The Science of
Deep Specification

Benjamin C. Pierce
University of Pennsylvania

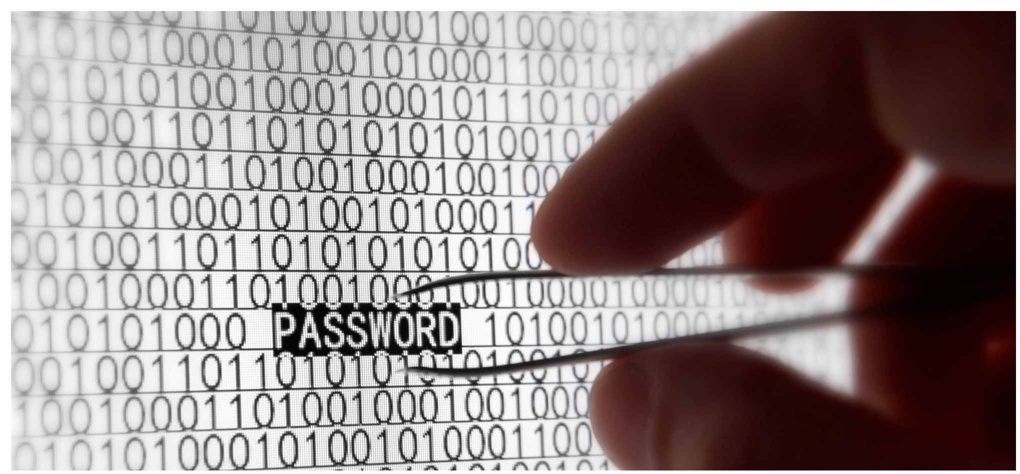
SPLASH
November, 2016

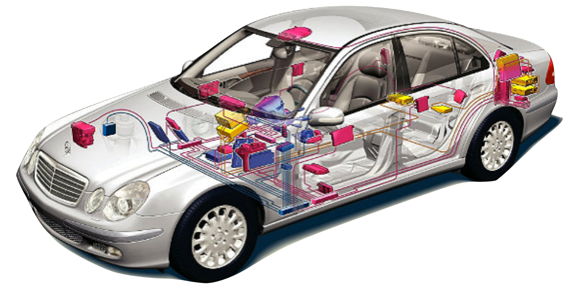
The Science of
Deep Specification

Benjamin C. Pierce
University of Pennsylvania

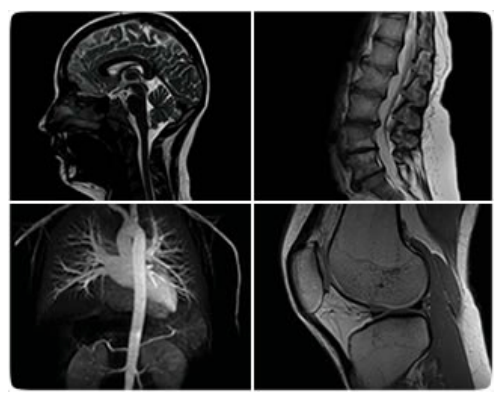
SPLASH
November, 2016

"We can't build software that works!"





Or can we??



How did that happen?

- Better programming languages
  - Basic *safety guarantees* built in
  - Powerful mechanisms for *abstraction* and *modularity*
- Better software development methodology
- Stable platforms and frameworks
- Better use of specifications

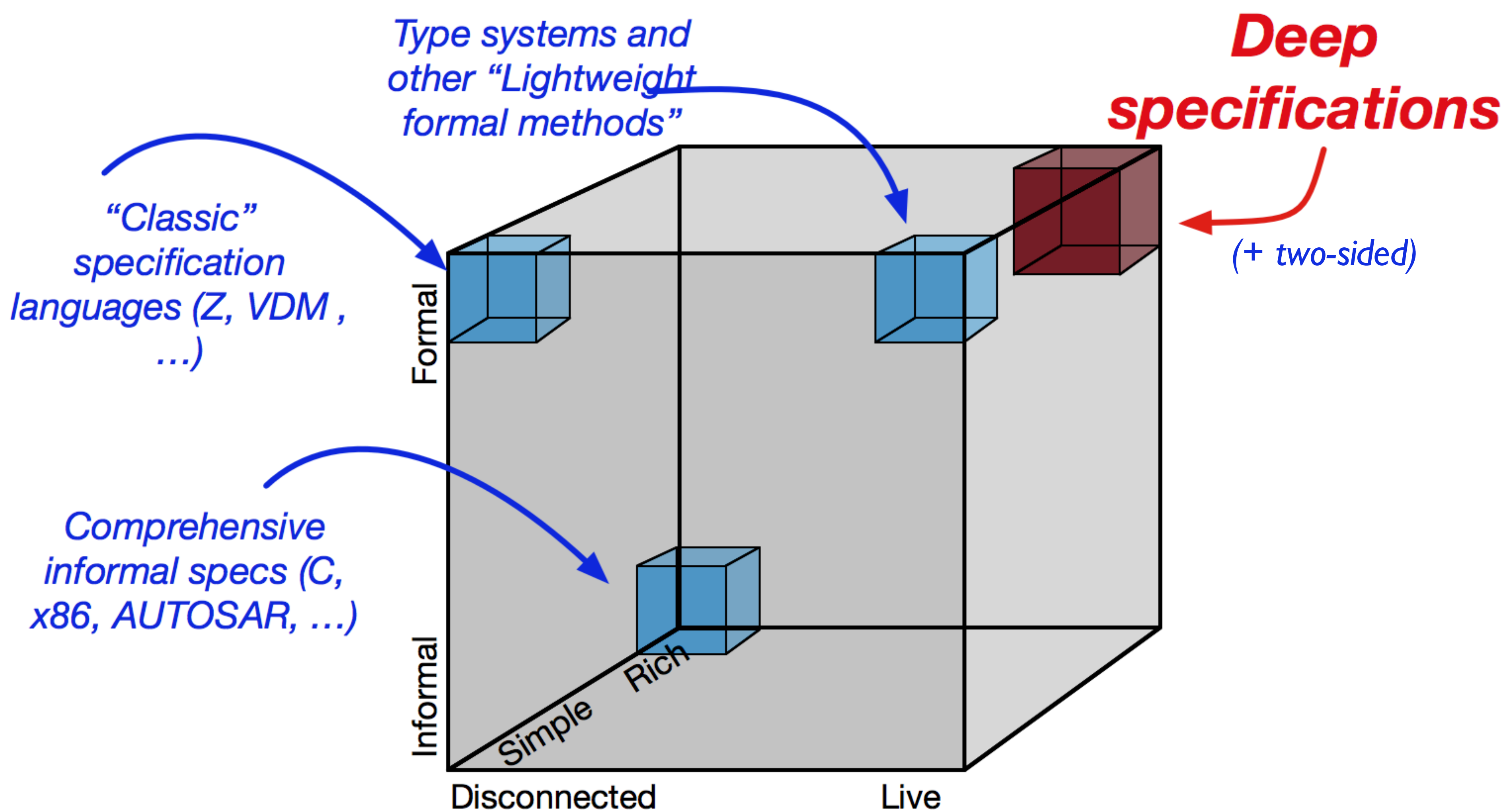
- Better programming languages
  - Basic *safety guarantees* built in
  - Powerful mechanisms for *abstraction* and *modularity*
- Better software development methodology
- Stable platforms and frameworks
- Better use of **specifications**

I.e., descriptions of what software does (as opposed to how to do it)

What are
“deep” specifications?

Deep specifications are…

Formal	mathematically precise
Rich	precisely expressing intended behavior of complex software (a spectrum!)
Live	automatically checked against actual code (not just a model)
Two-sided	exercised by both implementations and clients


Type systems and other "Lightweight formal methods"
Deep specifications
(+ two-sided)
"Classic" specification languages (Z, VDM , …)
Comprehensive informal specs (C, x86, AUTOSAR, …)
Formal
Informal
Simple
Rich
Disconnected
Live

A Short Story

about a tiny compiler

and its specification(s)…

A datatype of stack machine instructions

```
Inductive instr : Type :=
| PUSH : nat -> instr
| PLUS : instr
| MINUS : instr
| MULT : instr.

Definition my_favorite_instructions
                                    : list instr :=
  [PUSH 10; PUSH 4; MULT; PUSH 2; PLUS].
```

An example instruction sequence

(All examples in Gallina, the functional language of the Coq proof assistant)

Starting stack

Program

Final stack

```
Fixpoint execute (s : list nat) (p : list instr) : list nat :=
  match (s, p) with
  | (_,        nil)              => s
  | (_,        (PUSH n) ::p') => execute (n    ::s)  p'
  | (m::n::s', PLUS     ::p') => execute ((m+n)::s') p'
  | (m::n::s', MINUS    ::p') => execute ((m-n)::s') p'
  | (m::n::s`, MULT     ::p') => execute ((m*n)::s') p'
  | (_,        _        ::p') => execute s           p'
  end.
```

Default: Skip this instruction

Operational semantics of the stack machine

A datatype of arithmetic expressions

```
Inductive exp : Type :=
  | Num : nat -> exp
  | Plus : exp -> exp -> exp
  | Minus : exp -> exp -> exp
  | Mult : exp -> exp -> exp.

Definition my_favorite_number : exp :=
  Plus (Mult (Num 10) (Num 4)) (Num 2).
```

An example value belonging to the type exp

```
Fixpoint compile (e : exp) : list instr :=
  match e with
  | Num n => [PUSH n]
  | Plus e1 e2 => compile e1 ++ compile e2 ++ [PLUS]
  | Minus e1 e2 => compile e1 ++ compile e2 ++ [MINUS]
  | Mult e1 e2 => compile e1 ++ compile e1 ++ [MULT]
  end.
```

A compiler from arithmetic expressions to stack instructions

Specifying our compiler…

An Informal Specification

Compiling an arithmetic expression should yield stack-machine instructions that compute the corresponding numeric result:

- (Plus e1 e2) means add the results of e1 and e2
- (Minus e1 e2) means subtract the results of e1 and e2
- (Mult e1 e2) means multiply the results of e1 and e2

Formal	✘
Live	✘
Rich	✔

A (Very) Simple Formal Specification

```
Fixpoint compile (e : exp) : list instr :=
  match e with
  | Num n => [PUSH n]
  | Plus e1 e2 => compile e1 ++ compile e2 ++ [PLUS]
  | Minus e1 e2 => compile e1 ++ compile e2 ++ [MINUS]
  | Mult e1 e2 => compile e1 ++ compile e1 ++ [MULT]
  end.
```

Types!

Formal	✔
Live	✔
Rich	✘

Another Simple Formal Specification

```
Fixpoint compile (e : exp) : list instr :=
  match e with
  | Num n => [PUSH n]
  | Plus e1 e2 => compile e1 ++ compile e2 ++ [PLUS]
  | Minus e1 e2 => compile e1 ++ compile e2 ++ [MINUS]
  | Mult e1 e2 => compile e1 ++ compile e1 ++ [MULT]
  end.

Example e1 : assert (eq (compile (Num 42))
                         [PUSH 42]).

Example e2 : assert (eq (compile (Plus (Num 2) (N
                         [PUSH 2; PUSH 2; PLUS]).
```

Unit tests

Formal	✔
Live	✔
Rich	✔ / ✘

Can we do better?

```
Fixpoint compile (e : exp) : list instr :=
  match e with
  | Num n => [PUSH n]
  | Plus e1 e2 => compile e1 ++ compi
  | Minus e1 e2 => compile e1 ++ com
  | Mult e1 e2 => compile e1 ++ compi
  end.

Example e1 : assert (eq (compile (Num 4
                        [PUSH 42]).

Example e2 : assert (eq (compile (Plus (Num 2) (Num 2)))
                        [PUSH 2; PUSH 2; PLUS]).
```

We don't really care what instructions we generate: we just want executing them to give the right answer!

For Coq savants:

```
Definition assert b := (b = true).
```

```
Fixpoint eval (e : exp) : nat :=
  match e with
  | Num n => n
  | Plus e1 e2 => (eval e1) + (eval e2)
  | Minus e1 e2  => (eval e1) - (eval e2)
  | Mult e1 e2 => (eval e1) * (eval e2)
  end.

Example e3 :
  assert (eq (execute [] (compile (Plus (Num 2) (Num 2))))
             [eval (Plus (Num 2) (Num 2))]).
```

"Executing the compiled code in an empty stack…

yields a stack containing the result of `eval`*uating the original expression."*

Operational semantics of the source language

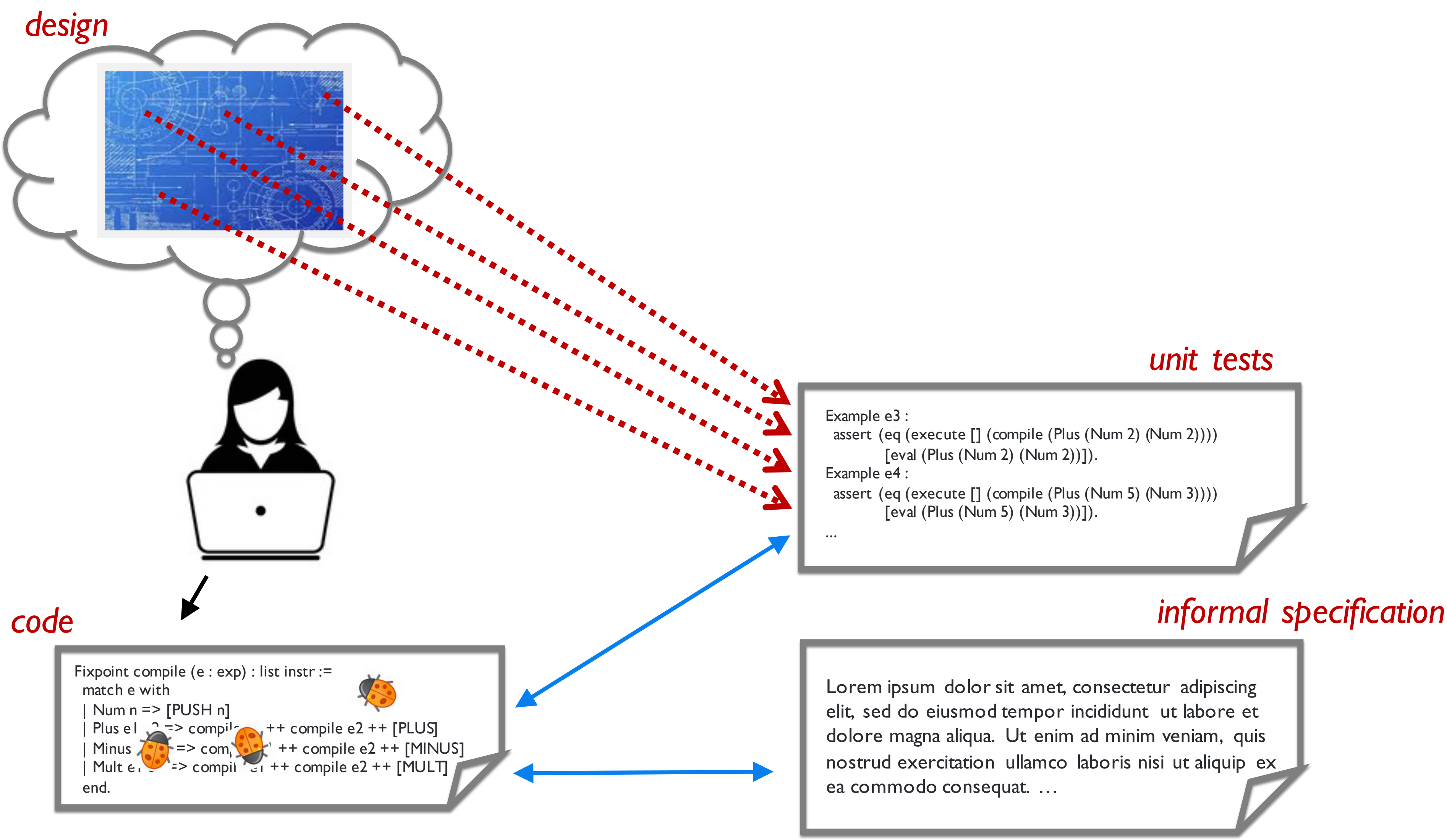
```
Example e3 :
  assert (eq (execute [] (compile (Plus (Num 2) (Num 2))))
               [eval (Plus (Num 2) (Num 2))]).

Example e4 :
  assert (eq (execute [] (compile (Plus (Num 5) (Num 3))))
               [eval (Plus (Num 5) (Num 3))]).

Example e5 :
  assert (eq (execute [] (compile (Mult (Num 0) (Num 3))))
               [eval (Mult (Num 0) (Num 3))]).

Example e6 :
  assert (eq (execute [] (compile (Mult (Num 2) (Num 2))))
               [eval (Mult (Num 2) (Num 2))]).
```

```
Example e7 :
  assert (eq (execute [] (compile (Mult (Num 3) (Num 1))))
             [eval (Mult (Num 3) (Num 1))]).
```

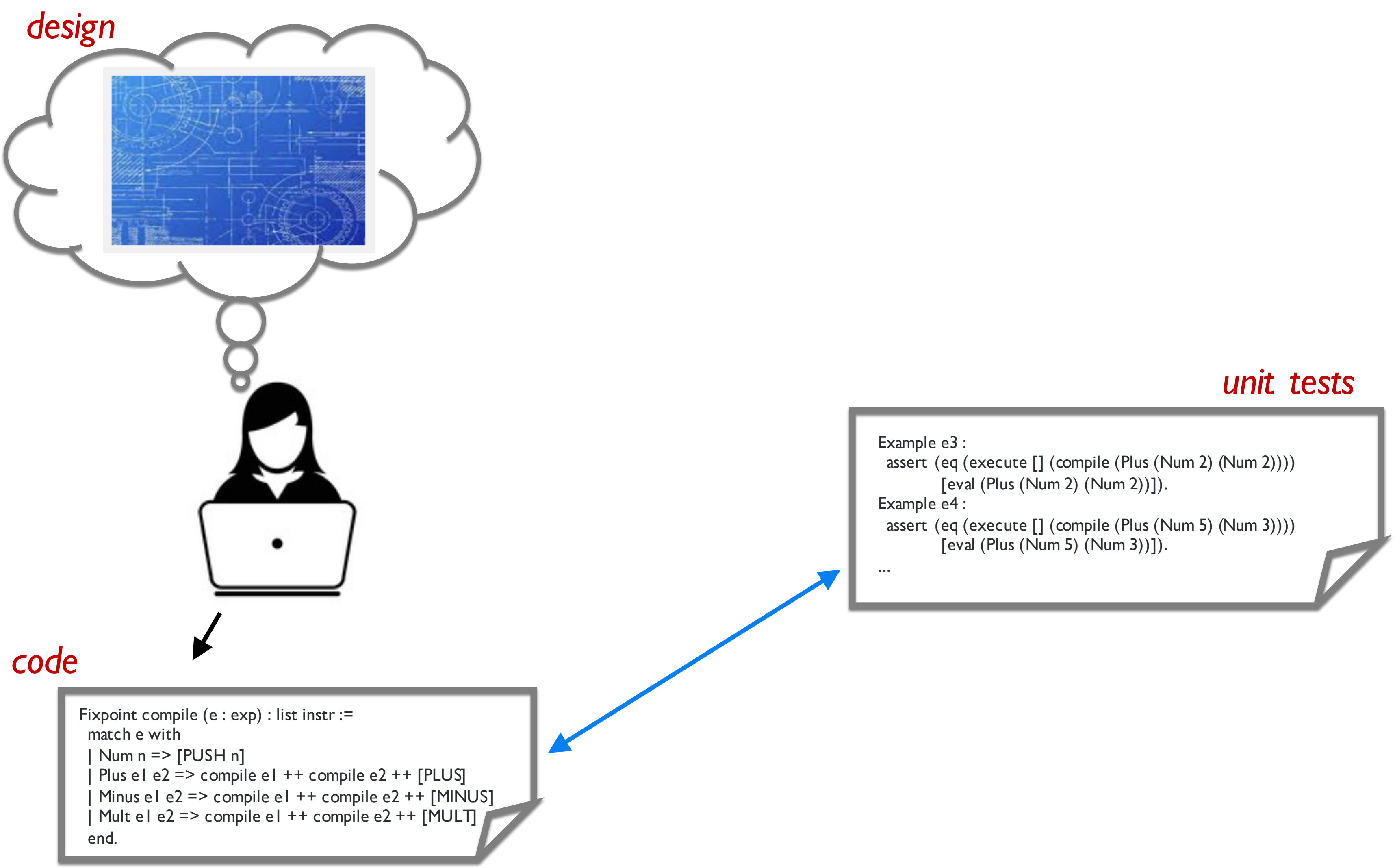
```
Fixpoint compile (e : exp) : list instr :=
  match e with
  | Num n => [PUSH n]
  | Plus e1 e2 => compile e1 ++ compile e2 ++ [PLUS]
  | Minus e1 e2 => compile e1 ++ compile e2 ++ [MINUS]
  | Mult e1 e2 => compile e1 ++ compile e1 ++ [MULT]
  end.
```

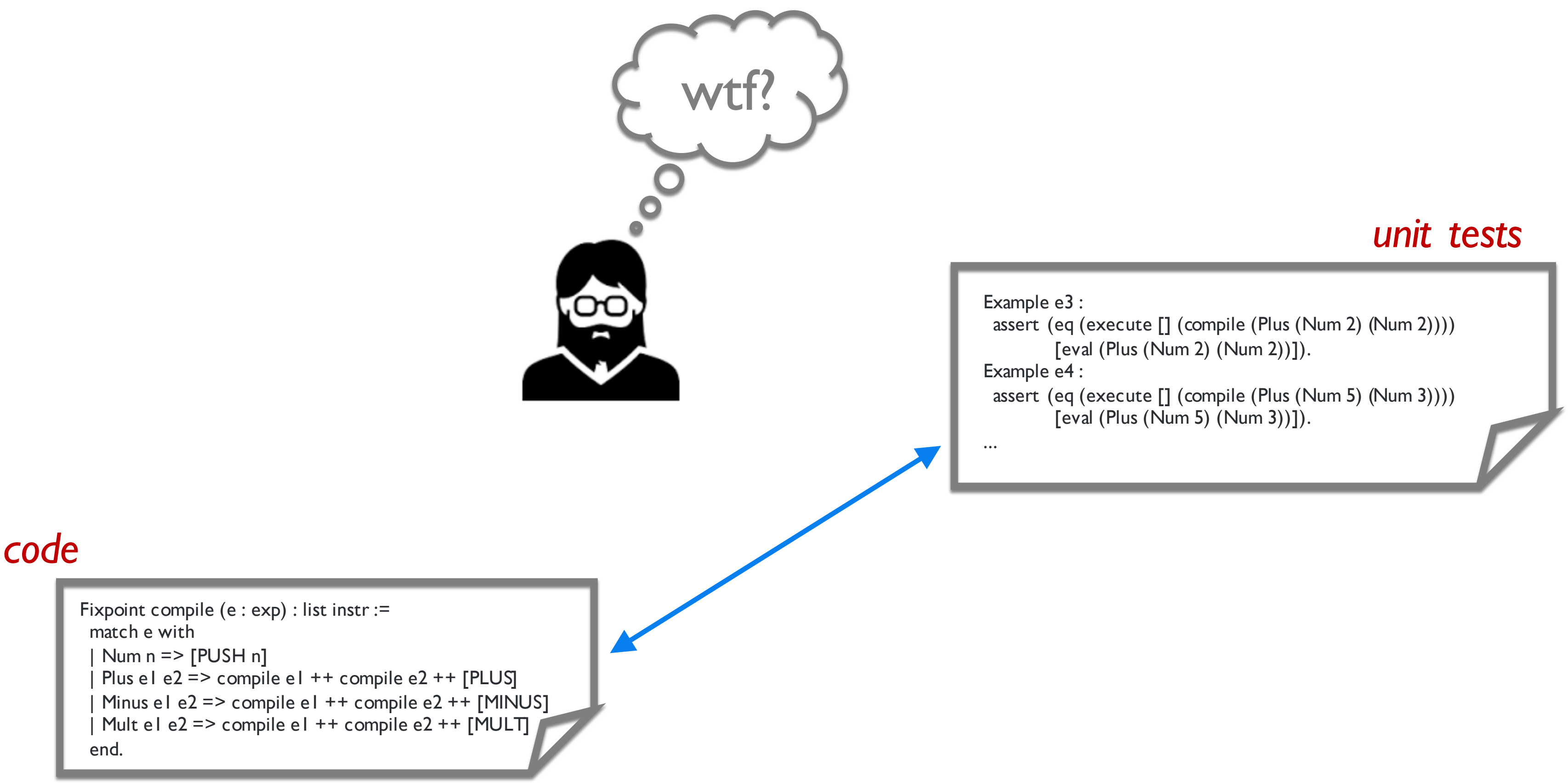
```
Example e7 :
  assert (eq (execute [] (compile (Mult (Num 3) (Num 1))))
             [eval (Mult (Num 3) (Num 1))]).
```

```
Fixpoint compile (e : exp) : list instr :=
  match e with
  | Num n => [PUSH n]
  | Plus e1 e2 => compile e1 ++ compile e2 ++ [PLUS]
  | Minus e1 e2 => compile e1 ++ compile e2 ++ [MINUS]
  | Mult e1 e2 => compile e1 ++ compile e2 ++ [MULT]
  end.
```



design
unit tests
Example e3 :
assert (eq (execute [] (compile (Plus (Num 2) (Num 2))))
[eval (Plus (Num 2) (Num 2))]).
Example e4 :
assert (eq (execute [] (compile (Plus (Num 5) (Num 3))))
[eval (Plus (Num 5) (Num 3))]).
...
informal specification
Lorem ipsum dolor sit amet, consectetur adipiscing elit, sed do eiusmod tempor incididunt ut labore et dolore magna aliqua. Ut enim ad minim veniam, quis nostrud exercitation ullamco laboris nisi ut aliquip ex ea commodo consequat. …
code
Fixpoint compile (e : exp) : list instr :=
match e with
| Num n => [PUSH n]
| Plus e1 … => compil… ++ compile e2 ++ [PLUS]
| Minus … => com… ++ compile e2 ++ [MINUS]
| Mult e… => compil… ++ compile e2 ++ [MULT]
end.


design
unit tests
Example e3 :
assert (eq (execute [] (compile (Plus (Num 2) (Num 2))))
[eval (Plus (Num 2) (Num 2))]).
Example e4 :
assert (eq (execute [] (compile (Plus (Num 5) (Num 3))))
[eval (Plus (Num 5) (Num 3))]).
...
code
Fixpoint compile (e : exp) : list instr :=
match e with
| Num n => [PUSH n]
| Plus e1 e2 => compile e1 ++ compile e2 ++ [PLUS]
| Minus e1 e2 => compile e1 ++ compile e2 ++ [MINUS]
| Mult e1 e2 => compile e1 ++ compile e2 ++ [MULT]
end.


wtf?
unit tests
Example e3 :
assert (eq (execute [] (compile (Plus (Num 2) (Num 2))))
[eval (Plus (Num 2) (Num 2))]).
Example e4 :
assert (eq (execute [] (compile (Plus (Num 5) (Num 3))))
[eval (Plus (Num 5) (Num 3))]).
...
code
Fixpoint compile (e : exp) : list instr :=
match e with
| Num n => [PUSH n]
| Plus e1 e2 => compile e1 ++ compile e2 ++ [PLUS]
| Minus e1 e2 => compile e1 ++ compile e2 ++ [MINUS]
| Mult e1 e2 => compile e1 ++ compile e2 ++ [MULT]
end.

```
Example e3 :
  assert (eq (execute [] (compile (Plus (Num 2) (Num 2))))
             [eval (Plus (Num 2) (Num 2))]).

Example e4 :
  assert (eq (execute [] (compile (Plus (Num 5) (Num 3))))
             [eval (Plus (Num 5) (Num 3))]).

Example e5 :
  assert (eq (execute [] (compile (Mult (Num 0) (Num 3))))
             [eval (Mult (Num 0) (Num 3))]).

Example e6 :
  assert (eq (execute [] (compile (Mult (Num 2) (Num 2))))
             [eval (Mult (Num 2) (Num 2))]).
```

```
Example e3 :
  assert (eq (execute [] (compile (Plus (Num 2) (Num 2))))
             [eval (Plus (Num 2) (Num 2))]).

Example e4 :
  assert (eq (execute [] (compile (Plus (Num 5) (Num 3))))
             [eval (Plus (Num 5) (Num 3))]).

Example e5 :
  assert (eq (execute [] (compile (Mult (Num 0) (Num 3))))
             [eval (Mult (Num 0) (Num 3))]).

Example e6 :
  assert (eq (execute [] (compile (Mult (Num 2) (Num 2))))
             [eval (Mult (Num 2) (Num 2))]).
```

```
Definition compiles_correctly (e : exp) :=
  eq (execute [] (compile e)) [eval e].


Example e3 :
  assert (eq (execute [] (compile (Plus (Num 2) (Num 2))))
             [eval (Plus (Num 2) (Num 2))]).

Example e4 :
  assert (eq (execute [] (compile (Plus (Num 5) (Num 3))))
             [eval (Plus (Num 5) (Num 3))]).

Example e5 :
  assert (eq (execute [] (compile (Mult (Num 0) (Num 3))))
             [eval (Mult (Num 0) (Num 3))]).

Example e6 :
  assert (eq (execute [] (compile (Mult (Num 2) (Num 2))))
             [eval (Mult (Num 2) (Num 2))]).
```

```
Definition compiles_correctly (e : exp) :=
  eq (execute [] (compile e)) [eval e].

Example e3 :
  assert (compiles_correctly (Plus (Num 2) (Num 2))).

Example e4 :
  assert (compiles_correctly (Plus (Num 5) (Num 3))).

Example e5 :
  assert (compiles_correctly (Mult (Num 0) (Num 3))).

Example e6 :
  assert (compiles_correctly (Mult (Num 2) (Num 2))).
```

Enumerative

Specification-Based Testing

Random

Concolic

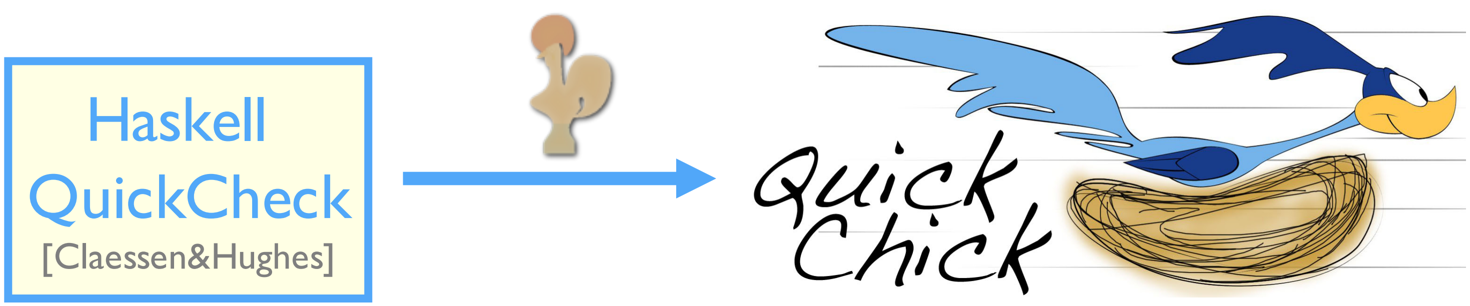
etc.

etc.

Specification-Based *Random* Testing

Idea:

- Generate lots of random values of type `exp`
- See if `compiles_correctly` returns `true` for each of them



```
QuickChick compiles_correctly.


Counterexample found after 4 tests:

Plus ( Plus ( Minus ( Num ( 3 ) ) ( Num ( 0 ) )
) ( Plus ( Num ( 3 ) ) ( Num ( 2 ) ) ) ) ( Plus
( Minus ( Num ( 0 ) ) ( Num ( 0 ) ) ) ( Mult (
Num ( 0 ) ) ( Num ( 3 ) ) ) ) )
```

```
QuickChick compiles_correctly.


Counterexample found after 4 tests:

Plus ( Plus ( Minus ( Num ( 3 ) ) ( Num ( 0 ) ) )
) ( Plus ( Num ( 3 ) ) ( Num ( 2 ) ) ) ) ( Plus
( Minus ( Num ( 0 ) ) ( Num ( 0 ) ) ) ( Mult (
Num ( 0 ) ) ( Num ( 3 ) ) ) ) )
```

Idea:

- Generate lots of random values of type `exp`
- For each, see if `compiles_correctly` returns `true`
- If a failing example is found, perform a greedy search for a minimal failing example (“shrinking”)

```
QuickChick compiles_correctly.
```

Counterexample found after 4 tests and 8 shrinks:

Minus (Num

```
Fixpoint compile (e : exp) : list instr :=
  match e with
  | Num n => [PUSH n]
  | Plus e1 e2 => compile e1 ++ compile e2 ++ [PLUS]
  | Minus e1 e2 => compile e1 ++ compile e2 ++ [MINUS]
  | Mult e1 e2 => compile e1 ++ compile e2 ++ [MULT]
  end.
```

with shrinking on…

QuickChick

Counterexam
shrinks:

Minus (Num

```
Fixpoint execute (s : list nat) (p : list instr) : list nat :=
  match (s, p) with
  | (_,         nil)          => s
  | (_,         (PUSH n) ::p') => execute (n     ::s)  p'
  | (m::n::s',  PLUS     ::p') => execute ((m+n)::s')  p'
  | (m::n::s',  MINUS    ::p') => execute ((m-n)::s')  p'
  | (m::n::s',  MULT     ::p') => execute ((m*n)::s')  p'
  | (_,         _        ::p') => execute s            p'
  end.
```

```
Fixpoint compile (e : exp) : list instr :=
  match e with
  | Num n => [PUSH n]
  | Plus e1 e2 => compile e1 ++ compile e2 ++ [PLUS]
  | Minus e1 e2 => compile e1 ++ compile e2 ++ [MINUS]
  | Mult e1 e2 => compile e1 ++ compile e2 ++ [MULT]
  end.
```

`compile` leaves the arguments of `Minus` in the wrong order on the stack!

Beyond Testing…

What else can we do with a specification?

- Synthesize programs that satisfy it
- Build run-time monitors that check for violations
- Prove that an implementation satisfies it

```
Theorem compile_correct : forall e,
  assert (compiles_correctly e).
```

```
Lemma execute_app : forall p1 p2 stack,
    execute stack (p1 ++ p2)
  = execute (execute stack p1) p2.

Lemma execute_eval_comm : forall e stack,
  execute stack (compile e) = eval e :: stack.

Theorem compile_correct : forall e,
  assert (compiles_correctly e).
```

```
Lemma execute_app : forall p1 p2 stack,
    execute stack (p1 ++ p2)
  = execute (execute stack p1) p2.

Lemma execute_eval_comm : forall e stack,
  execute stack (compile e) = eval e :: stack.

Theorem compile_correct : forall e,
  assert (compiles_correctly e).
```

```
Lemma execute_app : forall p1 p2 stack,
    execute stack (p1 ++ p2)
  = execute (execute stack p1) p2.
Proof.
  induction p1.
    - reflexivity.
    - destruct a.
      + intros. simpl. rewrite IHp1.
        reflexivity.
      + intros. simpl.
        destruct stack as [|x [|y stack']].
        * rewrite IHp1. reflexivity.
        * rewrite IHp1. reflexivity.
        * rewrite IHp1. reflexivity.
      + intros. simpl.
        destruct stack as [|x [|y stack']].
        * rewrite IHp1. reflexivity.
        * rewrite IHp1. reflexivity.
        * rewrite IHp1. reflexivity.
      + intros. simpl.
        destruct stack as [|x [|y stack']].
        * rewrite IHp1. reflexivity.
        * rewrite IHp1. reflexivity.
        * rewrite IHp1. reflexivity.
Qed.
```

```
Lemma execute_app : forall p1 p2 stack,
    execute stack (p1 ++ p2)
  = execute (execute stack p1) p2.
Proof.
  induction p1.
    - reflexivity.
    - destruct a.
      + intros. simpl. rewrite IHp1.
        reflexivity.
      + intros. simpl.
        destruct stack as [|x [|y stack']].
        * rewrite IHp1. reflexivity.
        * rewrite IHp1. reflexivity.
        * rewrite IHp1. reflexivity.
      + intros. simpl.
        destruct stack as [|x [|y stack']].
        * rewrite IHp1. reflexivity.
        * rewrite IHp1. reflexivity.
        * rewrite IHp1. reflexivity.
      + intros. simpl.
        destruct stack as [|x [|y stack']].
        * rewrite IHp1. reflexivity.
        * rewrite IHp1. reflexivity.
        * rewrite IHp1. reflexivity.
Qed.
```

No automation

```
Lemma execute_app : forall p1 p2 stack,
    execute stack (p1 ++ p2)
  = execute (execute stack p1) p2.
Proof.
  induction p1.
    - reflexivity.
    - destruct a; simpl; intros;
      destruct stack as [|x [|y stack']];
      try rewrite IHp1; reflexivity.
Qed.
```

Simple automation

```
Lemma execute_app : forall p1 p2 stack,
    execute stack (p1 ++ p2)
  = execute (execute stack p1) p2.
Proof.
  induction p1.
    - reflexivity.
    - destruct a.
      + intros. simpl. rewrite IHp1.
        reflexivity.
      + intros. simpl.
        destruct stack as [|x [|y stack']].
        * rewrite IHp1. reflexivity.
        * rewrite IHp1. reflexivity.
        * rewrite IHp1. reflexivity.
      + intros. simpl.
        destruct stack as [|x [|y stack']].
        * rewrite IHp1. reflexivity.
        * rewrite IHp1. reflexivity.
        * rewrite IHp1. reflexivity.
      + intros. simpl.
        destruct stack as [|x [|y stack']].
        * rewrite IHp1. reflexivity.
        * rewrite IHp1. reflexivity.
        * rewrite IHp1. reflexivity.
Qed.
```

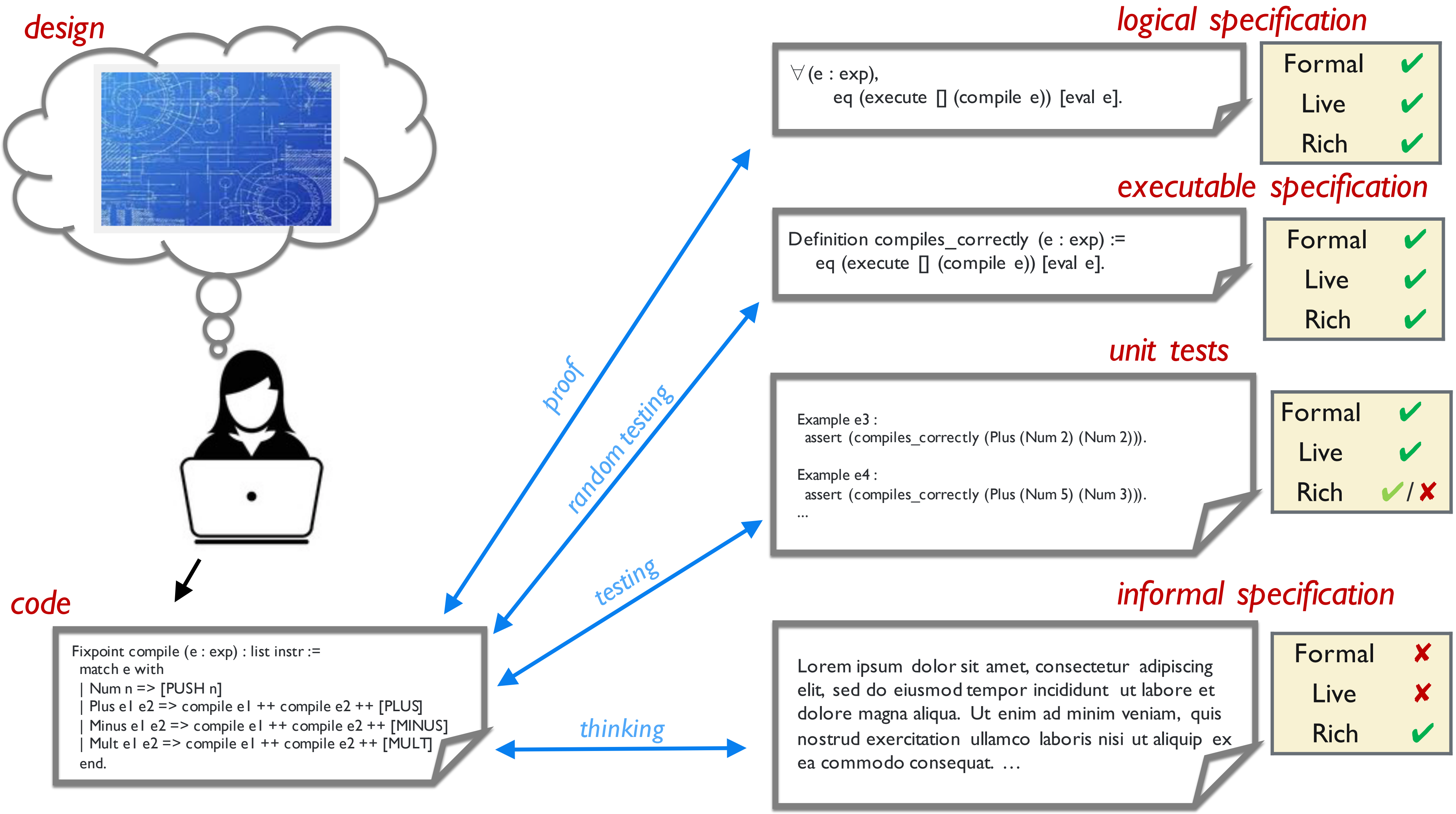
No automation

```
Lemma execute_app : forall p1 p2 stack,
    execute stack (p1 ++ p2)
  = execute (execute stack p1) p2.
Proof.
  induction p1.
    - reflexivity.
    - destruct a; simpl; intros;
      destruct stack as [|x [|y stack']];
      try rewrite IHp1; reflexivity.
Qed.
```

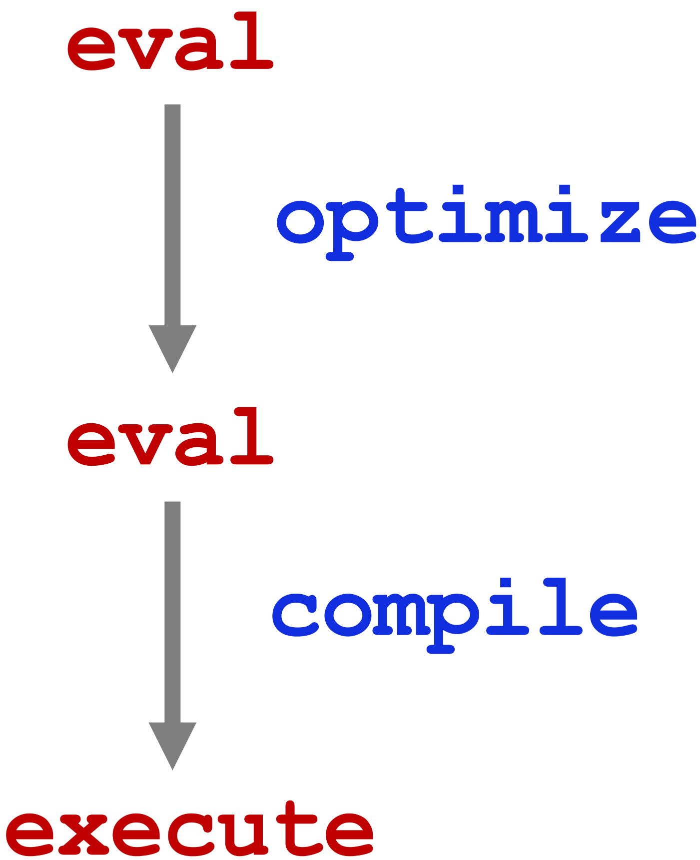
Simple automation

```
Lemma execute_app : forall p1 p2 stack,
    execute stack (p1 ++ p2)
  = execute (execute stack p1) p2.
Proof.
  induction p1;
    try (destruct a);
    try (destruct stack
              as [|x [|y stack']]);
    crush.
Qed.
```

Chlipala automation


design
code
Fixpoint compile (e : exp) : list instr :=
match e with
| Num n => [PUSH n]
| Plus e1 e2 => compile e1 ++ compile e2 ++ [PLUS]
| Minus e1 e2 => compile e1 ++ compile e2 ++ [MINUS]
| Mult e1 e2 => compile e1 ++ compile e2 ++ [MULT]
end.
proof
random testing
testing
thinking
logical specification
∀ (e : exp),
eq (execute [] (compile e)) [eval e].
Formal ✔
Live ✔
Rich ✔
executable specification
Definition compiles_correctly (e : exp) :=
eq (execute [] (compile e)) [eval e].
Formal ✔
Live ✔
Rich ✔
unit tests
Example e3 :
assert (compiles_correctly (Plus (Num 2) (Num 2))).
Example e4 :
assert (compiles_correctly (Plus (Num 5) (Num 3))).
...
Formal ✔
Live ✔
Rich ✔ / ✘
informal specification
Lorem ipsum dolor sit amet, consectetur adipiscing elit, sed do eiusmod tempor incididunt ut labore et dolore magna aliqua. Ut enim ad minim veniam, quis nostrud exercitation ullamco laboris nisi ut aliquip ex ea commodo consequat. ...
Formal ✘
Live ✘
Rich ✔

What about “two-sided”?

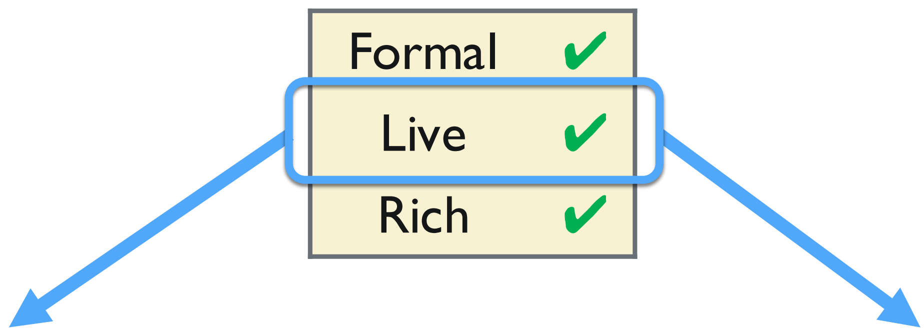



```
Theorem optimize_correct :
  forall e,
      eval (optimize e)
    = eval e.
```

nice story

does it scale??

Some recent “deep specifications”



rigorously *tested* specifications of existing real-world artifacts

formally *verified* specifications of new artifacts “live” = “exhaustively tested”…

REMS

http://rems.io

"Rigorous Engineering of Mainstream Systems"

- Full-scale formal specifications of a range of critical interfaces
  - 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 (Cerberus – also see Krebbers, K semantics, …)

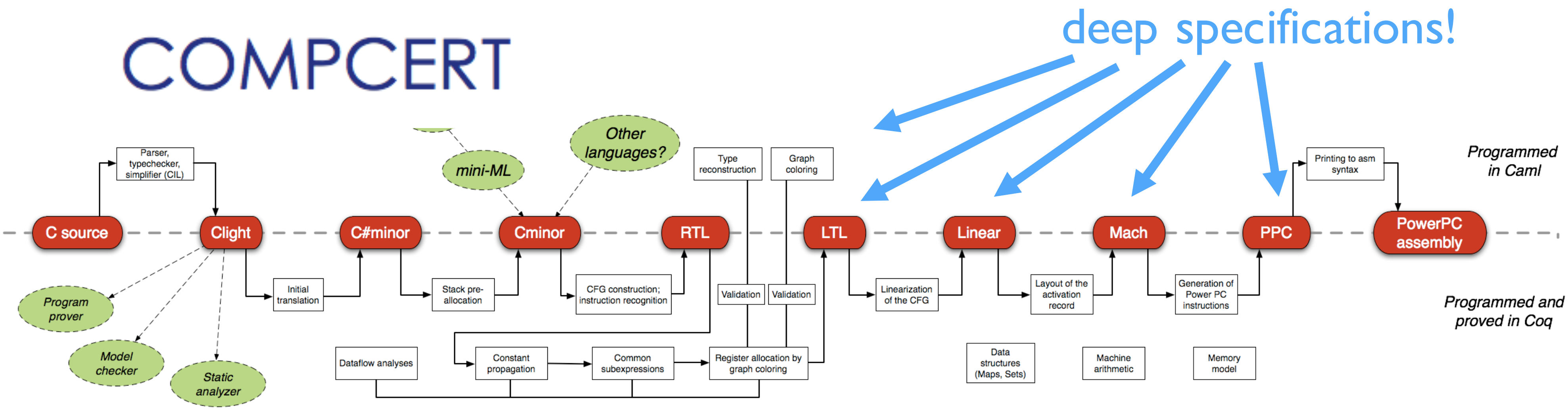
QuviQ

AUTOSAR

- Engineers at 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

“live” = “verified”…

COMPCERT



- Accepts most of ISO C 99
- Produces machine code for PowerPC, ARM, and IA32 (x86 32-bit) architectures
- 90% of the performance of GCC (v4, opt. level 1)
- Fully verified (at the source-code level)

COMPCERT

- 50,000 lines of Coq
  - 8k code (~= 40k of C or Java)
  - 42k specification and proof

Verification really works!

Regehr's Csmith project used random testing to assess all popular C compilers, and reported:

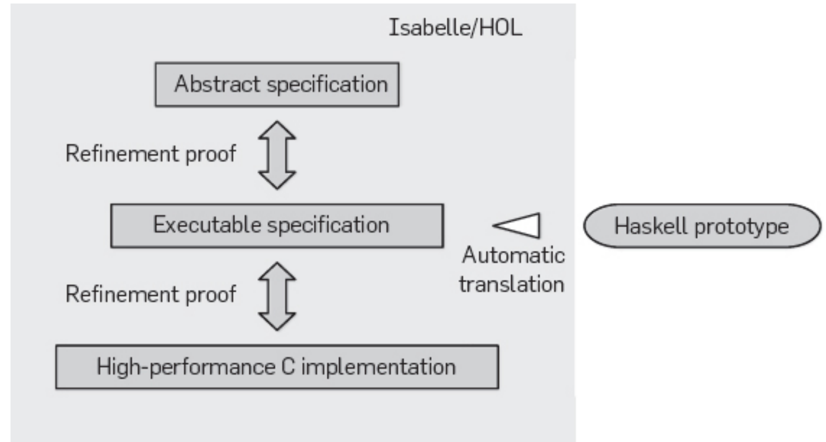
``The striking thing about our CompCert results is that the middle-end bugs we found in all other compilers are absent. As of early 2011, the under-development version of CompCert is the only compiler we have tested for which Csmith cannot find wrong-code errors. This is not for lack of trying: we have devoted about six CPU-years to the task. *The apparent unbreakability of CompCert supports a strong argument that developing compiler optimizations within a proof framework, where safety checks are explicit and machine-checked, has tangible benefits for compiler users."*

John Regehr
Univ. of Utah

CAKEML

A Verified Implementation of ML

- Verified compiler from a substantial subset of Standard ML to x86-64 machine code (ARM, MIPS, and RISC-V are anticipated)
- Bootstrapped!
  - The compiler itself is implemented in CakeML, so its executable is guaranteed to implement the compilation algorithm described by its source code
- Correctness proofs use validated ISA models for machine code
- Goal is to implement a proof assistant in CakeML and use it to verify CakeML's own correctness proof
  - TinyTCB!!



- Real-world operating-system kernel with an end-to-end proof of implementation correctness and security enforcement
- Verified down to machine code

NICTA

Ironclad

- Ironclad Apps: verifying the security of a complete software stack

User can securely transmit her data to a remote machine with the guarantee that every instruction executed on that machine adheres to a formal abstract specification of the app's behavior.

- IronFleet: verifying safety and liveness of distributed systems

Dafny

Microsoft Research

Project Everest

Verified Secure Implementations of the HTTPS Ecosystem

- Ongoing project aiming to build and deploy a verified HTTPS stack
- drop-in replacement for the HTTPS library in mainstream web browsers, servers, etc.

Microsoft Research

CertiKOS

Certified Kit Operating System

YALE

Certified OS Kernels

Clean-slate design with end-to-end guarantees on extensibility, security, and resilience. Without Zero-Day Kernel Vulnerabilities.

Layered Approach

Divides a complex system into multiple certifeid abstraction layers, which are deep specifications of their underlying implementations.

Languages and Tools

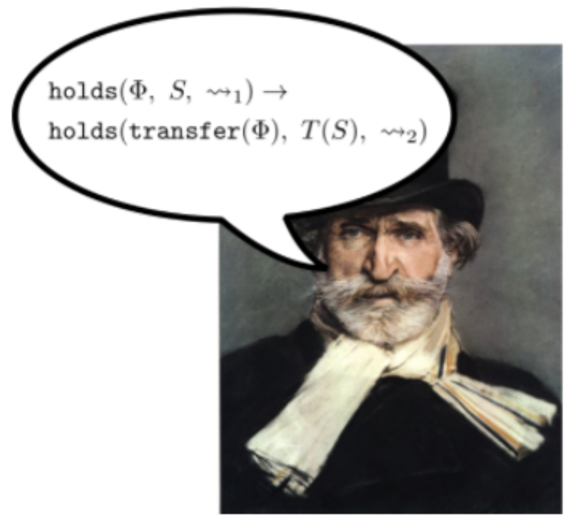
New formal methods, languages, compilers and other tools for developing, checking, and automating specs and proofs.

Kami

- Coq framework for implementing, specifying, verifying, and compiling Bluespec-style hardware components.
- E.g., a RISC-V implementation (w 4-stage pipeline), fully verified down to RTL

Verdi

- Framework for implementing and formally verifying distributed systems
  - E.g. verified implementation of the Raft distributed consensus protocol
- *Verified system transformers* encapsulate common fault tolerance techniques
  - Developers verify an application in an idealized fault model, then apply a VST to obtain an application with analogous properties in a more adversarial environment



- The Vellvm project has built a formal specification of the intermediate representation used by the popular LLVM compiler.
- This spec has been used to build *verified compiler transformations* that can be plugged into LLVM. Their performance is competitive with unverified transformations.
- The specification has been validated against the LLVM test suite.

Penn
UNIVERSITY *of* PENNSYLVANIA

CertiCoq

- Certified compiler from Coq to C
  - and then, via CompCert, to assembly
- (in progress)

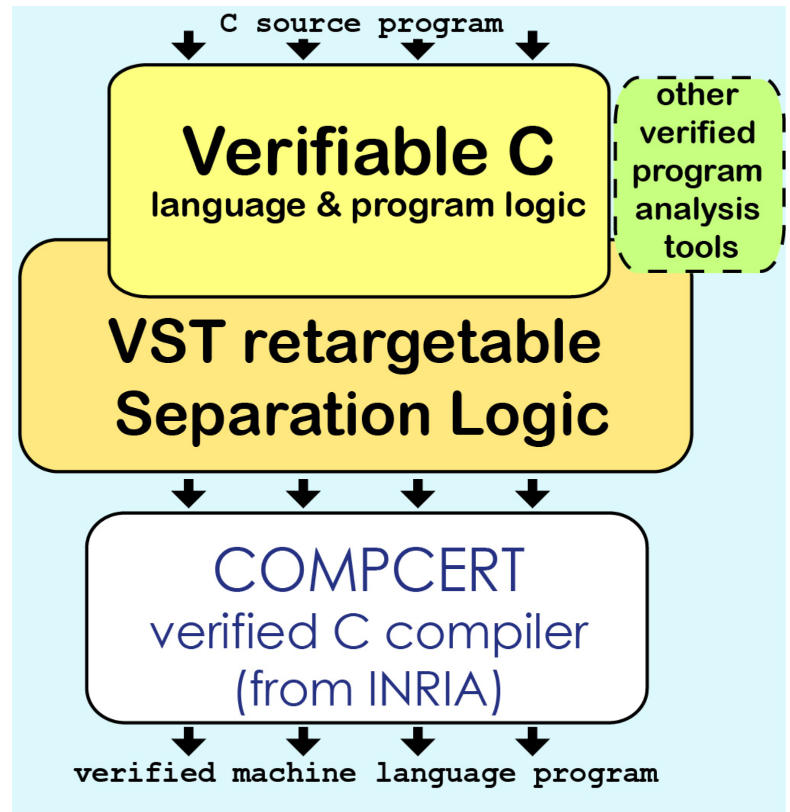
Core Spec

Haskell CoreSpec is an ongoing effort to formally specify the core intermediate language of the GHC compiler and verify key compiler passes

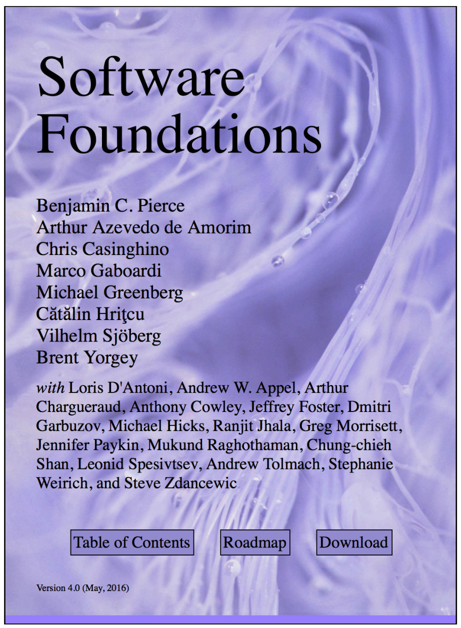
Penn
University of Pennsylvania

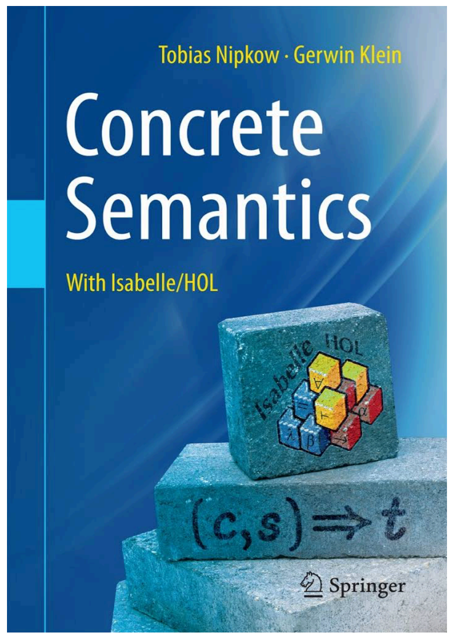
Verified
Software
Toolchain

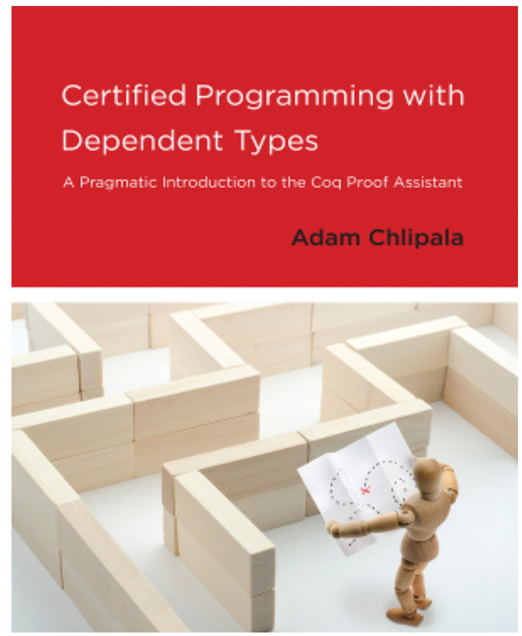
- C verification framework based on higher-order separation logic in Coq
- Verified implementations of OpenSSL-HMAC and SHA-256
- working on additional cryto primitives (HMAC-based Deterministic Random Byte Generation, AES), parts of TweetNaCL



Verified Textbooks!







And more!

- Bedrock system
- Ur/Web compiler
- CompCert TSO compiler
- CompCert static analysis tools
- Jitk and Data6 verified filesystems
- Verified Fscq from MIT
- …

Why now?

Urgent need for increased confidence

+

Diminishing value of "paper proofs"

+

Progress on enabling technologies

Enabling Technologies

Better theory

- Operational semantics, etc.
- Domain-specific logics
  - E.g. Separation logic

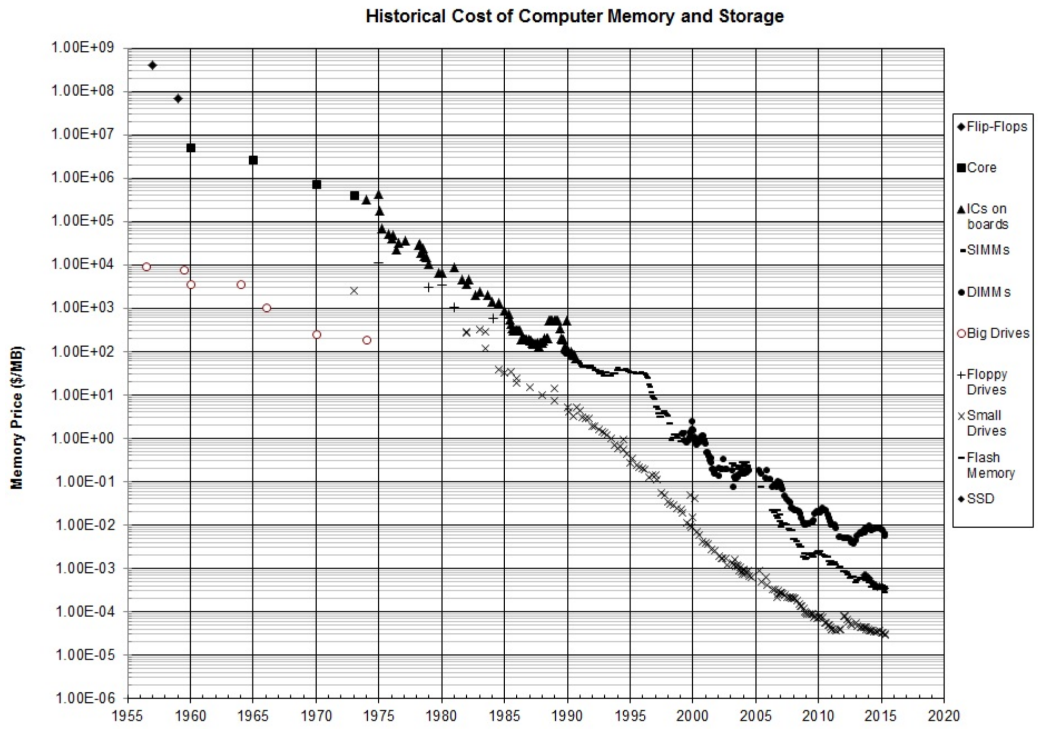
Enabling Technologies

Better tools

- Proof assistants
  - Coq, Isabelle, ACL2, Twelf, HOL-light, …
- Testing tools and methodologies
  - QuickCheck, QuickChick, …
- DSLs for writing specifications
  - OTT, Lem, Redex, …
- Languages with integrated specifications
  - Dafny, Boogie, JML, F*, Liquid Types, Verilog PSL, Dependent Haskell, …

Enabling Technologies

Faster hardware
also helps!



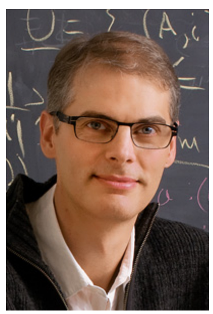
What next?

The Science of Deep Specification

Andrew Appel
Princeton

Steve Zdancewic
University of Pennsylvania

Adam Chlipala
MIT



Yours truly
University of Pennsylvania

Zhong Shao
Yale

Stephanie Weirich
University of Pennsylvania

Goal:

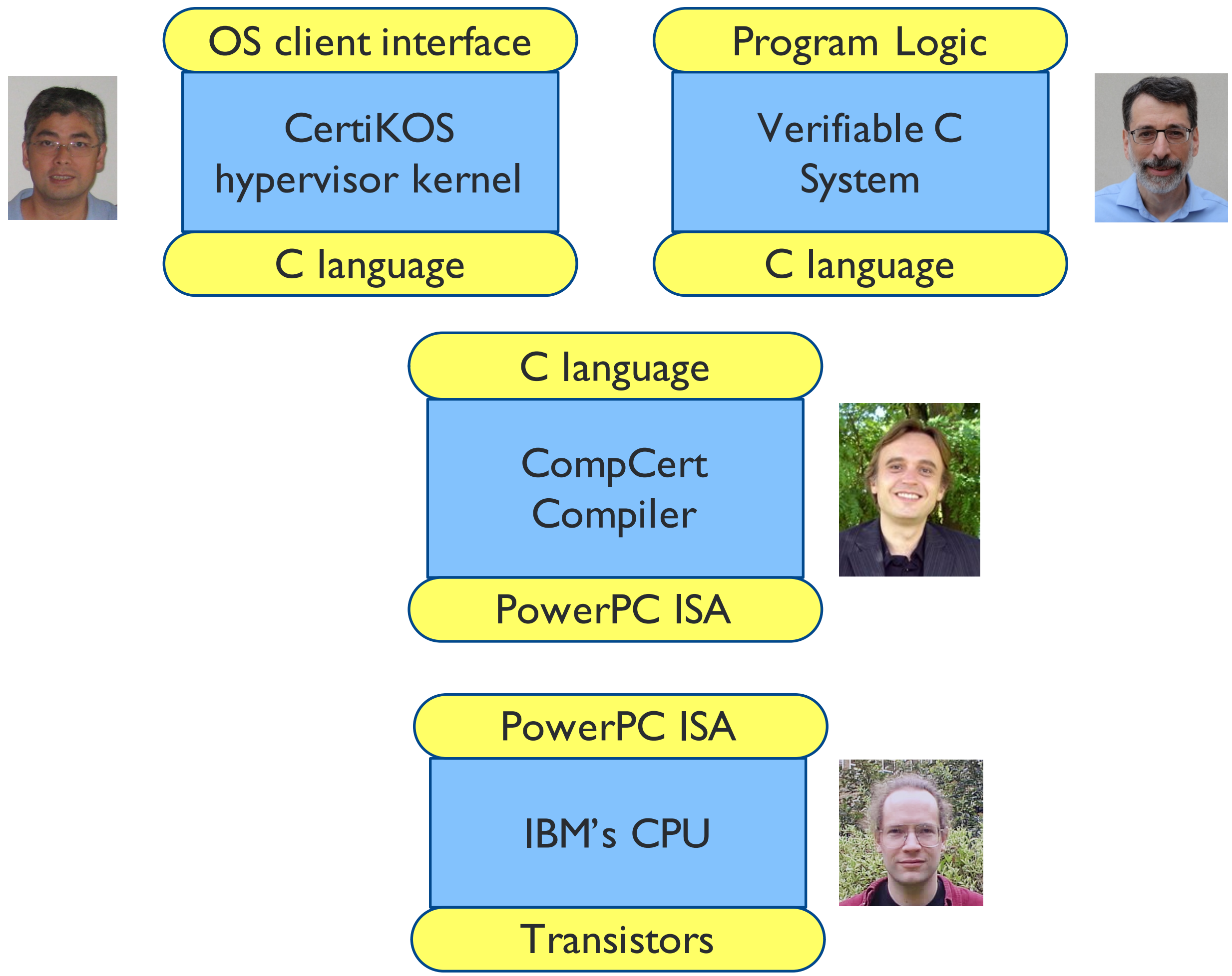
Move from

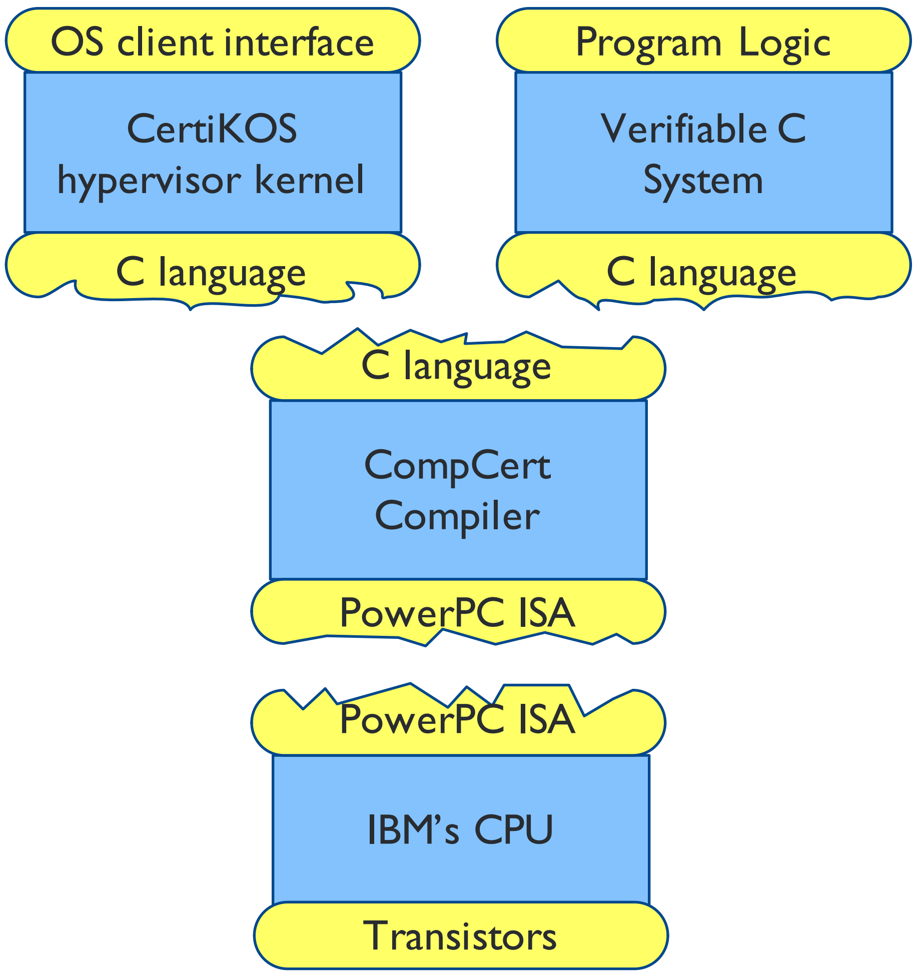
one-off success stories

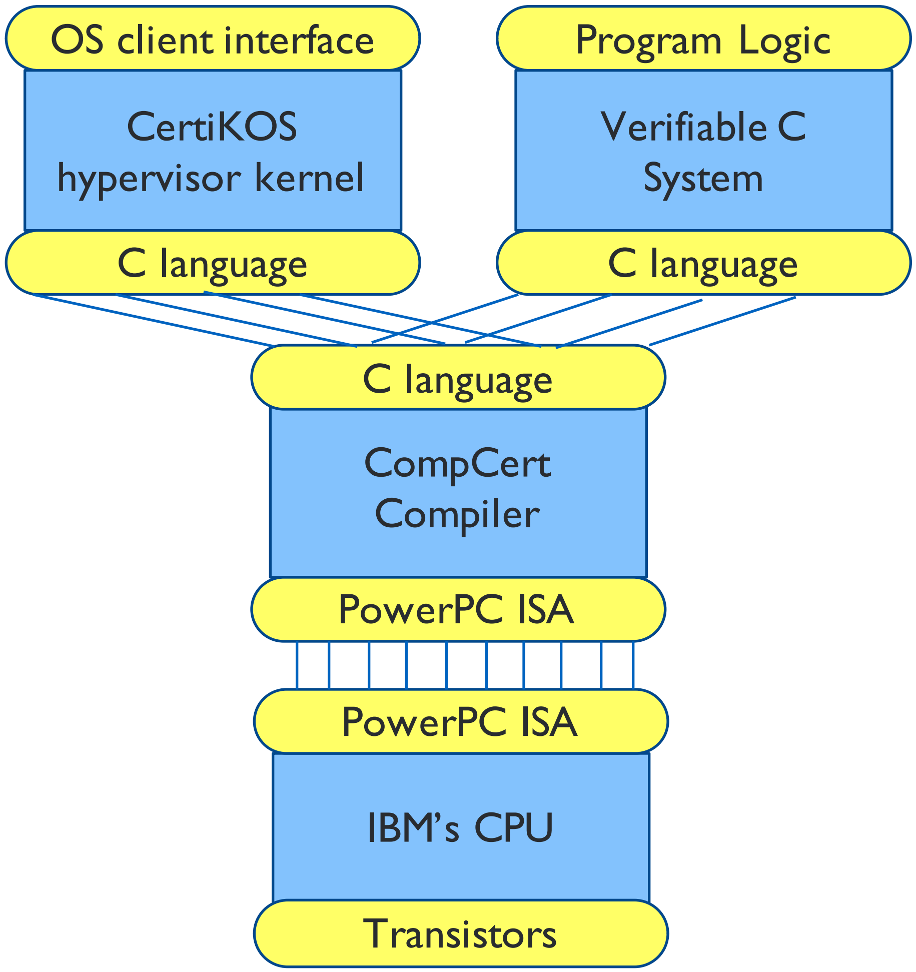
to

sustainable engineering practice
at industrially relevant scale

Lessons from CompCert

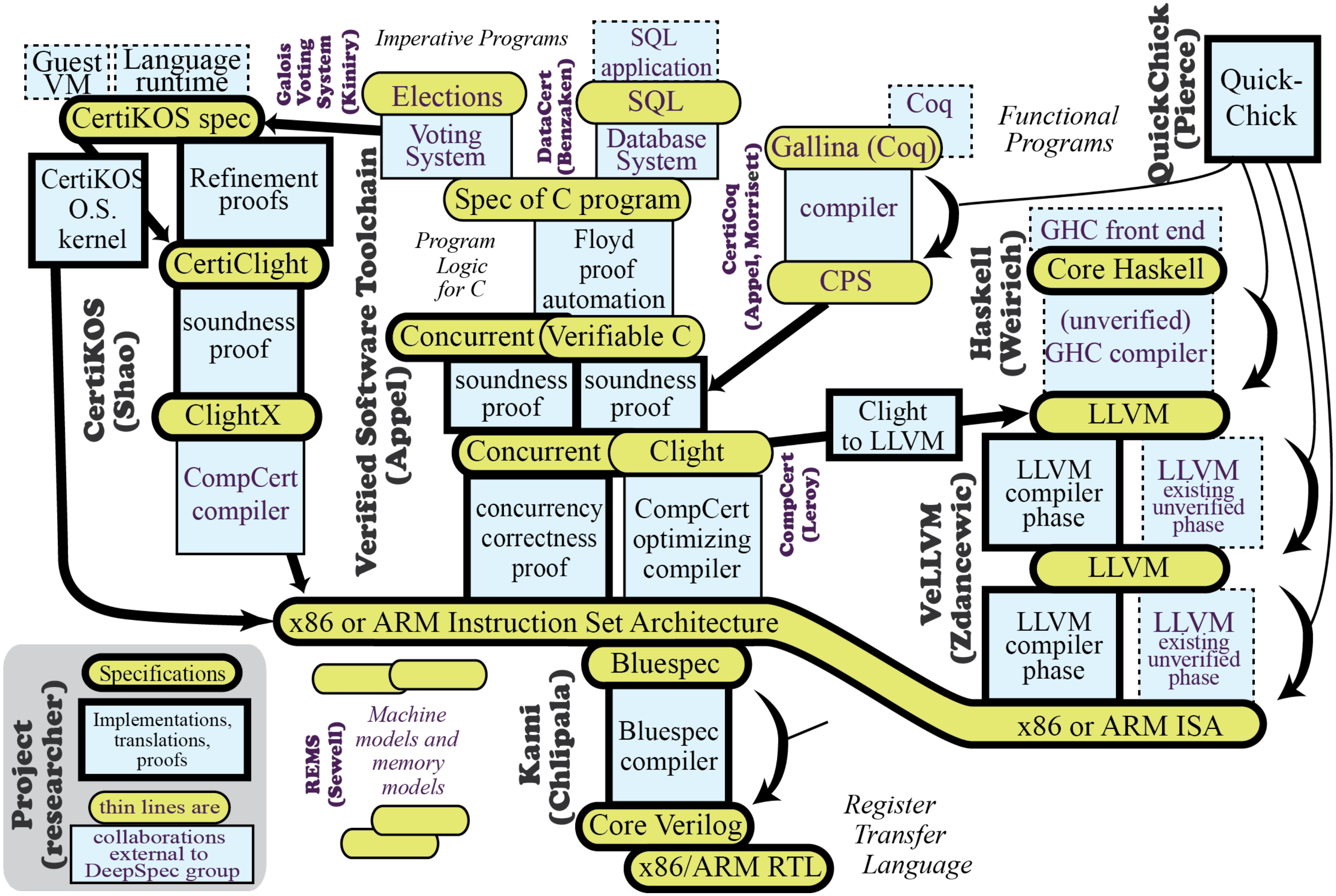



OS client interface
CertiKOS hypervisor kernel
C language
Program Logic
Verifiable C System
C language
C language
CompCert Compiler
PowerPC ISA
PowerPC ISA
IBM's CPU
Transistors


OS client interface
CertiKOS hypervisor kernel
C language
Program Logic
Verifiable C System
C language
C language
CompCert Compiler
PowerPC ISA
PowerPC ISA
IBM's CPU
Transistors

Research threads

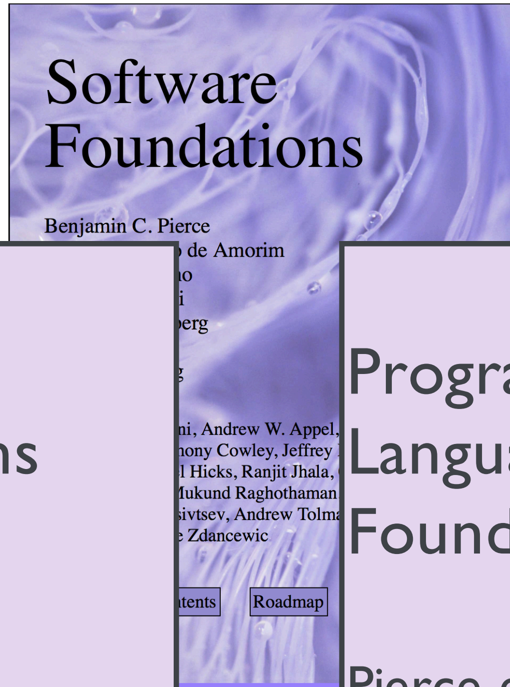
Fiat/Kami	RISC-V implementation, verified down to RTL
CertiCoq	Verified Gallina-to-CompCert-C compiler
CertiKOS	Verified OS / hypervisor
VST	Verified Software Toolchain for C
Vellvm	Verified LLVM
Core Haskell	Formal model of GHC core
QuickChick	Specification-based random testing in Coq



Guest VM
Language runtime
CertiKOS spec
CertiKOS O.S. kernel
Refinement proofs
CertiClight
soundness proof
ClightX
CompCert compiler
CertiKOS (Shao)
Galois Voting System (Kiniry)
Imperative Programs
Elections
Voting System
Verified Software Toolchain (Appel)
DataCert (Benzaken)
SQL application
SQL
Database System
Spec of C program
Program Logic for C
Floyd proof automation
Concurrent
Verifiable C
soundness proof
soundness proof
Concurrent
Clight
concurrency correctness proof
CompCert optimizing compiler
CompCert (Leroy)
CertiCoq (Appel, Morrisett)
Coq
Gallina (Coq)
compiler
CPS
Functional Programs
QuickChick (Pierce)
Quick-Chick
GHC front end
Core Haskell
(unverified) GHC compiler
Haskell (Weirich)
Clight to LLVM
LLVM
LLVM compiler phase
LLVM existing unverified phase
LLVM
LLVM compiler phase
LLVM existing unverified phase
VeLLVM (Zdancewic)
x86 or ARM Instruction Set Architecture
x86 or ARM ISA
Bluespec
Bluespec compiler
Core Verilog
x86/ARM RTL
Kami (Chlipala)
Register Transfer Language
REMS (Sewell)
Machine models and memory models
Project (researcher)
Specifications
Implementations, translations, proofs
thin lines are
collaborations external to DeepSpec group

Course design

- Undergrad
  - Drop-in replacements for standard compiler and OS courses
  - Built around pedagogical versions of Vellvm and CertiKOS
  - Students will learn to read and interact with specifications (but not proofs)
  - Code connected to specifications via random testing
- Grad
  - New course on formally specifying and verifying systems software and hardware



Logical Foundations

Pierce et al.

Programming Language Foundations

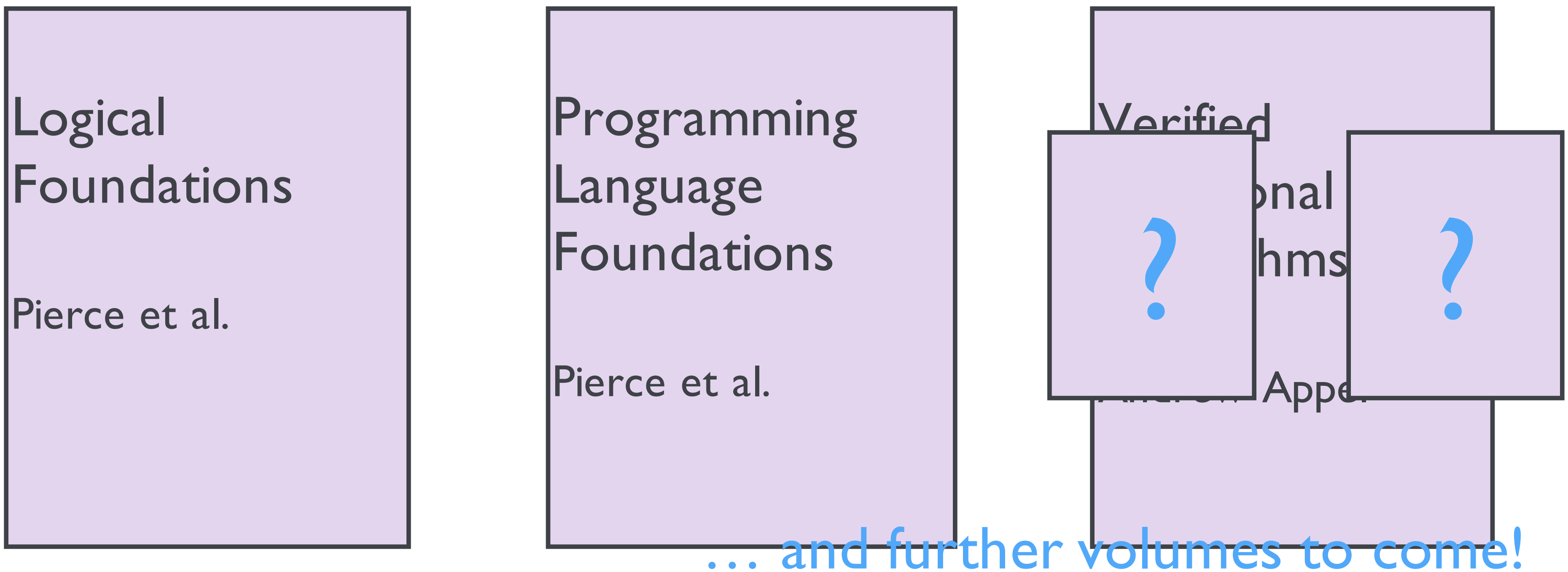
Pierce et al.

Verified Functional Algorithms

Andrew Appel

Late 2016

Early 2017


Logical
Foundations
Pierce et al.
Programming
Language
Foundations
Pierce et al.
Verified
?
?
… and further volumes to come!

Join us!

Thank you!
(any (more) questions?)

Summer schools
(July 13-28, 2017,
in Philadelphia)

Technical workshops
(one last spring; several more to follow)

visitors program

PhD and postdoc positions

Visit **deepspec.org** to see what's happening
and join our mailing list