Mutation is not evil, actually

Dimi Racordon - LAMP, EPFL

10 January 2024

Hello everyone, my name is Dimi. I'm a post-doc at EPFL and today I'd like to talk about mutation, why it's necessary, why it's not evil, and how I think we can embrace it safely. So let's get started.

To motivate my thesis, let's talk about this gentleman for bit. This is Eratosthenes of Cyrene, a Greek mathematician who's credited for having discovered a pretty neat algorithm to compute lists of prime numbers.

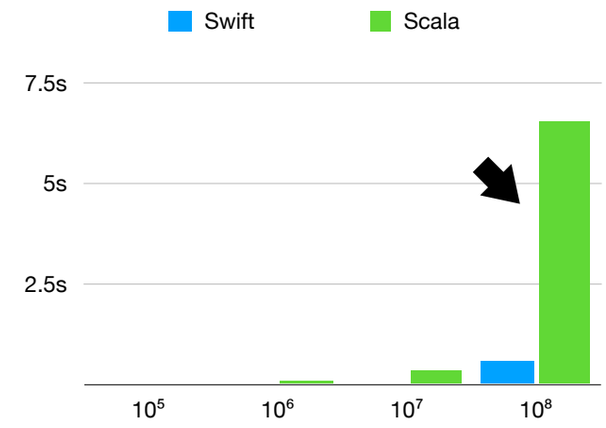
The spirit of the algorithm is to start with a sequence of numbers and to cross off those that are multiple of known primes, starting with 2.

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

def sieve(h: List[Int], t: List[Int]): List[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)
```


If you look online, you might stumble upon an implementation looking like this Scala program.

>> The bulk of the work is done by this function, which takes two ordered lists. The first is a sequence of prime numbers 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.



Here are the results of a benchmark I ran to compare the Scala implementation we just saw and a competing imperative version written in Swift. The x-axis shows the length of the input and the y-axis is the time it took identify prime numbers.

As we can see, the Scala version doesn't scale very well, pun intended.

>> It took Scala close to a 8 seconds to process an input of 100 million elements, whereas the same computation took Swift half a second. I don't have numbers for larger inputs because I prepared the slides yesterday and the computation is still running back at home.

Also note that VM warmup has nothing to do with these numbers. I used Scala native. So what do we make of that?

*Melissa E. O'Neill: **The Genuine Sieve of Eratosthenes**. J. Funct. Program. 19(1): 95-106 (2009)*

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

def sieve(h: List[Int], t: List[Int]): List[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)
```


The good news is that I didn't prove Scala was worse than Swift.

The problem is more that this program doesn't actually implement the sieve of Eratosthenes. It's a half-clever implementation of a much more naive algorithm called *trial division*, which identifies prime numbers by testing whether they are divisible by smaller prime numbers. Melissa O'Neill, who wrote a really beautiful paper about the subject, calls this version the "unfaithful sieve".

So maybe before we look at a better implementation, let's review the description of the algorithm that we actually want to implement.

Wikipedia

1. *Create a list of consecutive integers from 2 through n: (2, 3, 4, ..., n).*
2. *Initially, let p equal 2, the smallest prime number.*
3. *Enumerate the multiples of p by counting in increments of p from 2p to n, and mark them in the list*
4. *Find the smallest number in the list greater than p that is not marked. If there was no such number, stop. Otherwise, let p now equal this new number (which is the next prime), and repeat from step 3.*
5. *When the algorithm terminates, the numbers remaining not marked in the list are all the primes below n.*

According to Wikipedia, the sieve of Eratosthenes can be described as follows.

read the algorithm

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

*Melissa E. O'Neill: **The Genuine Sieve of Eratosthenes**. J. Funct. Program. 19(1): 95-106 (2009)*

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

def sieve(h: List[Int], t: List[Int]): List[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)
```


Here, the filter operates on all elements that are not divisible by the primes smaller than `p`. So while the speed of the Eratosthenes's algorithms depends on the number of unique primes that are factors of the numbers it examines, the "unfaithful" algorithm depends on the numbers that aren't factors. I really invite you to read O'Neill's paper for a more detailed about that. My personal take away, and 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. But even if that were true, 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.

Besides, optimizers are in fact not able to fix everything. I did write the actual sieve of Eratosthenes in a functional way, but it still lagged pretty far behind the imperative version. It's only after I went fully imperative that I could finally get in the same ballpark.

```
def primesUntil2(n: Int): List[Int] =
  val s = ArraySeq.fill(n)(true)
    .updated(0, false)
    .updated(1, false)
  val t = loop1(2, s)
  (2 until n).collect({ case i if t(i) => i }).toList

def loop1(x: Int, s: ArraySeq[Boolean]): ArraySeq[Boolean] =
  if (x * x) >= s.length
  then s
  else loop1(x + 1, if s(x) then loop2(x * x, x, s) else s)

def loop2(y: Int, p: Int, s: ArraySeq[Boolean]): ArraySeq[Boolean] =
  if y >= s.length
  then s
  else loop2(y + p, p, s.updated(y, false))
```


For what it's worth, here's the implementation I wrote. I should say first that I find this version ridiculously difficult to read. But at the very least it's implementing the right algorithm.

>> Specifically, this loop is crossing off the the multiple of each newly discovered prime number by counting in increments. So we're no longer iterating all elements not divisible by the primes smaller than `p`.

Sadly we're still not out of the weeds, because this program is still very slow. In fact, it does barely better than the "unfaithful" sieves for the numbers that I tested. So let's understand why.

```
def primesUntil2(n: Int): List[Int] =
  val s = ArraySeq.fill(n)(true)
    .updated(0, false)
    .updated(1, false)
  val t = loop1(2, s)
  (2 until n).collect({ case i if t(i) => i }).toList

def loop1(x: Int, s: ArraySeq[Boolean]): ArraySeq[Boolean] =
  if (x * x) >= s.length
  then s
  else loop1(x + 1, if s(x) then loop2(x * x, x, s) else s)

def loop2(y: Int, p: Int, s: ArraySeq[Boolean]): ArraySeq[Boolean] =
  if y >= s.length
  then s
  else loop2(y + p, p, s.updated(y, false))
```


The method `updated` on `ArraySeq` creates a copy of the sequence, updated at the given index. It effectively lets us write a "functional update", which is a pattern one can use to emulate part-wise mutation with reassignment. We can use this pattern to write a saner version of that code, at least in my opinion:

```
import scala.collection.immutable.ArraySeq

def primesUntil(n: Int): List[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))
    for (y <- x * x until n by x)
      s = s.updated(y, false)
  (2 until n).collect({ case i if s(i) => i }).toList
```


Here I used imperative-looking loops instead of tail-recursive functions to express iteration, at the price of local `var`s.

>> These `var`s are innocuous, though, because we're not using any mutable data structure. ArraySeq is an immutable collection, and so we can't accidentally mutate its contents with a sneaky side-effect. I'll come back to that point in a minute. For now, let's just observe that our code got arguably closer to the English description of the algorithm.

>> In particular, it's now much more obvious that the inner loop does indeed iterate by increments of the discovered prime number. But this function is still too slow.
>> That shouldn't be surprising because it's in fact the exact same implementation as the previous one, only written with functional updates. But while this pattern may help us emulate mutation, it can't reproduce the efficiency of an actual part-wise, in-place update.

So eventually I couldn't get my Scala implementation to even play in the same ballpark as the Swift one until I fully embraced imperative programming.

```scala
import scala.collection.mutable.ArraySeq

def primesUntil(n: Int): List[Int] =
  var s = mutable.ArraySeq.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 }).toList
```


This implementation is about a 100 times faster than the previous one for an input of a 100 million. How come? Well I'm not particularly knowledgeable about details of Scala's immutable collections, but I know for a fact that it's typically impossible to beat the efficiency of an update like this one. >>

This is a part-wise, in-place mutation. It's really not obvious for an optimizer to understand how an arbitrary functional update should translate to that because constructing a whole new instance of a sophisticated data structure involves a lot of arbitrary code. And figuring out that this code is actually useless is a hard thing to do.

Hopefully this example is enough to demonstrate the problem. Scala has a pretty decent optimizer and yet it was unable to handle the situation. But if you're still not convinced, know that there are even trickier use cases and I'm happy to talk about them later if you want.

Is mutation a necessary evil?

Let's assume you *are* convinced that in-place updates matter but still believe that they should be relegated to small pockets of highly optimized code, because mutation is essentially evil. That wouldn't be an outrageous position to have, especially in a forum about software verification. After all, mutation makes everything harder to reason about, right?

But wait a minute. If functional updates are notionally equivalent to in-place part-wise mutation, then what exactly is causing the headaches?

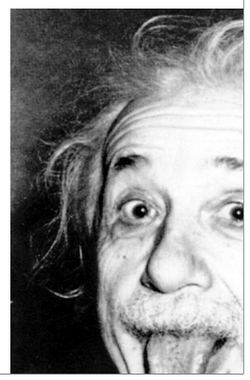
```
class Vec2(var x: Int, var y: Int):

  def offset(delta: Vec2) =
    this.x += delta.x
    this.y += delta.y

  def offsetTwice(delta: Vec2) =
    this.offset(delta)
    this.offset(delta)

val v = Vec2(1, 2)
v.offsetTwice(v)
println(s"(${v.x}, ${v.y})") // (4, 8)
```


Here is an example of the kind of program that quickly convinces everyone that mutation is evil. We have a class representing 2-D vectors as coordinates relative to an origin and a pair of methods.

>> The first offsets an instance by some delta, and the second does it twice.
>> The problem occurs when we apply `offsetTwice`. The first call to `offset` may modify the value of our delta behind our back. So the second call to `offset` may end up operating on a different argument.
>> The result, as we observe with the final print statement, is that `offsetTwice` didn't produce the expected result.
>> Note also that the `val` didn't help us, as it only prevents "shallow" mutation. But since the class is declared with variable fields, "deep" mutation is still allowed.

So what in this example makes mutation evil that didn't in the sieve of Eratosthenes?

```
class Vec2(var x: Int, var y: Int):

  def offset(delta: Vec2) =
    this.x += delta.x
    this.y += delta.y

  def offsetTwice(delta: Vec2) =
    this.offset(delta)
    this.offset(delta)

val v = Vec2(1, 2)
v.offsetTwice(v)
println(s"(${v.x}, ${v.y})") // (4, 8)
```

The answer is aliasing! When aliasing is on the table, a mutation of a variable may have an unforeseen effect on another variable.

There's a constellation of problems that step from these unintended mutations. Some people, myself included, call that spooky action at a distance, with apologies to Einstein

```
struct Vec2(var x: Int, var y: Int):

  def offset(delta: Vec2) =
    this.x += delta.x
    this.y += delta.y

  def offsetTwice(delta: Vec2) =
    this.offset(delta)
    this.offset(delta)

val v = Vec2(1, 2)
v.offsetTwice(v)
println(s"(${v.x}, ${v.y})") // (3, 6)
```


So imagine a world where aggregates like Vec2 behave like values, i.e., like a plain integer. In such a world, passing a value as argument to a function would create a copy, at least notionally, and therefore a mutation of the receiver could not possibly cause unintended mutation of the argument.

Language for systems programming

Fast by definition

Safe by default

Extensive support for generic programming

Well, that is the world in which I live when I'm not working with Scala. For the past two years I've been developing Hylo with a group of amazing people coming from the industry.

Hylo is a programming language for high-level systems programming.

It 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. Further, the language has very transparent semantics, making it relatively easy to predict allocation and indirection 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.

Finally, Hylo has an extensive support for generic programming based on type classes.

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


An important feature of the language is that it has a linear type system, meaning that all values must be used exactly once. Linear type systems are great at modeling resources, like memory, or even capabilities, like access to IO. But they are also notoriously difficult to use.

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

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


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. Fortunately, there are a couple of other things we can do to make the user's life easier.

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


The first is to relax the definition of "using" a value and make a distinction between reading it and consuming it. Most functions only need to read the contents of a value and do not care if that value might be read again in the future. So a first observation we can make is that accessing a value for reading is not 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.

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


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, like memory. We'll see an example later.

```
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)
}
```


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.

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


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

```
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, and 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.

```
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.

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

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


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.

```
type FileOutputStream {
  var h: Pointer<FILE>
  public init(path: String) {
    &h = fopen(path.c_string(), "w+".c_string())
  }
  public fun close() sink { fclose(h) }
}

public fun main() {
  let m = ["Hello, ", "World!"]
  var o = FileOutputStream(path: "/tmp/a.out")
  if Bool.random() {
    m.map((s) => &o.write(s))
  }
} // error: 'FileOutputStream' does not model 'Deinitializable'
```


It's quite amazing how powerful linearity can be and how convenient we can make it by using temporary let or inout accesses. I'll just give one example.

>> Here's a bare-bones implementation of a file output stream that just wraps low-level functions from libc.
>> We can create instances of this type and use them just like we would use a regular object in Scala, including in control-flow structure, without any particular ceremony.
>> And then, thanks to linearity, we can enforce protocols that are typically very difficult to check in GCed languages.

Here, the only way to get rid of a stream instance in this program is to call `close`. Otherwise the linearity constraint will prevent us from closing the scope in which the owner of the instance lives.

So let's address the elephant in the room, because I'm sure most of you have figured out Hylo is playing in the same ballpark as Rust. I also have no doubt this audience is at least heard of the language, so I'll skip the introduction and move directly to the salient differences.

```
struct Matrix2 { data: [f64; 4] }

impl Matrix2 {
  fn zero() -> Matrix2 { Matrix2 { data: [0.0; 4] } }
  fn row(&mut self, row: usize) -> &mut [f64] {
    let n = 2 * row;
    &mut self.data[n .. (n + 2)]
  }
}

fn main() {
  let mut m = Matrix2::zero();
  let r = m.row(1);
  r[0] = 4.2;
  println!("{}", m.data[2]);
 }
```


Here's a simple Rust program.

>> It defines a 2-D matrix with a row-major representation.
>> It also defines a method to access the contents of a single row, identified by its index in the matrix. In Rust, one uses references to implement this operation.
>> That lets us write efficient in-place updates later on, such as this one.

The type system guarantees that mutation requires uniqueness, so I don't have to worry about spooky actions in this program. But this type system comes at the cost of a sophisticated annotation system. The actual signature of the function looks like this. >>

```
struct Matrix2 { data: [f64; 4] }

impl Matrix2 {
  fn zero() -> Matrix2 { Matrix2 { data: [0.0; 4] } }
  fn row<'a>(&'a mut self, row: usize) -> &'a mut [f64] {
    let n = 2 * row;
    &mut self.data[n .. (n + 2)]
  }
}

fn main() {
  let mut m = Matrix2::zero();
  let r = m.row(1);
  (*r)[0] = 4.2;
  println!("{}", m.data[2]);
 }
```


The "'a" is a lifetime parameter and it is used by the compiler to track the lifetime and ownership status of the matrix that is passed as receiver of the method. Further, the value of `r` is actually a reference. And so without sugar, we'd have to write the assignment like this. >>

All of this additional syntax is not necessary in the original program, so you may wonder why I'm making a big deal about it. Well, the reason is that the user must ultimately understand how the system works, regardless of whether or not sugar can be used to remove recurring boilerplate. So if we really want to understand this program, we still have to think about lifetime annotations and dereferencing.

Besides, this additional complexity can not always be hidden under the rug, especially in generic contexts. At the very least, the the complexity *will* eventually leak in error messages.

```
type Matrix2: Regular {
  var data: Float64[4]
  memberwise init
  static fun zero() -> Self { Matrix2(data: .zero()) }

  fun with_row<T>(_ i: Int, _ action: (inout Slice<Float64[4]>) -> T) inout {
    let n = 2 * row
    action(&data[n ..< (n + 2)])
  }
}

public fun main() {
  var m = Matrix2.zero()
  m.with_row(1, fun (r) {
    &r[0] = 4.2
  })
  print(m.data[2])
}
```


The way we can deal with this situation in a world without references is to use inversion of control. Rather than extracting a reference from a data structure, we can simply ask the data structure to apply the operation we want done on its part. That way, we can trivially guarantee that the data structure is able to track the lifetime of its parts and maintain its invariants.

And because working with higher order functions all over the place brings poor ergonomics, we can capture this pattern in a language construct. This trick's been implemented in Swift. I just made it more powerful in Hylo.

```
type Matrix2: Regular {
  var data: Float64[4]
  memberwise init
  static fun zero() -> Self { Matrix2(data: .zero()) }

  subscript row(_ i: Int): Slice<Float64[2]> inout {
    let n = 2 * row
    yield &data[n ..< (n + 2)]
  }
}

public fun main() {
  var m = Matrix2.zero()
  inout r = &m.row[1]
  &r[0] = 4.2

  print(m.data[2])
}
```


I replaced the higher order function with a subscript, which is a construct that essentially does the same thing. When we `yield` a value, we're essentially projecting it into the caller, which can use it the way it wants.

>> That is what lets us write the mutation very naturally, as we would do in Rust for example.
>> When he projection ends, control goes back to the subscript which may perform additional operations before giving control back.

It's a really powerful trick. In fact, it is even more expressive than references because a subscript can allocate memory to synthesize values on demand. That's a thing references can't do.

I'd love to talk more about the power of subscripts, but I'm running out of time. So please come talk to me after if you're interested and I'll show you other examples.

Mutation is necessary!

Mutation isn't evil, sharing mutable state is!

Linear types can change the world!

Mutation is necessary. Pure functional programming is beautiful, but it's not a panacea. There are algorithms whose descriptions are intrinsically imperative, like the sieve of Eratosthenes. While a correct and efficient implementation is likely possible to write, it is certainly simpler to write the imperative one.

Mutation isn't evil and it can't be if we admit unless we're ready to claim that functional updates are too. If there exists a notional translation from every functional update to an in-place update, then clearly the problem isn't about the mutation. As we briefly discussed, the real culprit is shared mutable state.

That leads us to linear types and uniqueness. If we're able to guarantee uniqueness of every mutable variable, at the point of mutation, then we can recover almost all reasoning capabilities that we enjoy with pure functional programming. Side effects do make evaluation order significant, but I thank that is a proper that most developers are willing to accept, including those leaning on the functional side.

It is interesting to note all systems with unrestricted aliasing ends up advocating for immutability. That's because without an efficient way to track uniqueness, the only only way to avoid spooky action is to ban mutation. But linear types provide a more permissive cure while also opening interesting opportunities to ask more about our type systems, like enforcing protocols.

VIMPL 2024

Call for Papers

VIMPL (Value Independence in Modern Programming Languages) intends to welcome a wide range of topics and perspectives relevant to value independence. We will accept three kinds of submissions:

- Research papers (10 pages, excluding references) documenting past or ongoing effort to use and/or leverage value independence in new or existing programming languages.
- Extended abstracts (2 pages) summarizing the design and implementation of applications or libraries centered around value independence.
- Position papers (2 pages) presenting the authors' opinion on a topic related to the workshop.

Topics of interest include, but are not limited to:

- Programming languages designed to support value independence;
- Inclusion of value types in reference-oriented languages (e.g., Java, Python, or Javascript);
- Aliasing restriction mechanisms designed to support value independence in reference-oriented languages (e.g., ownership and uniqueness);
- Memory representation and garbage collection of value types;
- Optimization strategies based on value independence;
- Empirical studies on the use, usability, and/or performance of mechanisms to promote value independence.

Papers should strictly adhere to the ACM `acmart` format v1.87 or newer and be submitted as PDF: https://www.acm.org/publications/proceedings-template. Please use the following LaTeX class configuration: `\documentclass[sigconf,screen]{acmart}`.

Important Dates AoE (UTC-12h)

Mon 22 Jan 2024
Submission Deadline

Submission Link

https://easychair.org/conferences/?conf=vimpl2024

And finally a bit of shameless advertising. I'm organizing a workshop on value semantics in March, which will be co-located with <Programming24>. If you found any of what I said remotely interesting, you may also like the topics of the workshop.

We're still looking for submissions. We're accepting research papers, extended abstracts, and talk proposals. So don't hesitate to if you or someone in your circles is working on anything related to the use of values for program verification, compilation, optimization, or interpretation, even if it's only about a burgeoning idea or some early results.