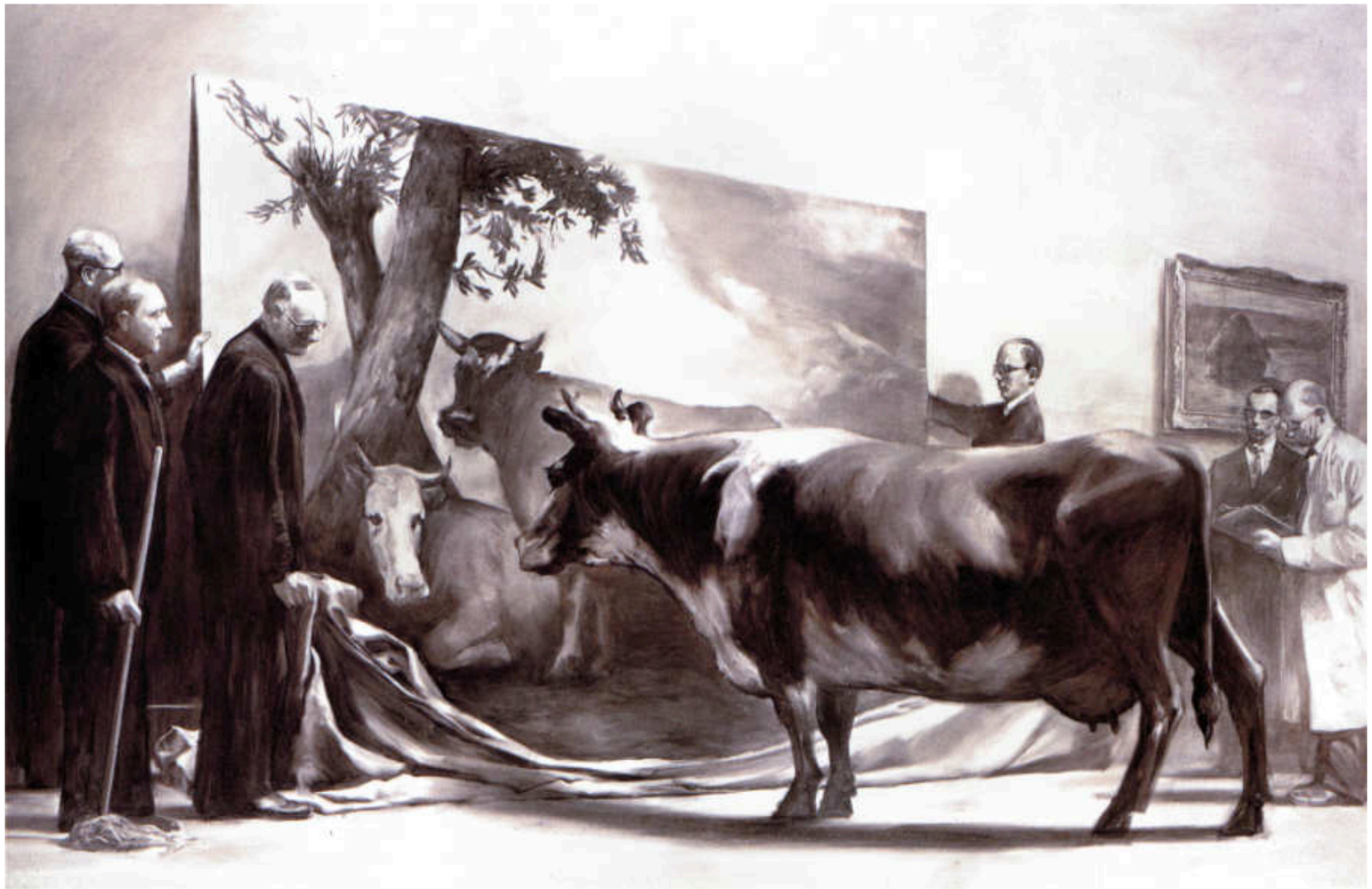
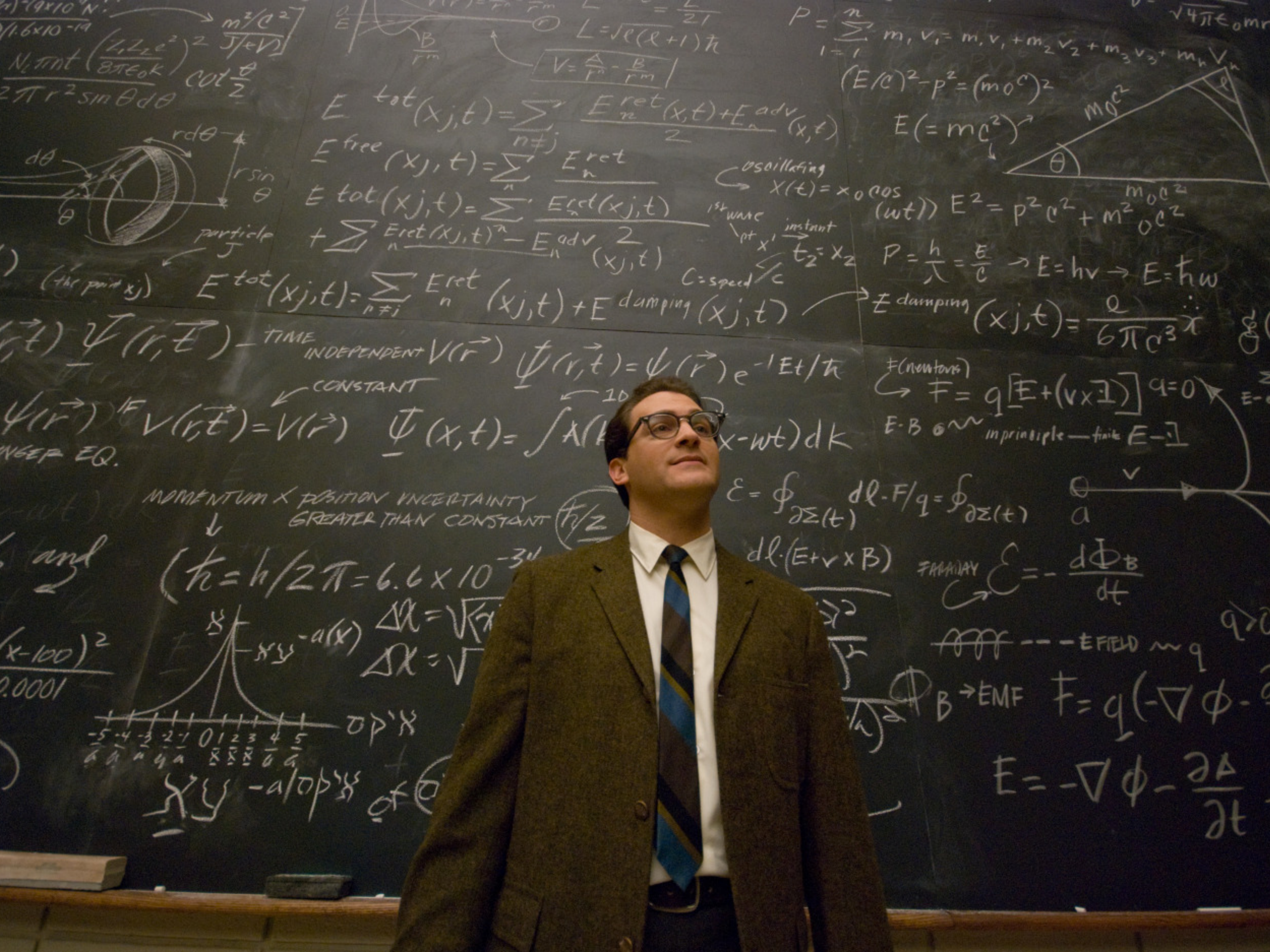
Separation Logic



Derek Dreyer
MPI for Software Systems
Cornell/Maryland/Max Planck Summer School
Saarbrücken, August 2017

Separation Logic in One Slide

- Extension of **Hoare logic** for reasoning modularly about pointer-manipulating code
- O'Hearn, Reynolds, et al. (~2000)
  - One of the most fundamental advances in program verification in the past 20 years
  - **"Concurrent separation logic"** (2007) won the **2016 Gödel Prize** (Nobel prize in theory)
  - Underpinning of all my recent research



MOMENTUM × POSITION UNCERTAINTY
GREATER THAN CONSTANT
TIME
INDEPENDENT
CONSTANT
oscillating
instant
C=speed
E damping
F(newtons)
inprinciple
FARADAY
E FIELD
EMF
particle

RustBelt:

Logical Foundations for the Future of Safe Systems Programming

Derek Dreyer
MPI for Software Systems

Cornell/Maryland/MPI Summer School
Saarbrücken, August 2017

Goal of These Lectures

- Tell you about a major ongoing research project in PL/verification (**RustBelt**)
- Teach you something about a cutting-edge language (**Rust**), a cutting-edge separation logic (**Iris**), and how they are connected

About Me

- Born in NYC, grew up in Great Neck
- Undergrad in Math/CS at NYU (1993-1996)
- PhD in CS at CMU (1997-2004)
- Postdoc at TTI-Chicago (2005-2007)
- MPI for Software Systems (2008-present)

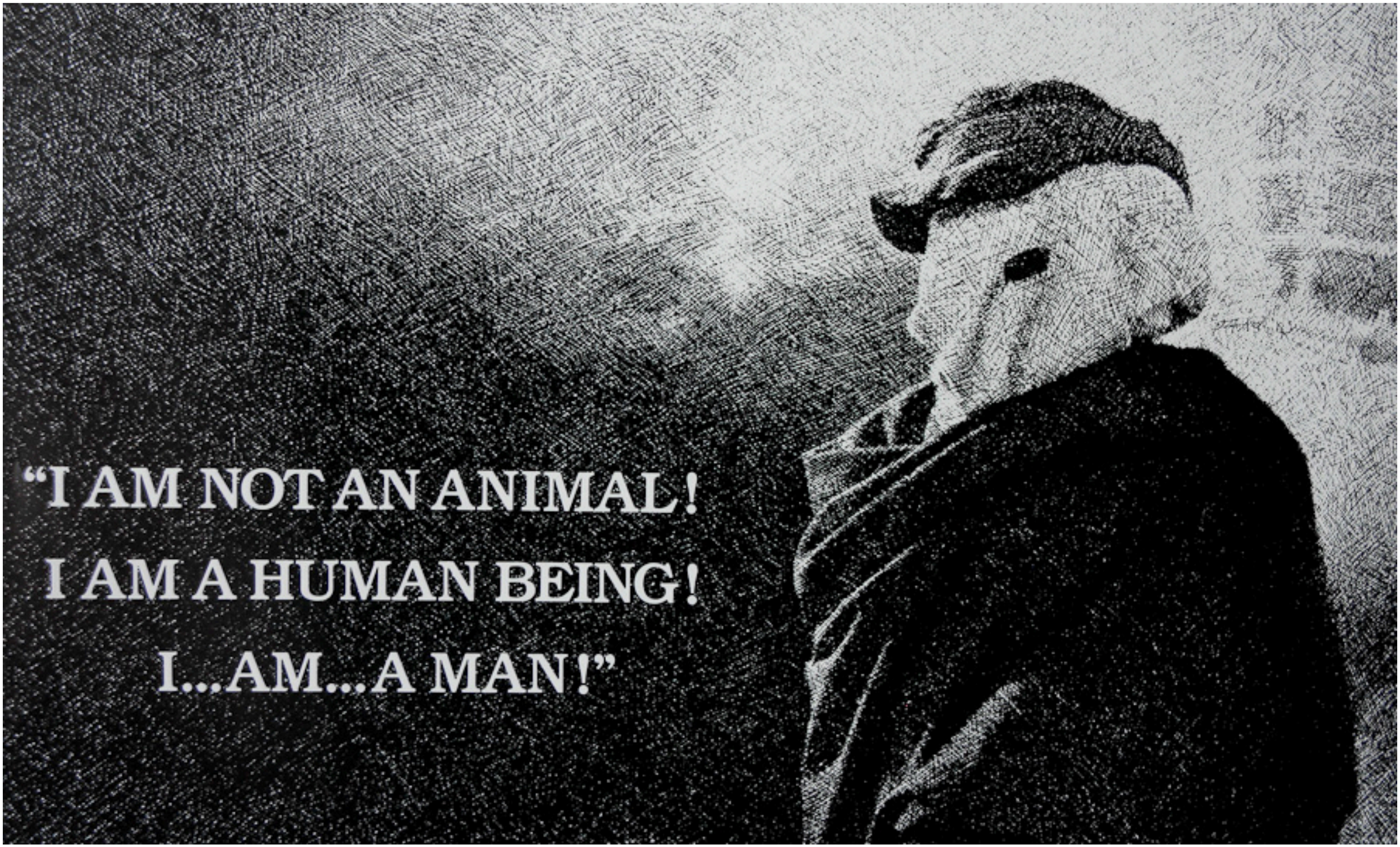
About Me

- Started out mainly interested in PL design
  - Particularly “functional” languages (ML, Haskell, etc.)
  - PhD thesis and postdoc work on extensions of the ML module system
- Module systems research was fun, but a bit lonely, and it was hard to have much impact

About Me

- After coming to MPI-SWS, became more interested in foundational PL questions:
  - How can we verify "real" programs?
  - How can we prove safety of "real" PLs?

- Definition of "real" has changed over time…
  - And gotten progressively more "grungy"


“I AM NOT AN ANIMAL!
I AM A HUMAN BEING!
I...AM...A MAN!”

Check out my blog at

herrdreyer.wordpress.com

The RustBelt Team

@MPI-SWS & UdS

Ralf Jung

Jan-Oliver Kaiser

David Swasey

Hai Dang

Jacques-Henri Jourdan

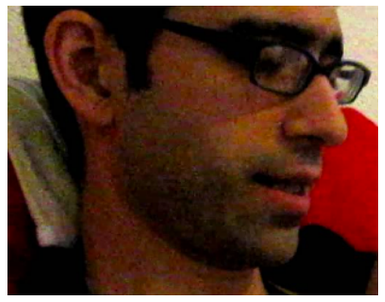
Ori Lahav

Viktor Vafeiadis

Derek Dreyer

@Mozilla

Aaron Turon



Niko Matsakis

@Aarhus

Robbert Krebbers

Lars Birkedal

@Seoul Nat. Univ.

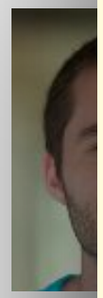
Jeehoon Kang

Chung-Kil Hur

The RustBelt Team

Two new members in July!

Ralf Jung



Jan K

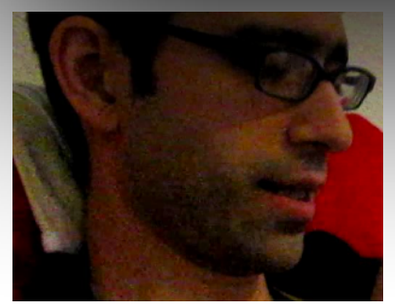
Azalea Raad

Josh Yanovski

Derek Dreyer

@M Nat. Univ.

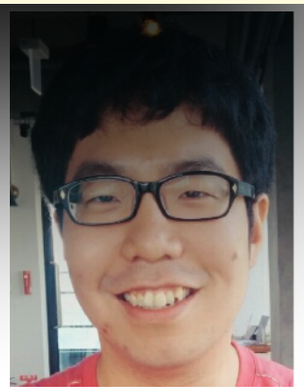
Aaron Turon



Niko Matsakis

Robbert Krebbers

Lars Birkedal



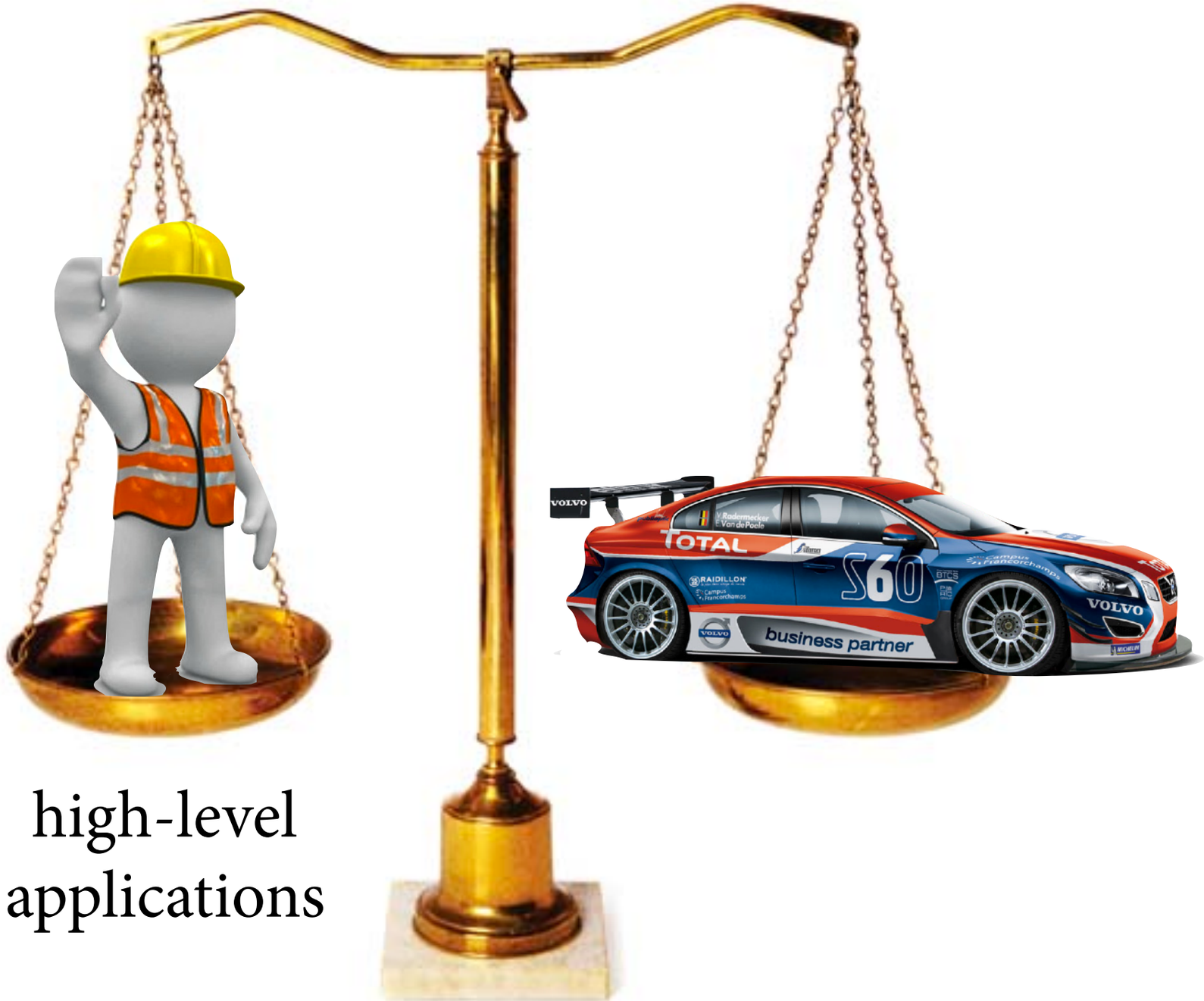
Jeehoon Kang

Chung-Kil Hur

Safety vs. Control

Safety vs. Control

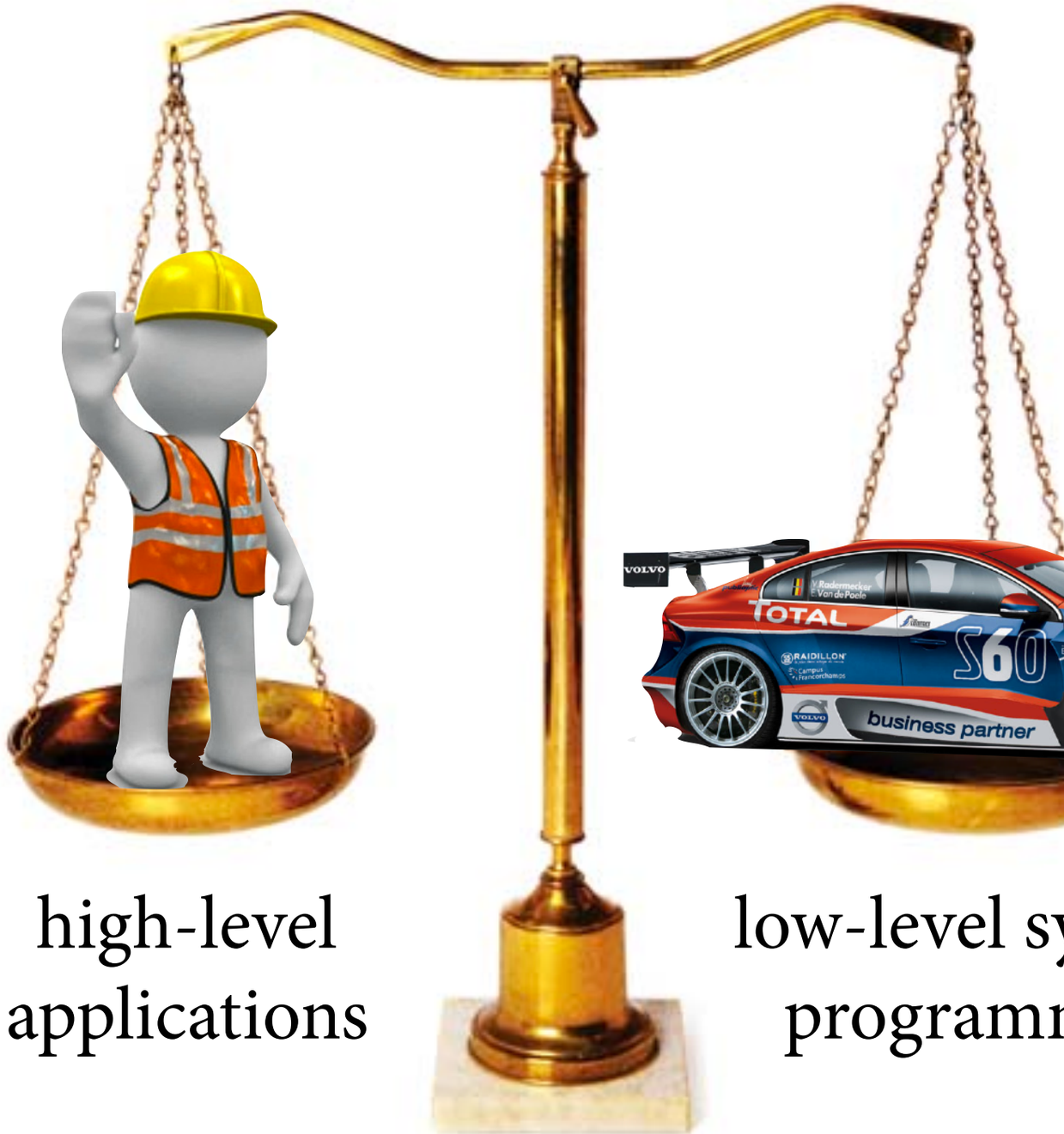
Java
C#
Go
Haskell
…



high-level
applications

Safety vs. Control

Java
C#
Go
Haskell
…



high-level
applications

low-level systems
programming

C
C++
…

Safety vs. Control

Java
C#
Go
Haskell
…

hig
appl

ystems
ning

C
C++
…

Rust:

The Future of Safe Systems Programming?

Rust has been developed at **Mozilla** since 2010

- Mozilla is using Rust to build Servo, a next-gen browser engine with better parallel performance

Rust is the only "systems PL" to provide…

- Low-level control à la modern C++
- Strong safety guarantees
- Industrial development and backing

15 companies using Rust in production

- **Dropbox** is rewriting block storage engine from Go into Rust to control memory footprint

Rust:

The Future of Safe Systems Programming?

Rust has been developed at **Mozilla** since 2010

- Mozilla is using Rust to build Servo, a next-gen browser engine with better parallel performance

Rust has the potential to become the "next big thing" in systems programming

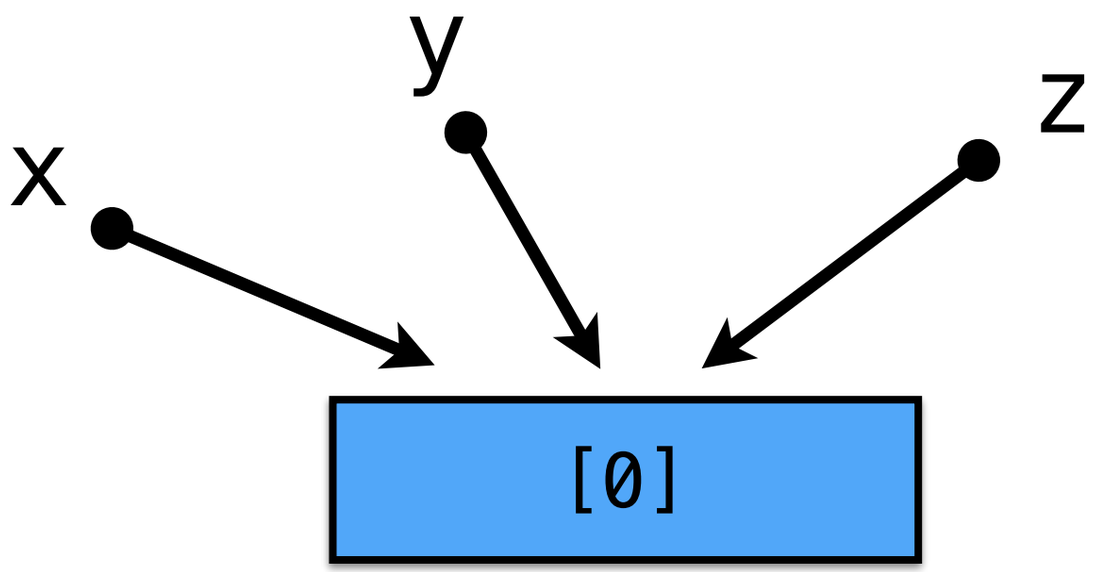
Dropbox

15 companies using Rust in production

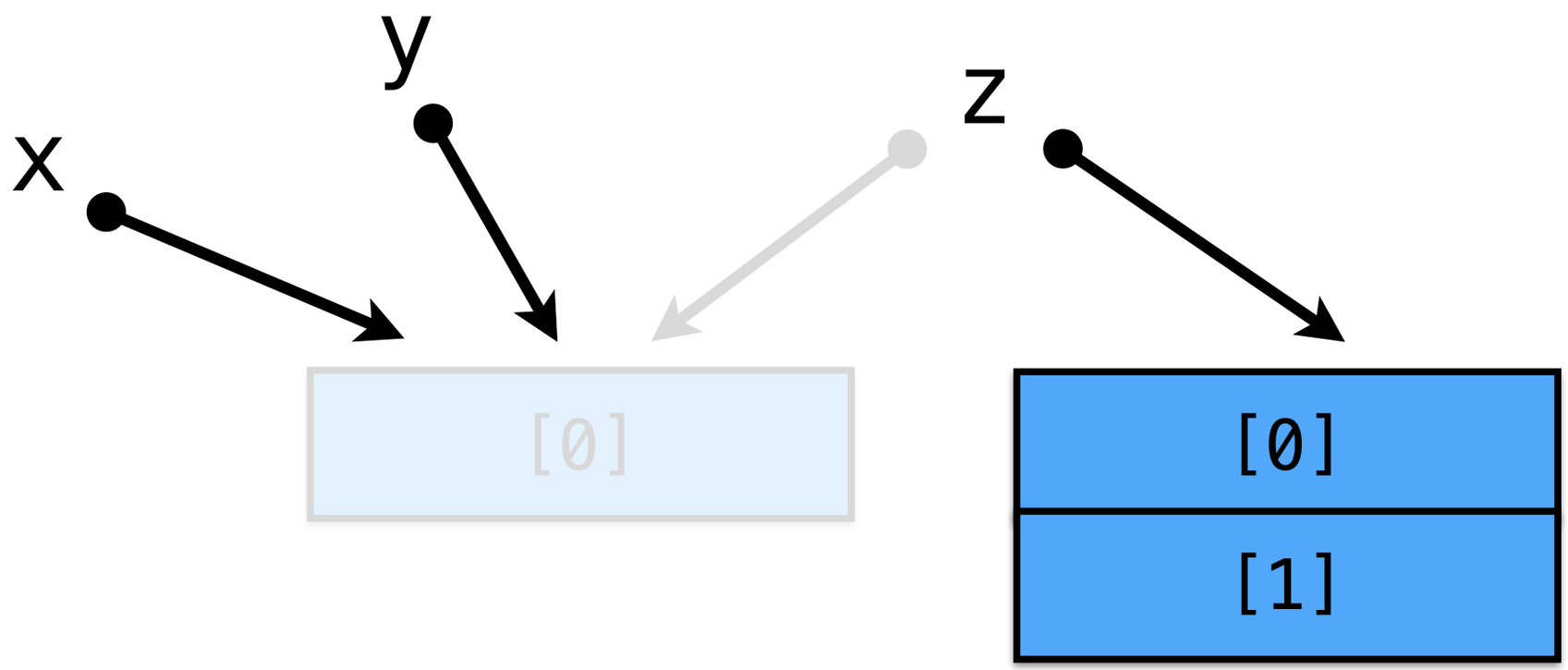
- **Dropbox** is rewriting block storage engine from Go into Rust to control memory footprint

Core Idea of Rust

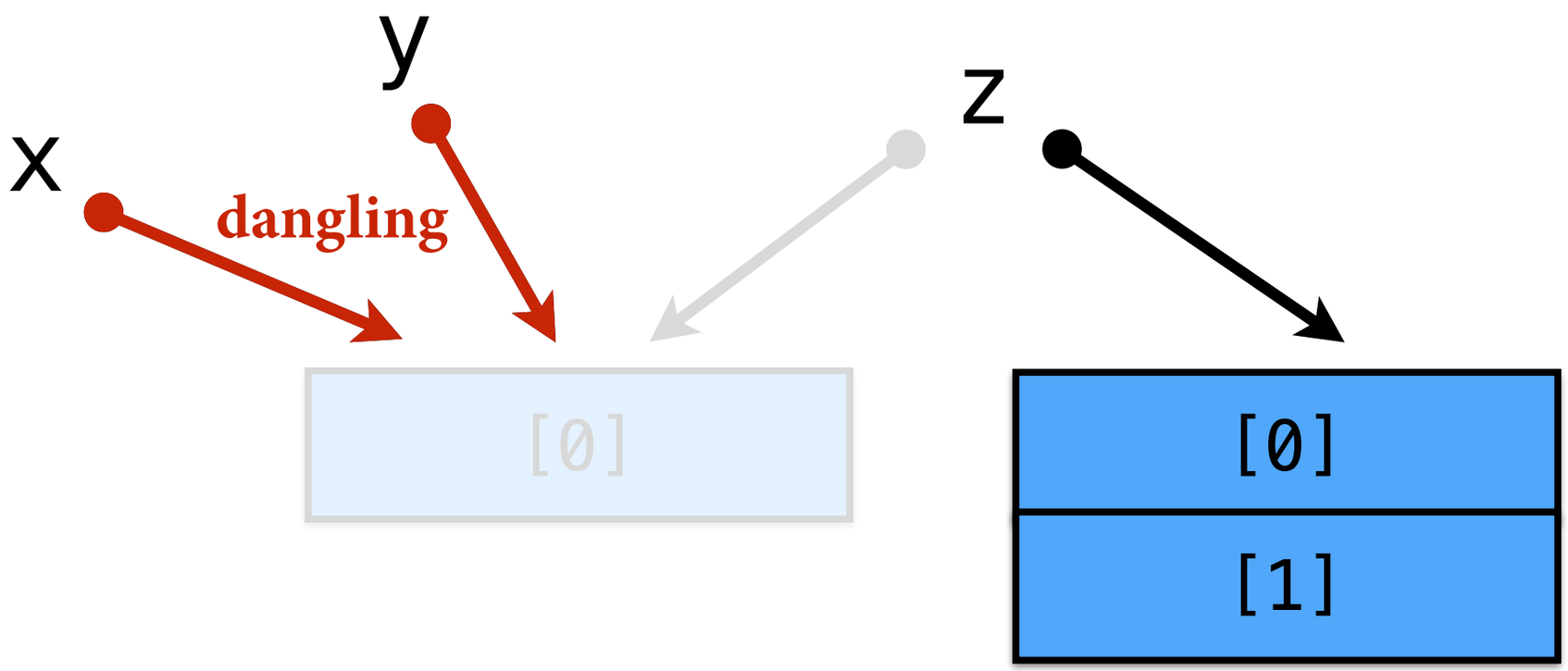
Core Idea of Rust



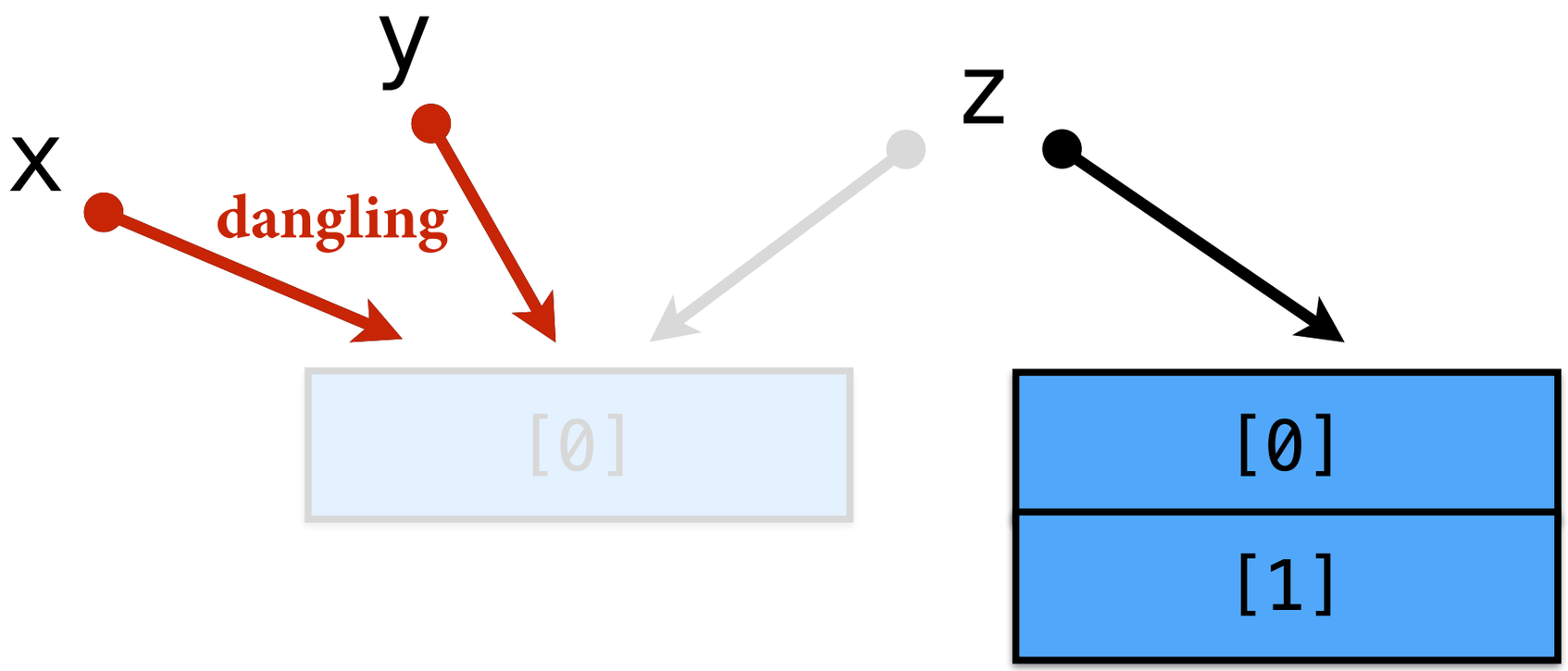
Core Idea of Rust



Core Idea of Rust



Core Idea of Rust



Unrestricted mutation and aliasing lead to:

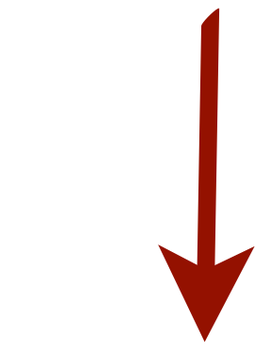
- use-after-free errors (dangling references)
- data races
- iterator invalidation

Data Races

```
x := 1
y := 2
x := 3
```

Data Races

```
x := 1
y := 2
x := 3
```

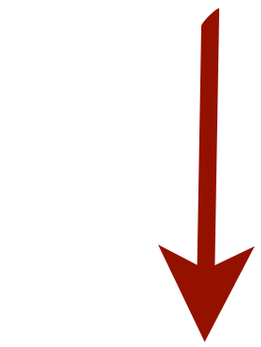


~~`x := 1`~~

```
y := 2
x := 3
```

Data Races

```
x := 1          ||    print y
y := 2          ||    print x
x := 3          ||
```



```
x := 1          ||    print y
y := 2          ||    print x
x := 3          ||
```

Data Races

```
x := 1          ‖    print y
y := 2          ‖    print x
x := 3          ‖
```

```
x := 1          ‖    print y
y := 2          ‖    print x
x := 3          ‖     ⟶ 2 0
```

Data Races

```
x := 1          ‖    print y
y := 2          ‖    print x
x := 3          ‖     ↛ 2 0
```

```
x := 1          ‖    print y
y := 2          ‖    print x
x := 3          ‖     ⟶ 2 0
```

Data Races

```
x := 1   ‖   print y
y := 2   ‖   print x
```

Standard compiler optimizations change the "meaning" of racy concurrent code

```
y := 2   ‖   print x
x := 3   ‖   ⟶ 2 0
```

Data Races

```
x := 1     ‖     print y
y :
```

Many architectures do, too!

Standard compiler optimizations change the “meaning” of racy concurrent code

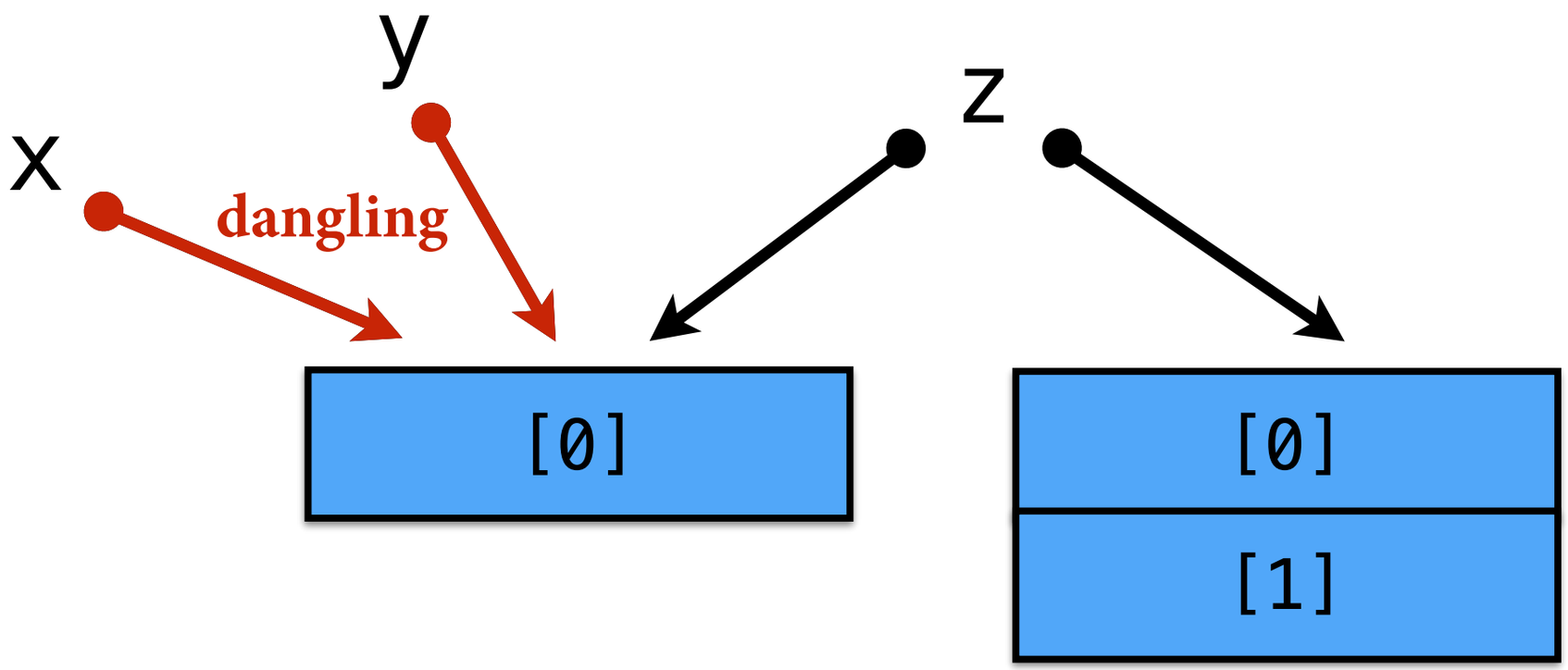
```
y := 2     ‖     print x
x := 3     ‖     ⟶ 2 0
```

What's a PL to do?

Get used to disappointment.

- **Java**: Data races → **Very weak** behavior
- **C/C++**: Data races → **Undefined** behavior

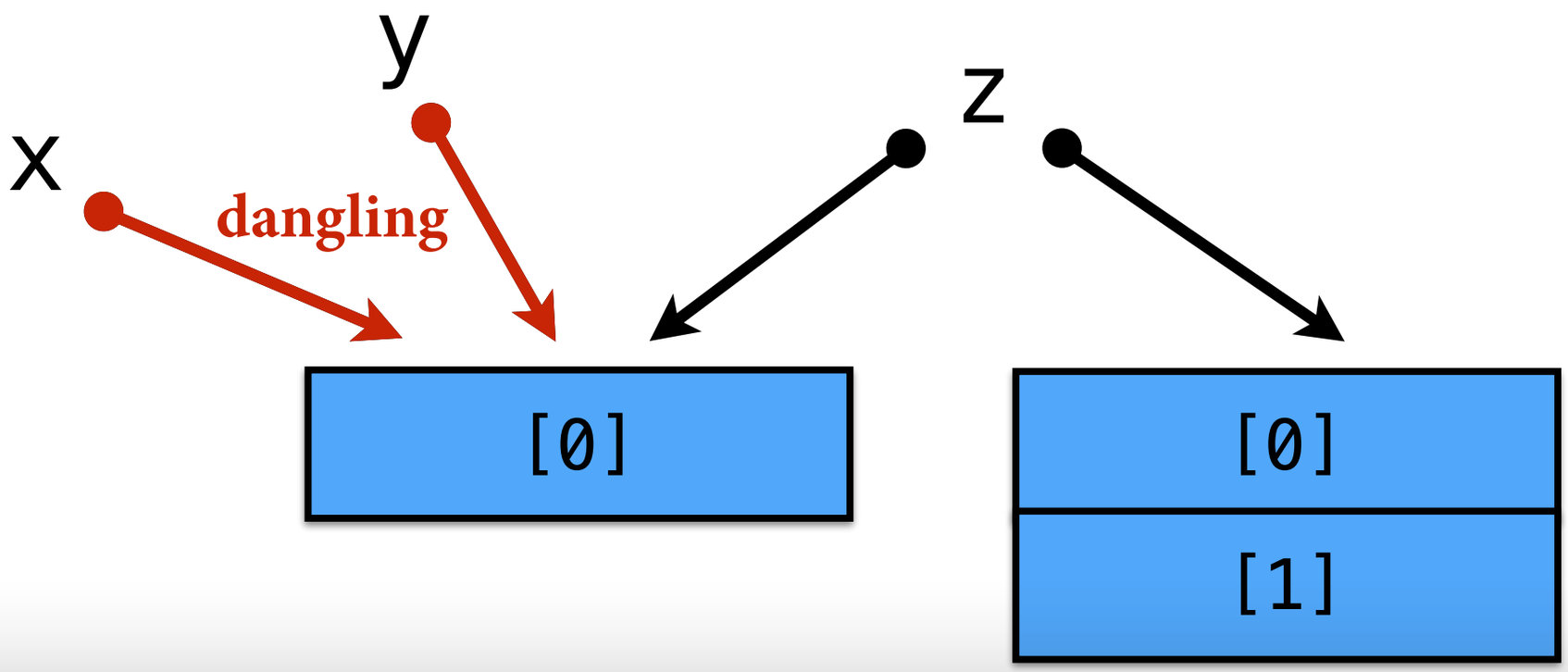
Core Idea of Rust



Unrestricted mutation and aliasing lead to:

- use-after-free errors (dangling references)
- data races
- iterator invalidation

Core Idea of Rust



Rust prevents all these errors using a sophisticated "ownership" type system

Ownership & Borrowing

- Having a value of type **`T`** means you "own" it fully.
- **`T`** can be "borrowed" (e.g. passed by reference):
  - ✦ **`&T`** — **shared, immutable** borrow
  - ✦ **`&mut T`** — **unique, mutable** borrow

But sometimes you ***need***

aliased mutable state!

But sometimes you ***need***

aliased mutable state!

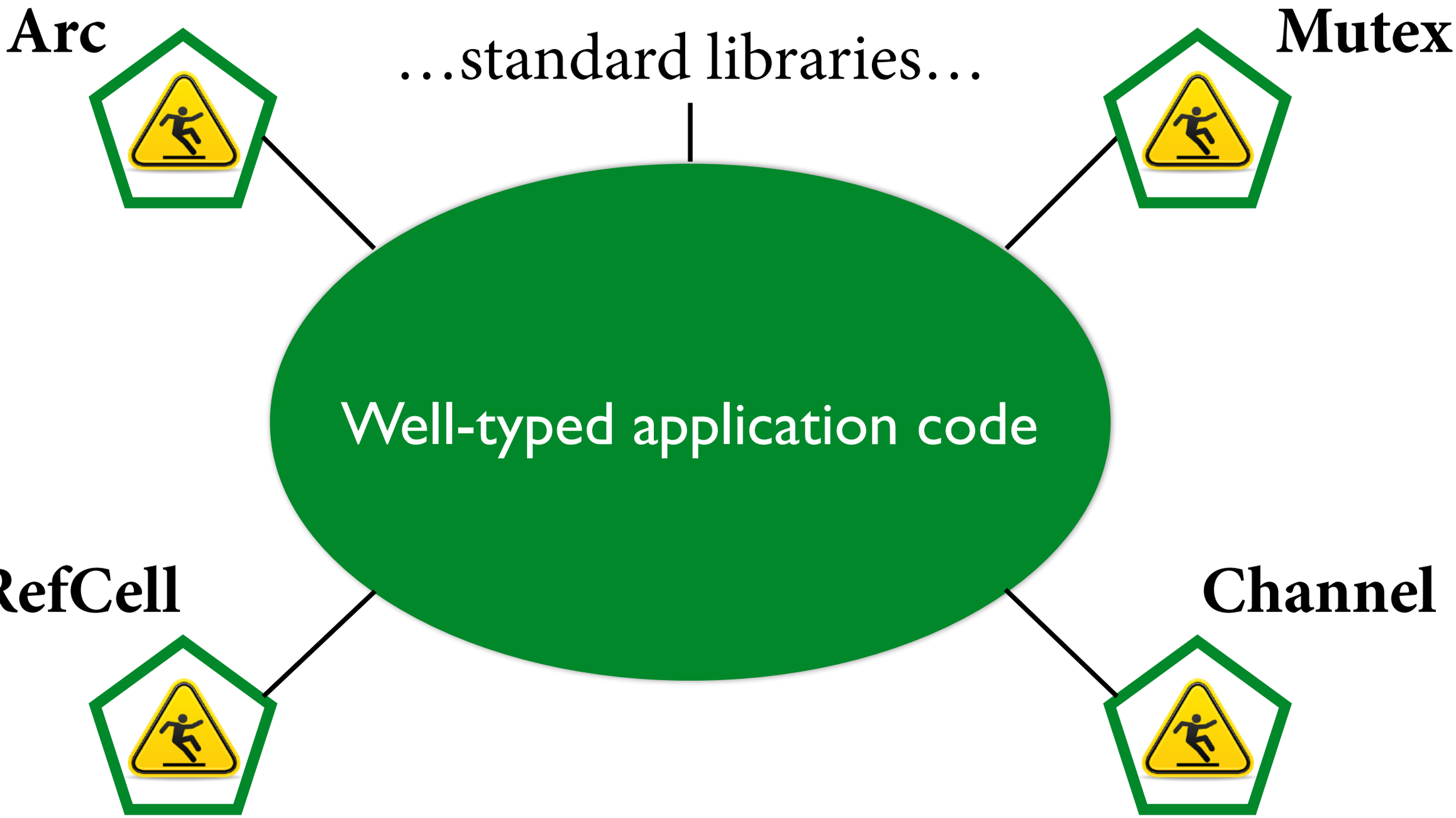
Synchronization mechanisms:

- e.g. Locks, channels, semaphores

Memory management:

- e.g. Reference counting

The Reality of Rust



The Reality of Rust

Arc

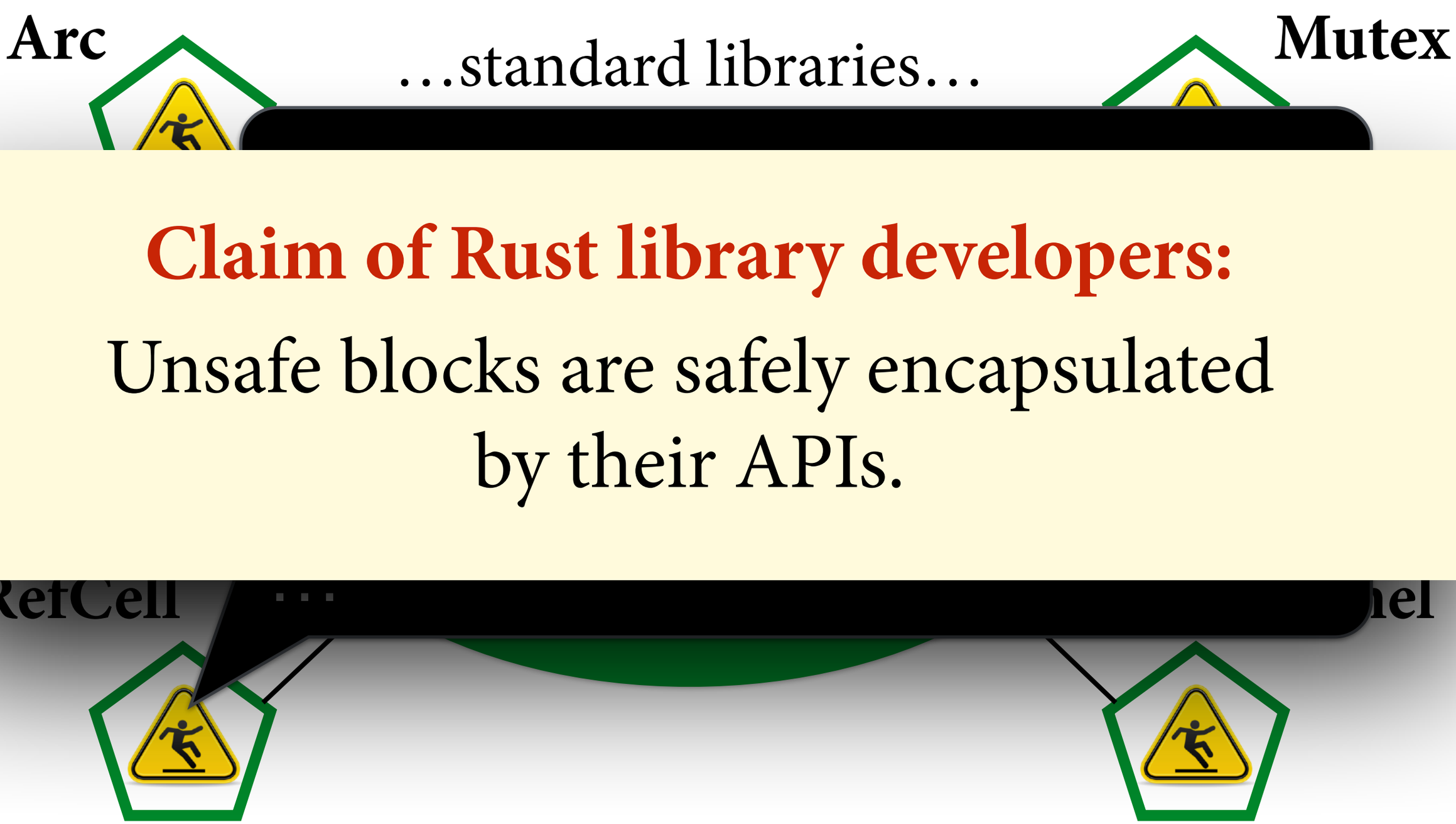
Mutex

…standard libraries…

```
...
pub fn borrow(&self) -> Ref<T> {
  match BorrowRef::new(&self.borrow) {
    Some(b) => Ref {
      _value: unsafe { &*self.value.get() },
      _borrow: b,
    }, ...
  }
}
...
```

RefCell

The Reality of Rust



Claim of Rust library developers:

Unsafe blocks are safely encapsulated
by their APIs.

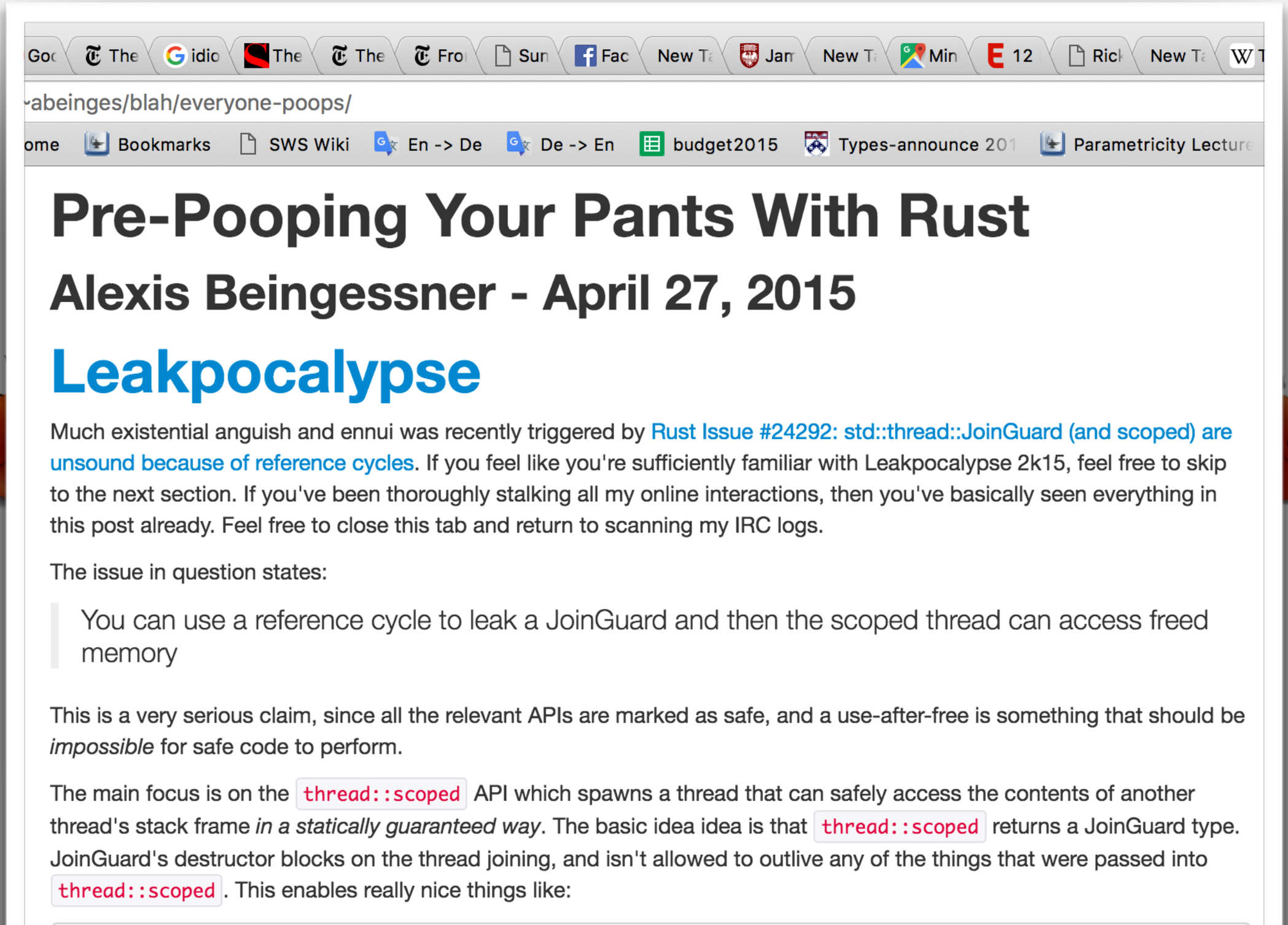
Is Rust Safe?

Several bugs found in Rust safety so far:

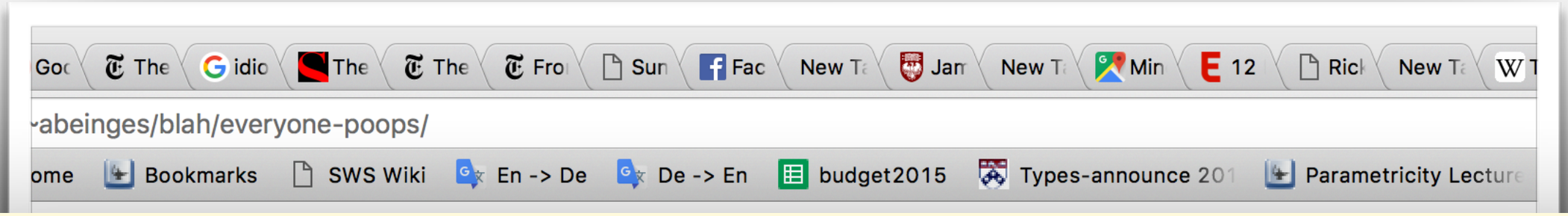
- Due to **unsafe blocks in Rust libraries**
  - e.g. “scoped threads” API

- Due to **dark corners of the type system**
  - e.g. “dropck” rule for checking safety of generic destructor methods

Is Rust Safe?


abeinges/blah/everyone-poops/
Bookmarks
SWS Wiki
En -> De
De -> En
budget2015
Pre-Pooping Your Pants With Rust
Alexis Beingessner - April 27, 2015
Leakpocalypse
Much existential anguish and ennui was recently triggered by Rust Issue #24292: std::thread::JoinGuard (and scoped) are unsound because of reference cycles. If you feel like you're sufficiently familiar with Leakpocalypse 2k15, feel free to skip to the next section. If you've been thoroughly stalking all my online interactions, then you've basically seen everything in this post already. Feel free to close this tab and return to scanning my IRC logs.
The issue in question states:
You can use a reference cycle to leak a JoinGuard and then the scoped thread can access freed memory
This is a very serious claim, since all the relevant APIs are marked as safe, and a use-after-free is something that should be impossible for safe code to perform.
The main focus is on the thread::scoped API which spawns a thread that can safely access the contents of another thread's stack frame in a statically guaranteed way. The basic idea idea is that thread::scoped returns a JoinGuard type. JoinGuard's destructor blocks on the thread joining, and isn't allowed to outlive any of the things that were passed into thread::scoped. This enables really nice things like:

Is Rust Safe?



Rust is at the **bleeding edge** of language design for safe systems programming

- We need **formal foundations** in order to build confidence in its safety guarantees!

memory

This is a very serious claim, since all the relevant APIs are marked as safe, and a use-after-free is something that should be *impossible* for safe code to perform.

The main focus is on the `thread::scoped` API which spawns a thread that can safely access the contents of another thread's stack frame *in a statically guaranteed way*. The basic idea idea is that `thread::scoped` returns a JoinGuard type. JoinGuard's destructor blocks on the thread joining, and isn't allowed to outlive any of the things that were passed into `thread::scoped`. This enables really nice things like:

RustBelt

Goal: **Develop 1st logical foundations for Rust**

- Use these foundations to verify the safety of the Rust core type system and std libraries
- Give Rust developers the tools they need to safely evolve the language

What is “Safety”?

What is “Safety”?

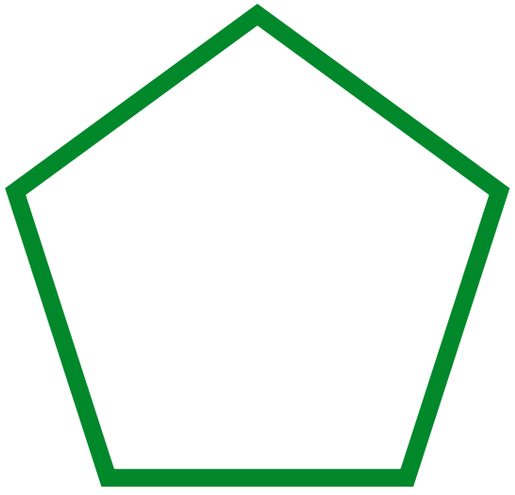
- Standard **“syntactic safety”** approach of Wright and Felleisen (1994) **will not work for Rust!**
  - Requires whole program to be well-typed!

What is “Safety”?

- Standard **“syntactic safety”** approach of Wright and Felleisen (1994) **will not work for Rust!**
  - Requires whole program to be well-typed!

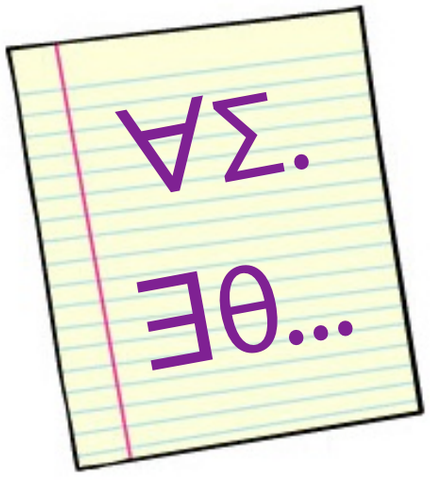
- Need to generalize to **semantic safety**
  - A library is semantically safe if no well-typed application using it can have undefined behavior

Semantic Safety



Library
interface

semantic
model

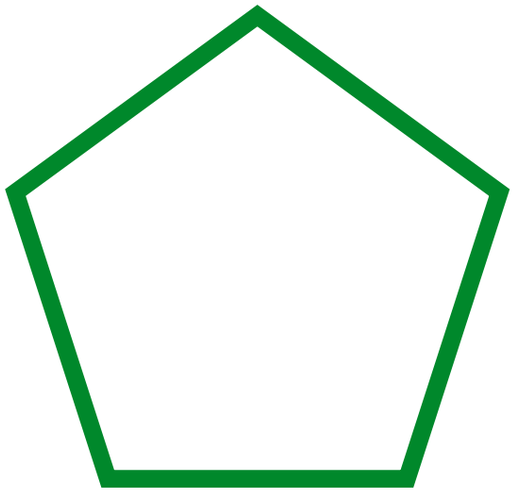


Safety
contract

logical
satisfaction

Library
implementation

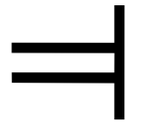
Semantic Safety



Library interface

semantic model

Safety contract

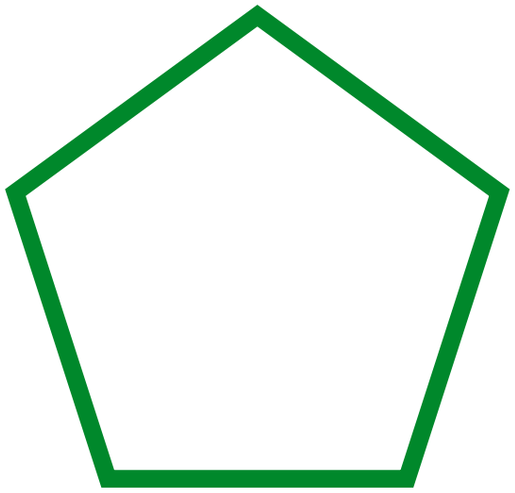


logical satisfaction

Library implementation

Well typed

Semantic Safety



Library
interface

semantic
model

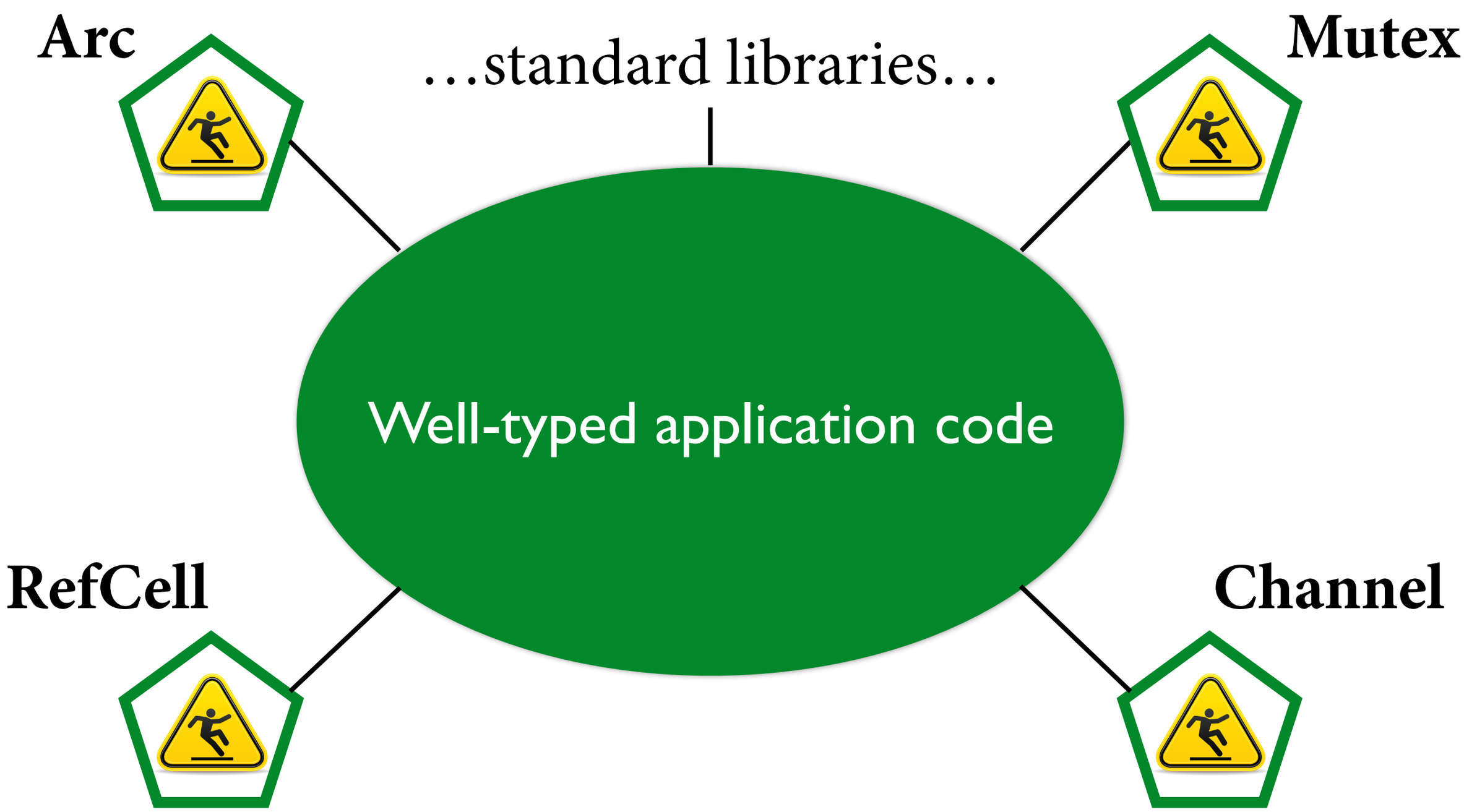
Safety
contract

?

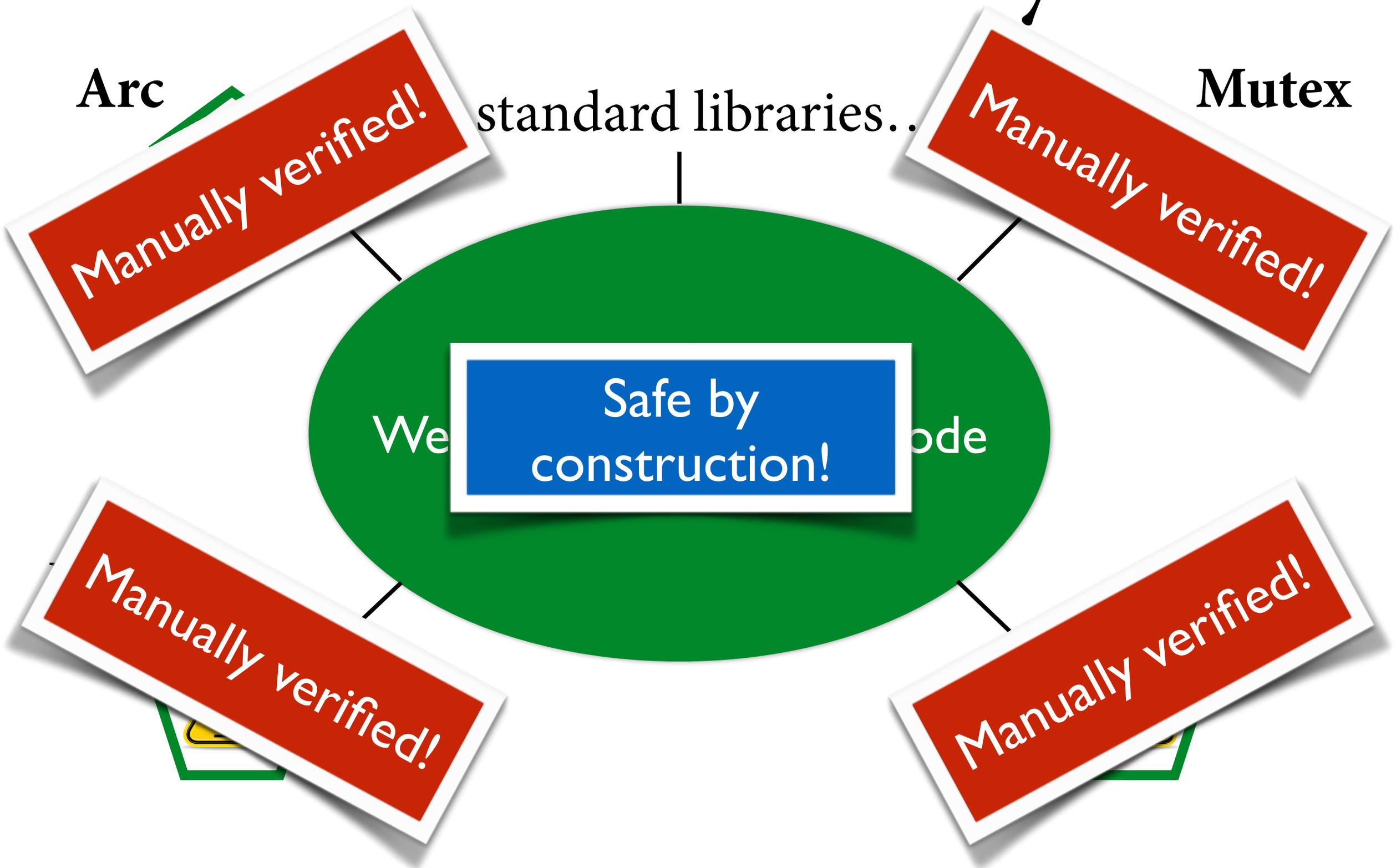
logical
satisfaction

Library
implementation

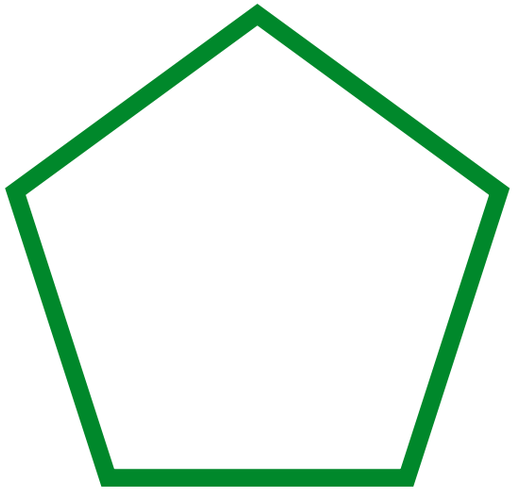
Semantic Safety



Semantic Safety

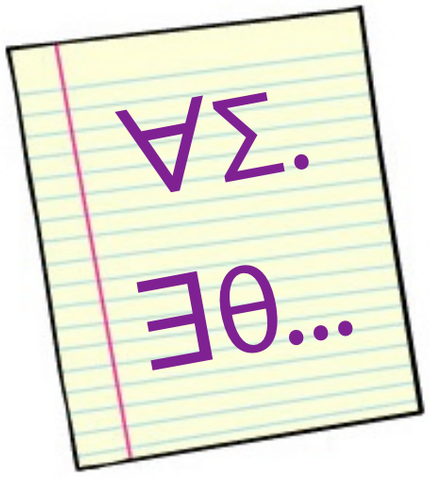


Semantic Safety

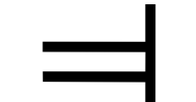


Library interface

semantic model



Safety contract



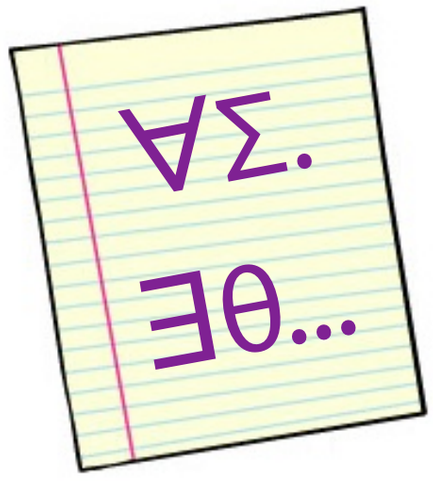
logical satisfaction

Library implementation

Challenge:

Verify semantic safety for 

Heart of the Problem



Which logic to use?

Separation Logic to the Rescue!

Separation Logic to the Rescue!

Extension of Hoare logic (O'Hearn, Reynolds…, ~2000)

- For reasoning about pointer-manipulating programs

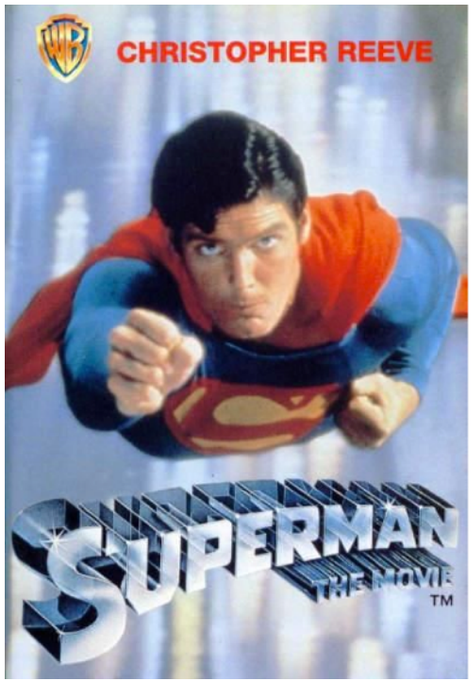
Major influence on many verification & analysis tools

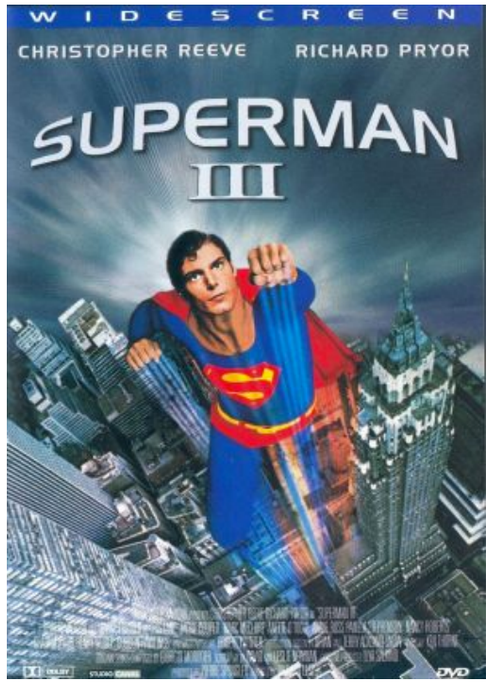
- e.g. Infer, VeriFast, Chalice, Bedrock, jStar, …

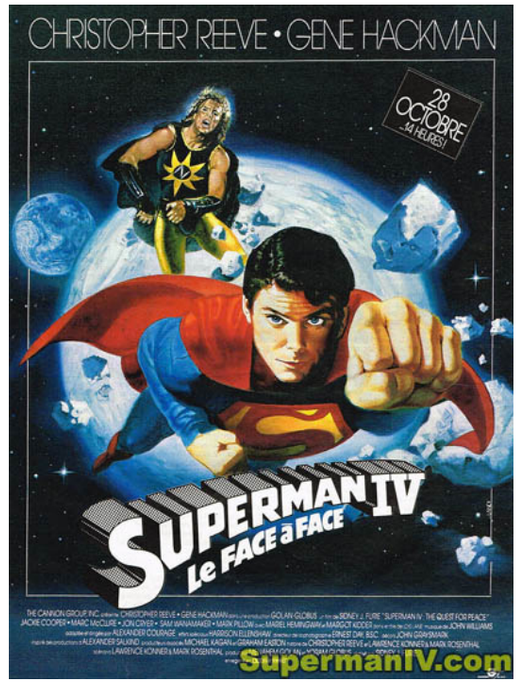
Separation logic = Ownership logic

- Perfect fit for modeling Rust's ownership types!

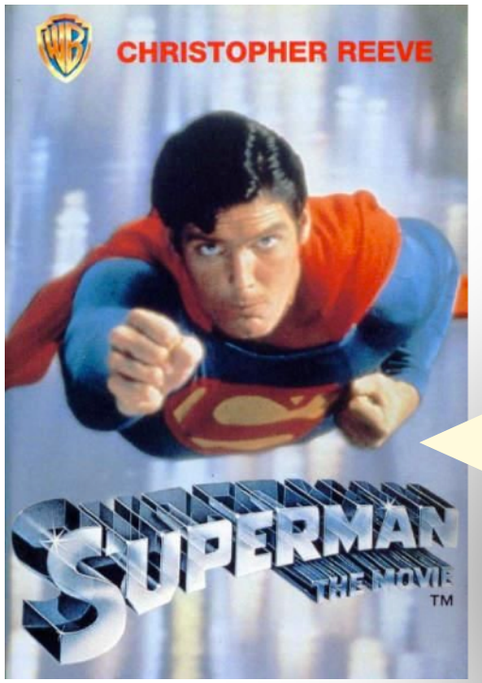
Problem 1: Which One?

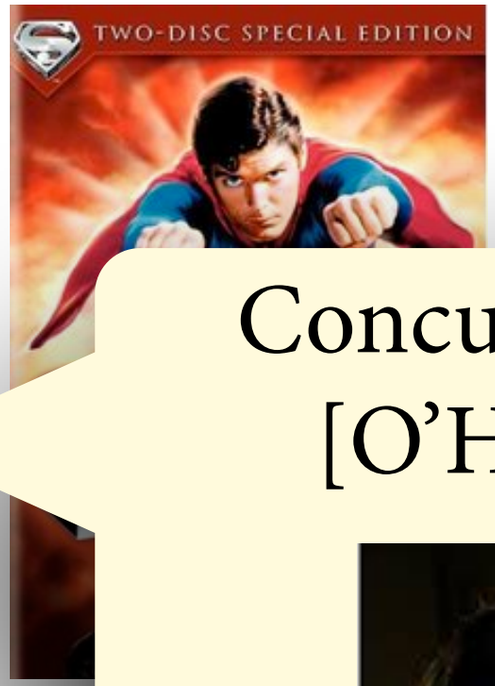


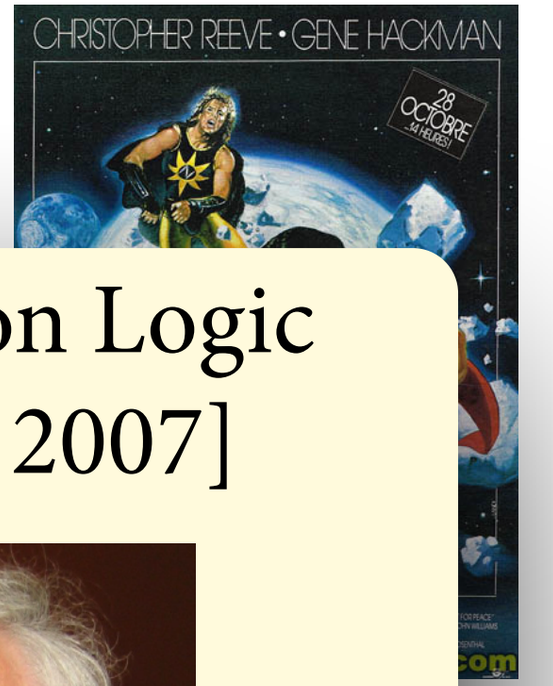




Problem 1: Which One?



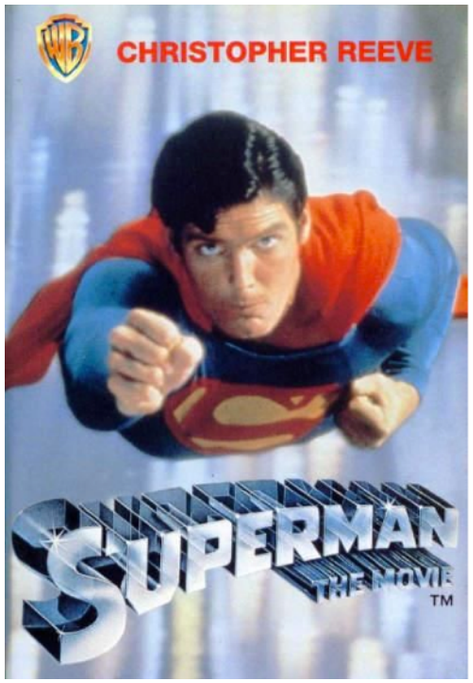


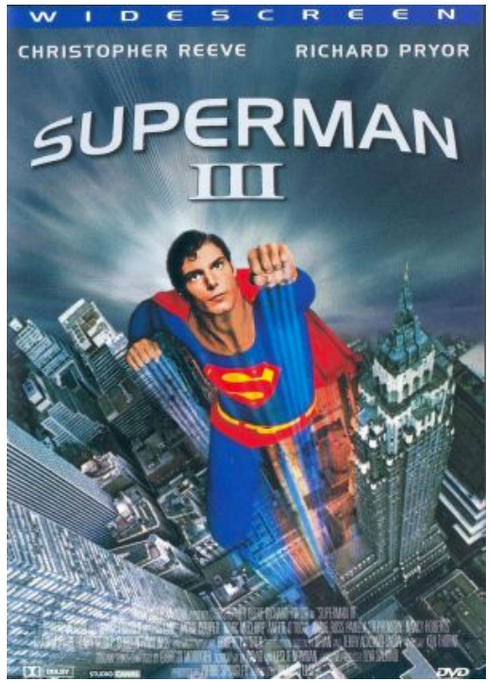


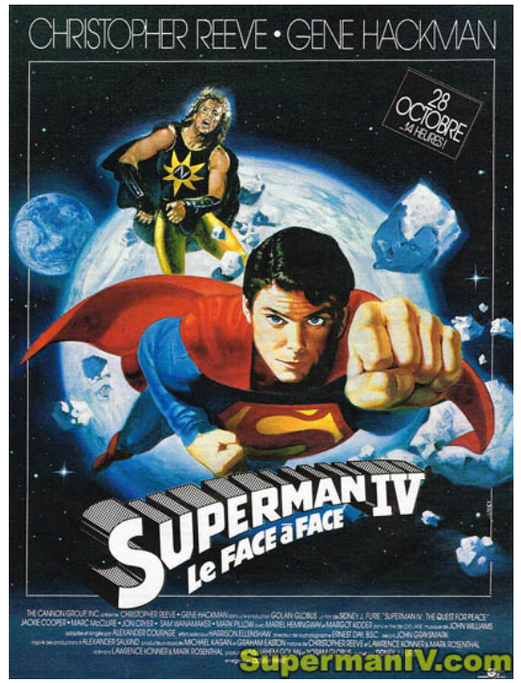
Concurrent Separation Logic
[O'Hearn/Brookes, 2007]

Won the 2016 Gödel Prize!

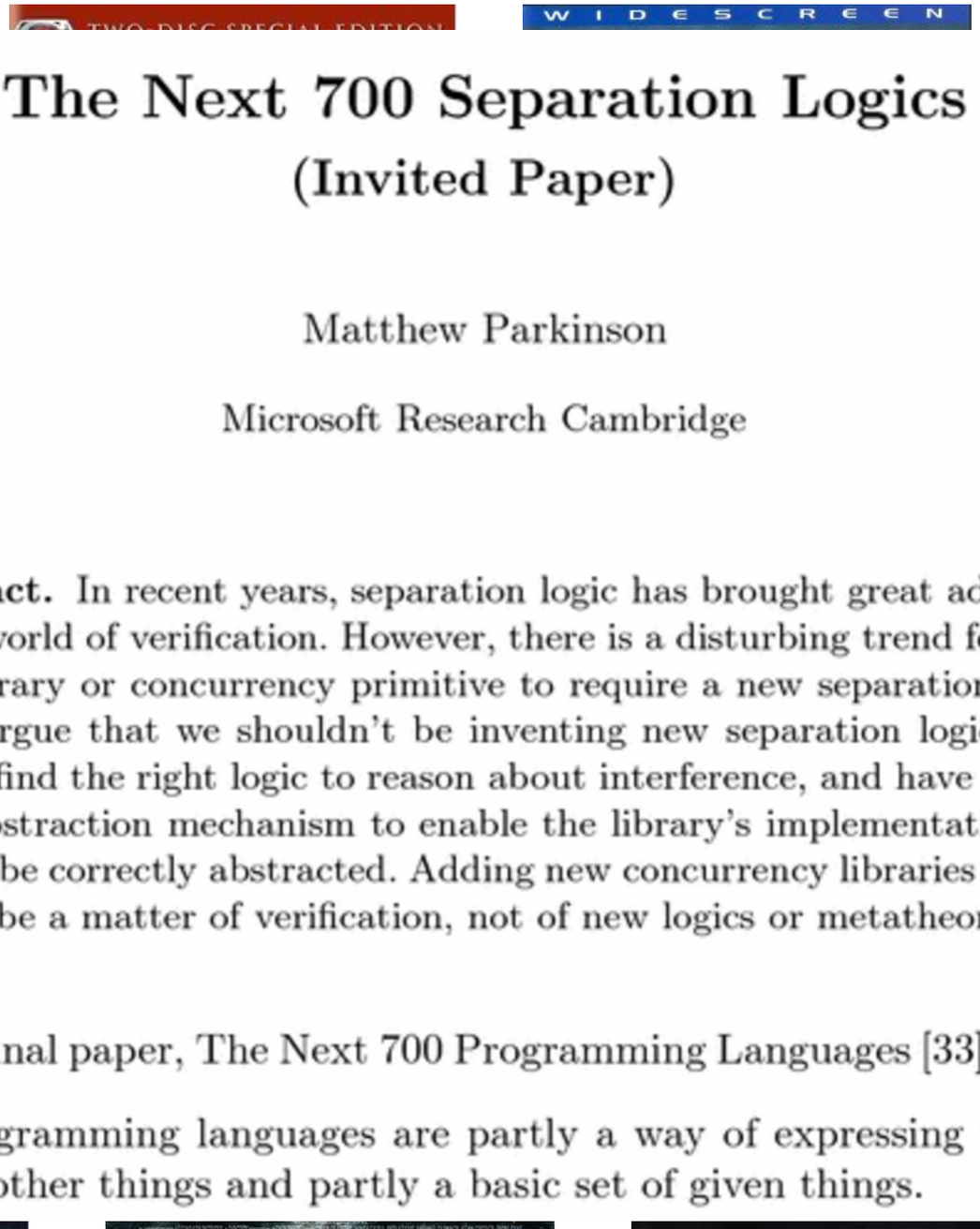
Problem 1: Which One?







Problem 1: Which One?



The Next 700 Separation Logics
(Invited Paper)

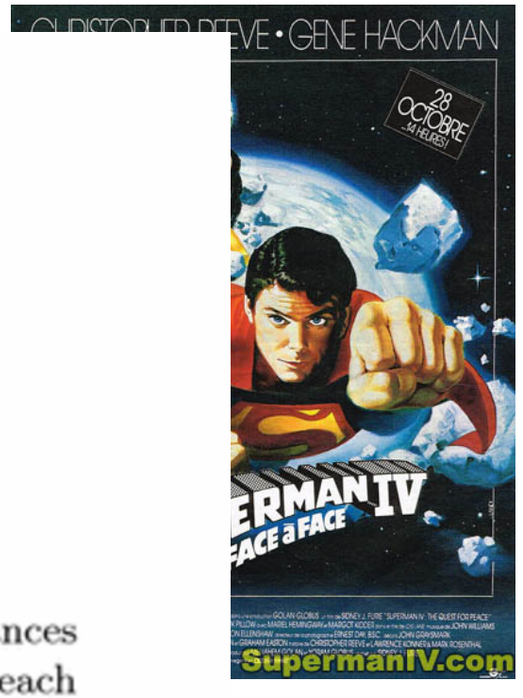
Matthew Parkinson

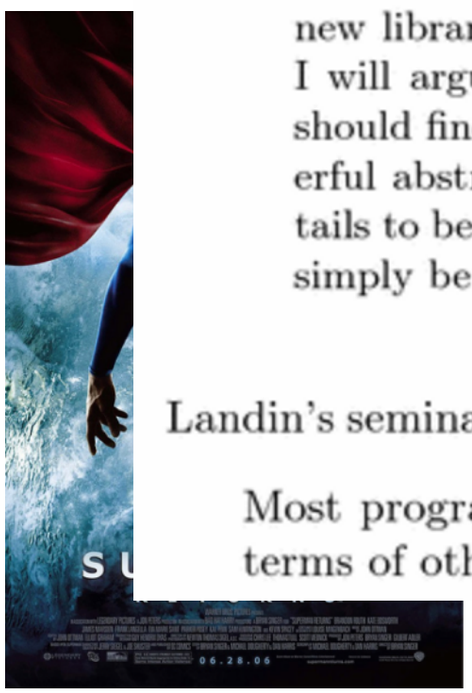
Microsoft Research Cambridge

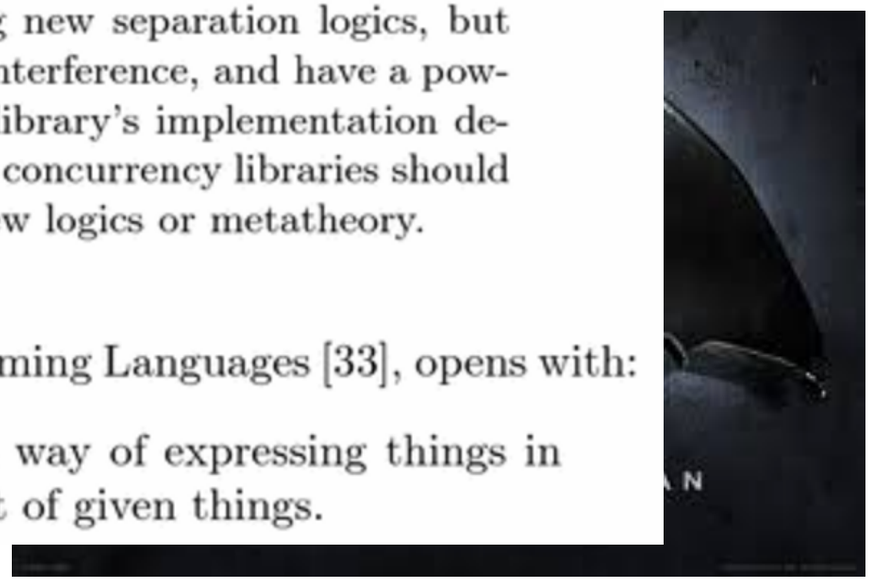
Abstract. In recent years, separation logic has brought great advances in the world of verification. However, there is a disturbing trend for each new library or concurrency primitive to require a new separation logic. I will argue that we shouldn't be inventing new separation logics, but should find the right logic to reason about interference, and have a powerful abstraction mechanism to enable the library's implementation details to be correctly abstracted. Adding new concurrency libraries should simply be a matter of verification, not of new logics or metatheory.

Landin's seminal paper, The Next 700 Programming Languages [33], opens with:

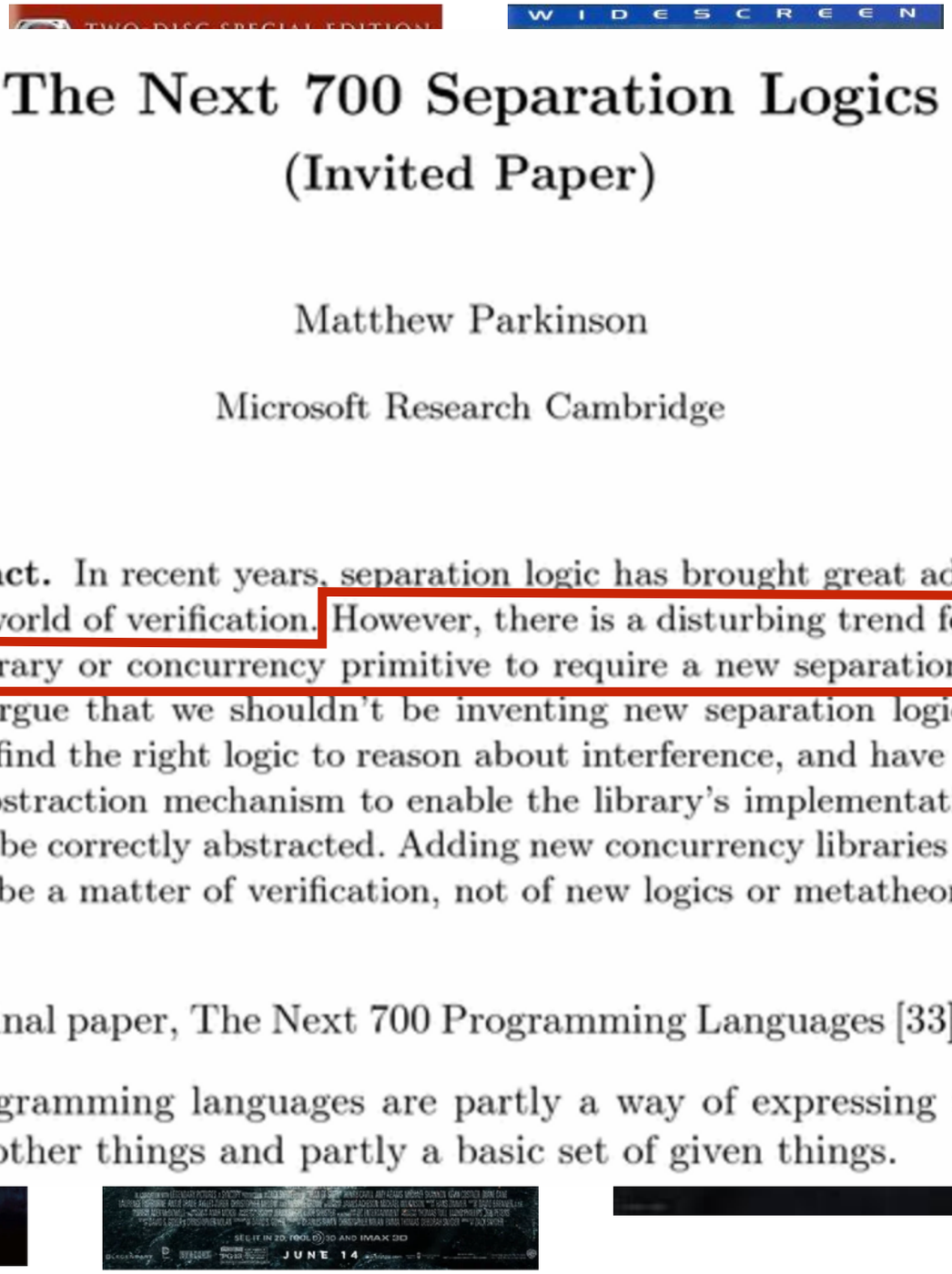
> Most programming languages are partly a way of expressing things in terms of other things and partly a basic set of given things.







Problem 1: Which One?



The Next 700 Separation Logics
(Invited Paper)

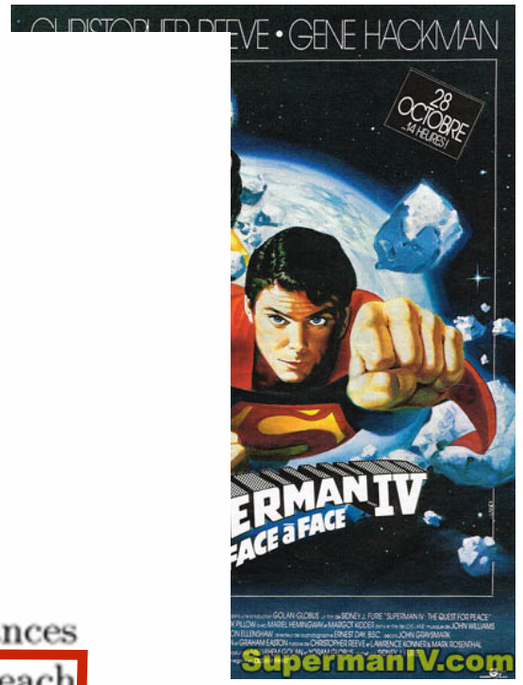
Matthew Parkinson

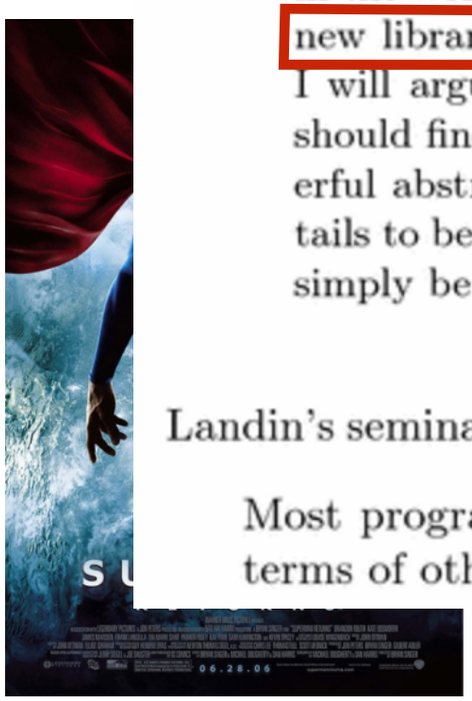
Microsoft Research Cambridge

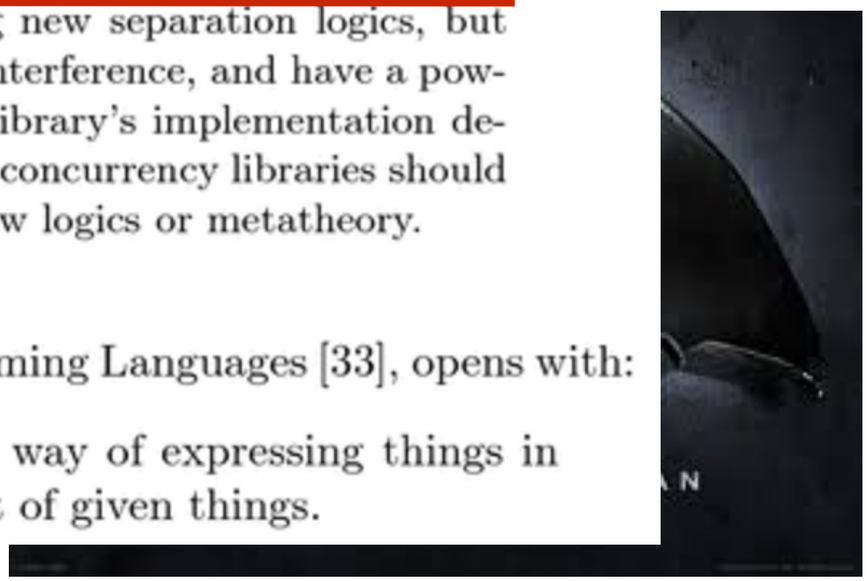
Abstract. In recent years, separation logic has brought great advances in the world of verification. However, there is a disturbing trend for each new library or concurrency primitive to require a new separation logic. I will argue that we shouldn't be inventing new separation logics, but should find the right logic to reason about interference, and have a powerful abstraction mechanism to enable the library's implementation details to be correctly abstracted. Adding new concurrency libraries should simply be a matter of verification, not of new logics or metatheory.

Landin's seminal paper, The Next 700 Programming Languages [33], opens with:

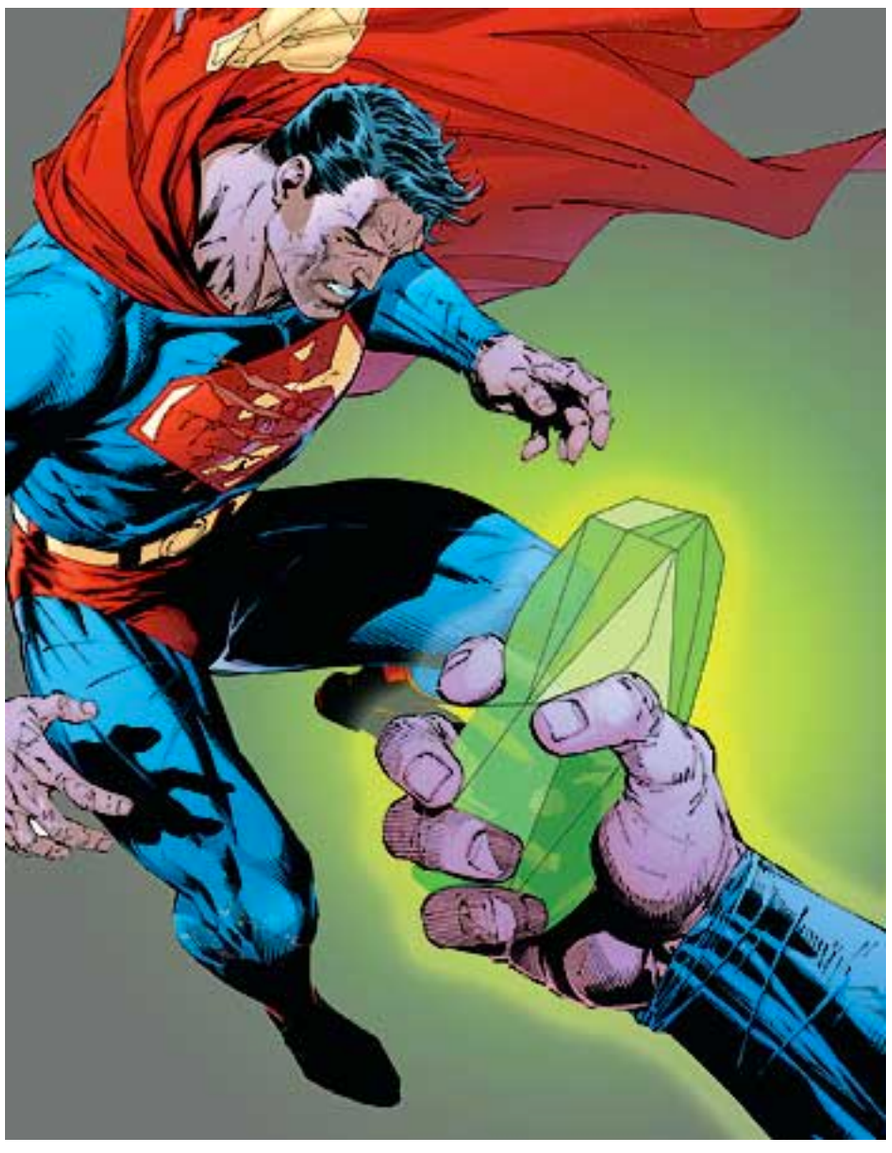
> Most programming languages are partly a way of expressing things in terms of other things and partly a basic set of given things.







Problem 2: Memory Model



All these logics assume:

- **sequential consistency** for memory accesses

*This is totally **unrealistic** for high-performance concurrency!*

- e.g. Rust's Arc library uses **C++'s weak memory ops**

Towards a Logic for Rust

- **Iris** [POPL'15, ICFP'16, POPL'17, ESOP'17]:
  Simplifying & unifying modern separation logics
  + support for machine-checked proof in Coq

- **GPS** [OOPSLA'14, PLDI'15, ECOOP'17]:
  First modern sep. logic for C++ memory model

In these lectures…

- **Day 1**: Ownership types

- **Day 2**: Concurrent separation logic

- **Day 3**: Introduction to Iris framework
  & how we are using it to verify safety of Rust

In these lectures

- **Day 1**: Ownership types in Rust!
- **Day 2**: Concurrent separation logic in Coq!
- **Day** mework
  & how we are using it to verify safety of Rust

Interactive demos!