# PhD, Research, and Programming Languages

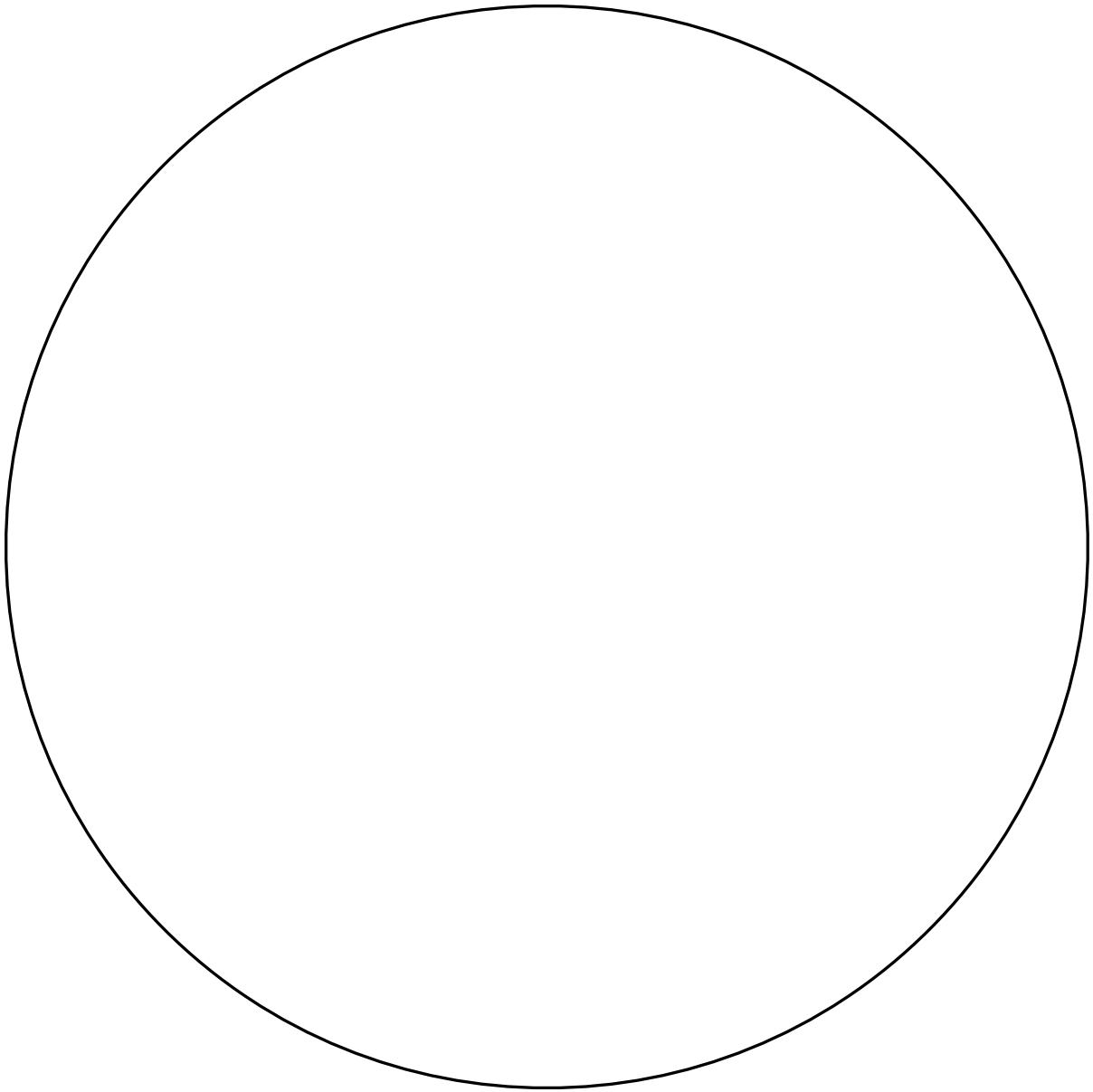
Gabriel Scherer

Gallium – INRIA

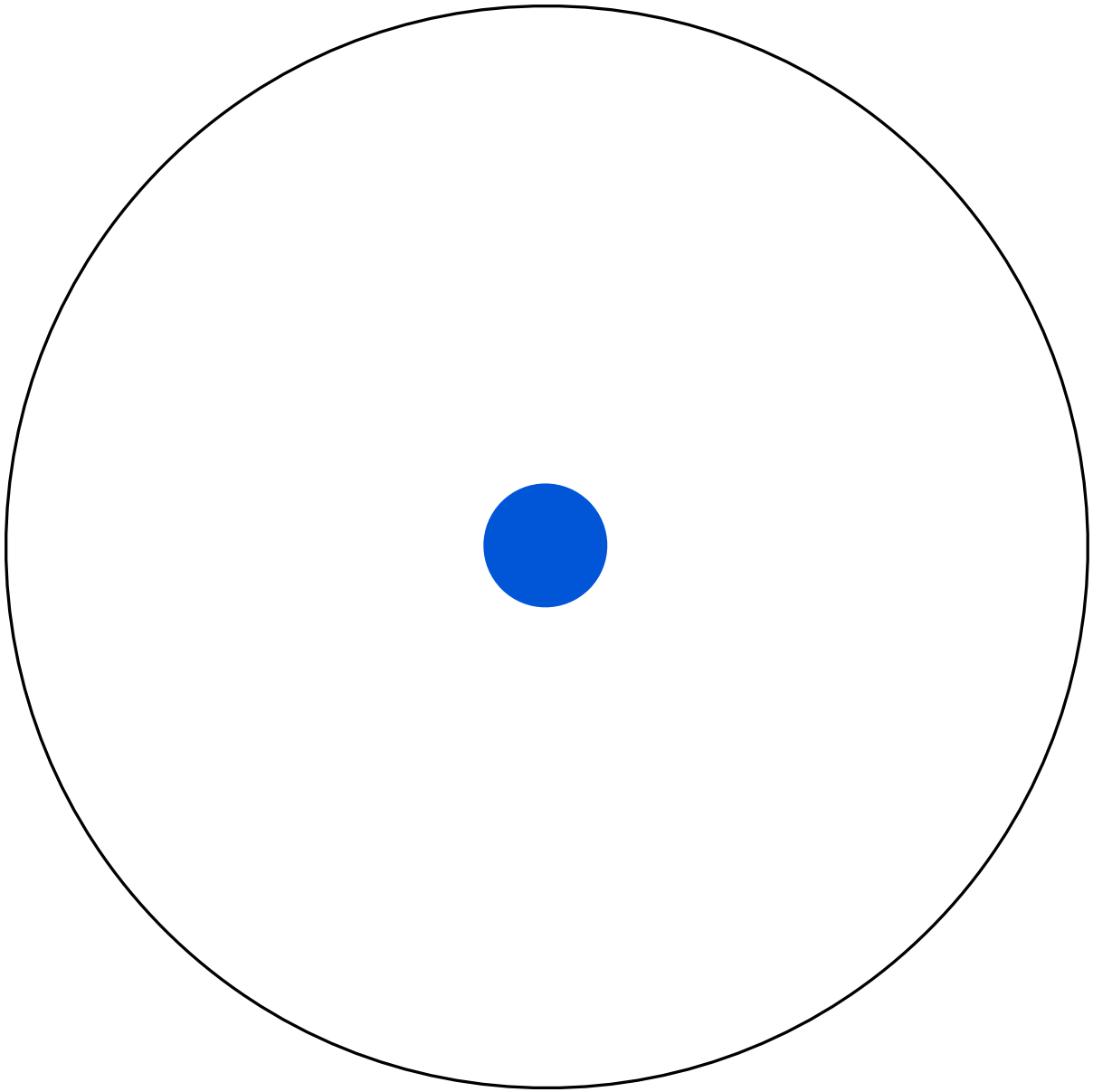
# The illustrated guide to a Ph.D.

Matt Might
matt.might.net

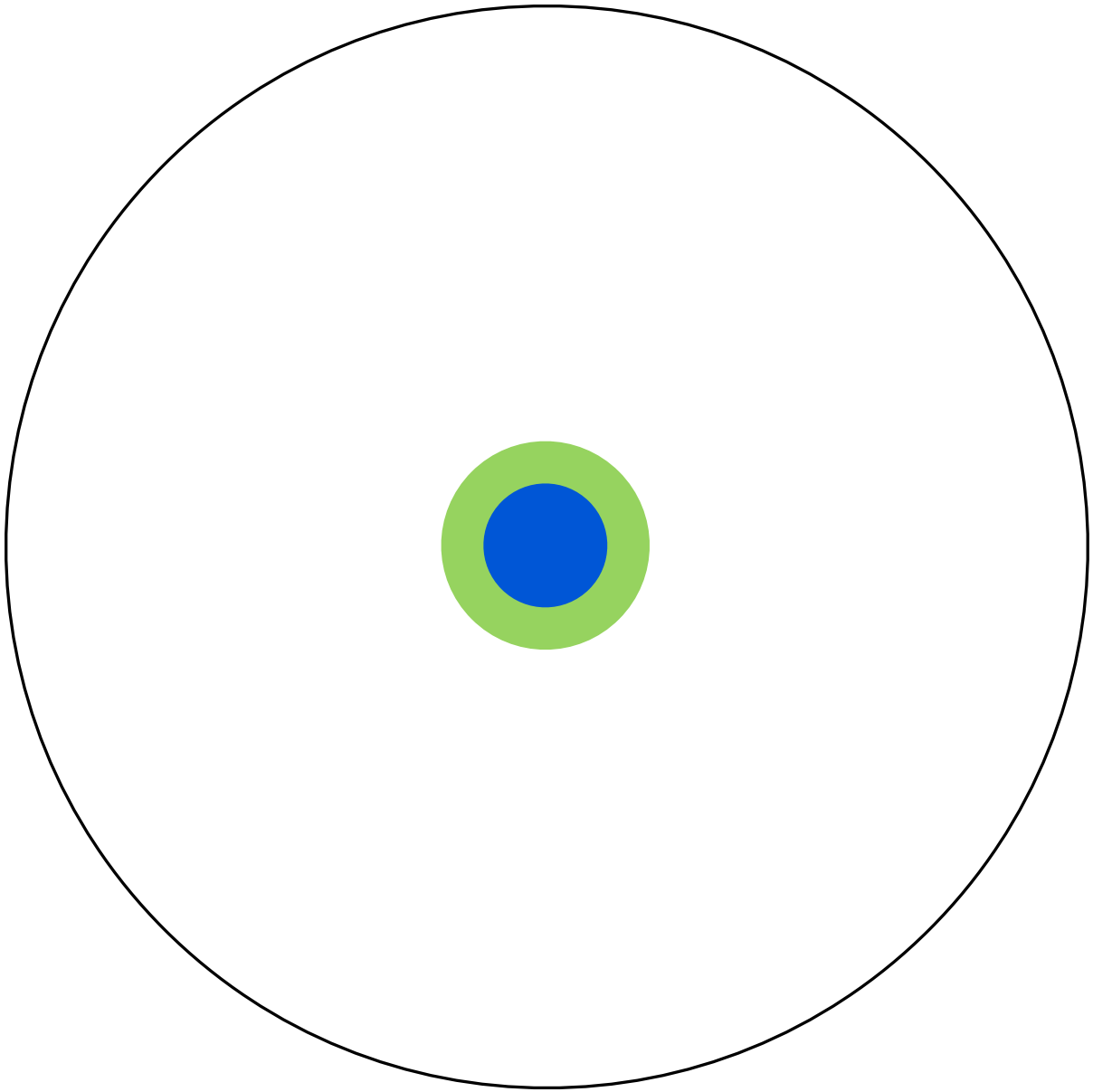
Imagine a circle that contains all of human knowledge:



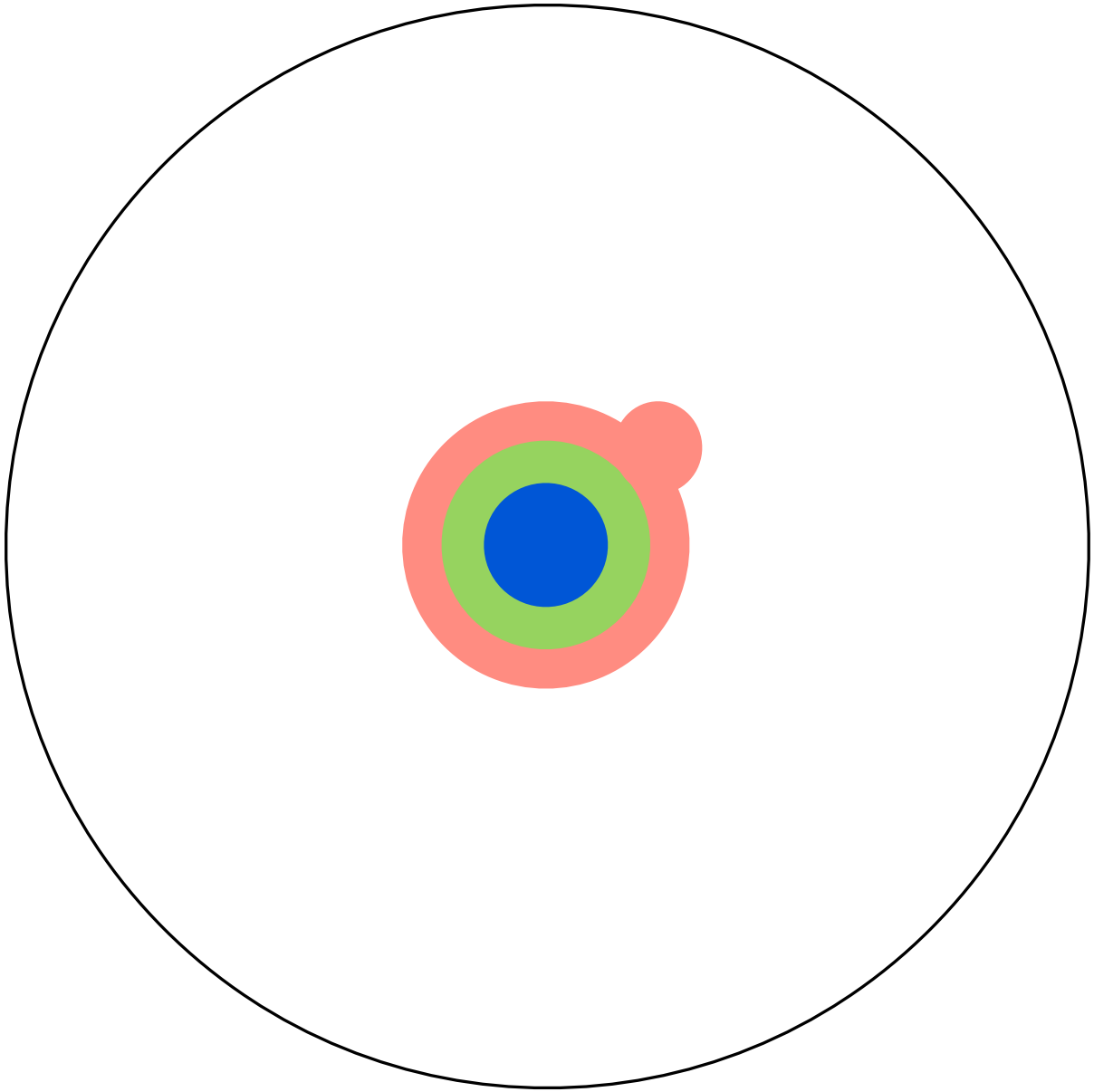
By the time you finish elementary school, you know a little:



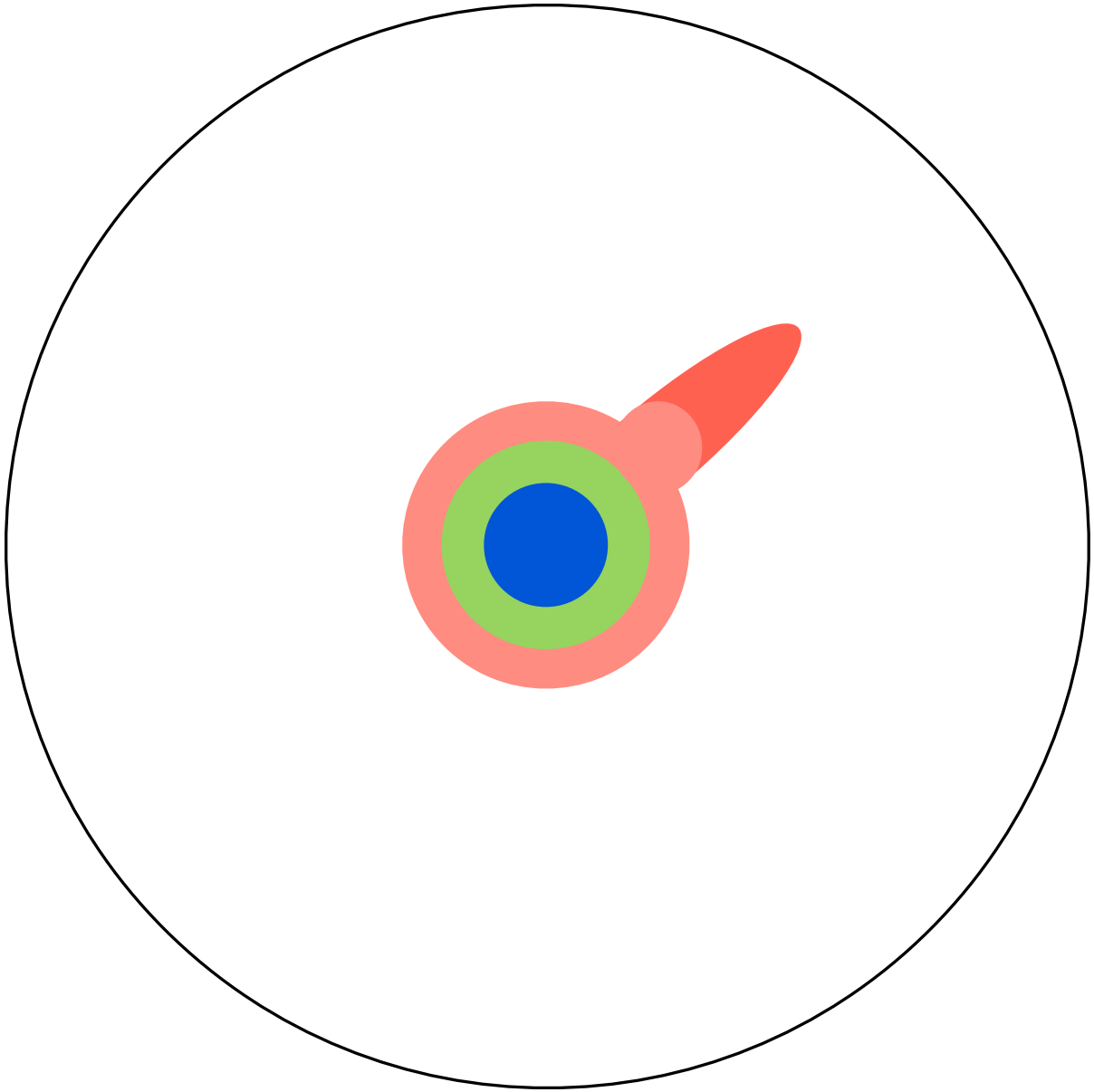
By the time you finish high school, you know a bit more:



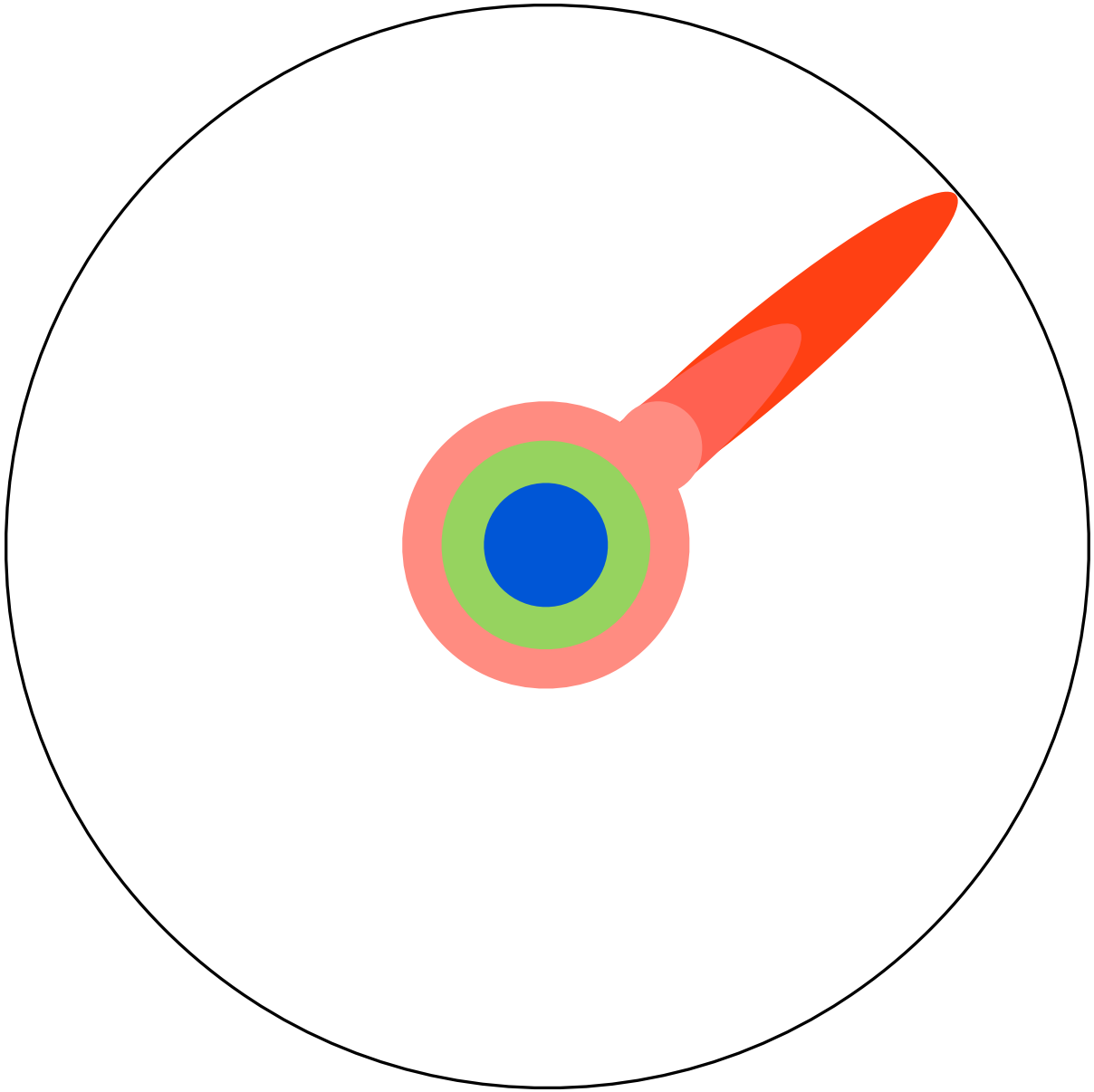
With a bachelor's degree, you gain a specialty:



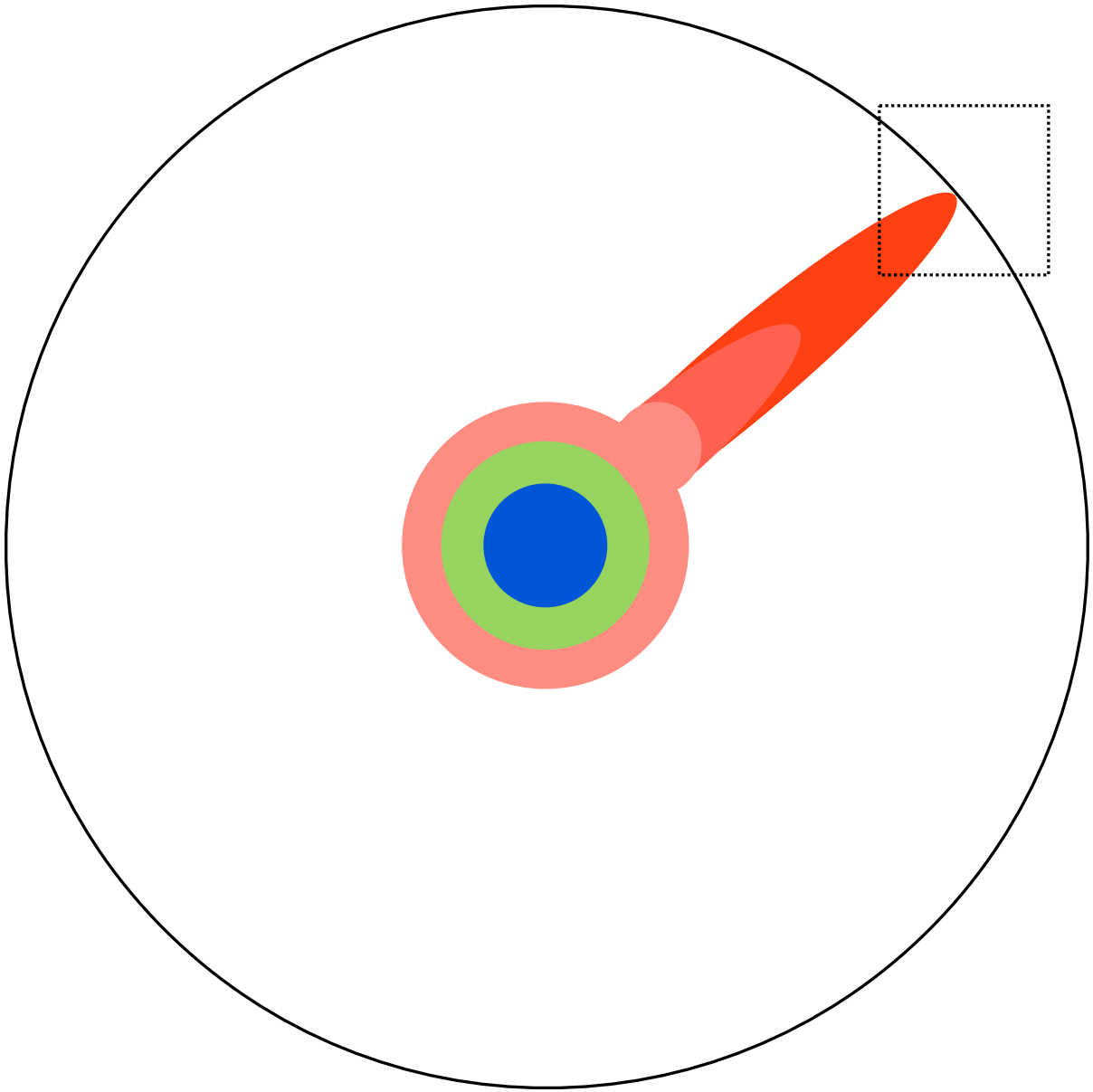
A master's degree deepens that specialty:



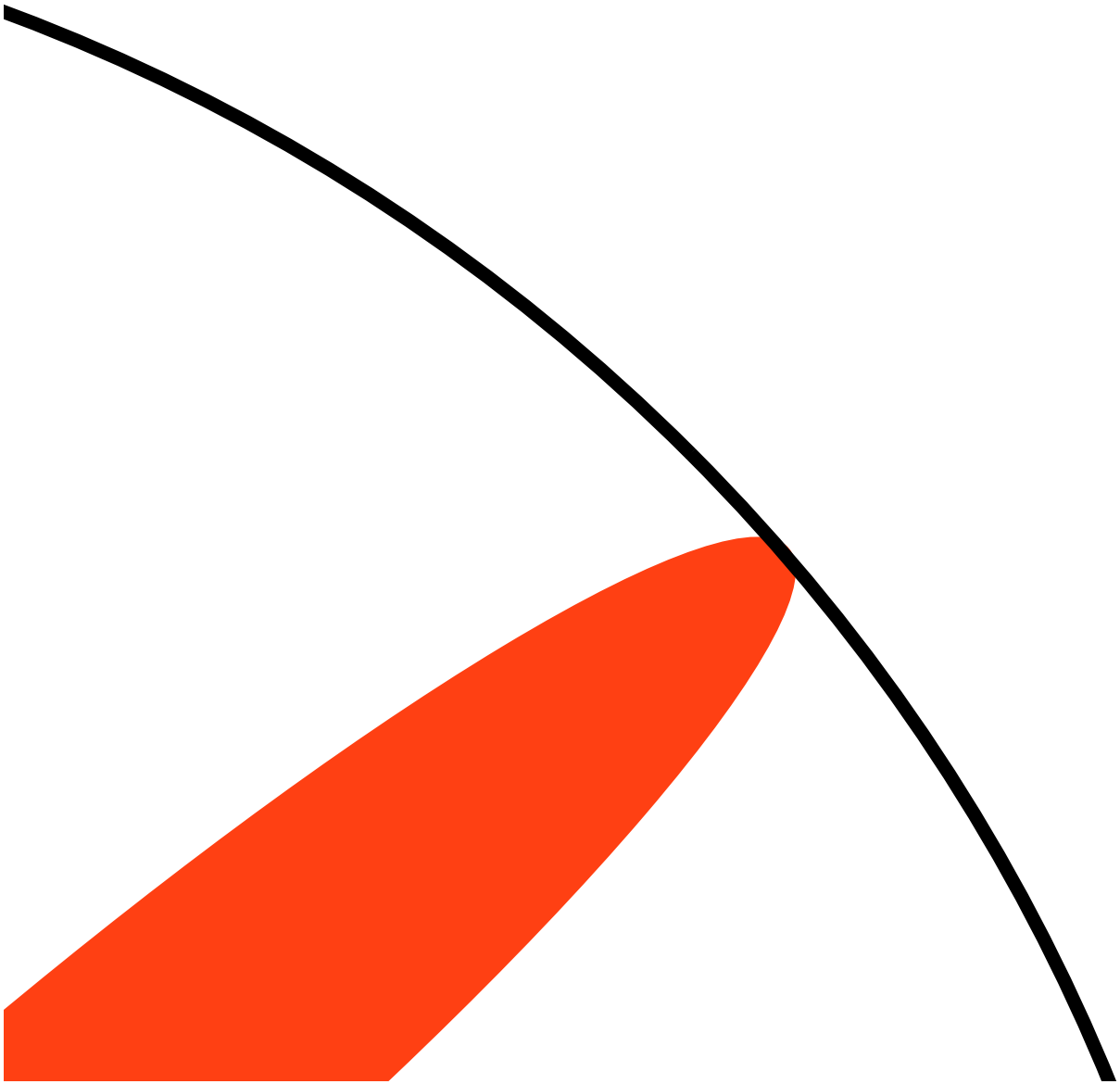
Reading research papers takes you to the edge of human knowledge:



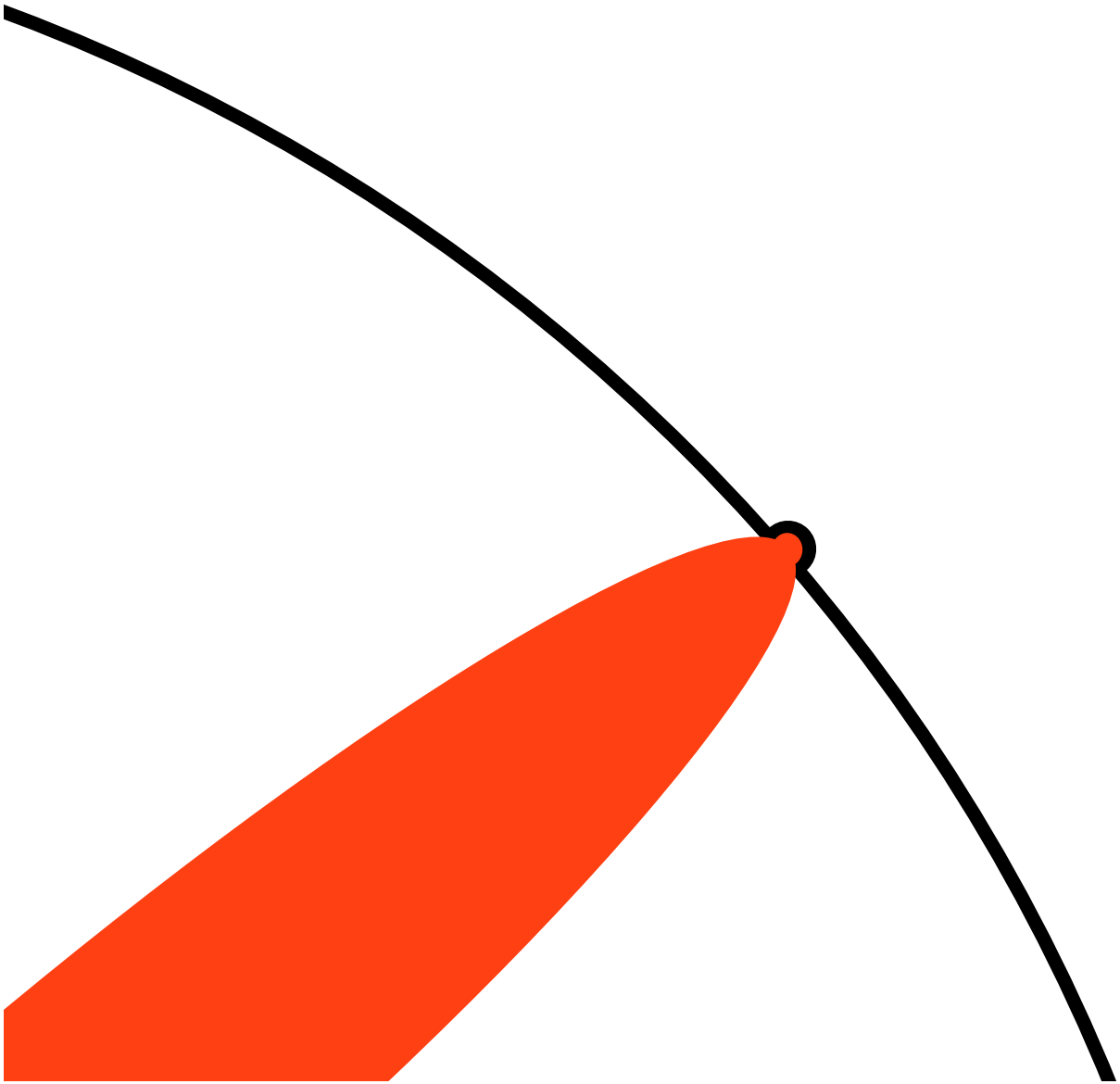
Once you're at the boundary, you focus:



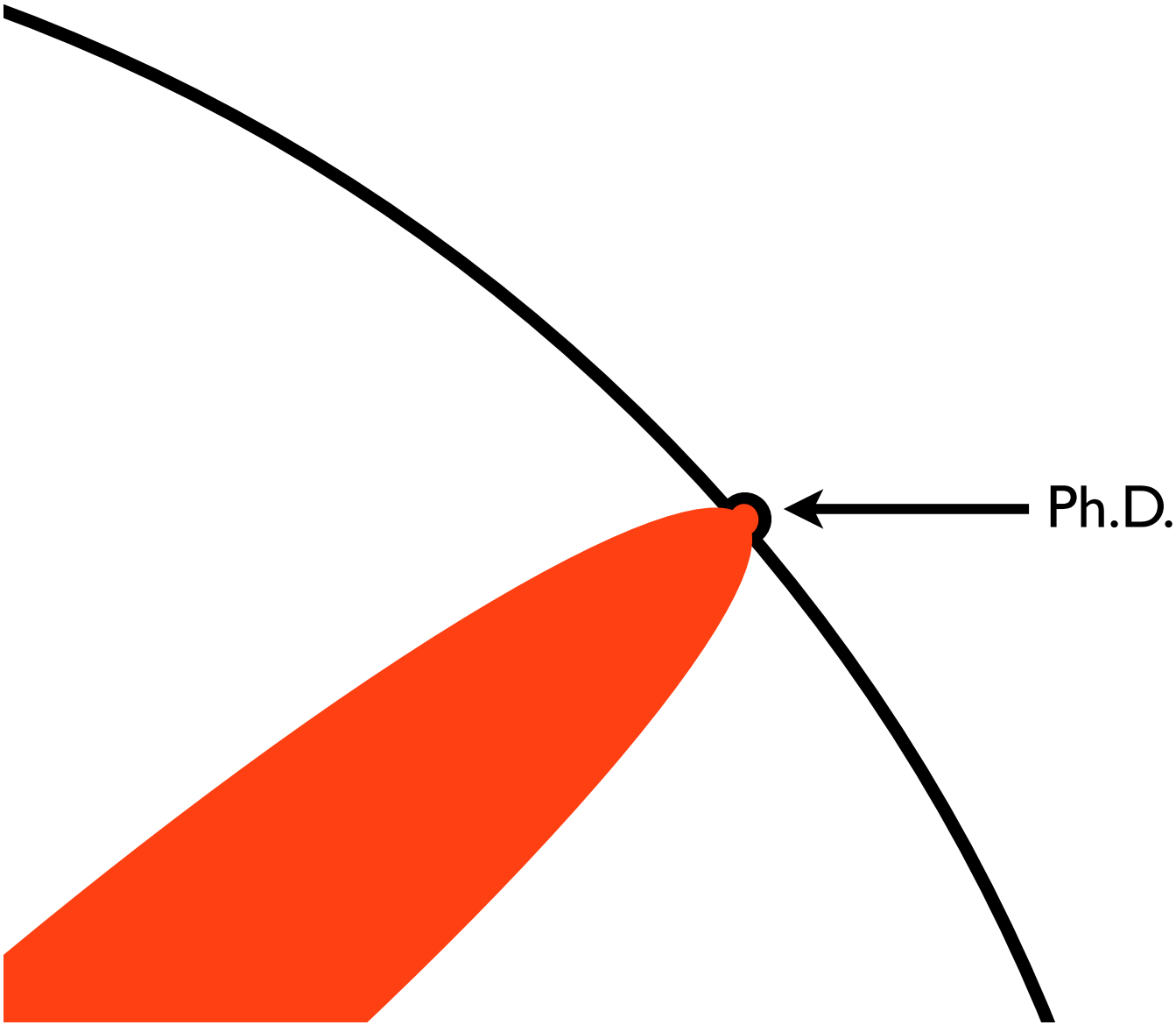
You push at the boundary for a few years:



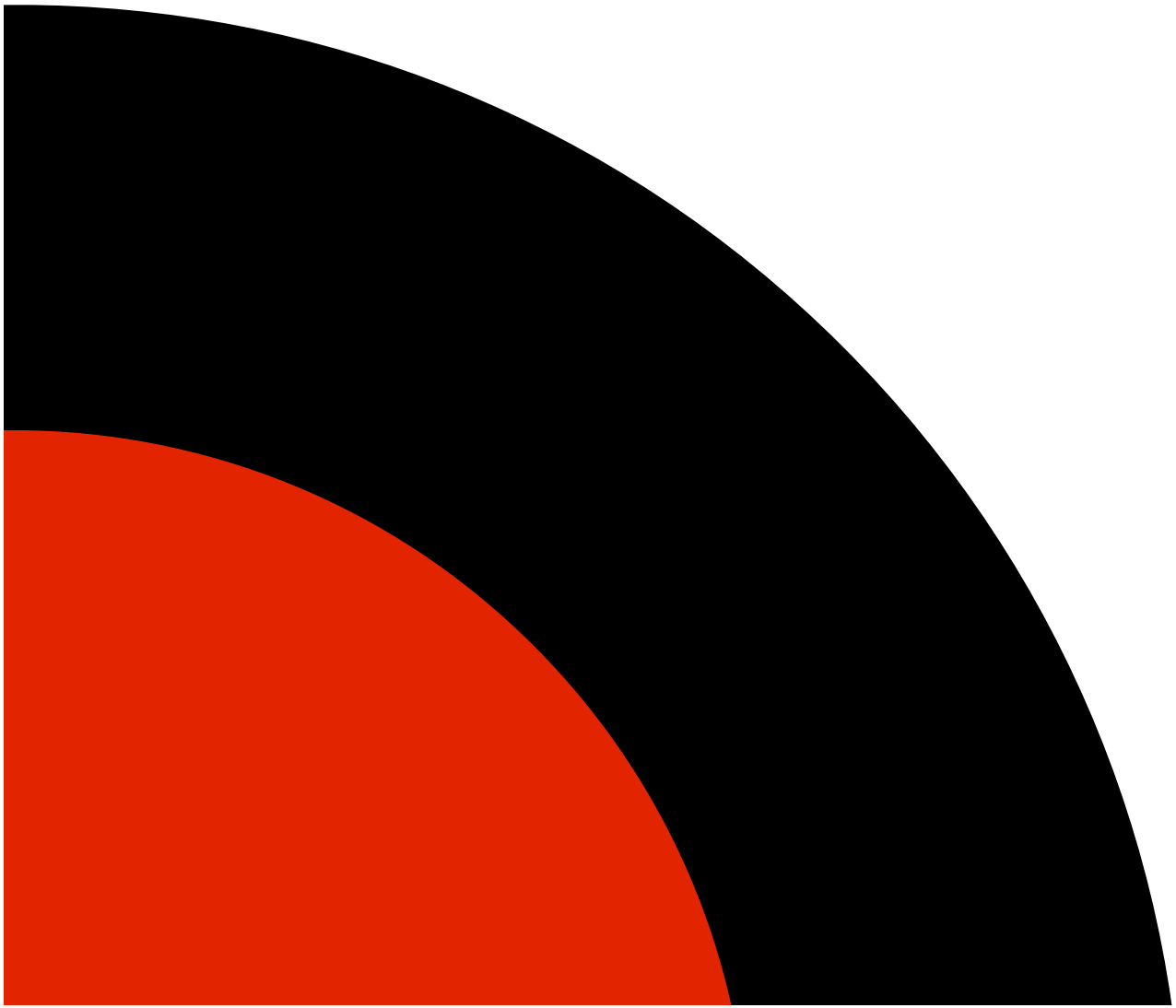
Until one day, the boundary gives way:



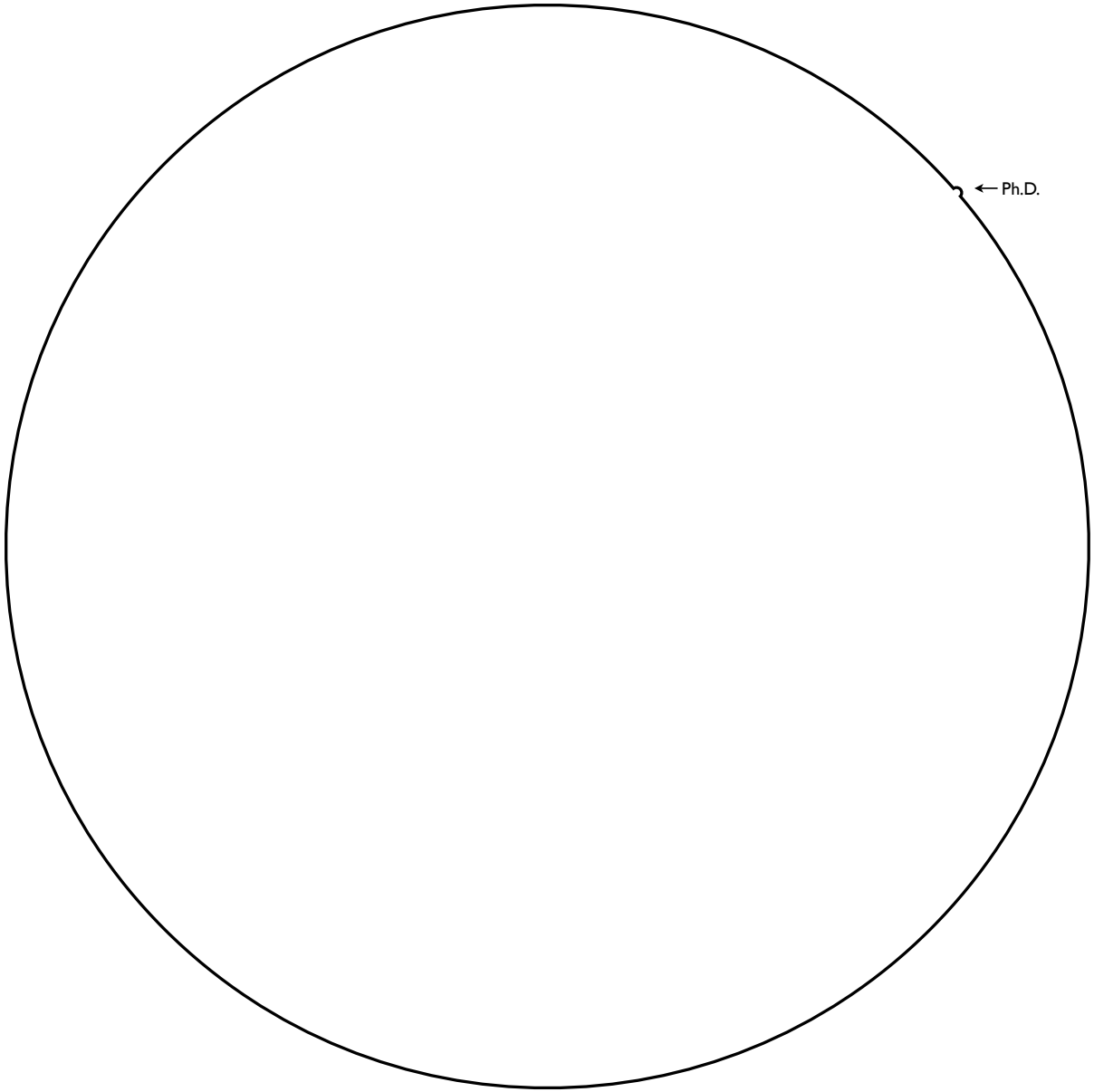
And, that dent you've made is called a Ph.D.:



Of course, the world looks different to you now:



So, don't forget the bigger picture:



Keep pushing.

# Debriefing

This is written by a PhD, for PhD students.

It does not say that PhD know *more*,
They only are expert in one sub-sub-field of *science*
– experience is not depicted here, and at least as important.

There is a lot more, good and bad, to understand about research.

But it's still a pretty good high-level description.


(photo by David Van Horn, CC-By)

# Section 1

## Research

# What is research about?

Working on *hard* problems with no known solution

- Can we detect and remove *all/most* bugs in a large codebase?
- Can the computer check that my implementation respects a *specification*?
- Can concurrent programs be *both* efficient and correct?
- Can this program running on my server leak personal information?

(What is a research problem? Sometimes we get them wrong.)

Presenting results using the Scientific Method

# Scientific Method

Not set in stone, but a sheaf of *social conventions*:
Research is what researchers do – collectively.

Precisely describe your *contribution* – what is *new*.

*Prove* that your idea works.
(The accepted notion of proof depends on the field.)

Know the *previous and related work*. Discuss it copiously.

Highlight the *limitations* of your work.

# Hard problems

There is increasing demand for customizable applications. As applications became more complex, customization with simple parameters became impossible: users now want to make configuration decisions at execution time; users also want to write macros and scripts to increase productivity [1,2,3,4].

How do we build a system that manipulates and stores information whose secrecy and integrity must be preserved?

# Precisely describe your contribution

## Lua – an extensible extension langage – 1996

This paper describes Lua, an extensible procedural language with powerful data description facilities, designed to be used as a general purpose extension language.

## The SLam Calculus: Programming with Secrecy and Integrity – 1998

The SLam calculus is a typed $\lambda$-calculus that maintains security information as well as type information. [...] We prove that the type system *prevents security violations* and give some examples of its power. [...]
Our work is not the first to use a programming language framework for security. [...] The main novelties of our work are the use of both access protection and information ow, and the incorporation of higher-order functions and data structures these are both essential for a development of practical languages that provide mechanis

# Prove that your idea works

## Lua – an extensible extension langage – 1996

Currently, Lua is being extensively used in production for several tasks, including user configuration, general-purpose data-entry, description of user interfaces, storage of structured graphical metafiles, and generic attribute configuration for finite element meshes.

## The SLam Calculus: Programming with Secrecy and Integrity – 1998

**Theorem 4.1 (Non-interference)** Suppose $\emptyset \vdash e : (t, (r, ir))$ and $\emptyset \vdash C[e] : (t', (r', ir'))$, $t'$ is a transparent ground type and $ir \not\sqsubseteq ir'$. If $e'$ is an expression where $\emptyset \vdash e' : (t, (r, ir))$, then $C[e] \simeq C[e']$.

# Cite related work

Python [23] is an interesting new language that has also been proposed as an extension language. However, according to its own author, there is still a need for "improved support for embedding Python in other applications, e.g., by renaming most global symbols to have a 'Py' prefix" [24]. Python is not a tiny language, and has many features not necessary in extension languages, like modules and exception handling. These features add extra cost to applications using the language.

# Know your limitations

Lua, with more adequate data types and pre-compilation, runs 10 to 20 times faster than Tcl. [...] On the other hand, Lua is approximately 20 times slower than C.

The system we have presented is vulnerable to timing attacks.

[SLAM] deals with the essence of computing with secure information, but a number of important issues remain:

- the type system we have presented is monomorphic; clearly this is too restrictive [...]
- Second, our type system is static, but the security of objects changes dynamically; [...]
- Third, [we] must provide ways to reduce the amount of type information that must be specified by a programmer; [...]

# Section 2

## Pragmatics of research

# Foreground (in parallel)

A few months thinking about *your question* and other stuff
Whiteboard, corridor talks, implementation, experiments, *failures*

A few weeks writing about it (for a deadline)

Submission, rejected or accepted, with reviews

Presenting one's results at a conference

# Background

Teaching

Listening to talks

Learning about other people's questions. Ideas, links, collaborations.

Reviewing articles

# Roles

PhD: what is *your* (first) question?

Full-time researchers: watch out for administrativia

Post-docs: publications pressure, the position game

Interns: learn about stuff, pretend to *do* as well

Research engineers: a delicate cooperation

Users: not planned for

# Section 3

# Programming languages and research

# Some research areas

Compilation, Optimization, Runtimes (GC, targeting GPUs, debugging).
Adding expressive programming features without a performance cost.
Benefiting from advanced hardware capabilities without productivity cost.

# Some research areas

Compilation, Optimization, Runtimes (GC, targeting GPUs, debugging).
Adding expressive programming features without a performance cost.
Benefiting from advanced hardware capabilities without productivity cost.

New language constructs (pattern matching, restartable exceptions).
Give us new means of expression.

# Some research areas

Compilation, Optimization, Runtimes (GC, targeting GPUs, debugging).
Adding expressive programming features without a performance cost.
Benefiting from advanced hardware capabilities without productivity cost.

New language constructs (pattern matching, restartable exceptions).
Give us new means of expression.

Type systems and program analysis (System F, Astrée)
Helping program design, construction, documentation and maintenance.

# Some research areas

Compilation, Optimization, Runtimes (GC, targeting GPUs, debugging).
Adding expressive programming features without a performance cost.
Benefiting from advanced hardware capabilities without productivity cost.

New language constructs (pattern matching, restartable exceptions).
Give us new means of expression.

Type systems and program analysis (System F, Astrée)
Helping program design, construction, documentation and maintenance.

Formal proofs (Coq, Compcert, SeL4)
The pursuit of *Bug-free* software – at a cost.

# Research and Industry

Stereotypes:

- engineers are ignorant of what has been done in research
- researchers don't care about practical problems and lack practical experiences

# Research and Industry

Stereotypes:

- engineers are ignorant of what has been done in research
- researchers don't care about practical problems and lack practical experiences

Real world:

- engineers working on *hard problems* very much care about research; dev teams for the Javascript engines at Mozilla, Google, Microsoft or Apple read and comment research articles
- a lot of researchers are also very good programmers; Xavier Leroy implemented the threading library used by `glibc` between 1998 and 2004; in my lab, people write code daily

Yet, research solution are not always practical, and communication is not always easy. Bridging the two worlds is an art.

# Inertia

# Inertia

- 1930-201x: first-class anonymous functions demanded – **80** years

# Inertia

- 1930-201x: first-class anonymous functions demanded – **80** years
- 1960-199x: garbage collection expected – **35** years

# Inertia

- 1930-201x: first-class anonymous functions demanded – **80** years
- 1960-199x: garbage collection expected – **35** years
- 1973-1998: Actors message-passing concurrency inspiring – **35** years

# Inertia

- 1930-201x: first-class anonymous functions demanded – **80** years
- 1960-199x: garbage collection expected – **35** years
- 1973-1998: Actors message-passing concurrency inspiring – **35** years
- 197x-200x: Generics (still source of trouble), type inference – **30** years

# Inertia

- 1930-201x: first-class anonymous functions demanded – **80** years
- 1960-199x: garbage collection expected – **35** years
- 1973-1998: Actors message-passing concurrency inspiring – **35** years
- 197x-200x: Generics (still source of trouble), type inference – **30** years
- 1989-2004: monads inspiring – **15** years

# Inertia

- 1930-201x: first-class anonymous functions demanded – **80** years
- 1960-199x: garbage collection expected – **35** years
- 1973-1998: Actors message-passing concurrency inspiring – **35** years
- 197x-200x: Generics (still source of trouble), type inference – **30** years
- 1989-2004: monads inspiring – **15** years

We're actually getting better!

# Inertia

- 1930-201x: first-class anonymous functions demanded – **80** years
- 1960-199x: garbage collection expected – **35** years
- 1973-1998: Actors message-passing concurrency inspiring – **35** years
- 197x-200x: Generics (still source of trouble), type inference – **30** years
- 1989-2004: monads inspiring – **15** years

We're actually getting better!

OCaml, Haskell, F♯, Scala,
Rust, Ceylon, Julia, ...

# Section 4

## My sub-domain

# Formalisation

How can we study programming languages?

Say you want to study a programming language L:

- write a very precise description of what is a *valid* L program
- write a very precise description of what it means to *run* a L program

We have a time-honored setting to express things very precisely: mathematics.

Our approach: formalize (parts of) programming languages in maths, and *prove* things about them. Maths can be cool when you have good intuitions about the objects of study.

(Not the only approach; you may do statistics on a wide body of existing programs, for example. Or ask cognitive scientists how programming constructs relate to current theories about how our mind work.)

# Semantics

The set of rules for *valid programs* is called the *static semantics* of a language: what you can tell without running the code. `const`, `private`, `List<A>`, `static_cast`...

The set of rules for *running* programs is called the *dynamic semantics* of the language. Functions, loops, exceptions, assignments, serialization, reflection...

Few existing languages have precisely defined semantics.
Standards documents (C, C++) are often a bit too vague.
Without standards or spec, look at the implementation (Python, Ruby)
Things are more complex than they should be (Python's `nonlocal`)

# Section 5

## Some projects in my lab

# OCaml

Xavier Leroy, Damien Doligez,
Alain Frisch, Jacques Garrigue, Didier Rémy, Jérôme Vouillon

A general-purpose programming language.

- functional, statically-typed, with algebraic datatypes
- efficient compiler, highly optimized GC
- expressive yet relatively-simple type system
- clunky syntax
- object-oriented programming possible

Active research until 2000, mostly maintenance now.
Free software (contribute!)
Large codebases depend on it, in research and/or industry.

We don't have much time for things people want: nice IDEs, etc.
Rely on the community.

# Mezzo

François Pottier, Jonathan Protzenko, Thibaut Balabonski

To reason about memory management and concurrent programs, most programmers think about *ownership* of data.

Mezzo brings this (unspoken) invariant directly in the type system.
Aliasing mutable data without explicit checks is an error.
Race-free concurrency by construction.

Mostly an experiment so far (there is an implementation, but it's still a prototype).
Long-term, we think robust languages will support this.

The theory of separation logic is complex. Presenting it as a functional language helped get a coherent design.

# Type systems

Didier Rémy, François Pottier, Julien Cretin, Gabriel Scherer...

Type inference: can we *guess* the type annotation needed for my program to compile? In a non-ambiguous way?

Coercions: when can we say that a type *is more general* than another? Can we forget about the difference during compilation?

Code inference: can typing information help us write the code? In a non-ambiguous way?

# Compcert

Xavier Leroy,
Sandrine Blazy, Zaynah Dargaye, Jacques-Henri Jourdan, and
Jean-Baptiste Tristan

A compiler for C that comes with its *correctness proof*.
Backends for ARM, x86 and PowerPC.
Performance close to `gcc -O1`.

(Silly? John Regher found *hundreds of bugs* in production compilers (GCC, Clang, ICC, etc.) by random testing – hard)

High-level idea:

- write a dynamic semantics of your source and target language
- prove that your compilation function preserves the semantics

In practice, a dozen of small passes (as in any compiler), with their proof.

# But also...

Verified compilation can be applied to high-level languages (functional langages, C++...) and more exotic settings. Thomas Braibant has worked on verified compilers for *hardware circuits*.

Today's CPUs are so complex that we need *experimental science* to model their behaviour. Luc Maranget tests and formalizes so-called *weak memory models*. Jacques-Pascal Deplaix helps testing the new C11 atomic memory operations.

Manycore systems with shared memory need new data structures and algorithm to express synchronization in parallel algorithms. Umut Acar, Arthur Charguéraud and Mike Rainey.

Language verification technology requires copious amounts of automated reasoning. Damien Doligez works on the Zenon automatic prover, and TLA+ temporal logic system.

# Conclusion

Research is great – as plenty of other things you can get paid for.

Researchers alone aren't that useful.
We need people to *create problems*.

Research implies communication of knowledge.
Anyone should take advantage of that.

Questions?