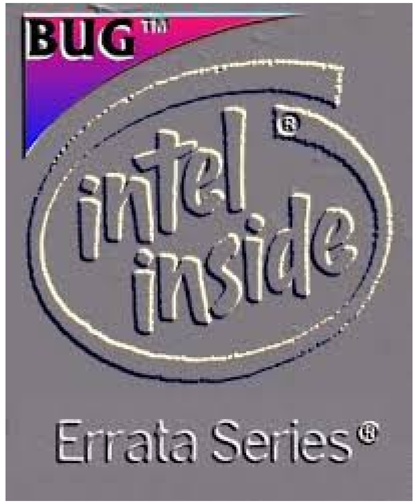
Raising the Level of Abstraction in Systems Programming with Fiat and Extensible, Correct-by-Construction Compilers

Adam Chlipala
MIT CSAIL
ENTROPY workshop
January 2018

Joint work with: Thomas Braibant, Santiago Cuellar, Benjamin Delaware, Samuel Duchovni, Jason Gross, Gregory Malecha, Clément Pit—Claudel, Sorawit Suriyakarn, Peng Wang, and Katherine Ye

How We're Doing

Software bug causes launch failure



Hardware bug causes massive recall

Software bug leaks secret information

Software bug causes loss of life

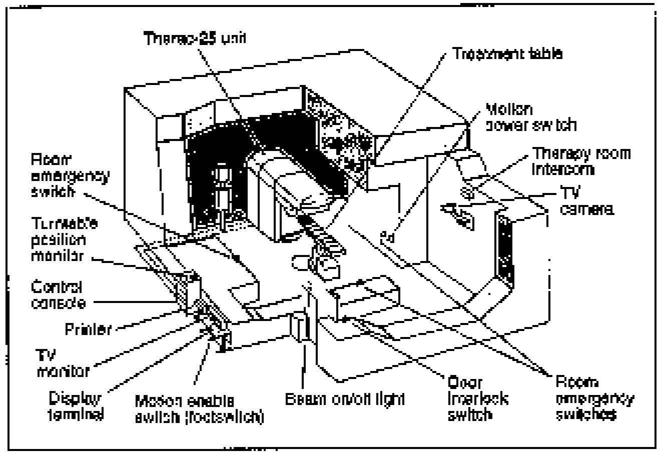
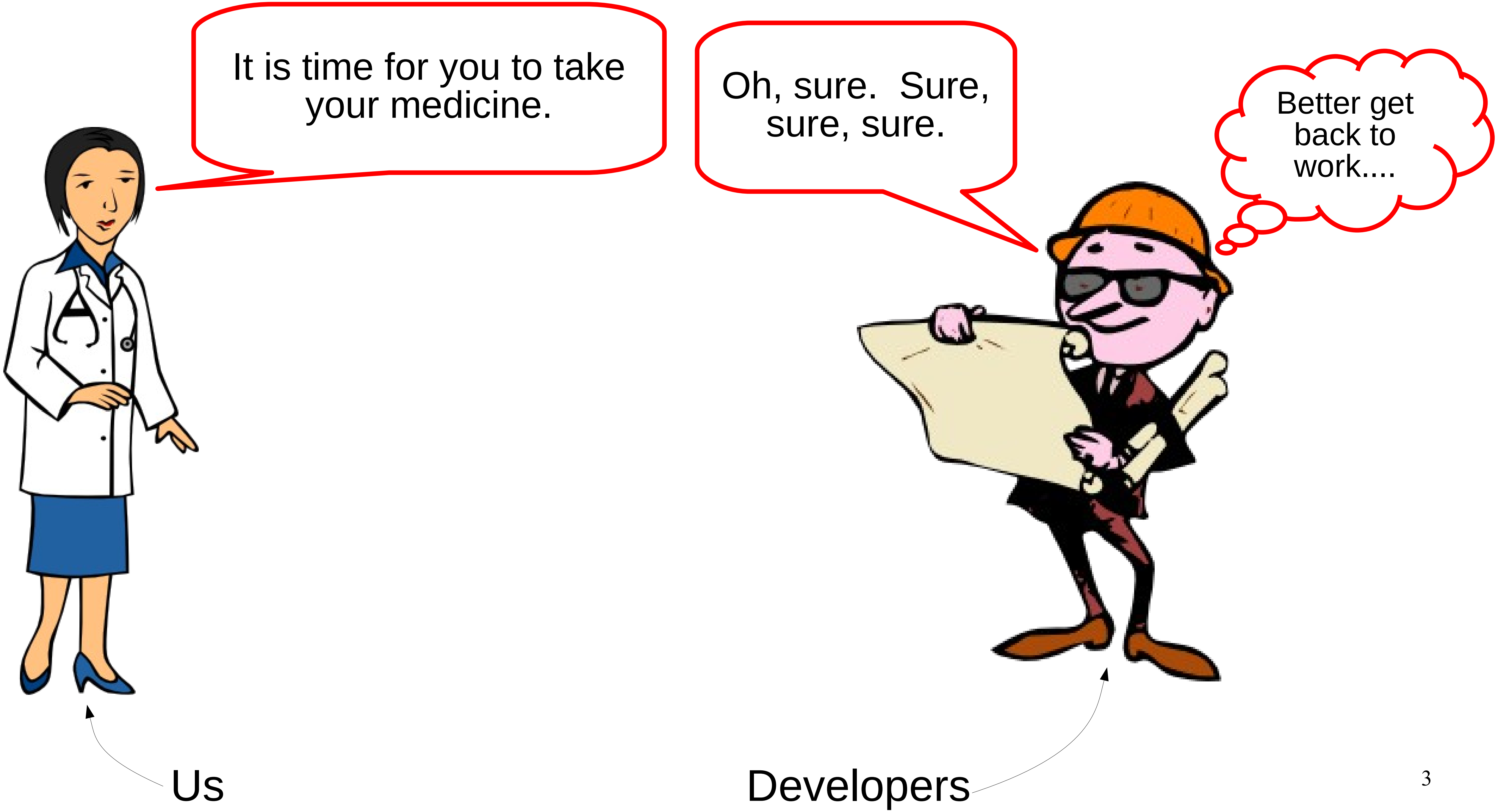
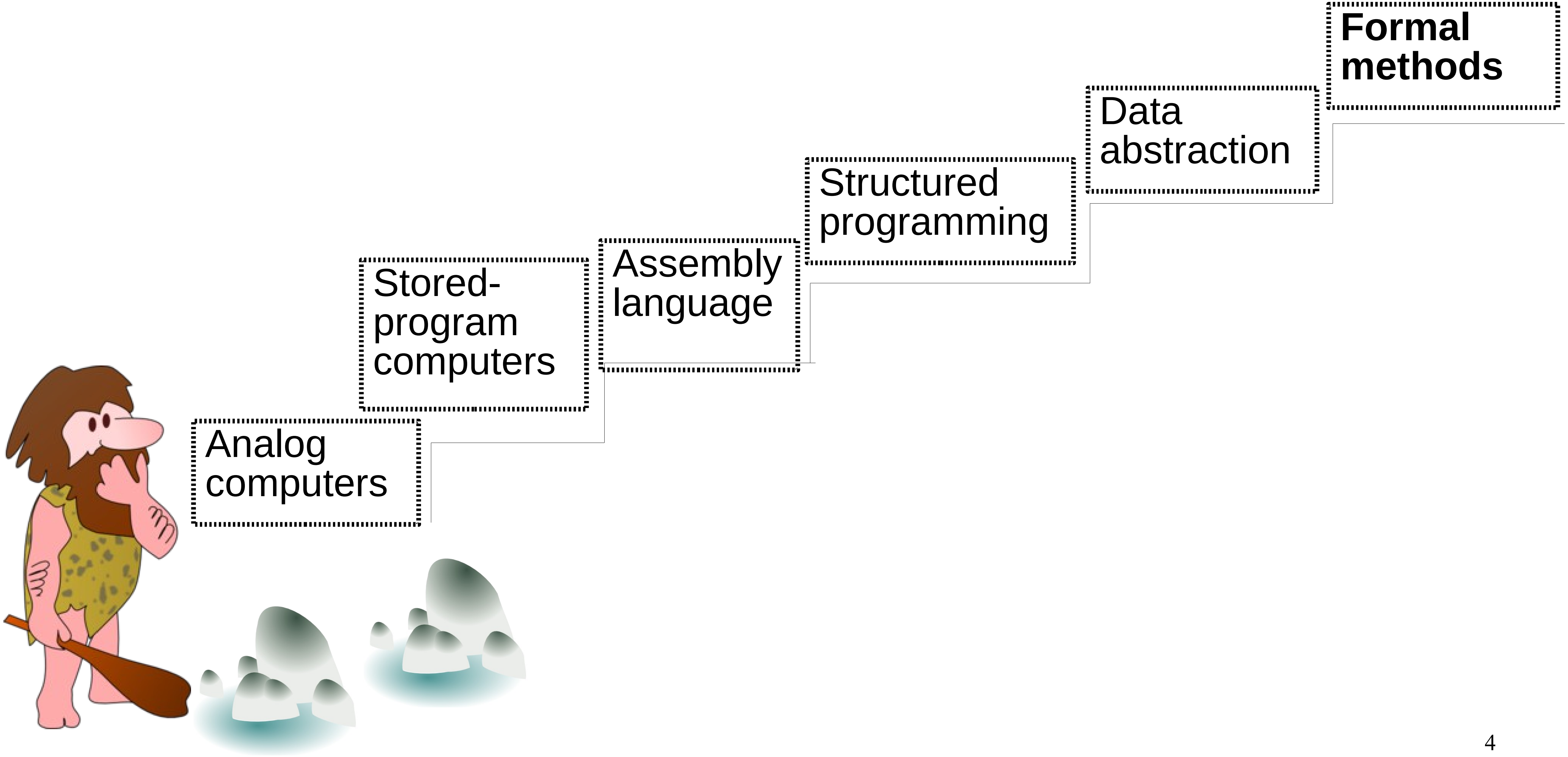


Figure 1. Typical Therac-25 facility.

<u>The time has come to settle for nothing less than high-assurance computer systems!</u>


It is time for you to take your medicine.
Oh, sure. Sure, sure, sure.
Better get back to work....
Us
Developers

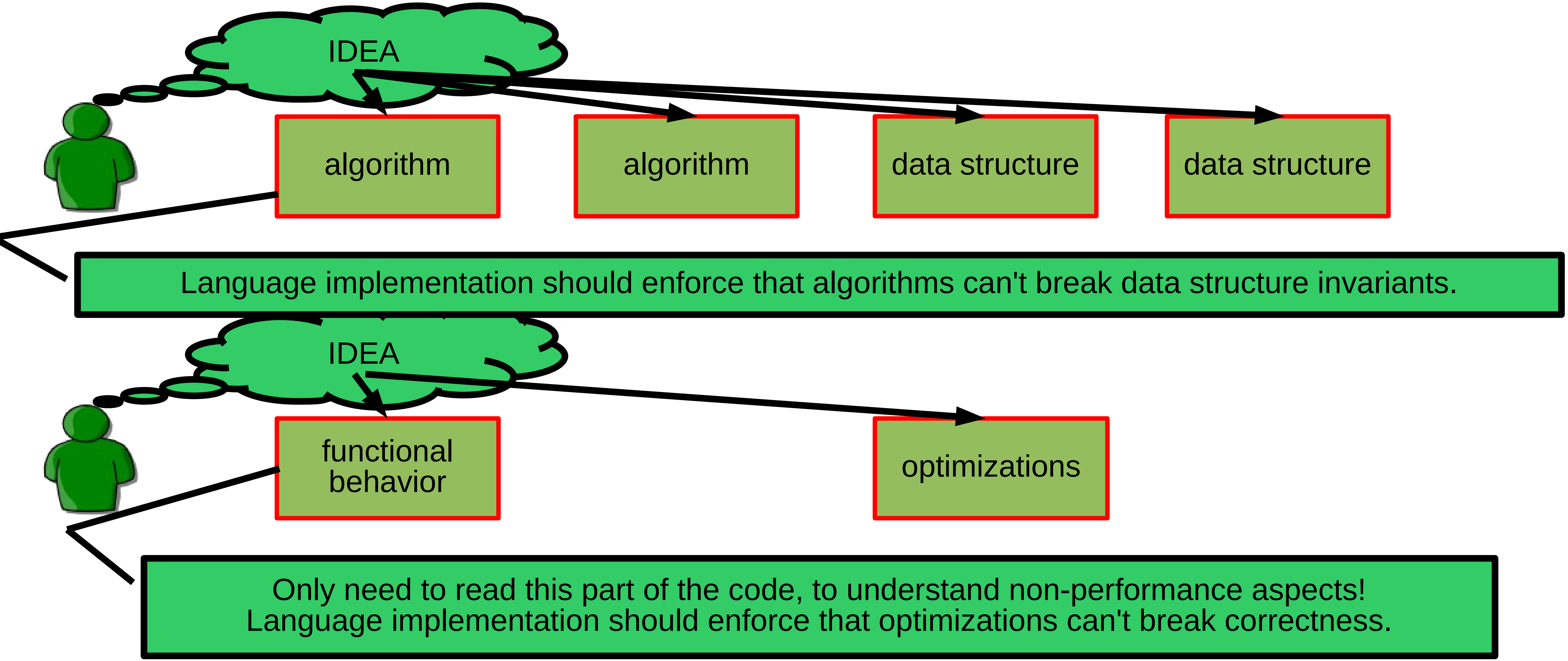


Analog computers
Stored-program computers
Assembly language
Structured programming
Data abstraction
Formal methods

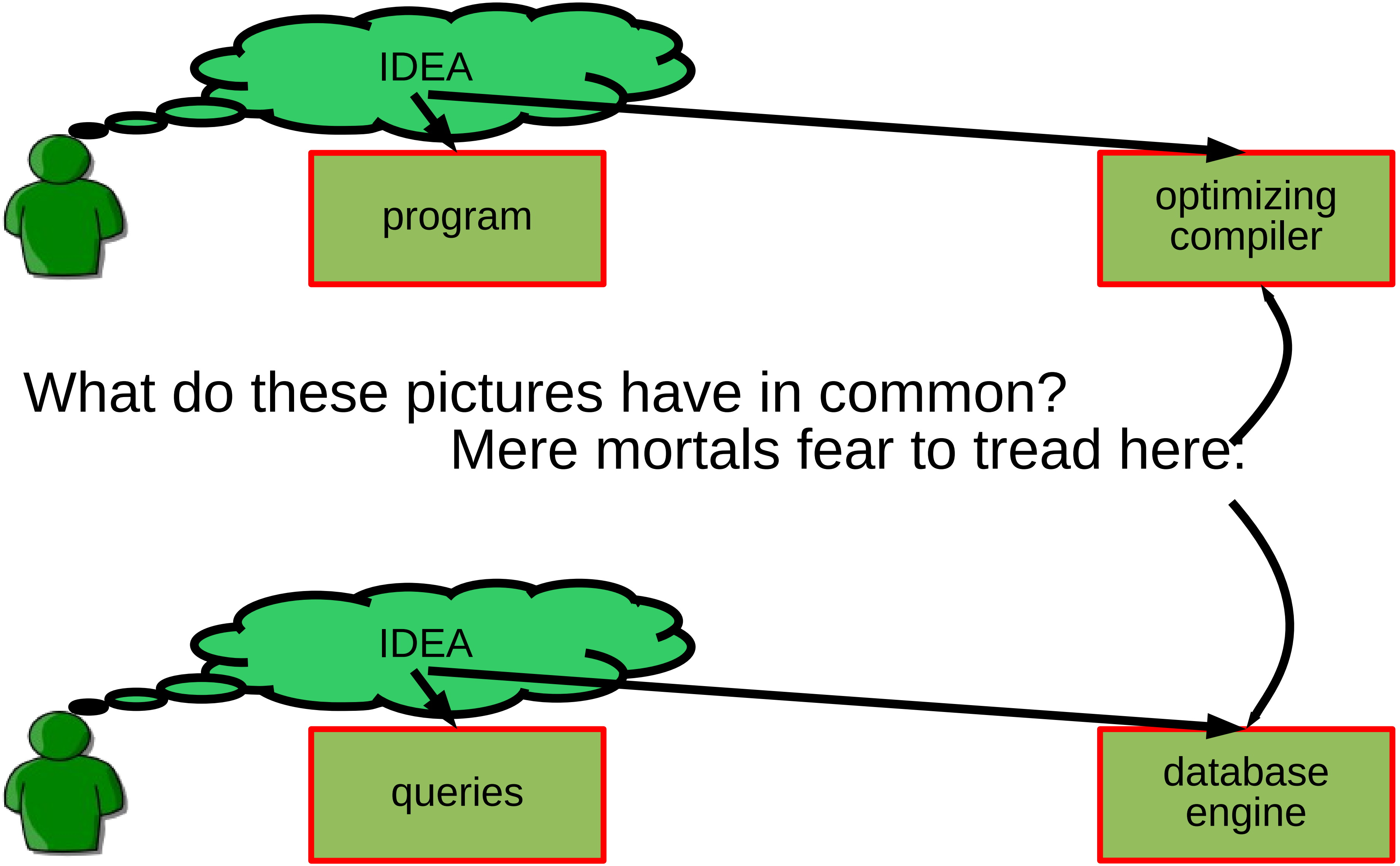
Formal specifications and proofs deserve to be *the new glue* holding together complex systems and helping us understand them and their parts.

The design of systems should *change* to take advantage of formal methods to *raise the level of abstraction*.

The Big Idea


IDEA
algorithm
algorithm
data structure
data structure
Language implementation should enforce that algorithms can't break data structure invariants.
IDEA
functional behavior
optimizations
Only need to read this part of the code, to understand non-performance aspects!
Language implementation should enforce that optimizations can't break correctness.

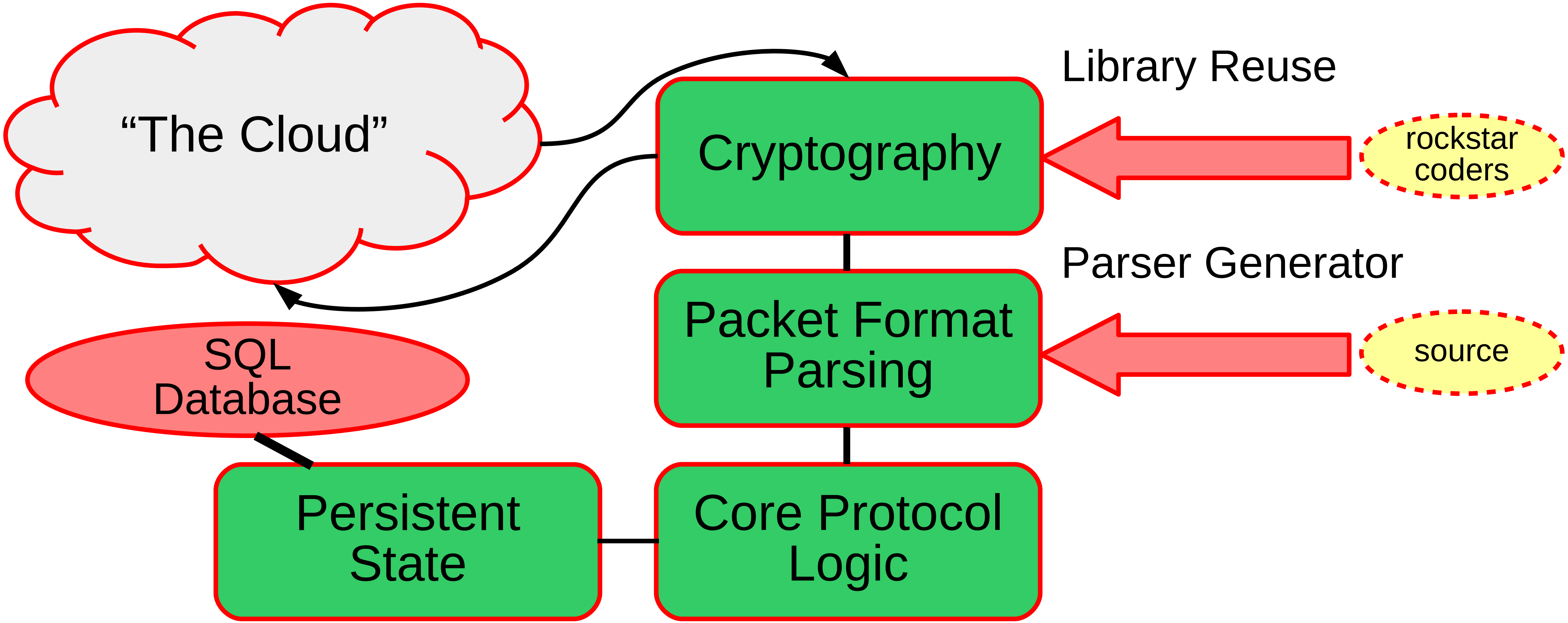
That Looks Familiar



What do these pictures have in common?

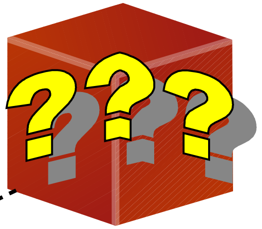
Mere mortals fear to tread here.

State of the Art: Building an Internet Server



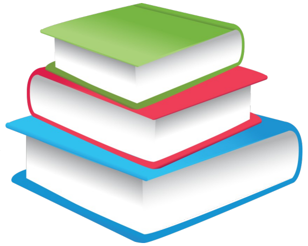
Complaints About: Talking to a Standard Server

And by the way, sometimes there are serious bugs.

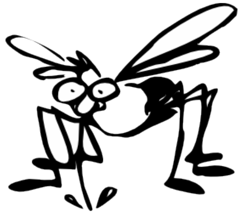


Database is a black box, maintained by an elite cadre, often not doing quite what you need

SQL Database



Yet another language, only understandable after reading a pile of documentation

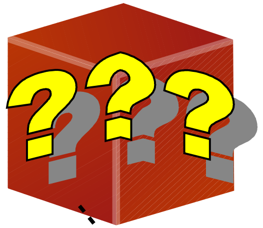


Awkward API, often based on string manipulation, allowing code-injection vulnerabilities

Persistent State

Complaints About: Using a Domain-Specific Language

And by the way, sometimes there are serious bugs.



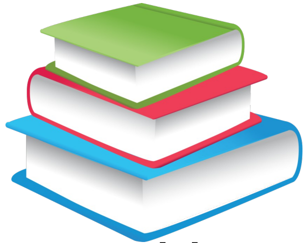
Compiler is a black box, maintained by an elite cadre, often not doing quite what you need

Parser Generator

source

Packet Format Parsing

Core Protocol Logic



Yet another language, only understandable after reading a pile of documentation

Awkward integration, with build processes instead of clean intra-language abstractions

What About *Embedded* DSLs?

Complaint	**Addressed?**
Yet another language	*Partly yes*, but still need to learn the semantics of the DSL, even if syntax may be standardized
Compiler is a black box	**No**!
Awkward integration	**Yes**!
Sometimes serious bugs	*Partly yes*, as we usually avoid type-safety bugs but not deeper semantic bugs

Complaints About: Using Libraries Coded by Wizards

Algorithms × Prime #s × HW Arches

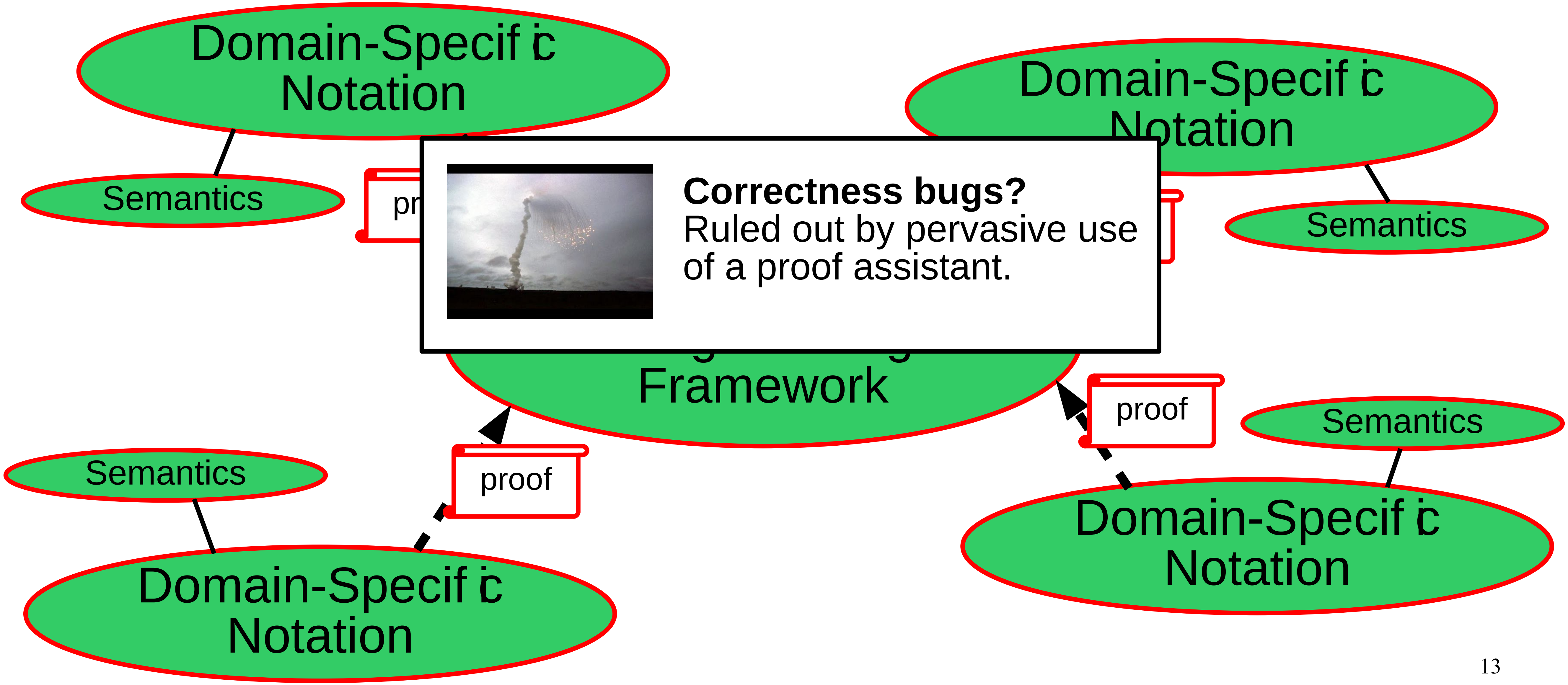
Labor-intensive adaptation, with each combination taking *at least* several days for an expert.

Library Reuse

Cryptography ← rockstar coders

And by the way, sometimes there are serious bugs.

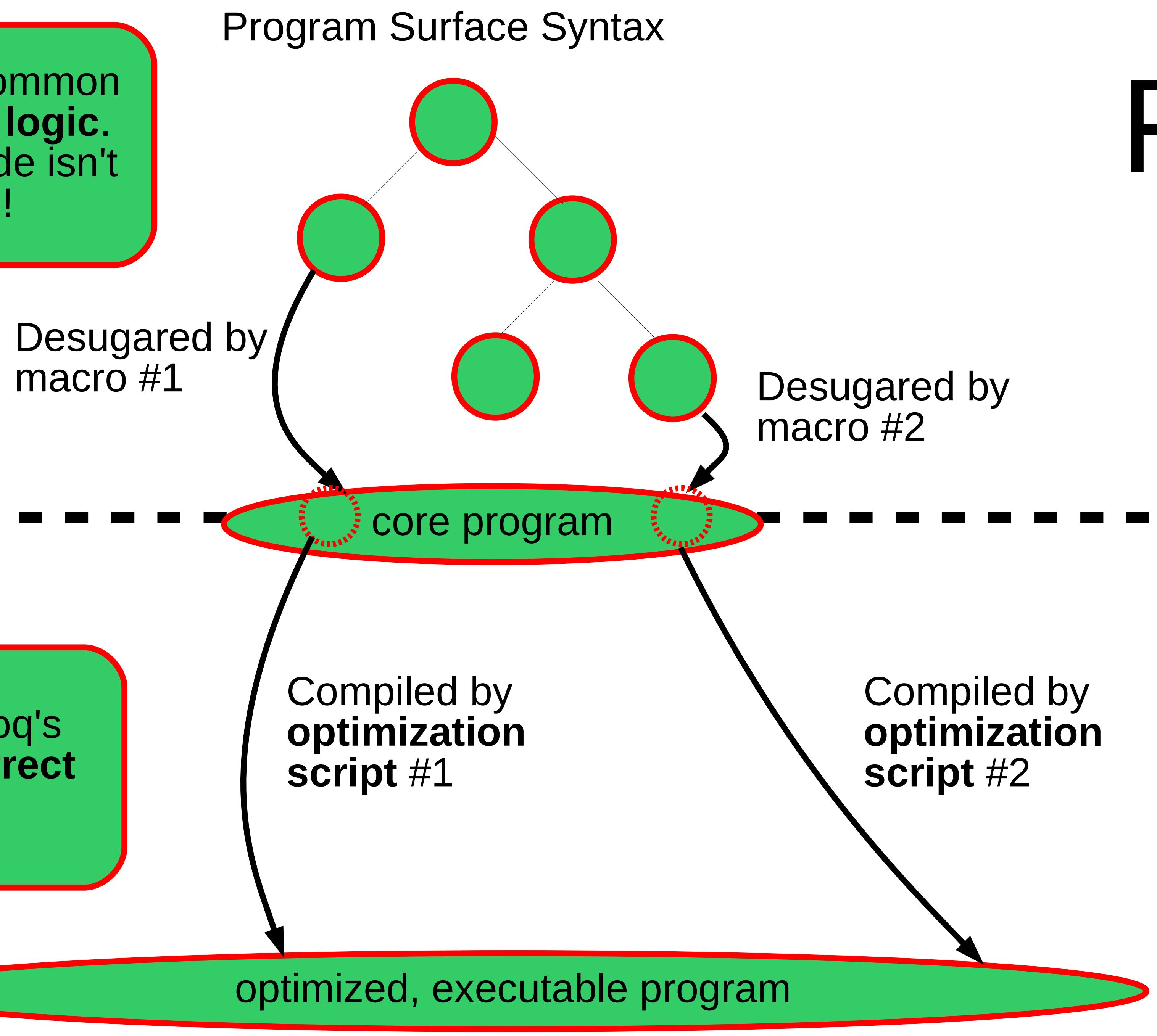
Rethinking the Programming Framework



Fiat

Macros desugar into the common language of **higher-order logic**. Often the most concise code isn't obviously executable!

Functionality



Performance

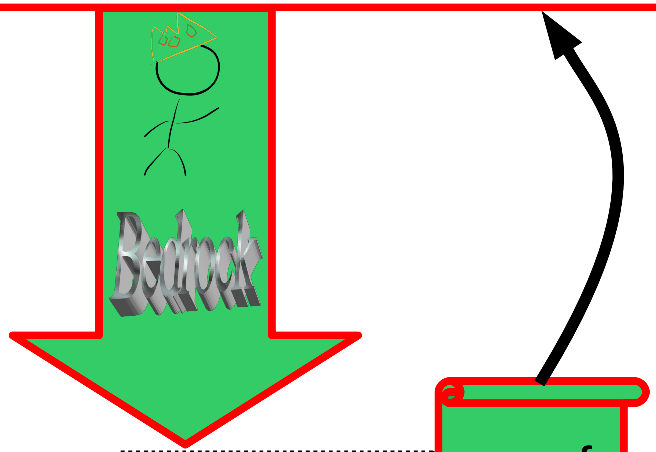
Optimization scripts use Coq's tactic language and are **correct by construction**.

Fiat's Layers

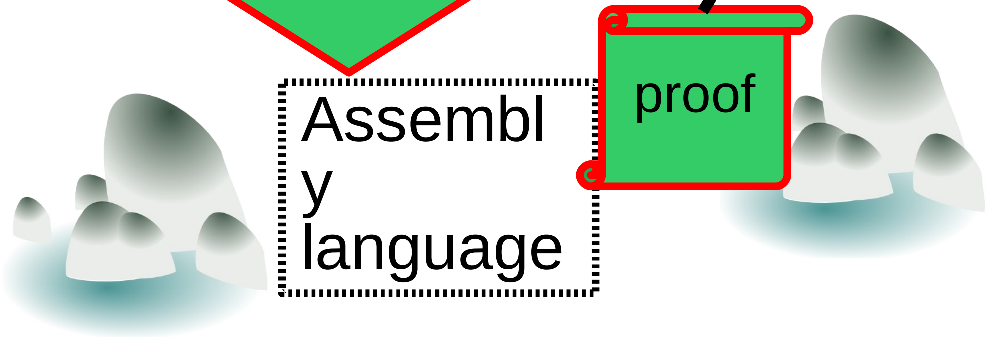
1. **Coq**: logic and tactic language
2. **Computations**: nondeterministic functional programs
3. **Abstract data types**: encapsulated state
4. **Domains**: libraries for particular spec styles
5. **Applications**

```
Definition SchedulerSchema :=
  Query Structure Schema [
          relation "Processes" has schema
          <"pid" :: W, "state" :: State, "cpu" :: W>
          where (UniqueAttribute ``"pid")
  ] enforcing [].

Definition SchedulerSpec : ADT _ :=
  QueryADTRep SchedulerSchema {
    (* … *)

    Def Method1 "Enumerate" (r : rep) (state : State) : rep * list W :=
      For (p in r!"Processes")
      Where (p!"state" = state)
      Return (p!"pid"),

    (* … *)
  }.
```

Automatic, proof-generating derivation of assembly code from relational specif ications

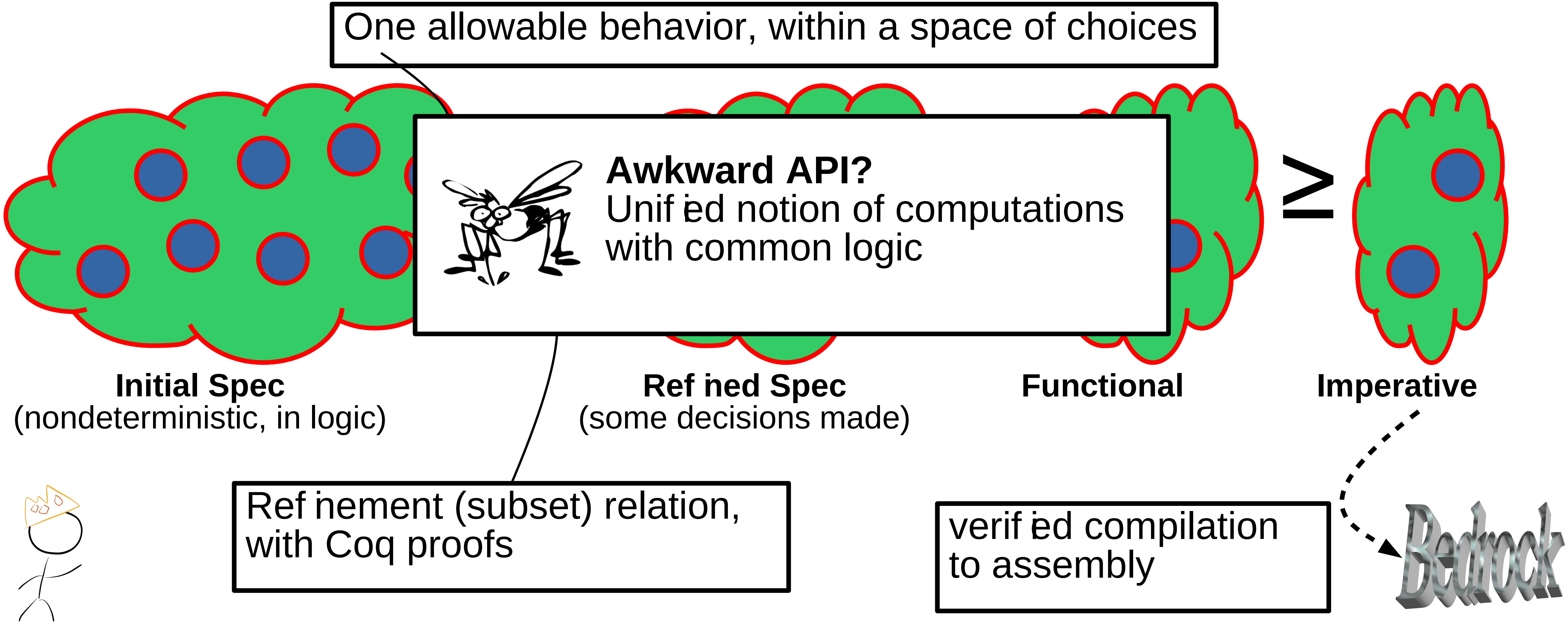


Foundational proofs: connect operational semantics of assembly to original spec, trusting little beyond standard Coq proof checker



Demo:
Query Structures
& the Bookstore example

Core Fiat: **Computations** as Sets of Results



Computations naturally form a **monad**.

$$\text{ret } v \stackrel{\text{def}}{=} \{v\}$$

$$x \qquad c_1;\ c_2(x) \stackrel{\text{def}}{=} \{v \in c_2(x) \mid x \in c_1\}$$

Example:

$x \qquad \{n \in \mathbb{N} \mid \exists m.\ n = 2 \times m\};$

$y \qquad \mathbb{N};$

$\text{ret } (x + 2 \times y)$

In other words, choose an even number,
by some very indirect means!

Refinement of computations is just **subset**.

In other words,
resolving *nondeterminism*
is the same as
moving to a *smaller set*.

ret 42

⊇

Example:

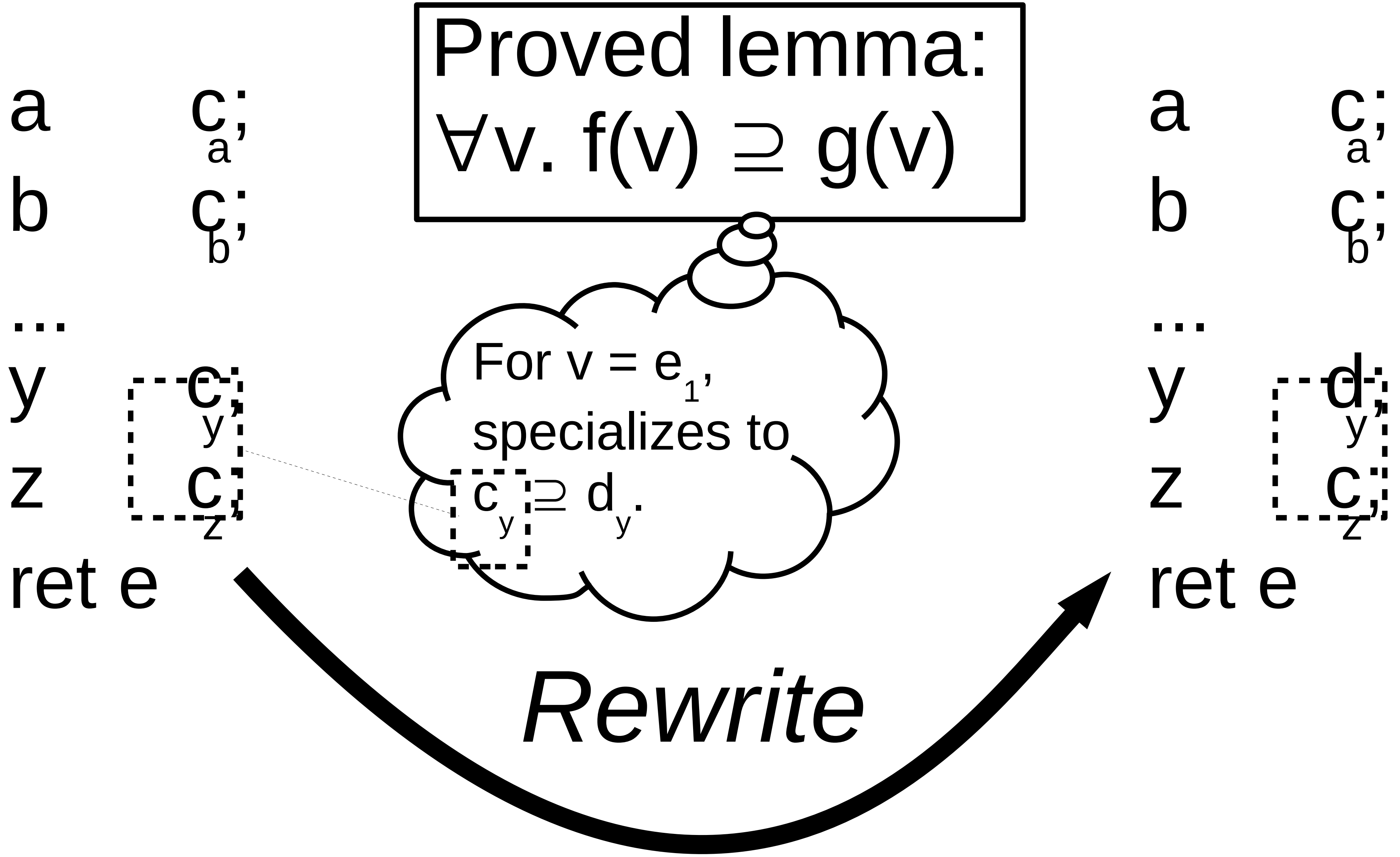
x {n∈ ℕ | ∃m. n = 2 × m};

y ℕ;

ret (x + 2 × y)

$\subseteq \{n \in \mathbb{N} \mid \text{even}(n)\}$

Refinement is compatible with **rewriting**.



Demo:
find an element not in a list

Fiat Principle #2: **Abstract Data Types** with computations for methods

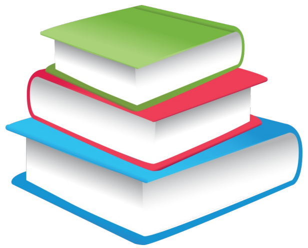
Type of private state (annotation: Type of **private state**)

ADT {
  rep = T
  method $m_1(x : D_1) : R_1 = c_1$
  method $m_2(x : D_2) : R_2 = c_2$
  …
  method $m_n(x : D_n) : R_n = c_n$
}

Method has computation as body, allowing **nondeterminism**.

Macros as Documentation

```
Def Method1 "NumOrders" (r : rep) (author : string) : rep * nat :=
    count <- Count (For (o in r!sORDERS) (b in r!sBOOKS)
             Where (author = b!sAUTHOR)
```



Yet another language?
Readable macros desugaring into a common language

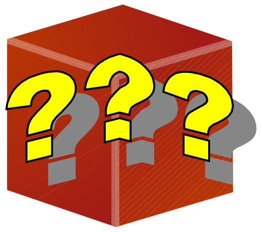
```
Count b ≡
results ← b;
return length(results)
```

```
return [a]
```

```
Where P b ≡
{l | P → l B b ش ¬P → l = []}
```

Ingredients for Optimization Scripts

filter … l1 l2)
= join



Compiler a black box?
Easy to add new
optimization rules w/ proofs

filter (λ(k, v) → k = k_0) (BST.enumerate t)
= BST.lookup t k_0

Example Derivation: SQL-style database

table island : {Name : string, Size : int, Temp : int}

```
sizeOf(db, name) =
    for i ∈ db.island
    where i.Name = name
    return i.Size
```

STEP 1: representation change

dictionary island : string {Size : int, Temp : int}

Abstraction relation:

db ~ fmap ≝ ∀r. r ∈ db fmap(r.Name) = {Size = r.Size, Temp = r.Temp}

sizeOf(fmap, name) =
 db {db | db ~ fmap}
 for i ∈ db.island
 where i.Name = name
 return i.Size

RULE: *representation change*

state x : τ_x
m(x, args) = e

state y : τ_y
m(y, args) =
 x {x | x ~ y}
 e

STEP 2: for+fmap

dictionary island : string   {Size : int, Temp : int}

sizeOf(fmap, name) =
  for k, v ∈ fmap
  let i = {Name = k, Size = v.Size, Temp = v.Temp}
  where i.Name = name
  retu

RULE: *use dictionary*

x   {x | ∀r. r ∈ x   y(k(r)) = v(r)}
for r ∈ x
q(r)

for k, v ∈ y
let r = $kv^{-1}(k, v)$
q(r)

STEP 3: simplify

dictionary island : string   {Size : int, Temp : int}

```
sizeOf(fmap, name) =
    for k, v ∈ fmap
    where k = name
    return v.Size
```

```
sizeOf(fmap, name) =
    for k, v ∈ fmap
    let i = {Name = k, Size = v.Size, Temp = v.Temp}
    where i.Name = name
    return i.Size
```

STEP 4: for-where-key-eq

dictionary island : string {Size : int, Temp : int}

```
sizeOf(fmap, name) =
    for v ∈ fmap.lookup(name)
    return v.Size
```

```
sizeOf(fmap, nam
    for k, v ∈ fmap
    where k = nam
    return v.Size
```

RULE: *equality test to lookup*

```
for k, v ∈ fmap
where k = x
q(k, v)
```

⇒

```
for v ∈ fmap.lookup(x)
q(x, v)
```

Example Derivation: binary decoder

```
T = {A : int, B : string, C : list int}

encode(t : T) = encodeInt(t.A)
                ++ encodeInt(len(t.C))
                ++ encodeString(t.B)
                ++ encodeList(encodeInt, t.C)

decode(s : bitstring) =
    {t | s = encode(t)}
```

Example Derivation: binary decoder

T = {A : int, B : string, C : list int}

```
encode(t : T) = encodeInt(t.A)
                ++ encodeInt(len(t.C))
                ++ encodeString(t.B)
                ++ encodeList(encodeInt, t.C)

decode(s : bitstring) =
    {t | s = eI(t.A) ++ eI(len(t.C)) ++ eS(t.B) ++ eL(eI, t.C)}
```

Example Derivation: binary decoder

decode(s : bitstring) =
    {t | s = eI(t.A) ++ eI(len(t.C)) ++ eS(t.B) ++ eL(eI, t.C)}

STEP 1: decode A

decode(s : bitstring) =
    let a, s = dI(s)
    {t | t.A = a ∧ s = eI(len(t.C)) ++ eS(t.B) ++ eL(eI, t.C)}

{t

RULE: *decode integer*

{x | P(x) ∧ s = eI(f(x)) ++ s'} ⇒ let v, s = dI(s)
{x | f(x) = v ∧ P(x) ∧ s = s'}

STEP 2: decode length

```
decode(s : bitstring) =
    let a, s = dI(s)
    let n, s = dI(s)
    {t | len(t.C) = n ∧ t.A = a ∧ s = eS(t.B) ++ eL(eI, t.C)}
```

```
let a, s = dI(s)
{t | t.A = a ∧ s = eI(len(t.C)) ++ eS(t.B) ++ eL(eI, t.C)}
```

STEP 3: decode B

decode(s : bitstring) =

let a, s = dI(s)

let n, s = dI(s)

let b, s = dS(s)

$\{t \mid t.B = b \wedge len(t.C) = n \wedge t.A = a \wedge s = eL(eI, t.C)\}$

RULE: *decode string*

$\{x \mid P(x) \wedge s = eS(f(x)) \mathbin{++} s'\}$ ⇒

let v, s = dS(s)

$\{x \mid f(x) = v \wedge P(x) \wedge s = s'\}$

STEP 4: decode C

```
decode(s : bitstring) =
    let a, s = dI(s)
    let n, s = dI(s)
    let b, s = dS(s)
    let c, s = dL(dI, s, n)
    {t | t.C = c ∧ t.B = b ∧ len(t.C) = n ∧ t.A = a ∧ s = []}
```

RULE: *decode list*

$\{x \mid P(x) \wedge s = eL(eI, f(x)) +\!+\ s'\}$ → let v, s = dL(eI, s, n)
$\{x \mid f(x) = v \wedge P(x) \wedge s = s'\}$

when $\forall x.\ P(x) \quad len(f(x)) = n$

STEP 5: construct t

```
decode(s : bitstring) =
    let a, s = dI(s)
    let n, s = dI(s)
    let b, s = dS(s)
    let c, s = dL(dI, s, n)
    if s = []:
        {A = a, B = b, C = c}
    else:
        fail
```

RULE: *use witness*

{x | P(x) ∧ s = []} ⇒

```
if s = []:
    v
else:
    fail
```

when P(v)

```
let a, s = dI(s)
let n, s = dI(s)
let b, s = dS(s)
let c, s = dL(dI, s, n)
{t | t.C = c ∧ t.B = b ∧ len(t.C) = n ∧ t.A = a ∧ s = []}
```

A Relational Abstract Data Type

```
ADT FiniteSet(α) {
   private set : ℘(α);

   constructor init() {
      set := {};
   }

   method add(x : α) {
      set := {x} ∪ set;
   }

   method member(x : α) {
      return {b : bool | b = true ↔ x ∈ set};
   }

   method toList() {
      return {l : list(α) | NoDup(l) ∧ ∀x. x ∈ set ↔ In(x, l)};
   }
}
```

A nondeterministic abstract data type formalizes expectations of a data structure, without committing to optimization details.

ADT Delegation

```
ADT MyBookCollection(FS : FiniteSet(nat × string)) {
   private books : FS;

   constructor init() {
      books := new FS();
   }

   method newBook(isbn : nat, title : string) {
      books.add((isbn, title));
   }

   method allTitles() {
      return map (fn (_, title) => title) (books.toList());
   }
}
```

Delegation allows one ADT to assume an implementation of another.

Translating to Lower-Level Imperative Code

```
ADT MyBookCollection(FS : FiniteSet(nat × string)) {
  private books : FS;

  method newBook(isbn : nat, title : string) {
    tup := new Tuple();
    tup.set(0, isbn);
    tup.set(1, title);
    books.add(tup); }

  method allTitles() {
    ls := books.toList();
    out := new List();
    while (!ls.isEmpty()) {
      x := ls.pop();
      title := tup.get(x, 1);
      out.push(title);
    }
    delete ls;
    out.reverse();
    return out; } }
```

Tactic-based derivation produces imperative code from functional, using **extensible hint databases**.

```
books.add((isbn, title));
```

```
map (fn (_, title) => title)
(books.toList())
```

Verified Compilation with ADTs

```
method allTitles() {
  ls := books.toList();
  out := new List();
  while (!ls.isEmpty()) {
    x := ls.pop();
    title := tup.ge
    out.push(title)
  }
  delete ls;
  out.reverse();
  return out;
}
```

```
{List(ls, hd::tl)}
x := ls.pop();
{Tuple(x, hd) * List(ls, tl)}
```

Two key parameters to *operational semantics* of imperative language:

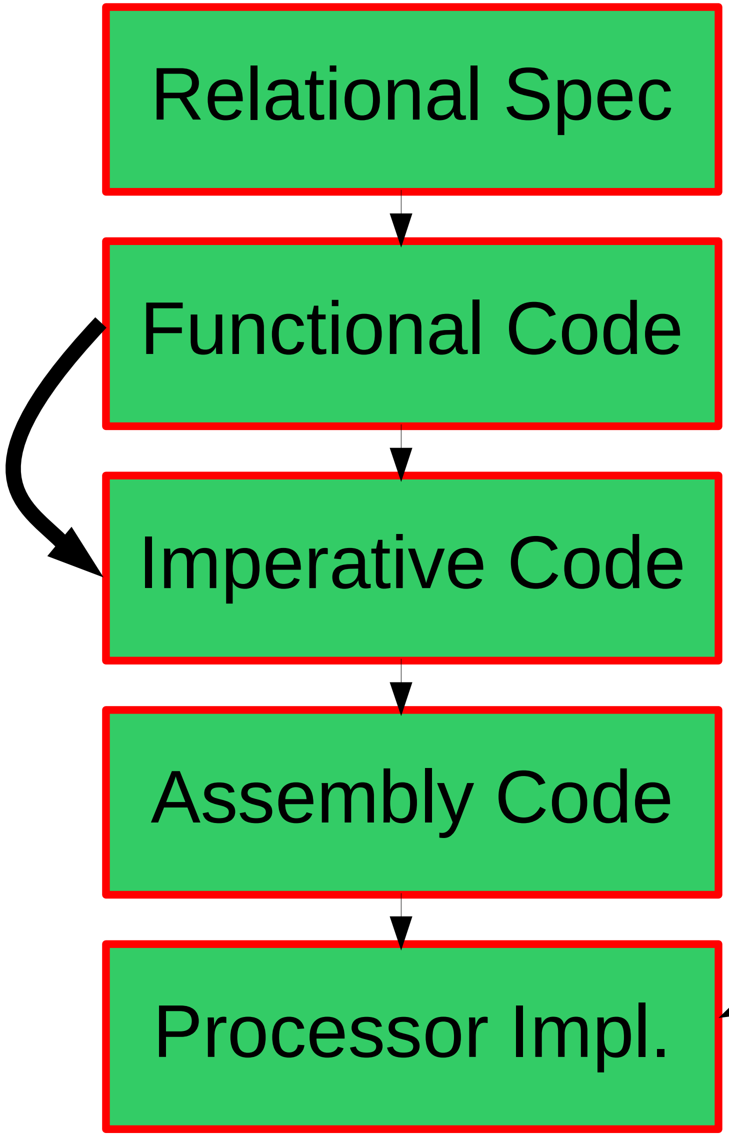
1. Domain of **abstract models** for foreign data types
2. Hoare-style **precondition and postcondition for every foreign function**, using abstract predicates for foreign data types

Verified compiler justifies linking with arbitrary implementations of those types and operations in other Bedrock languages.

Ongoing Related Work

Fiat Cryptography

Generates low-level ECC code automatically, with proof. Adopted by Google's **BoringSSL** library, thus transitively for TLS in **Chrome** and **Android**.



Implementations of **RISC-V** open instruction set, proving against **official formal semantics**

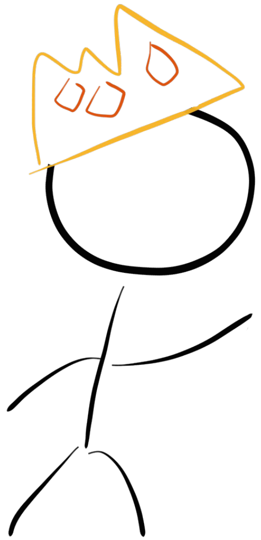
Kami

Separate **functionality** and **performance**

Functionality as macros desugaring to a common logic

Performance via proved optimization rules

http://plv.csail.mit.edu/fiat/



Work supported by:

I2O: HACMS and BRASS programs

National Science Foundation