(When) Will
Property-Based Testing
Rule The World?

Benjamin C. Pierce
University of Pennsylvania

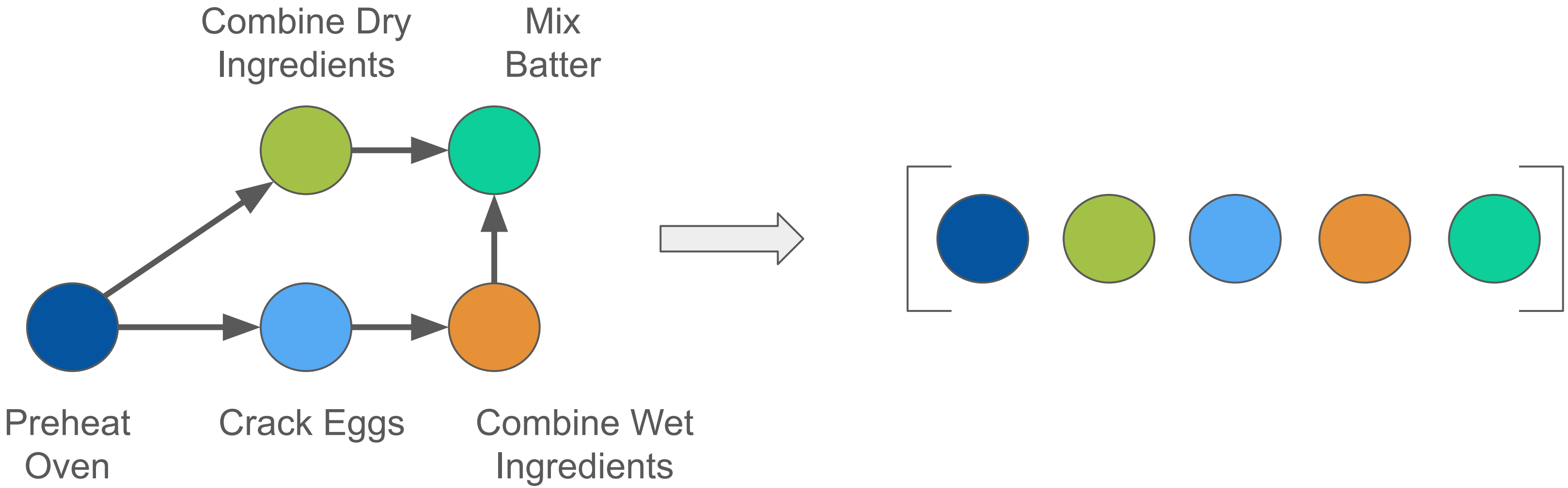
YOW! Lambda Jam 2022

What is property-based testing?

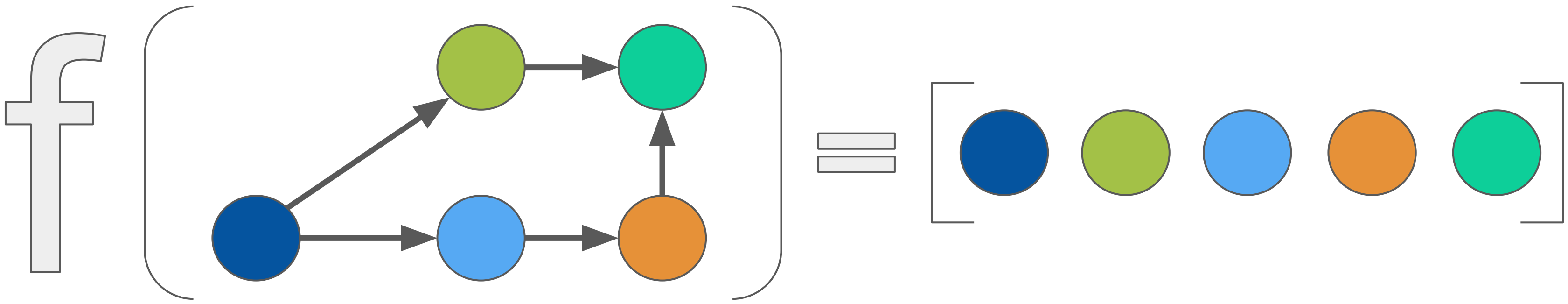
An Annoying Testing Scenario…

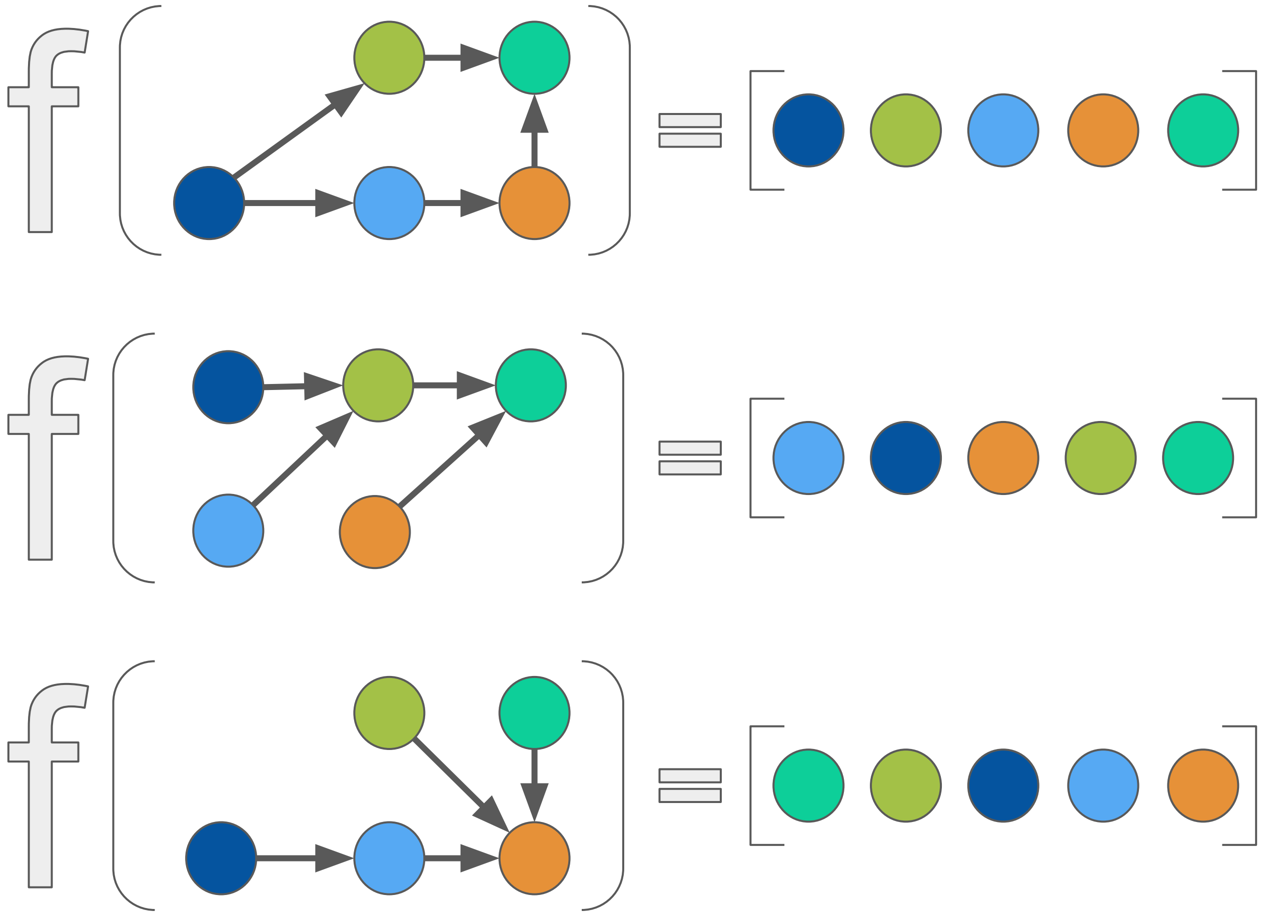
What has to happen before what?

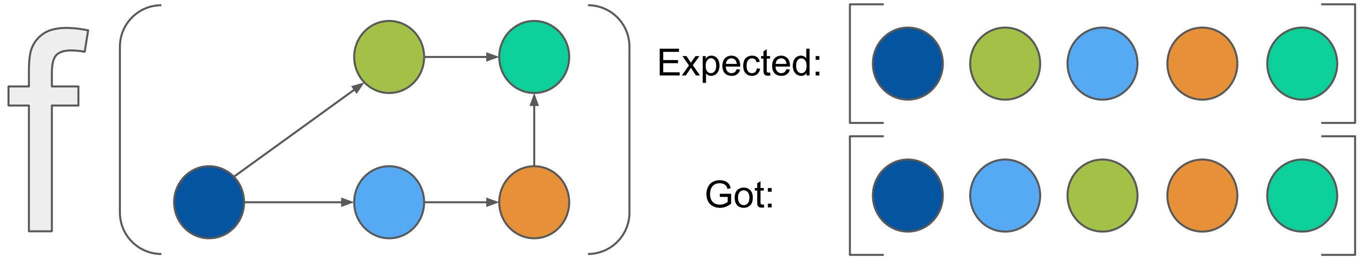
What order do I actually do things in?



(This could really be any set of actions with dependencies! Cf. “DAGs and topological sorting.”)

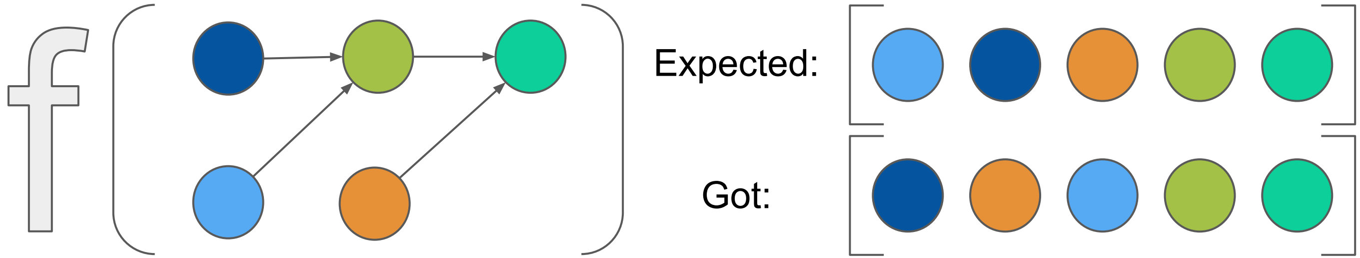

f


f
f
f



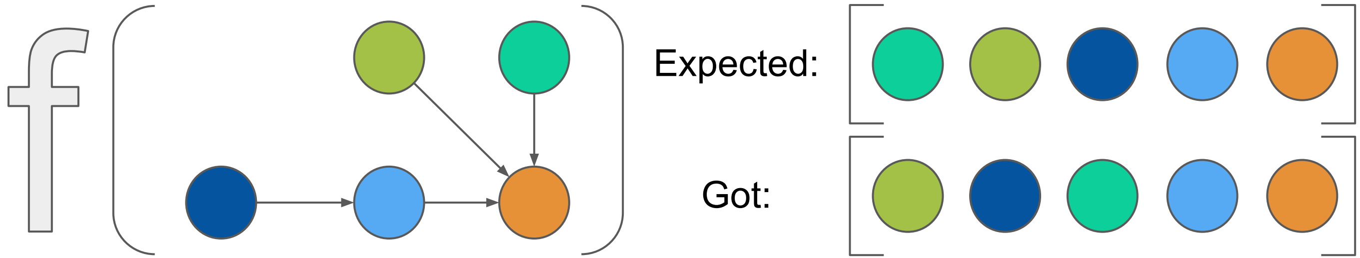
Expected:

Got:



Expected:

Got:



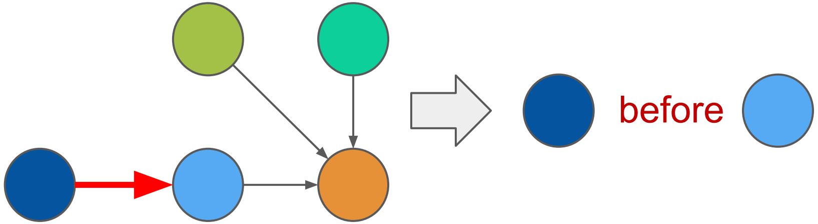
Expected:

Got:

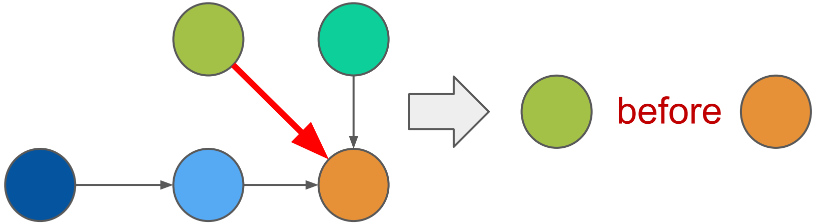
What to do?

1. Go back and fix up the expected outputs in all the tests?
2. Try something different?

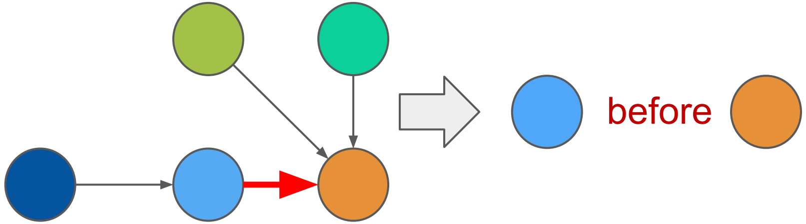
What does it **mean** to say that this function is “correct”?



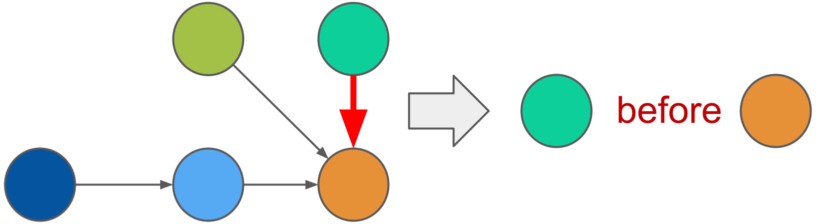
before



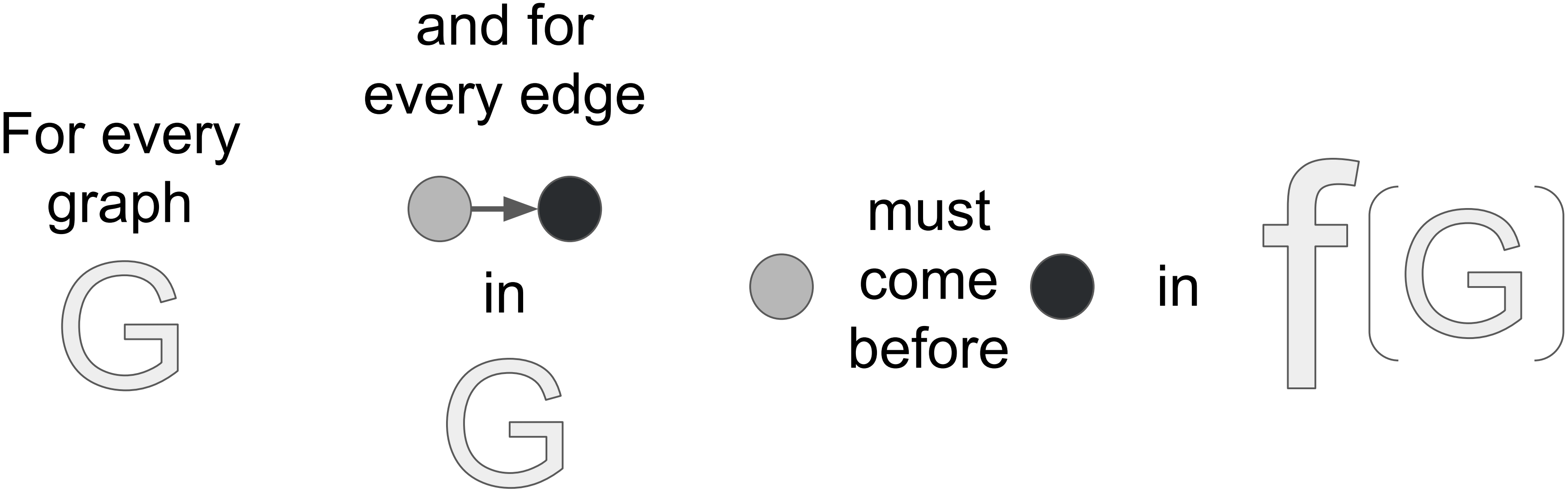
before



before



before



For every graph

G

and for every edge

○ → ●

in

G

○ must come before ● in f(G)

In Haskell…

```
fCorrectOn G =
  let s = f G in
    all (\(v, w) -> index v s < index w s) (edges g)
```

Property-Based Testing

Basic idea

1. Write down a **property**
    - … as a Boolean function taking a concrete input and yielding True if the system behaves as desired on this particular input
2. Apply it to many test **inputs**
3. If it ever yields False, report a **bug!**

… sampled from some random distribution

… or enumerated in some order

… or …

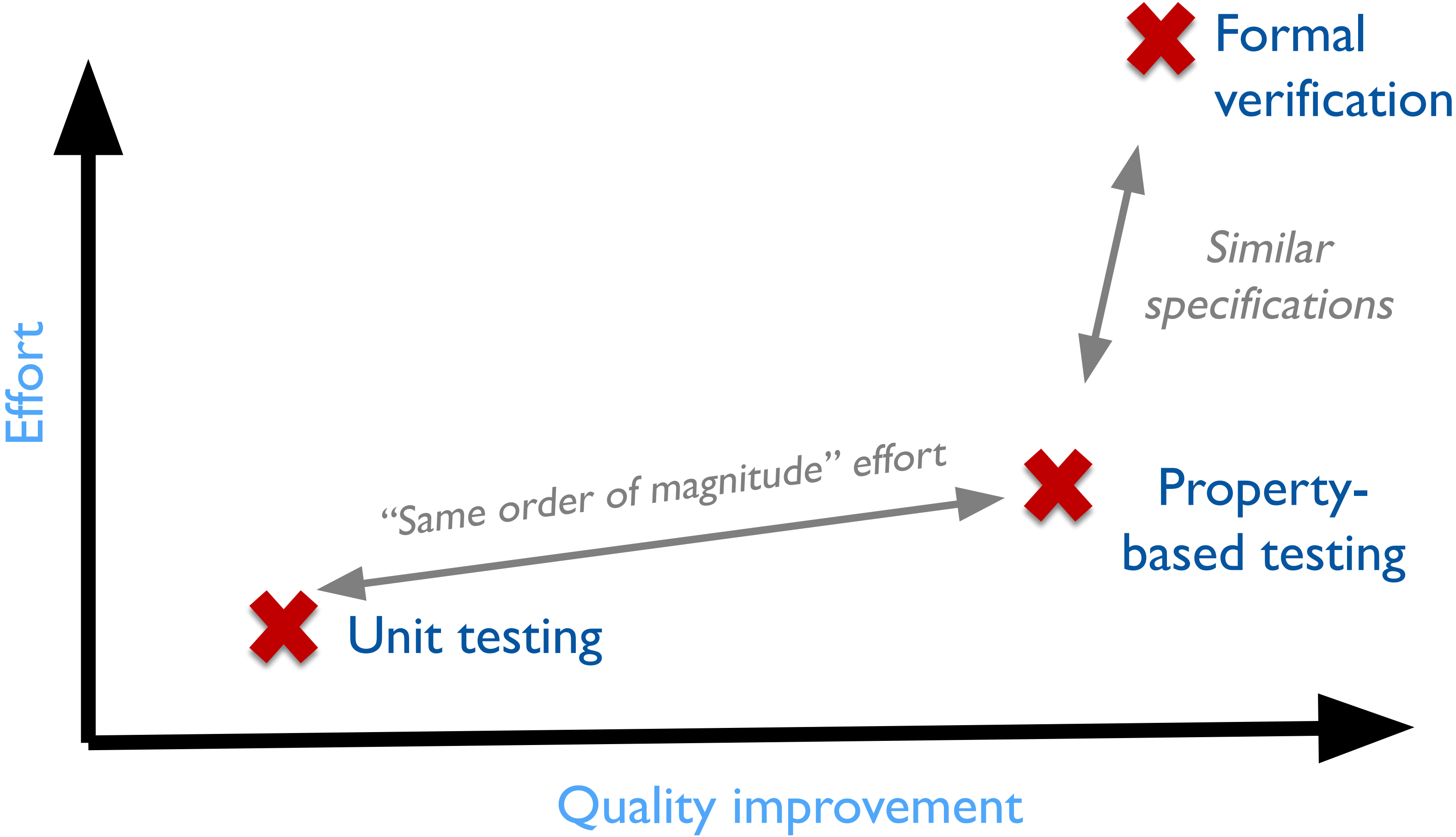
In Haskell…

```
fCorrectOn G =
  let s = f G in
    all (\(v, w) -> index v s < index w s) (edges g)

prop_fCorrect =
  forAll generateDAG (\G -> fCorrectOn G)

> quickCheck prop_fCorrect
+++ OK, passed 100 tests.
```

QuickCheck Family

- C (theft)
- C++ (CppQuickCheck)
- Clojure (test.check)
- Coq (QuickChick)
- F# (FsCheck)
- Go (gopter)
- Haskell (QuickCheck or Hedgehog)
- Java (QuickTheories)
- JavaScript (jsverify)
- PHP (Eris)
- Python (Hypothesis)
- Ruby (Rantly)
- Rust (Quickcheck)
- Scala (ScalaCheck)
- Swift (Swiftcheck)

And more!



Formal verification

Similar specifications

Property-based testing

"Same order of magnitude" effort

Unit testing

Effort

Quality improvement

(not to scale!)

Lightweight Formal Methods

Supports automation!

> Formal method: A mathematically rigorous technique for validating the actual behavior of a program against a description of its desired behavior.

Demands automation!

> *Lightweight* formal method: one that can be applied successfully by people that don’t understand it.”

:-)

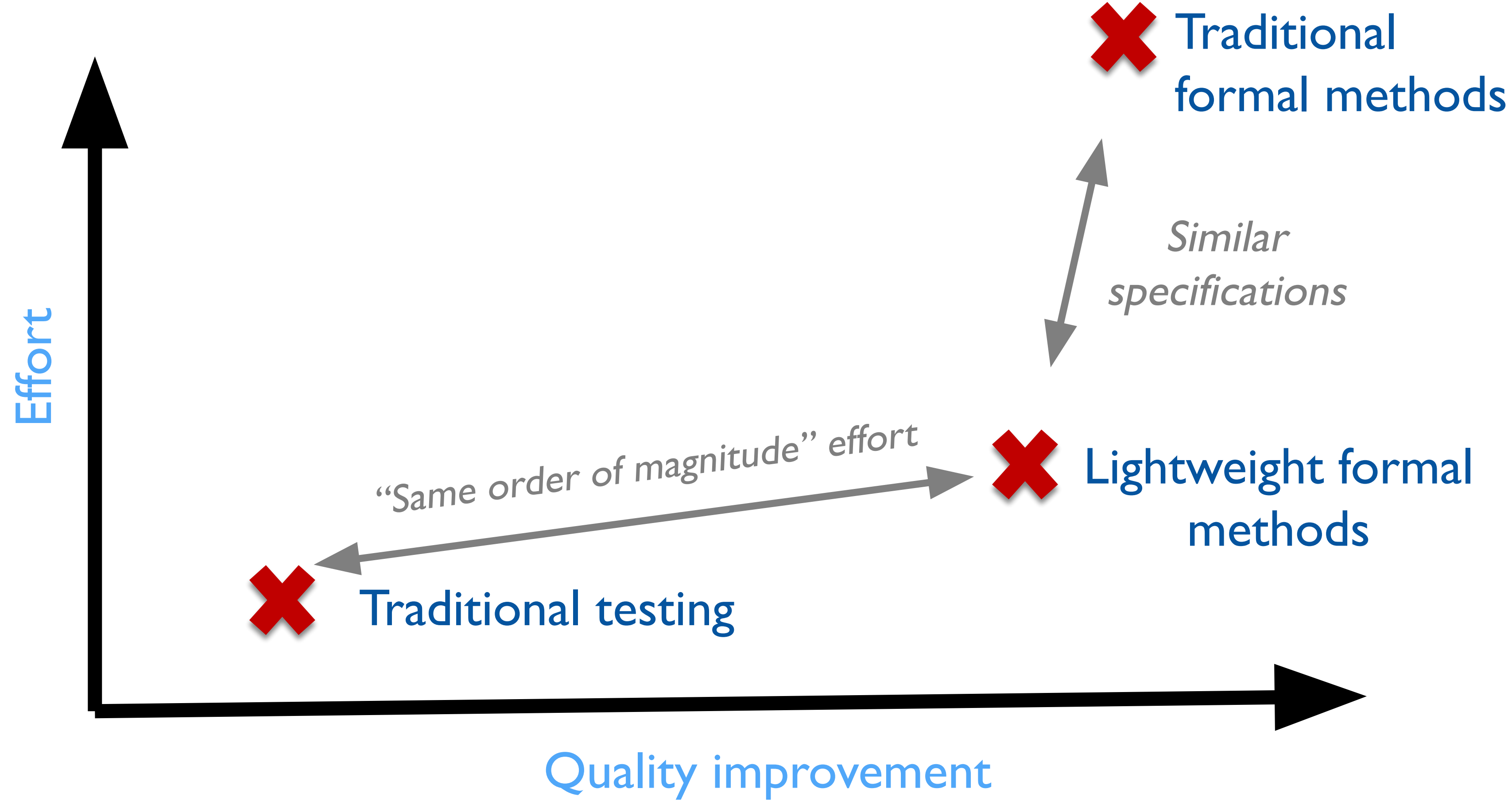
> “Industry will have no reason to adopt formal methods until the benefits of formalization can be obtained immediately.”

— Daniel Jackson and Jeannette Wing

https://people.csail.mit.edu/dnj/publications/ieee96-roundtable.html

Lightweight formal methods

- Property-based testing
- Model checking
- Types
- etc.



Effort

Quality improvement

Traditional testing

"Same order of magnitude" effort

Lightweight formal methods

Similar specifications

Traditional formal methods

(not to scale!)

> “The future is already here. It’s just not evenly distributed yet.”
> — (attributed to) William Gibson

Success Stories

Rust's PropTest tool was used to test that a new key-value store node implementation for S3 matches a reference implementation.

PBT is used in tandem with other lightweight formal methods like model checking.

REMS

http://rems.io

“Rigorous Engineering of Mainstream Systems”

Formal specifications of a range of critical interfaces, validated against real-world artifacts using PBT…

- X86 instruction set
- TCP protocol suite
- Posix file system interface
- Weak memory consistency models for x86, ARM, PowerPC
- ISO C / C++ concurrency
- Elf loader format
- C language

QuviQ

AUTOSAR

- Engineers at the PBT company Quviq built an executable specification based on the 3000-page AutoSAR standard for automotive software components
- QuickCheck-based testing found >200 faults in AutoSAR Basic Software, including >100 inconsistencies in the standard

QuviQ

> “We helped Basho test their no-SQL database, Riak, for the key property of eventual consistency—and found a bug (now fixed, of course) that was present, not only in Riak, but in the original Amazon paper … that kicked off the no-SQL trend.”
>
> - John Hughes
>   *Experiences with QuickCheck*

QuviQ

LEVELDB

- Used state-machine testing to generate large sequences of API calls
- Found long and hard-to-find sequences of operations that corrupted databases

What’s happening at Penn

(Shock, horror…)

Property-Based Testing Isn’t Perfect

- If PBT were a silver bullet for everything, it would be used for everything
- Three broad categories of problems:
  1. **Appropriateness** – PBT is *shockingly effective* in some domains; in others, it might not be the right tool; in others, we don’t know
  2. **Effectiveness** – even in the domains where it works well, there’s plenty of room for improvement
  3. **Usability** – it can be hard to know what properties to test and how to integrate PBT into software workflows

- We’ve done lots of work on (1) and (2)
- We’re starting to think very seriously about (3)

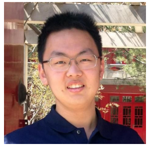
“Testing the Hard Stuff”

HT John Hughes

Case Studies

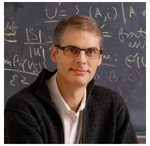
- Testing security properties (dealing with sparse preconditions and hard-to-falsify properties)
- Dropbox testing (flakey tests, distributed, time-sensitive, …)
- DeepSpec server (interactive systems)


Leonidas Lampropoulos


Yishuai Li


John Hughes

(And others!)


Benjamin C. Pierce


Li-Yao Xia

Dropbox testing

We used a QuickCheck specification of DropBox to find several new bugs in its behavior

Improving Random Generation

Picking Tests is Hard!

- Enumerating small inputs doesn’t give good coverage
- Effectiveness of random test generation depends a lot on sampling from the right distribution
- Lots of properties have preconditions that we need to worry about – “rejection sampling” doesn’t work!

```
prop_fCorrect =
  forAll generateDAG (\G -> fCorrectOn G)
```

Deriving Generators from Predicates

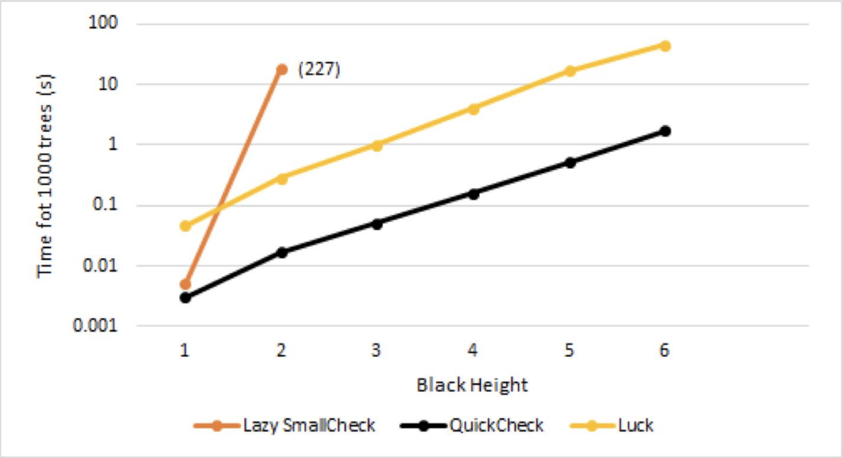
Generating Good Generators for Inductive Relations [POPL’18]

Beginner’s Luck [POPL’17]

Predicate: $\forall x.\ \phi(x)$

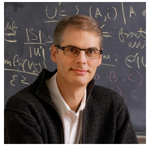
↓

Generator: G_ϕ




Leonidas Lampropoulos


Cătălin Hrițcu


Benjamin C. Pierce


John Hughes


Zoe Paraskevopoulou


Li-Yao Xia

Diane Gallois-Wong

Holey Generators! (Under Submission)

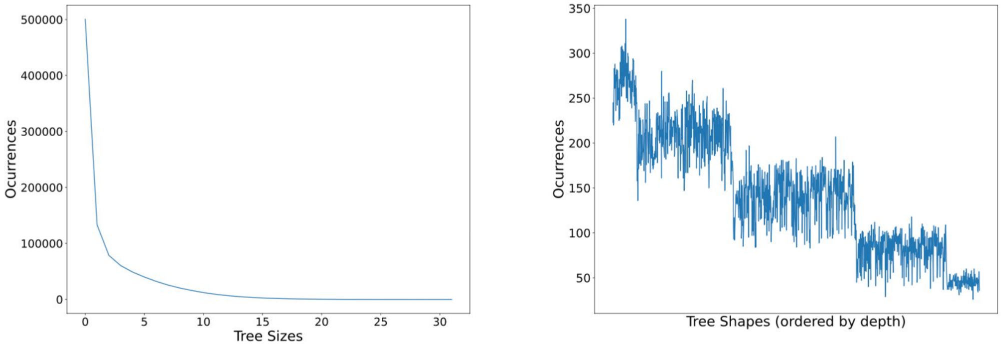


Fig. 1. Left: The size distribution of one million generated BSTs. Right: the shape distribution of BSTs of size 8, ordered shortest to tallest by depth (note that the smallest possible depth is 4).

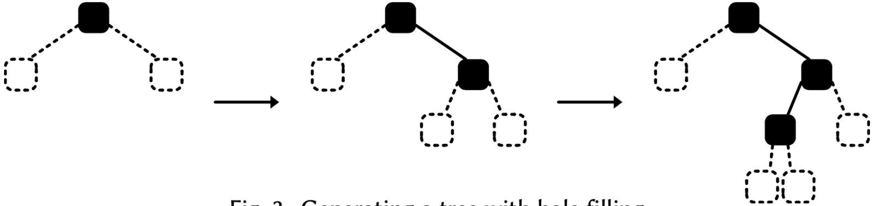
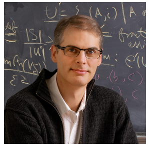


Fig. 3. Generating a tree with hole filling

Joseph W. Cutler



Benjamin C. Pierce

Harrison Goldstein

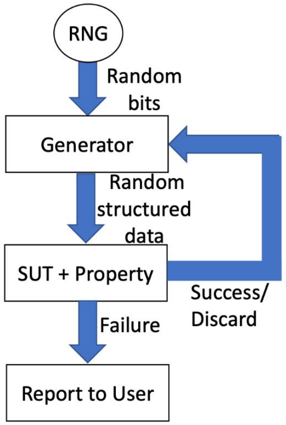
Koen Claessen

John Hughes

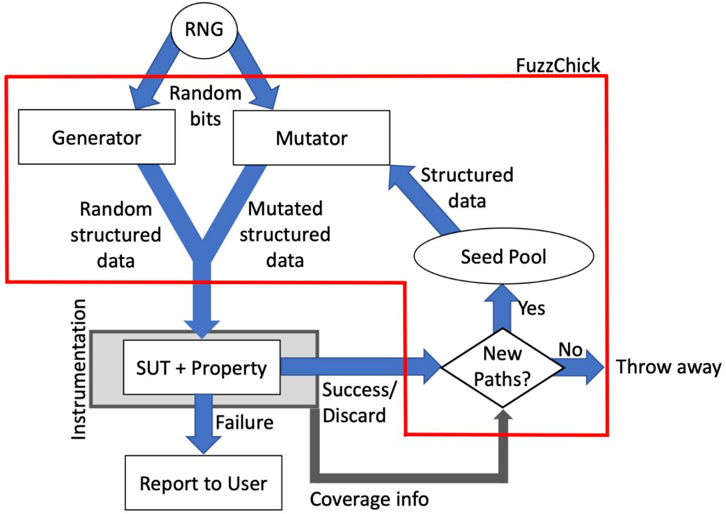
Incorporating Other Testing Techniques

Coverage Guided Property-Based Testing

Coverage Guided, Property Based Testing [OOPSLA'19]


RNG
Random bits
Generator
Random structured data
SUT + Property
Success/ Discard
Failure
Report to User

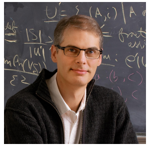
(a) QuickChick


RNG
Random bits
FuzzChick
Generator
Mutator
Random structured data
Mutated structured data
Structured data
Seed Pool
Yes
New Paths?
No
Throw away
Instrumentation
SUT + Property
Success/ Discard
Failure
Report to User
Coverage info

(b) FuzzChick


Leonidas Lampropoulos


Michael Hicks


Benjamin C. Pierce

Combinatorial Property-Based Testing

Do Judge a Test by its Cover [ESOP'21]

1. `w = False   x = False   y = False   z = False`
2. `w = False   x = True    y = True    z = True`
3. `w = True    x = False   y = True    z = True`
4. `w = True    x = True    y = False   z = True`
5. `w = True    x = True    y = True    z = False`

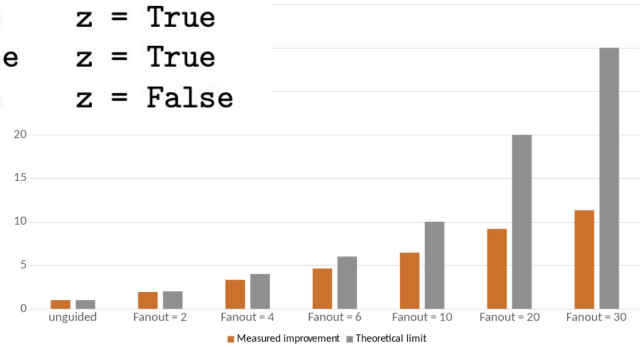
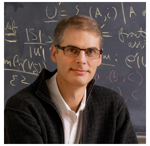


Fig. 2. System F, proportional reduction in total number of tests needed to find all bugs.

Harrison Goldstein

John Hughes

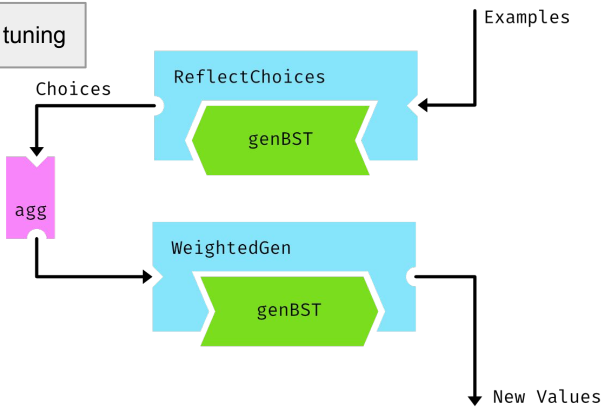
Leonidas Lampropoulos



Benjamin C. Pierce

Reflective Generators (Work in Progress!)

Example-based tuning

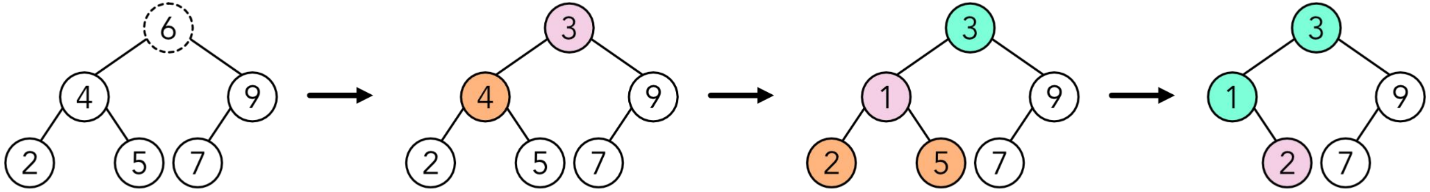


Run the generator “backward” to obtain the random choices used to arrive at particular examples

Tune distribution accordingly

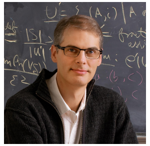
Run generator forward to get new inputs based on this distribution

Validity-preserving mutation



Samantha Frohlich

Harrison Goldstein



Benjamin C. Pierce

Meng Wang

Backtracking Generators (Work in Progress!)

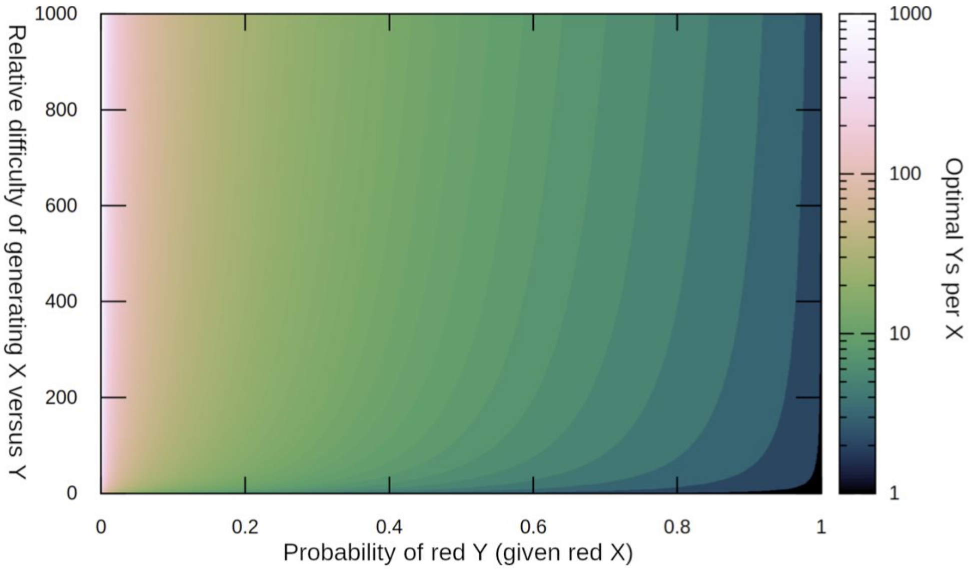
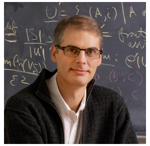


Fig. 1. A heatmap showing the optimal value for `how_many` under varying choices of p_y (on the X axis) and r (on the Y axis). Darker colors denote smaller optimal values.

Calvin Beck

Leonidas Lampropoulos



Benjamin C. Pierce

John Hughes

So… when will PBT rule the world??

When we get more scientific!

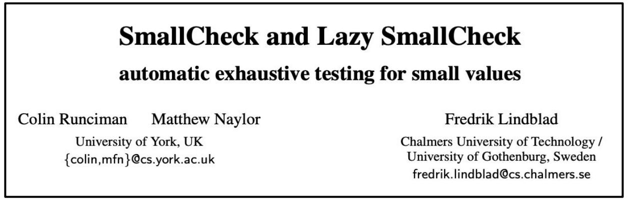
1. More rigorous ways of evaluating and comparing PBT techniques and technologies
2. Clearer picture of what potential users actually need and what are the barriers to adoption

A Common Benchmark Suite

**QuickCheck:
A Lightweight Tool for Random Testing
of Haskell Programs**

Koen Claessen
Chalmers University of Technology
koen@cs.chalmers.se

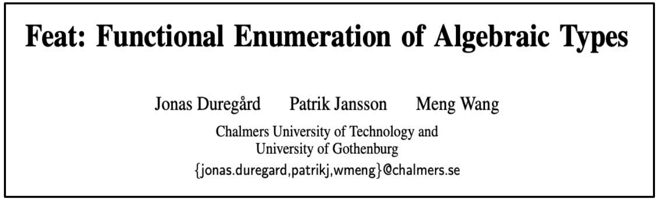
John Hughes
Chalmers University of Technology
rjmh@cs.chalmers.se



SmallCheck and Lazy SmallCheck
automatic exhaustive testing for small values

Colin Runciman Matthew Naylor
University of York, UK
{colin,mfn}@cs.york.ac.uk

Fredrik Lindblad
Chalmers University of Technology /
University of Gothenburg, Sweden
fredrik.lindblad@cs.chalmers.se



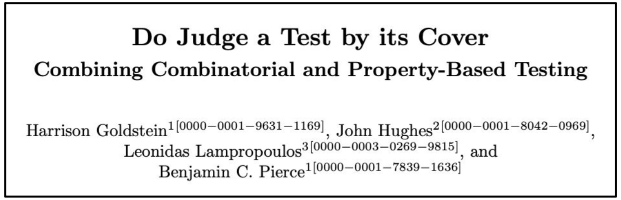
Feat: Functional Enumeration of Algebraic Types

Jonas Duregård Patrik Jansson Meng Wang
Chalmers University of Technology and
University of Gothenburg
{jonas.duregard,patrikj,wmeng}@chalmers.se

Deriving Compositional Random Generators

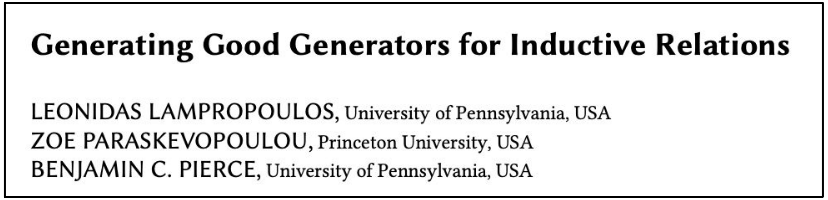
Agustín Mista
Chalmers University of Technology
Gothenburg, Sweden
mista@chalmers.se

Alejandro Russo
Chalmers University of Technology
Gothenburg, Sweden
russo@chalmers.se



Do Judge a Test by its Cover
Combining Combinatorial and Property-Based Testing

Harrison Goldstein[1][0000-0001-9631-1169], John Hughes[2][0000-0001-8042-0969],
Leonidas Lampropoulos[3][0000-0003-0269-9815], and
Benjamin C. Pierce[1][0000-0001-7839-1636]



Generating Good Generators for Inductive Relations

LEONIDAS LAMPROPOULOS, University of Pennsylvania, USA
ZOE PARASKEVOPOULOU, Princeton University, USA
BENJAMIN C. PIERCE, University of Pennsylvania, USA

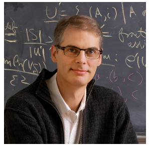
Coverage Guided, Property Based Testing

LEONIDAS LAMPROPOULOS, University of Maryland, USA and University of Pennsylvania, USA
MICHAEL HICKS, University of Maryland, USA
BENJAMIN C. PIERCE, University of Pennsylvania, USA

and more…

Goal: a benchmark framework for comparing property-based bug-finding methodologies

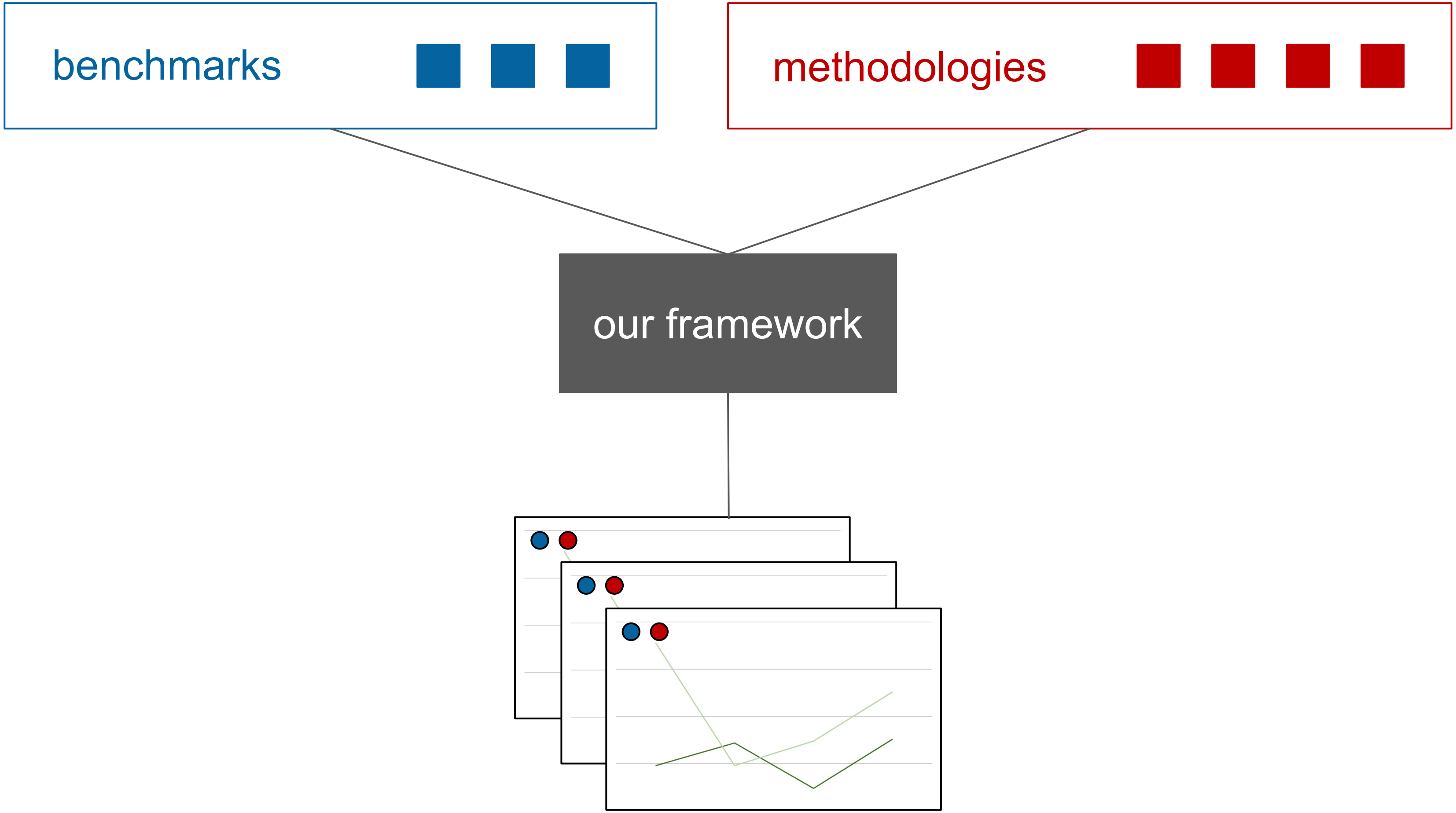

Alperen Keles


Benjamin C. Pierce


Jessica Shi


Harrison Goldstein


Leonidas Lampropoulos



benchmarks

methodologies

our framework

Outcomes

- A better way to know we’ve succeeded when we develop new testing tools!
- A canonical, comprehensive list of interesting testing problems
- As we see what tools succeed and fail at what, we may get new ideas about how to combine the strengths of multiple approaches

A Preliminary User Study

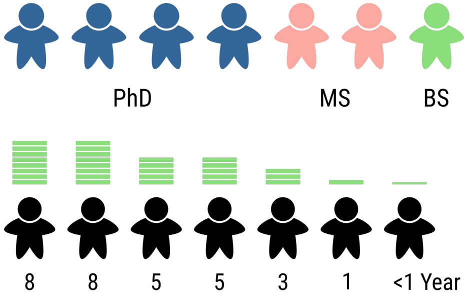
Property-Based Testing For Everyone?

How do we find out what would help more people use PBT?

Preliminary User Study

Focused on “interviews for need finding.”

Recruited 7 industrial Python programmers who use the *Hypothesis* PBT tool.



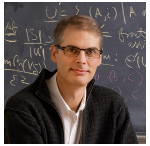
Interview Questions

1. “Tell us about your most memorable time doing PBT.”
(To get subjects thinking about a specific experience.)

2. “How did you come up with the properties that you tested?”

3. “Did you need custom generators? If so, what did they generate?”

Joseph W. Cutler



Benjamin C. Pierce

Harrison Goldstein

Adam Stein

Andrew Head

What Have We Learned (So Far)?

What have we learned?

1. People who like PBT **really** like it!

:-)

What have we learned?

2. There are two (surprisingly distinct) classes of users…

Power Users

- Fully “bought in”
- Often have strongly mathematical backgrounds (often PhD in Math/CS)
- Care about testing efficiency
- Tend to test properties corresponding to the math behind their code

Need better generators!

Occasional Users

- Use PBT occasionally
- More traditional software engineering backgrounds
- Tend to test simple, “extremal” properties:
  - “Program doesn’t crash”
  - “Program behaves exactly like oracle”

Need help “seeing” properties!

These groups can teach us different things!

What have we learned?

3. PBT requires cleanly abstracted code
    - In particular, functions tested with PBT should be relatively “pure”
    - Some informants reported that “carving out” an interface was much of their testing effort
    - Others reported resorting to “end-to-end” properties like “the whole system does not crash”
    - “I can’t see any properties to test” was a common refrain

What have we learned?

4. We need to do a better job of *teaching* PBT!
    - Several informants cited lack of examples / experience as a problem
    - PBT documentation often uses terminology unfamiliar to engineers
    - Incorporating PBT into CS education is critical!

Shriram Krishnamurthi has written a ton about how to do this!

Preliminary Takeaways

- For power users, a central problem is easily writing generators that effectively test the properties they care about
- For occasional users a central problem is understanding how to formulate even fairly simple properties
- PBT education (example repos, teaching materials, …) deserves more attention!

Scaling up

Comprehensive Benchmarking

- Our initial goals with benchmarking are modest, but eventually we hope to build the world's best PBT benchmarking framework
- This means we need **examples**!
- And we need people that want **test their tools** against our suite!
- Send us an email if you're interested in this kind of stuff

`bcpierce@cis.upenn.edu`

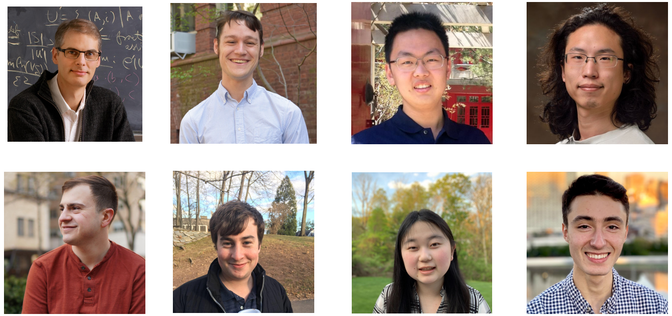
Full-Scale User Study

- We want to know *much more* about how PBT can be improved, especially for new / occasional users
    - Where is PBT especially useful? Especially difficult to implement?
    - What kinds of programs actually have useful properties? Do people see them?
    - How could we best integrate PBT into the software development process?
- Hope to talk to industry users, industry non-users (tried it, didn't like it?), and even tech leads and managers
- If you're willing to chat with us, fill out this form to let us know!

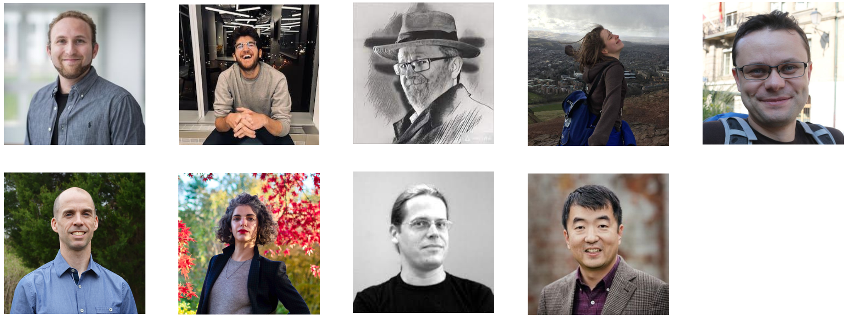
`https://tinyurl.com/pbt-at-penn`

Thank you!

Questions?



University of Pennsylvania



External Collaborators

PBT
@Penn

bcpierce@cis.upenn.edu
https://tinyurl.com/pbt-at-penn

PBT
@Penn

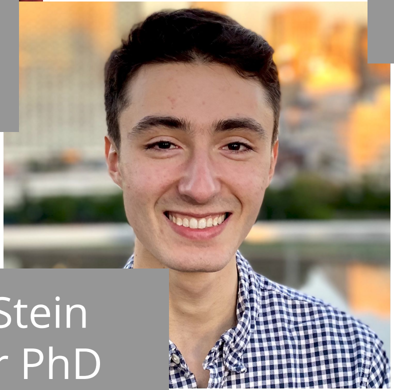
Thank you!
(Questions?)

Joseph W. Cutler
1st Year PhD

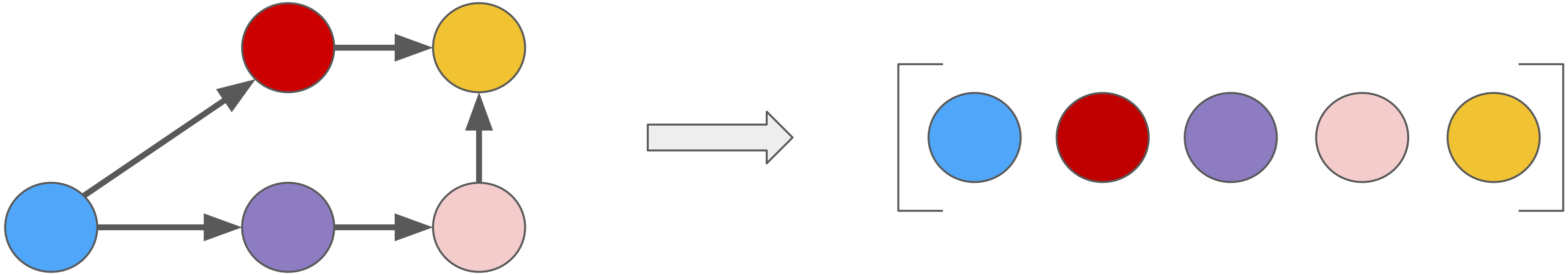
Jessica Shi
1st Year PhD

Harrison Goldstein
3rd Year PhD

Andrew Head
Asst. Prof



Adam Stein
1st Year PhD



Basic idea

1. Write down a property as a Boolean function mapping a concrete input to True if the system behaves as desired on this particular input
2. Apply this function to many test inputs
3. If the property ever yields False, report a bug

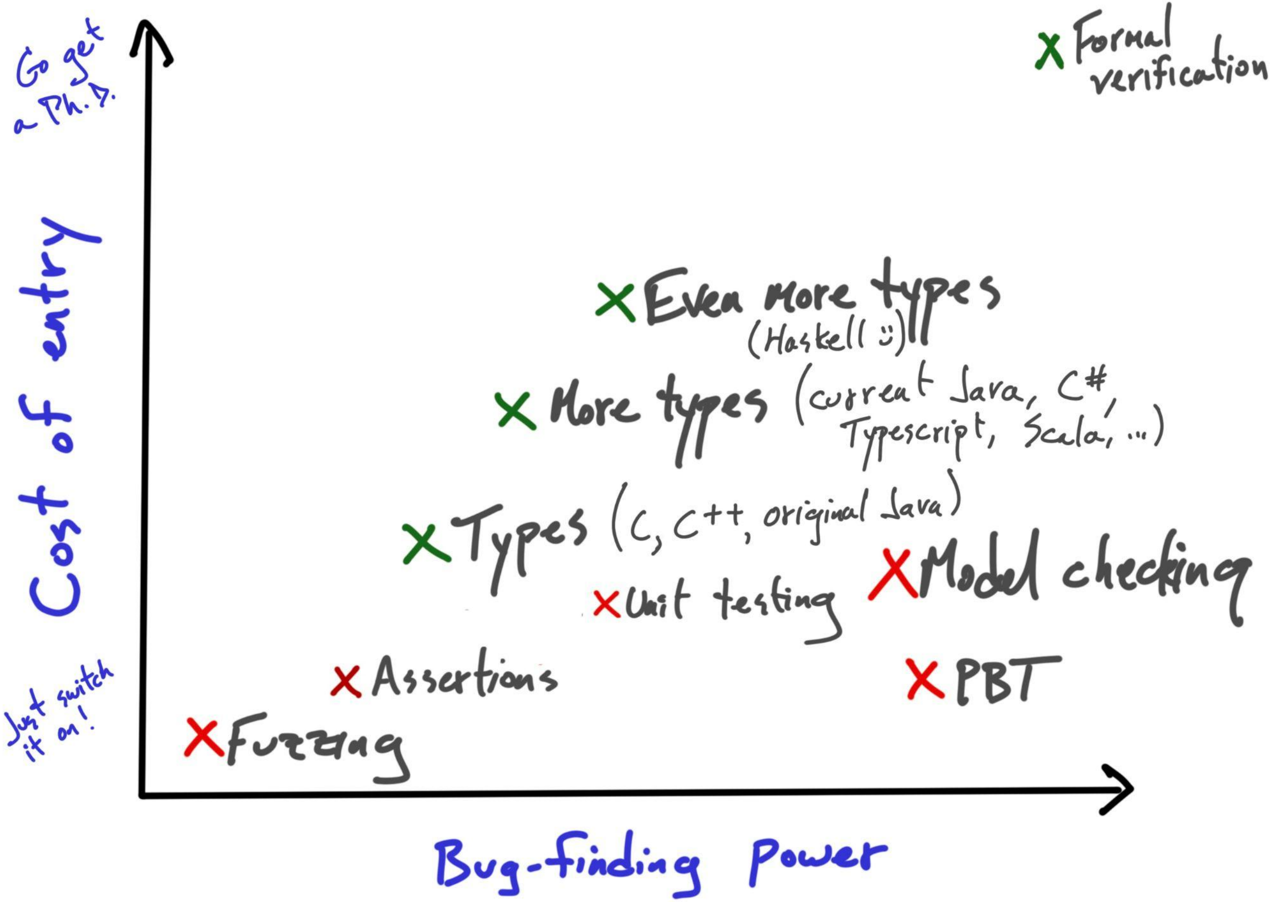
- … by enumerating small inputs exhaustively
- … or by generating larger inputs randomly
- … or by mutating past inputs that seem “interesting” (e.g., because they lead to a novel “branch coverage signature”)
- … etc.

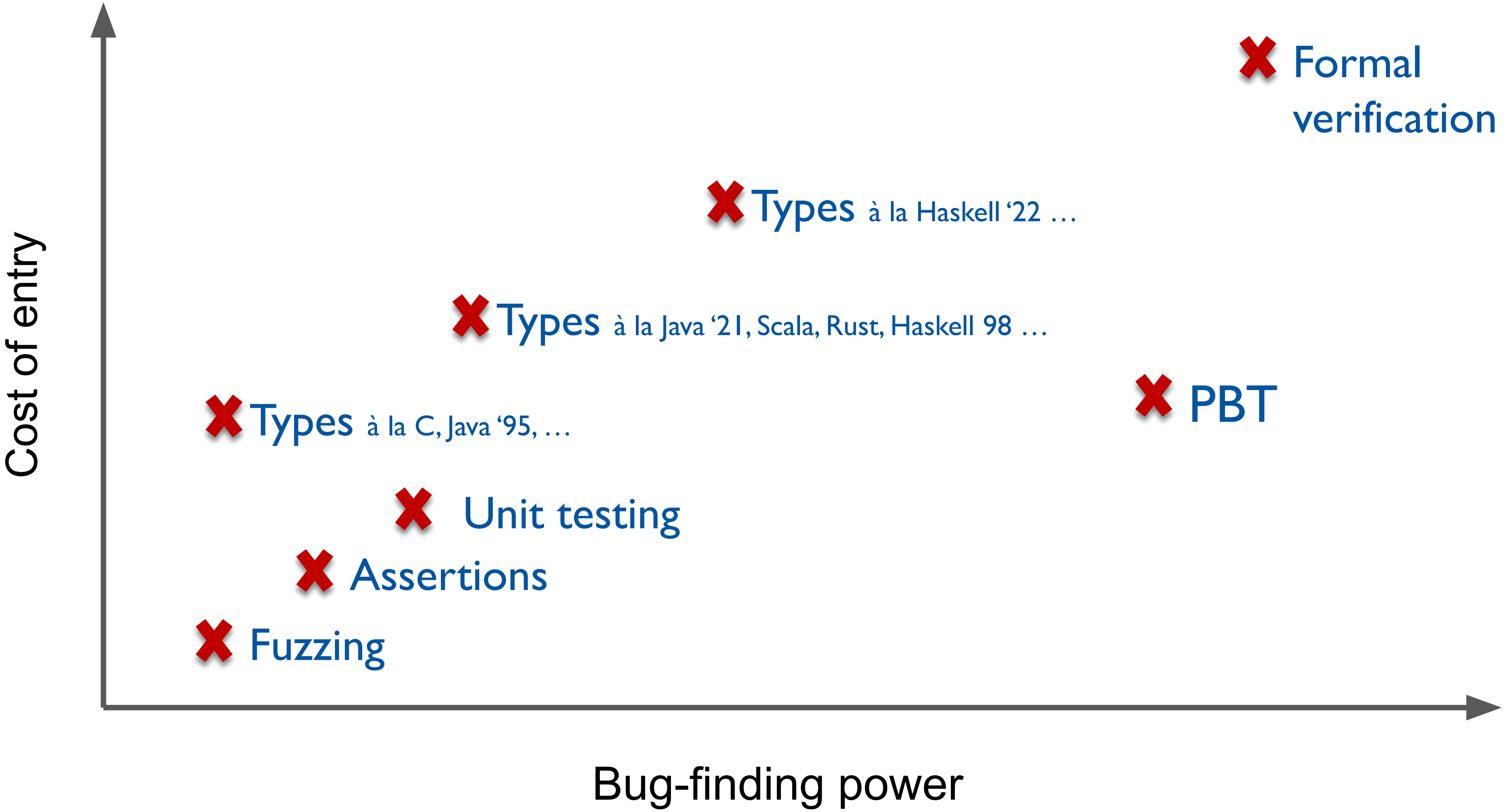
Better idea:

Split this into two slides. Show static approaches on the first, with stronger and stronger type systems (maybe both static and dynamic?) along the diagonal, with formal verification in the upper right corner.

(Model checking can go on this slide too, I guess, or on its own slide.)

Then switch to a slide on dynamic methods, with fuzzing way at the bottom but not all the way to the left, then assertions, then unit testing in the bottom middle, and then PBT almost but not quite all the way on the right…





Formal verification

Types à la Haskell ‘22 …

Types à la Java ‘21, Scala, Rust, Haskell 98 …

PBT

Types à la C, Java ‘95, …

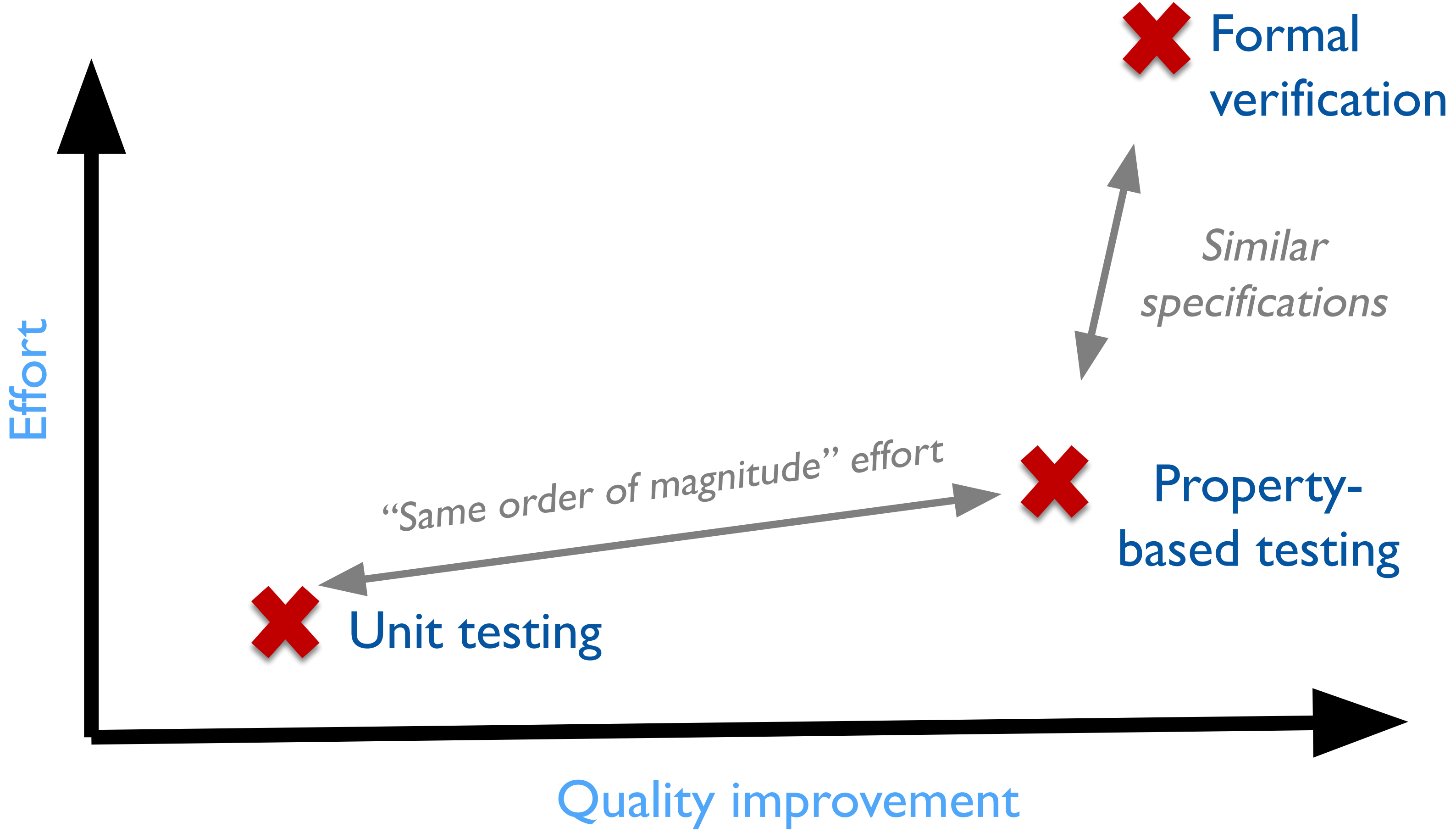
Unit testing

Assertions

Fuzzing

Cost of entry

Bug-finding power



Formal verification

Similar specifications

Property-based testing

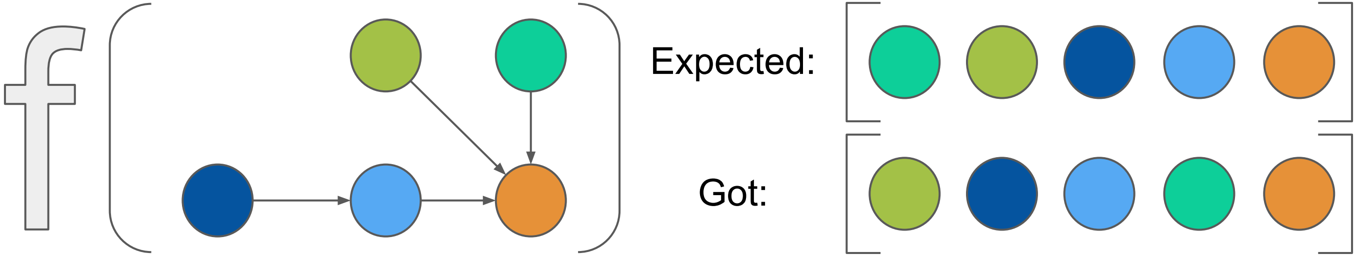
"Same order of magnitude" effort

Unit testing

Effort

Quality improvement

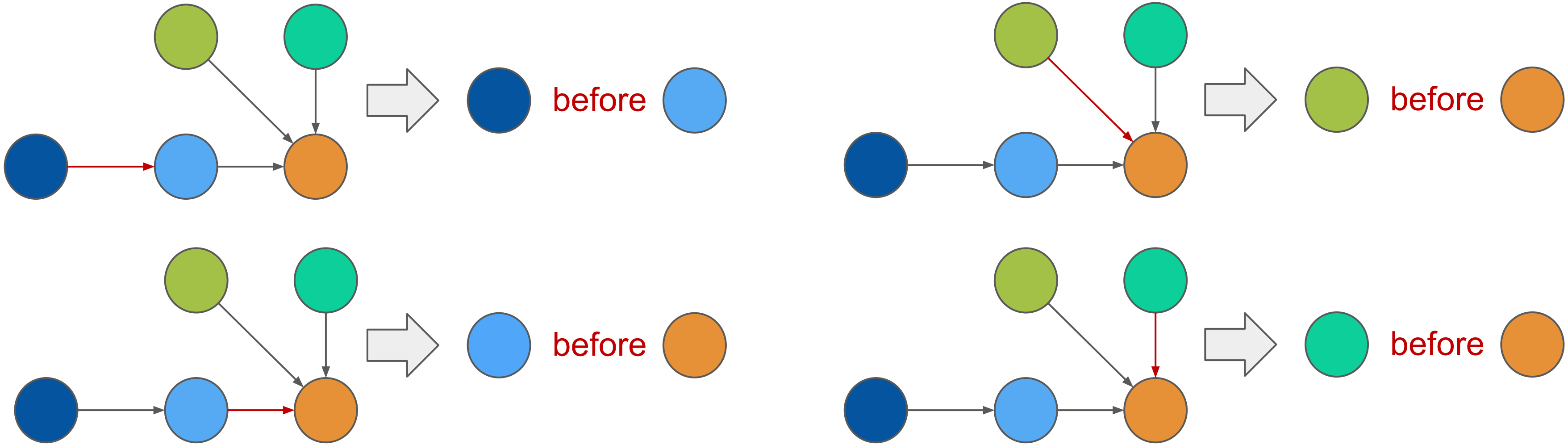
(not to scale!)



Expected:

Got:

What’s the common thread?

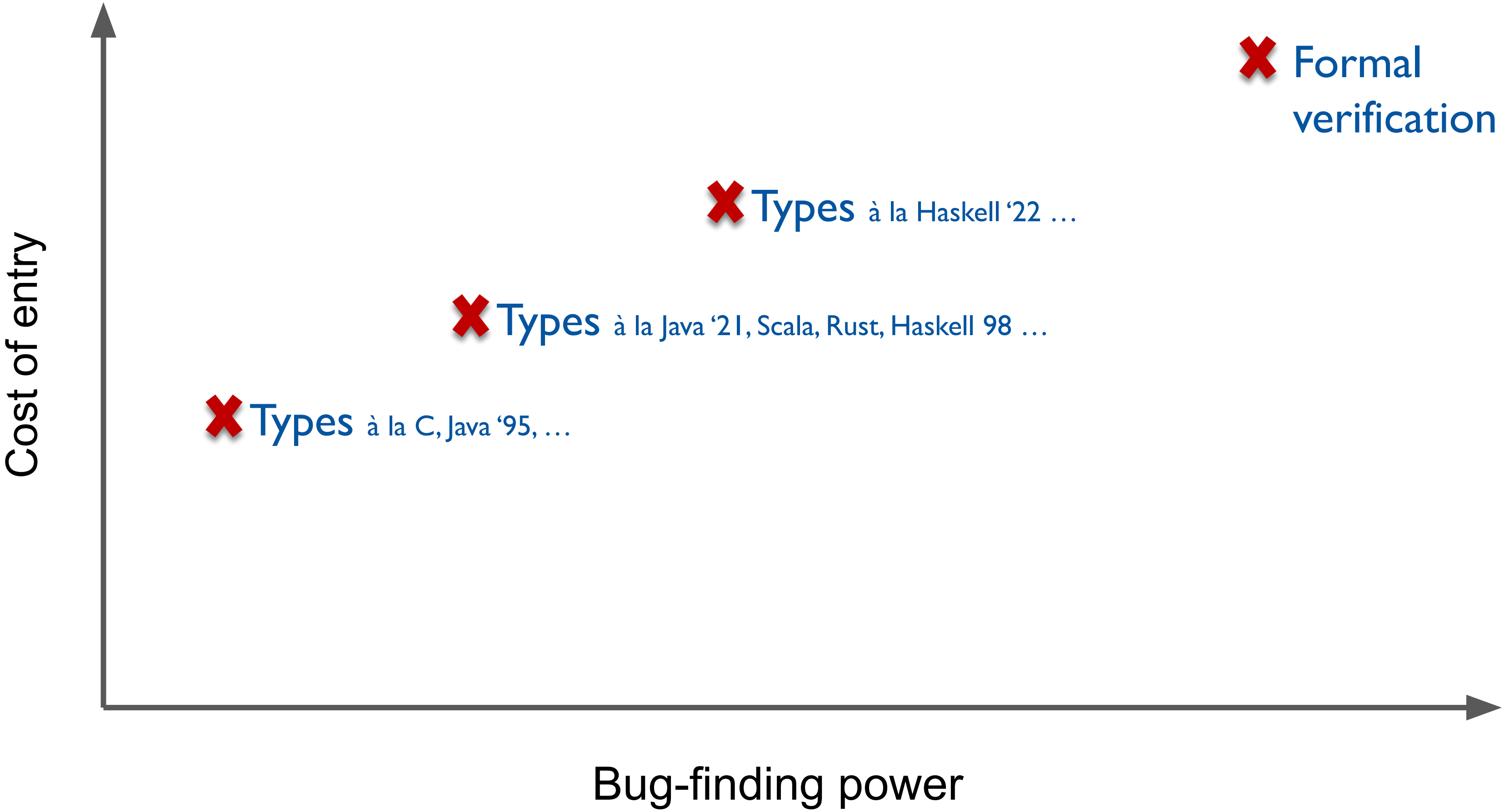


before

before

before

before



Formal verification

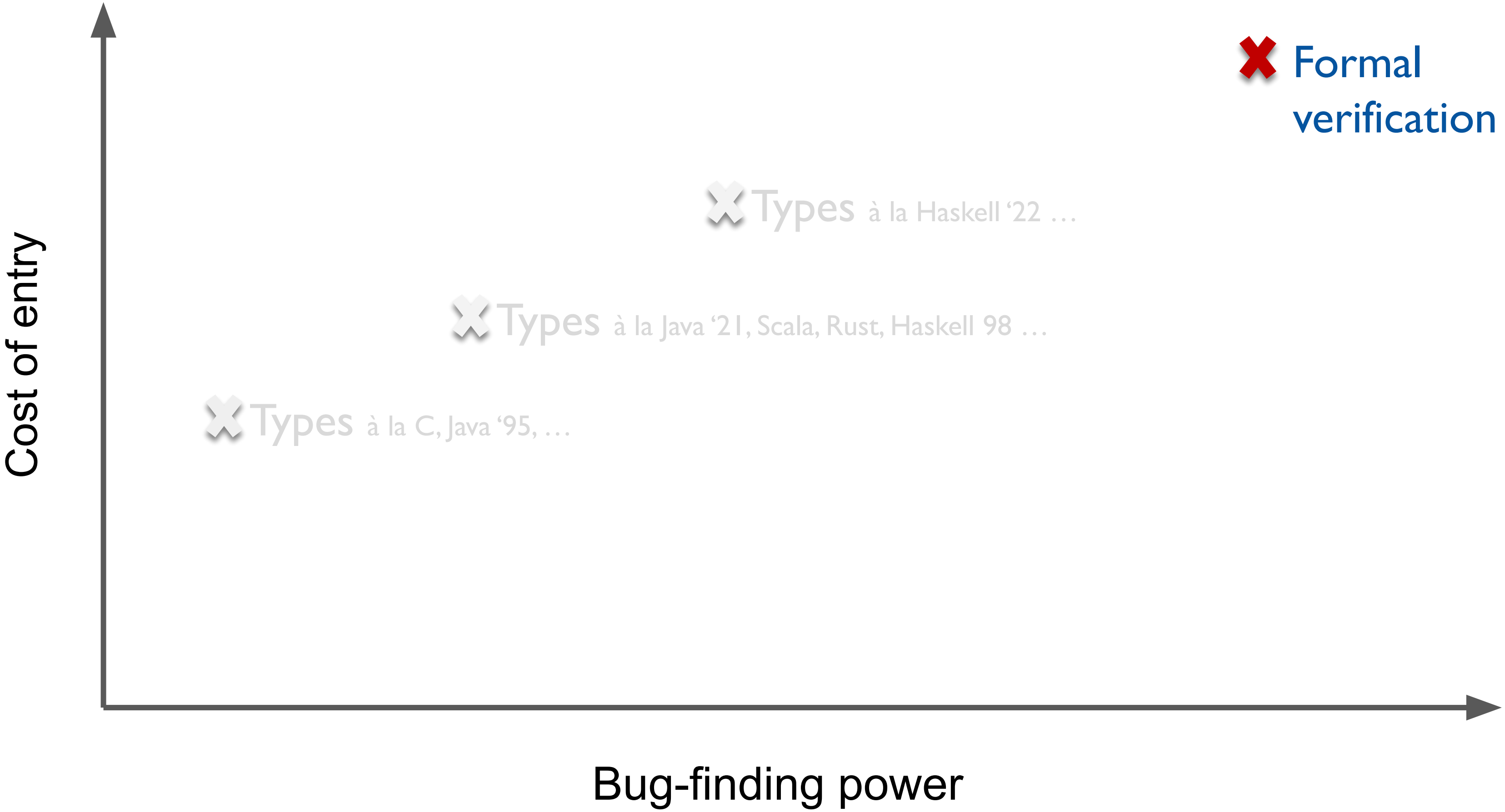
Types à la Haskell ‘22 …

Types à la Java ‘21, Scala, Rust, Haskell 98 …

Types à la C, Java ‘95, …

Cost of entry

Bug-finding power



Formal verification

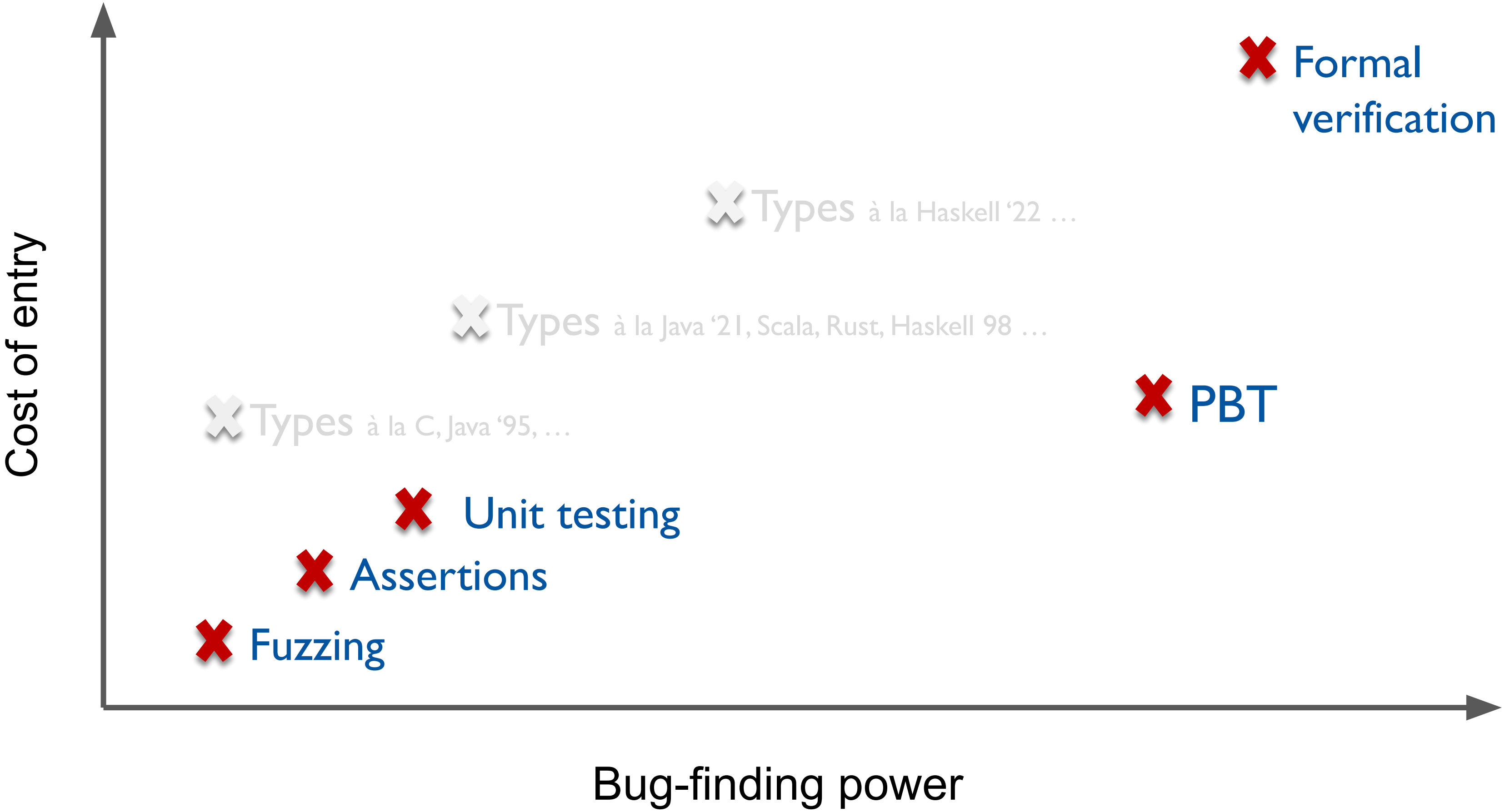
Types à la Haskell '22 …

Types à la Java '21, Scala, Rust, Haskell 98 …

Types à la C, Java '95, …

Cost of entry

Bug-finding power



Cost of entry

Formal verification

Types à la Haskell ‘22 …

Types à la Java ‘21, Scala, Rust, Haskell 98 …

PBT

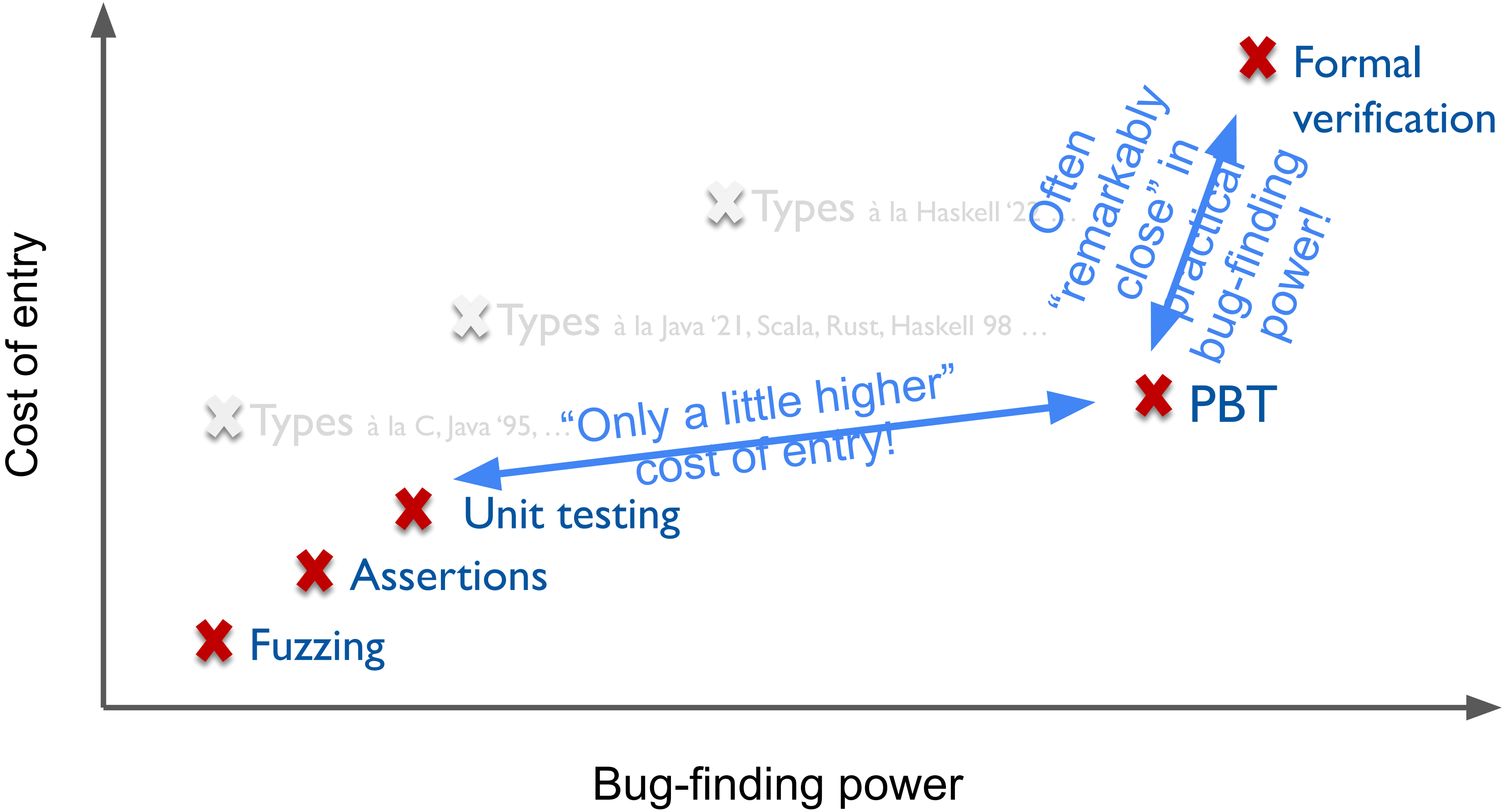
Types à la C, Java ‘95, …

Unit testing

Assertions

Fuzzing

Bug-finding power



Formal verification

Often “remarkably close” in practical bug-finding power!

Types à la Haskell ‘22 …

Types à la Java ‘21, Scala, Rust, Haskell 98 …

PBT

Types à la C, Java ‘95, …

“Only a little higher” cost of entry!

Unit testing

Assertions

Fuzzing

Cost of entry

Bug-finding power

To be written

This section can be short, just a little overview of what we are trying to achieve in the current phase of the project

- Sound basis for comparing different “generation methodologies” (enumerative, various flavors of random, coverage-based, etc.)
- Cover Haskell and Coq
- Provide boilerplate for analytics, presentation of results, …
- Make it very easy to add a new generation methodology and easy to add a new benchmark
- Each benchmark consists of one correct version and a number of “mutants” containing (hand-inserted) bugs of varying difficulties

Include pictures of collaborators :-)

Include a slide about QuickChick and PyTest Mutagen (as prior work that gives us some ideas for how to incorporate mutant suites into benchmarks)