updates that do not involve inefficient copies. And finally the last sentence that catches my attention is this one:

>> Like the world, [linear values] cannot be duplicated or destroyed.

It suggests that thinking in terms of linear values matches our understanding of the world. And indeed, actual objects certainly "behaves linearly". We can't duplicate things out of thin air and likewise things we don't need anymore don't just disappear. So not only can linear types serve to implement value semantics, perhaps they can also serve to build useful abstractions.



Linearity

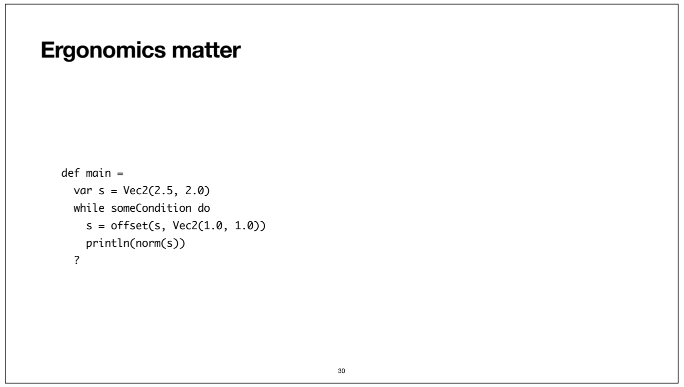
```
def offset(v: Vec2, d: Vec2): Vec2 =
  v.x = v.x + d.x
  v.y = v.y + d.y
  return v
```

✓ Mutable value semantics

29

As we've seen from Wadler's abstract, linearity means that values must be used exactly once. So if Scala had linear types, this definition of `offset` would be completely fine because there would be no way for `v` and `d` to alias each other.

>> Yet, we'd still be able to perform part-wise in-place mutation, what Wadler called "destructive array updates".
>> In other words, we'd get value semantics.



Ergonomics matter

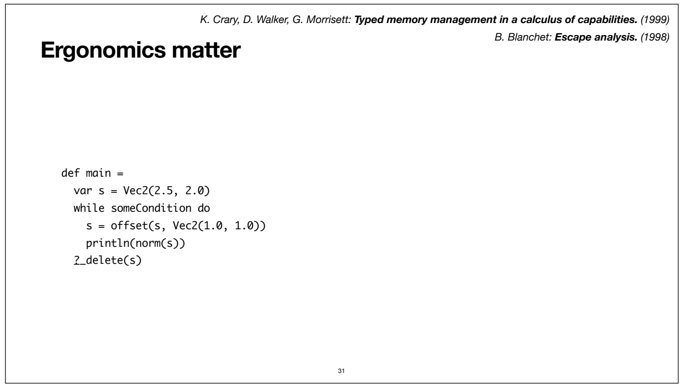
```
def main =
  var s = Vec2(2.5, 2.0)
  while someCondition do
    s = offset(s, Vec2(1.0, 1.0))
    println(norm(s))
  ?
```

30

Given that Wadler's paper is 30 years old, you may now wonder why linear types have not taken over the world. A big part of the answer is about language ergonomics. A formal system is of little practical use unless it is also conducive to a good programming experience. And as early critiques of linear types have pointed out, using linear values in conjunction with standard control flow is typically hard.

>> Here, for example, a linear type system would complain that `s` cannot be used in the loop because for any iteration but the first, it would have been consumed by the call to `norm` one line after.
>> We would probably get another error after the loop because if the condition never holds, then `s` won't have been used at all, which isn't permissible either.

We can settle for an affine system to alleviate these issues but if we want to stick with a true linear system, and I'll explain later why that's a good idea, there are a couple of observation we can make to improve ergonomics.



K. Crary, D. Walker, G. Morrisett: ***Typed memory management in a calculus of capabilities.*** *(1999)*

B. Blanchet: ***Escape analysis.*** *(1998)*

Ergonomics matter

```
def main =
  var s = Vec2(2.5, 2.0)
  while someCondition do
    s = offset(s, Vec2(1.0, 1.0))
    println(norm(s))
  ?_delete(s)
```

31

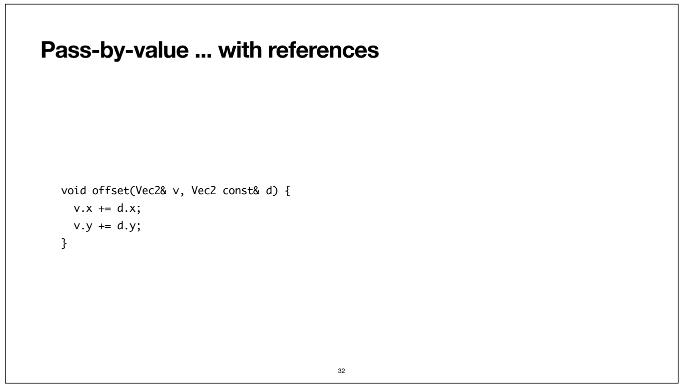
The first is that most operations only every need to inspect the contents of a value without modifying it. In that case, we can pretend that immutable values are independent, even if they share storage.

>> We should note that this idea has roots in early works on capability-based memory management systems. Here, we can use it to avoid consuming `s` for computing its norm, to get rid of the first error without changing anything.

The second observation we can make is that a compiler can figure out ahead of time when values are no longer needed and insert destruction on its own.

>> This approach is called escape analysis and it has been battle-tested in many programming languages. Here, we can use it to insert destructor calls at the end of useful lifetimes to give the illusion of an affine type system.

As the citations show, all of these ideas have deep roots in scientific literature but perhaps surprisingly, it's by looking at the way C++ developers write code that it all clicked for me.



Pass-by-value ... with references

```
void offset(Vec2& v, Vec2 const& d) {
  v.x += d.x;
  v.y += d.y;
}
```

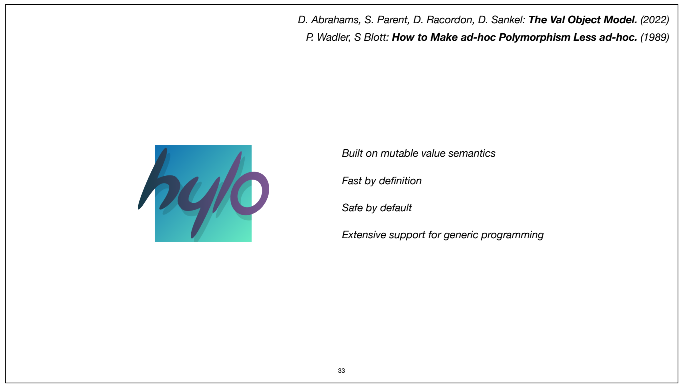
32

For example, here's how a C++ developer would write the offset function.

>> The first parameter accepts a mutable reference to the vector being mutated; and
>> The second parameter accepts a constant reference.

Using references gives this function similar semantics as the one we discussed in the beginning, which was written in Scala. But interestingly, the way C++ developers intuitively understand the signature is a slightly different. They will often assume that mutable references are unique and that immutable references are just a way to achieve pass-by-value semantics without paying for copies. In that sense, references are just a way to implement a function that's otherwise designed under a MVS-oriented mindset.

And this observation leads us to the third and final part of my lecture.


D. Abrahams, S. Parent, D. Racordon, D. Sankel: ***The Val Object Model.*** *(2022)*
P. Wadler, S Blott: ***How to Make ad-hoc Polymorphism Less ad-hoc.*** *(1989)*

Built on mutable value semantics

Fast by definition

Safe by default

Extensive support for generic programming

33

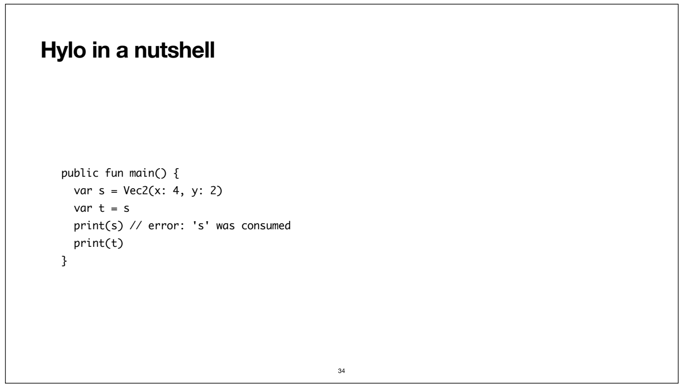
Meet Hylo, a programming language for high-level systems programming based entirely on MVS.

>> Hylo is a project on which I've been working for a little over three years in collaboration with great minds from the industry. Though it has started as a research endeavor, it has broke a little out of academia and caught some attention from the C++ community because of its goals.

Hylo is designed to be fast by definition, meaning that it embraces a "zero cost abstraction" principle. The user can rely on optimization guarantees to make sure that the use of abstractions like object or higher-order functions do not incur additional costs.

Hylo is also designed to be safe by default, meaning that you can't run operations that may cause undefined behavior unless you explicitly opt-out its safe subset. I don't have time to expand on this idea but that's an important problem in systems programming.

>> Finally, Hylo has an extensive support for generic programming. Again, we won't really have time to explore that today but one thing that's interesting to note is that MVS and linearity fit really well into the concept of type classes, and another import from the functional programming community.



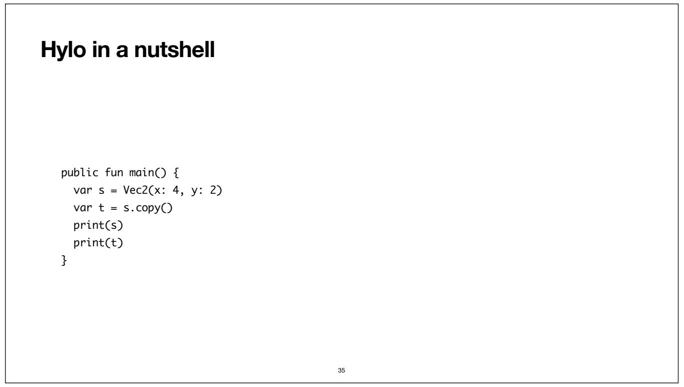
Hylo in a nutshell

```
public fun main() {
  var s = Vec2(x: 4, y: 2)
  var t = s
  print(s) // error: 's' was consumed
  print(t)
}
```

34

An important feature of the language is that it has a linear type system, meaning that all values must be used exactly once.

>> For example, linearity makes this program illegal, because we're using `s` twice.

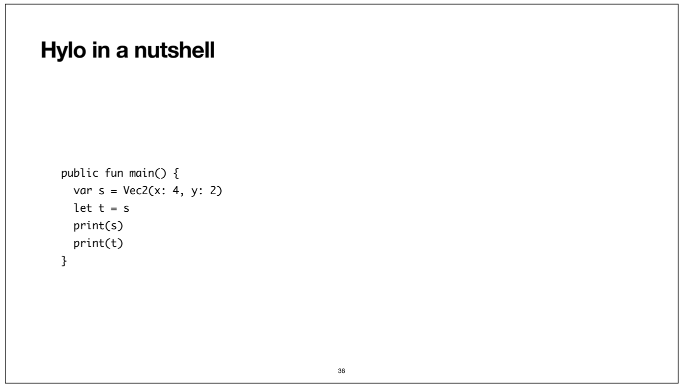

Hylo in a nutshell

```
public fun main() {
  var s = Vec2(x: 4, y: 2)
  var t = s.copy()
  print(s)
  print(t)
}
```

35

An easy fix is to make an explicit copy of that value, so that now `t` is its own, separate instance.

This solution is not very user-friendly, though. So that's where the observation we made earlier can come into play.


Hylo in a nutshell

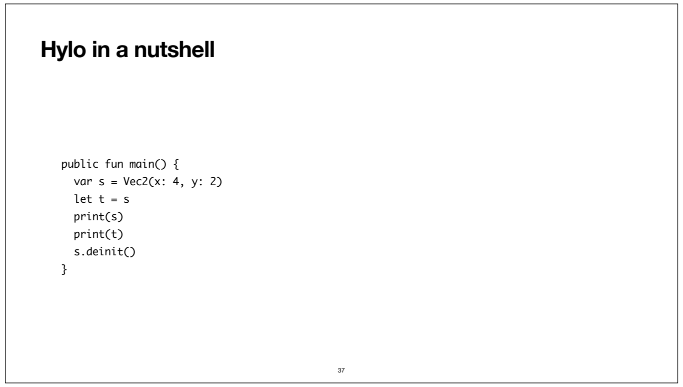
```
public fun main() {
  var s = Vec2(x: 4, y: 2)
  let t = s
  print(s)
  print(t)
}
```

36

First, we can relax the definition of "using" a value and make a distinction between reading it and consuming it. If we only need to inspect the contents of a value, then reading does not have to be a consuming operation.

>> In Hylo, we can represent a read access by declaring a `let` binding. In this program, `t` is simply a read access to the value of `s`. So now it is fine to print both `t` and `s`.

But what about consumption then? In Hylo, a value is consumed if it gets stored in another value, returned from a function, or deinitialized.



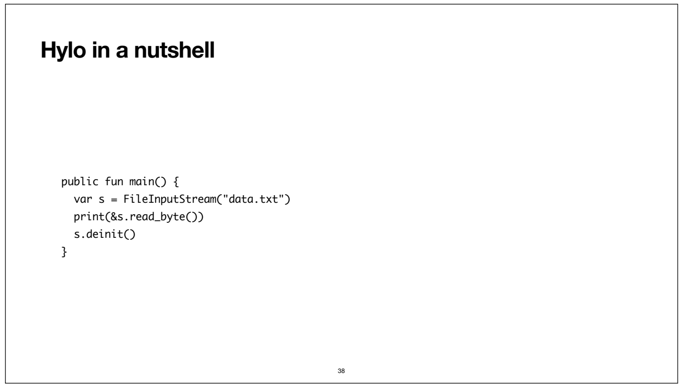
Hylo in a nutshell

```
public fun main() {
  var s = Vec2(x: 4, y: 2)
  let t = s
  print(s)
  print(t)
  s.deinit()
}
```

37

Deinitialization is typically inserted automatically by the compiler when it detects that a value has no further uses. That means Hylo does indeed has a linear type system rather than an affine one, because some form of consumption is guaranteed to eventually occur for all values. You can't just let values be defined and never used.

That is important because that means we can attach behavior to deinitialization to manage resources held by values.

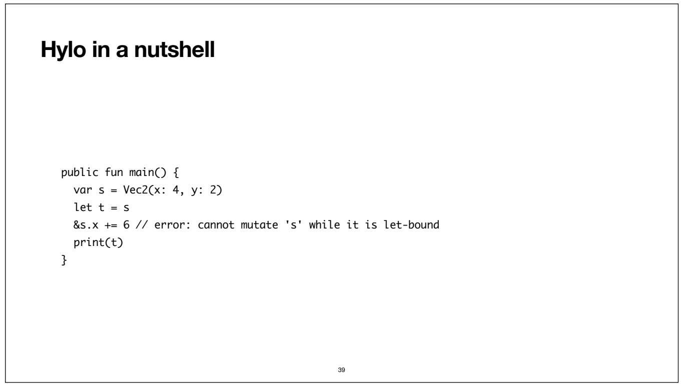


Hylo in a nutshell

```
public fun main() {
  var s = FileInputStream("data.txt")
  print(&s.read_byte())
  s.deinit()
}
```

38

For instance, we can imagine that `s` is not merely a vector but a handle on some file that we have opened. At the end of its lifetime, it is important that we release that handle and we can rely on destructor semantics to do that for us, automatically.



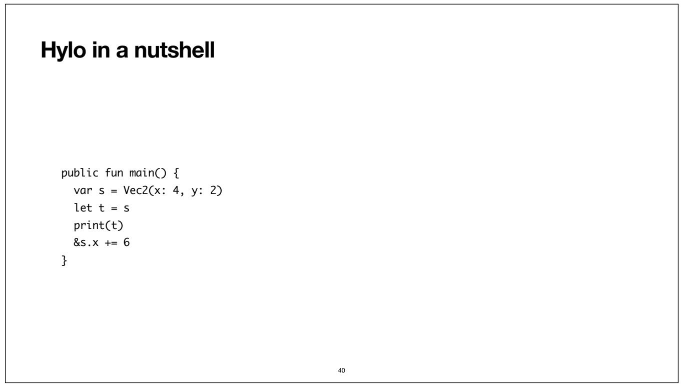
Hylo in a nutshell

```
public fun main() {
  var s = Vec2(x: 4, y: 2)
  let t = s
  &s.x += 6 // error: cannot mutate 's' while it is let-bound
  print(t)
}
```

39

At this point you may think that `t` is just as good as a reference. But like I said before, Hylo is all about value semantics. So when we define an access, the language guarantees that the accessed value can be thought as though it was completely independent, preventing operations that would violate this assumption.

>> Here, for example, it is illegal to modify `s` while `t` is alive, because that mutation would be observable through `t`, which is supposed to be independent.

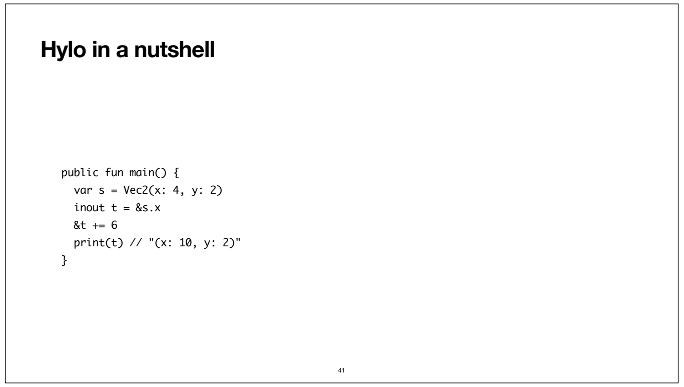


Hylo in a nutshell

```
public fun main() {
  var s = Vec2(x: 4, y: 2)
  let t = s
  print(t)
  &s.x += 6
}
```

40

Note that lifetimes are non-lexical, so swapping the two last statements of the function fixes the issue.

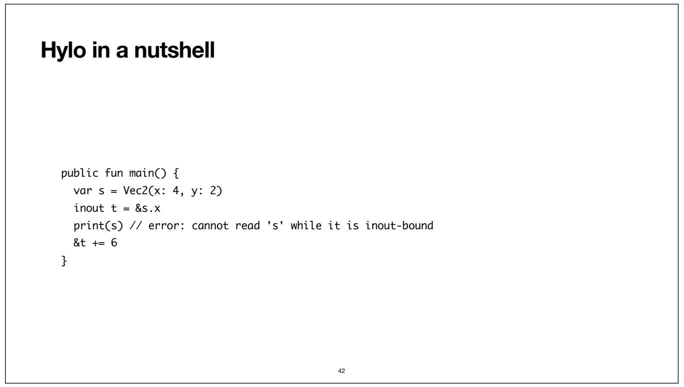


Hylo in a nutshell

```
public fun main() {
  var s = Vec2(x: 4, y: 2)
  inout t = &s.x
  &t += 6
  print(t) // "(x: 10, y: 2)"
}
```

We can create write accesses too, all of the same rules apply: for all intents and purposes, `t` can be thought of as an independent value.

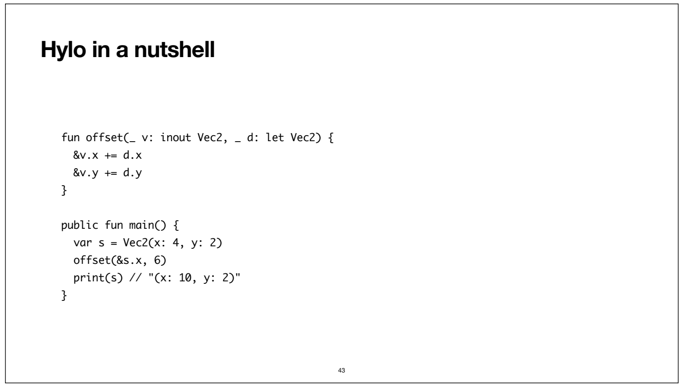
The fact that `t` has the write to modify its value adds a constraint, though. Mutation always requires exclusive access, so swapping the two last statements of the function makes the program illegal.



Hylo in a nutshell

```
public fun main() {
  var s = Vec2(x: 4, y: 2)
  inout t = &s.x
  print(s) // error: cannot read 's' while it is inout-bound
  &t += 6
}
```

Here, the compiler complains because we promised to `t` that it could behave as though it had exclusive access but then tried to break this premise by reading its value behind its back.

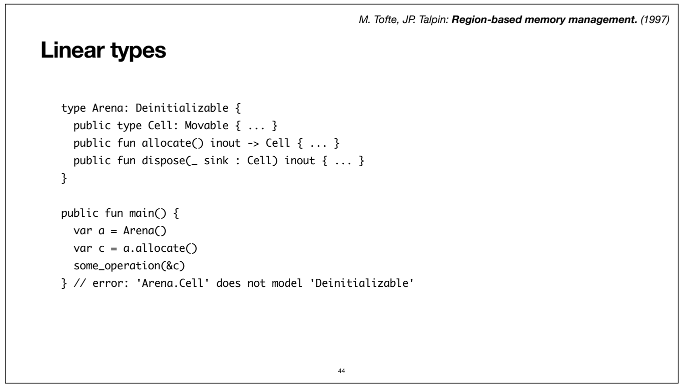


Hylo in a nutshell

```
fun offset(_ v: inout Vec2, _ d: let Vec2) {
  &v.x += d.x
  &v.y += d.y
}

public fun main() {
  var s = Vec2(x: 4, y: 2)
  offset(&s.x, 6)
  print(s) // "(x: 10, y: 2)"
}
```

43

Passing a parameter in Hylo can be seen as passing an access to some value defined in the caller. Here, for example, `offset` takes a write access on a value to modify and a read access on the offset to apply. When we call it, the same rules as the ones we saw before are applied.



M. Tofte, JP. Talpin: ***Region-based memory management.*** *(1997)*

Linear types

```
type Arena: Deinitializable {
  public type Cell: Movable { ... }
  public fun allocate() inout -> Cell { ... }
  public fun dispose(_ sink : Cell) inout { ... }
}

public fun main() {
  var a = Arena()
  var c = a.allocate()
  some_operation(&c)
} // error: 'Arena.Cell' does not model 'Deinitializable'
```

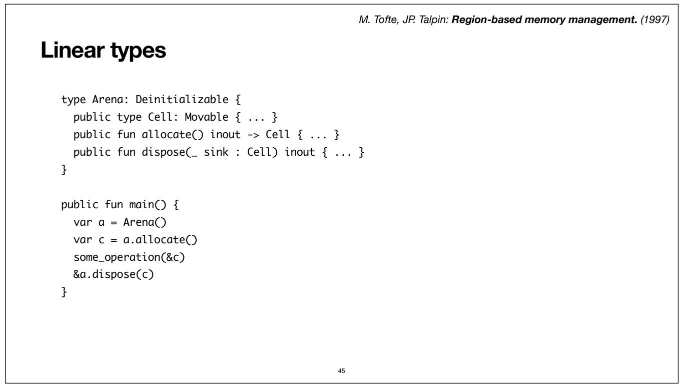
44

I promised earlier that I would show an example demonstrating the power of linearity.

>> Here, we're looking at a type representing some sort of memory arena, that is a region of memory in which we can allocate some objects. In real settings, one typically uses arenas to fine tune memory management.

>> The arena itself is deinitializable, meaning that the compiler is allowed to insert destructor calls on its own when instances are no longer used, as we already saw.
>> The cells, however, are only movable. That means we can pass them in and out of functions, store them in other values, but can't destroy them on our own.
>> Because of that, the compiler will refuse this program and complain in the main function that it doesn't know how to destroy the cell we have allocated

and never released.
>> Indeed, the only way to get rid of a cell is to call `dispose` on the arena. This method uses a `sink` passing convention, meaning that it *consumes* its arguments. And presumably, its implementation knows how to destroy a cell, having privilege access to its representation.



M. Tofte, JP. Talpin: **Region-based memory management.** (1997)

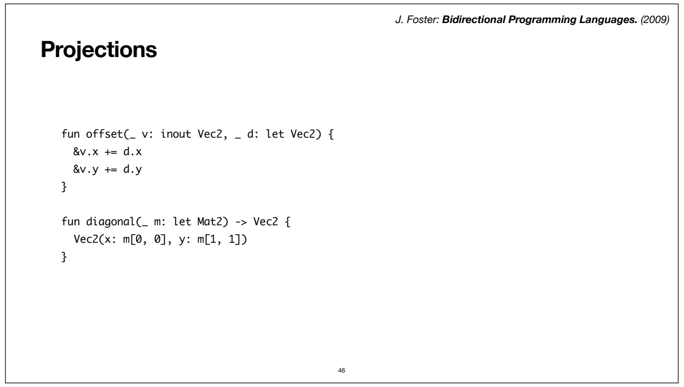
Linear types

```
type Arena: Deinitializable {
  public type Cell: Movable { ... }
  public fun allocate() inout -> Cell { ... }
  public fun dispose(_ sink : Cell) inout { ... }
}

public fun main() {
  var a = Arena()
  var c = a.allocate()
  some_operation(&c)
  &a.dispose(c)
}
```

45

There is no way to enforce such a protocol without a linear type system. At best, we could add some tests at run-time to catch violations.

Using linearity, we can create setups where invalid programs are systematically caught at compile-time.


J. Foster: **Bidirectional Programming Languages.** (2009)

Projections

```
fun offset(_ v: inout Vec2, _ d: let Vec2) {
  &v.x += d.x
  &v.y += d.y
}

fun diagonal(_ m: let Mat2) -> Vec2 {
  Vec2(x: m[0, 0], y: m[1, 1])
}
```

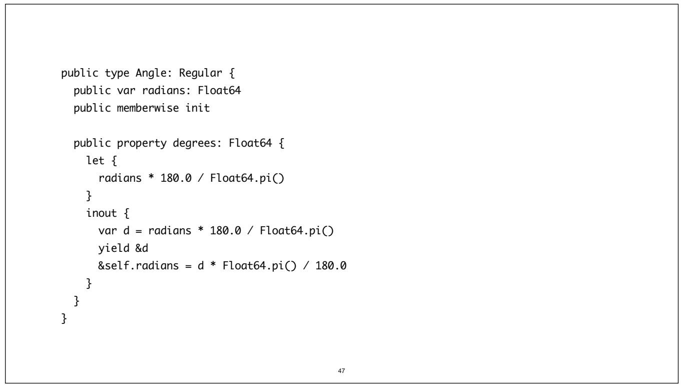
46

Finally I'd like to quickly talk about projections, which is a mechanism we can use to create accesses to parts of an object.

>> They are roughly equivalent to lenses in a pure functional setting.

>> We already encountered projections when we looked at the mutating implementation of `offset` in Hylo. Here, `v.x` is accessing the `x` component of the vector, leaving its other parts untouched. And as we discussed, the ability to select a part and mutate it in place is essential for efficiency.

>> Of course, we may also want to select parts "dynamically", for instance to access a specific element in an array or a matrix. The same idea applies: we want to access only a part of value and leave the others untouched. The only difference is that the specific part that we access is determined at run-time.



```
public type Angle: Regular {
  public var radians: Float64
  public memberwise init

  public property degrees: Float64 {
    let {
      radians * 180.0 / Float64.pi()
    }
    inout {
      var d = radians * 180.0 / Float64.pi()
      yield &d
      &self.radians = d * Float64.pi() / 180.0
    }
  }
}
```

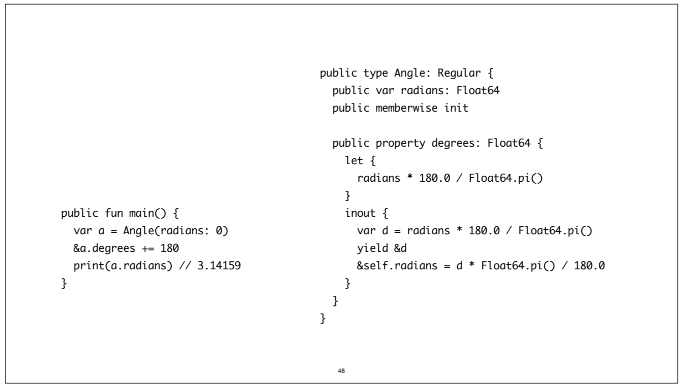
47

We may also want to manipulate values that are "notionally" part of some whole, but don't actually have a memory representation.

Imagine for instance that we define a data structure abstracting over angles. If you compose different libraries, it sometimes happen that one particular API deals in radians while another deals in degrees. But an angle is just a concept independent of the unit we use to measure it. So it might be helpful to abstract over this detail and have a way to just *project* the value of the angle with the right unit on demand. That's what this type is doing.

>> Though the value of the angle is stored in radians,
>> I have defined a projection that let's us access the same value but in degrees.
>> There are two implementations, for projecting immutable and mutable

values, respectively, and the compiler will select the right one depending on context.



```
public fun main() {
  var a = Angle(radians: 0)
  &a.degrees += 180
  print(a.radians) // 3.14159
}
```

```
public type Angle: Regular {
  public var radians: Float64
  public memberwise init

  public property degrees: Float64 {
    let {
      radians * 180.0 / Float64.pi()
    }
    inout {
      var d = radians * 180.0 / Float64.pi()
      yield &d
      &self.radians = d * Float64.pi() / 180.0
    }
  }
}
```

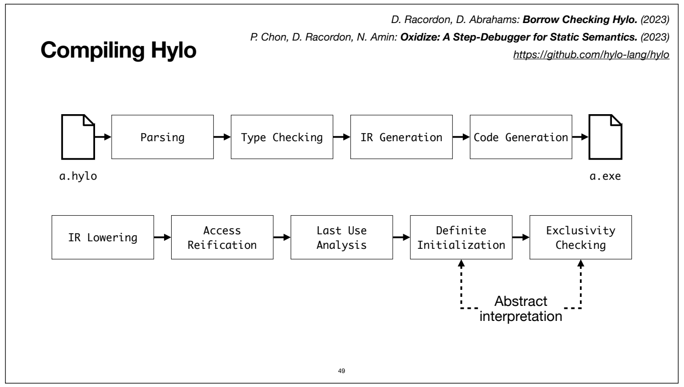
48

So here, for example, we're using the mutating projection because we're binding the part mutably.

>> The projection is made using inversion of control. When we're accessing the angle's `degrees`, control flows to the projection so that we can compute the value to project.
>> Once we reach the yield statement, control flows back to the caller that can now use the projected value.
>> When it's done, control flows back to the projection so that we can apply the mutation to the actual in-memory representation of the whole.

The same semantics could be achieved using callbacks, but using inversion of the control makes the whole process completely transparent to the user. Further, we do not need to rely on references, which is the road one would take to implement lenses in a language with reference semantics.

>> That has two advantages: First, note that we do not need to augment the type system with any particular type to represent references. The projection is producing a `Float64` like any other `Float64`. There is no dereferencing, boxing, method delegation, or any other trick involved. Second, note that using references would require the projected value to exist in memory, whereas here we can just synthesize anything we want *and* have the guarantee that control will come back so that we can restore the whole's invariants.



Compiling Hylo

*D. Racordon, D. Abrahams: **Borrow Checking Hylo.** (2023)*
*P. Chon, D. Racordon, N. Amin: **Oxidize: A Step-Debugger for Static Semantics.** (2023)*
https://github.com/hylo-lang/hylo

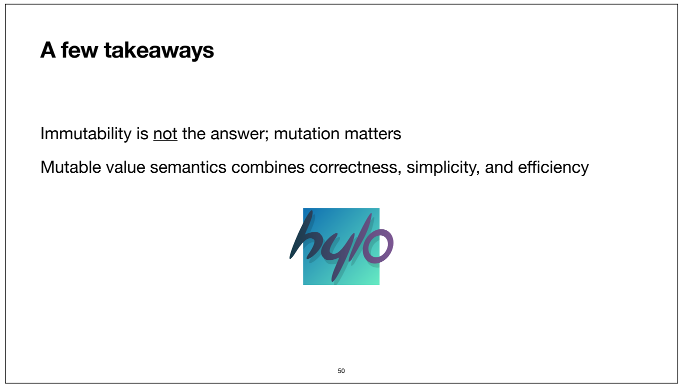
49

The compilation pipeline of Hylo is pretty traditional. The interesting part of the secret sauce is IR generation, which is where the access rules are checked. Note that, perhaps surprisingly, that means "borrow checking" isn't part of type checking.

>> IR Generation is decomposed into a few steps which operate on an intermediate representation, called Hylo IR. Let's look at them in more details.
>> The analysis we perform on the IR are pretty standard, but perhaps interestingly we're doing definite initialization and exclusivity checking, that is the two phases that enforce linearity and uniqueness using an abstract interpreter. That is in contrast to Rust, for example, which uses a constraint solver. The advantage of our approach is that we can just implement the formal semantics of our type system to get an interpreter, which also gives us a way to "step" through the formal semantics for debugging the compiler.

>> That's an idea I also explored in the context of Rust with a student from MIT.
>> The code is open sourced. You can find everything on GitHub.

Abstract interpretation is intraprocedural; it's not whole program.



A few takeaways

Immutability is not the answer; mutation matters

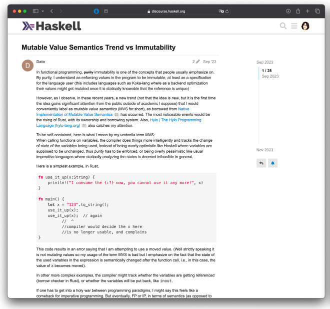
Mutable value semantics combines correctness, simplicity, and efficiency

50

The first thing I'd like to stress once again is that immutability is *not* the answer to the problems caused by reference semantics, because mutation matters. Functional programming is a beautiful paradigm, but it's not a catch-all solution. Imperative programming has its place at the table because it's the way we naturally describe some algorithms, and it's also the way real computers work.

>> The second thing I'd like you to remember from this lecture is mutable value semantics offers a much more compelling solution. While it does not provide equational reasoning, it upholds local reasoning without loss of efficiency.

>> And the last thing I'd like you to remember is Hylo, which I believe truly believe to be an interesting data point in the design space as a way to to bring correctness, simplicity, and efficiency together. In particular, I think it builds a very nice bridge between imperative and functional programming.



Mutable Value Semantics Trend vs Immutability

Dato 2 Sep '23

In functional programming, ~~purity~~ immutability is one of the concepts that people usually emphasize on. By purity, I understand as enforcing values in the program to be immutable, at least as a specification for the language user (this includes languages such as Koka-lang where as a backend optimization their values might get mutated once it is statically knowable that the reference is unique)

However, as I observe, in these recent years, a new trend (not that the idea is new, but it is the first time the idea gains significant attention from the public outside of academic I suppose) that I would conveniently label as *mutable value semantics* (MVS for short), as borrowed from Native Implementation of Mutable Value Semantics 22 has occurred. The most noticeable events would be the rising of Rust, with its ownership and borrowing system. Also, Hylo | The Hylo Programming Language (hylo-lang.org) 20 also catches my attention.

To be self-contained, here is what I mean by my umbrella term MVS:
When calling functions on variables, the compiler does things more intelligently and tracks the change of state of the variables being used, instead of being overly optimistic like Haskell where variables are supposed to be unchanged, thus purity has to be enforced, or being overly pessimistic like usual imperative languages where statically analyzing the states is deemed infeasible in general.

Here is a simplest example, in Rust,

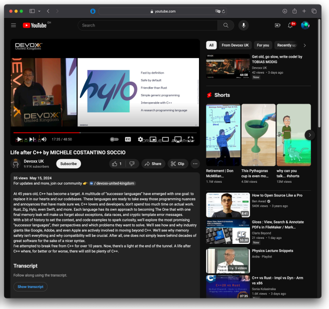
```
fn use_it_up(x:String) {
    println!("I consume the {:?} now, you cannot use it any more!", x)
}

fn main() {
    let x = "123".to_string();
    use_it_up(x);
    use_it_up(x);  // again
          //  ^
          //compiler would decide the x here
          //is no longer usable, and complains
}
```

This code results in an error saying that I am attempting to use a moved value. (Well strictly speaking it is not *mutating* values so my usage of the term MVS is bad but I emphasize on the fact that the state of the used variables in the expression is semantically changed after the function call, i.e., in this case, the value of x becomes moved).

In other more complex examples, the compiler might track whether the variables are getting referenced (borrow checker in Rust), or whether the variables will be put back, like inout.

If one has to get into a holy war between programming paradigms, I might say this feels like a comeback for imperative programming. But eventually, FP or IP, in terms of semantics (as opposed to

Sep 2023
1 / 28
Sep 2023
Nov 2023


youtube.com
YouTube
Search
DEVOXX United Kingdom
Fast by definition
Safe by default
Friendlier than Rust
Simple generic programming
Interoperable with C++
A research programming language
17:35 / 48:50
All
From Devoxx UK
For you
Get old, go slow, write code! by TOBIAS MODIG
Shorts
Retirement | Don McMillan...
This Pythagoras cup is even mo...
why can you talk... #shorts
How to Open Source Like a Pro

Life after C++ by MICHELE COSTANTINO SOCCIO

Devoxx UK 9.91K subscribers

35 views May 15, 2024
For updates and more, join our community 👉 / devoxx-united-kingdom

At 45 years old, C++ has become a target. A multitude of "successor languages" have emerged with one goal: to replace it in our hearts and our codebases. These languages are ready to take away those programming nuances and annoyances that have made sure we, C++ lovers and developers, don't spend too much time on actual work. Rust, Zig, Hylo, even Swift, and more. Each language has its own approach to becoming The One that with one final memory leak will make us forget about exceptions, data races, and cryptic template error messages.
With a bit of history to set the context, and code examples to spark curiosity, we'll explore the most promising "successor languages", their perspectives and which problems they want to solve. We'll see how and why industry giants like Google, Adobe, and even Apple are actively involved in moving beyond C++. We'll see why memory safety isn't everything and why compatibility will be crucial. After all, one does not simply leave behind decades of great software for the sake of a nicer syntax.
I've attempted to break free from C++ for over 10 years. Now, there's a light at the end of the tunnel. A life after C++ where, for better or for worse, there will still be plenty of C++.

Transcript

Follow along using the transcript.

Show transcript

And in fact, it's interesting to see that Hylo and MVS have caught the attention of both communities.

---

https://hylo-lang.org