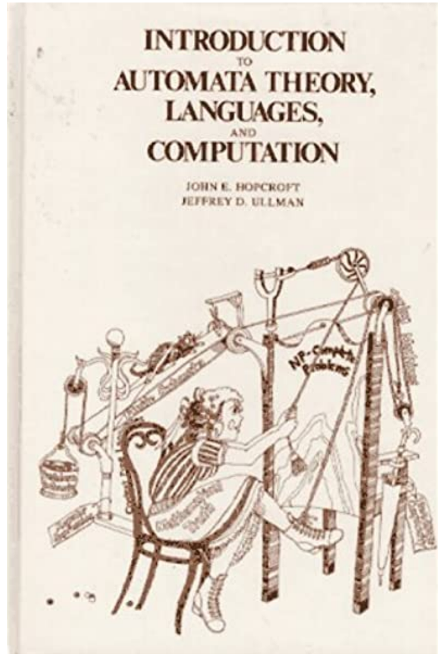
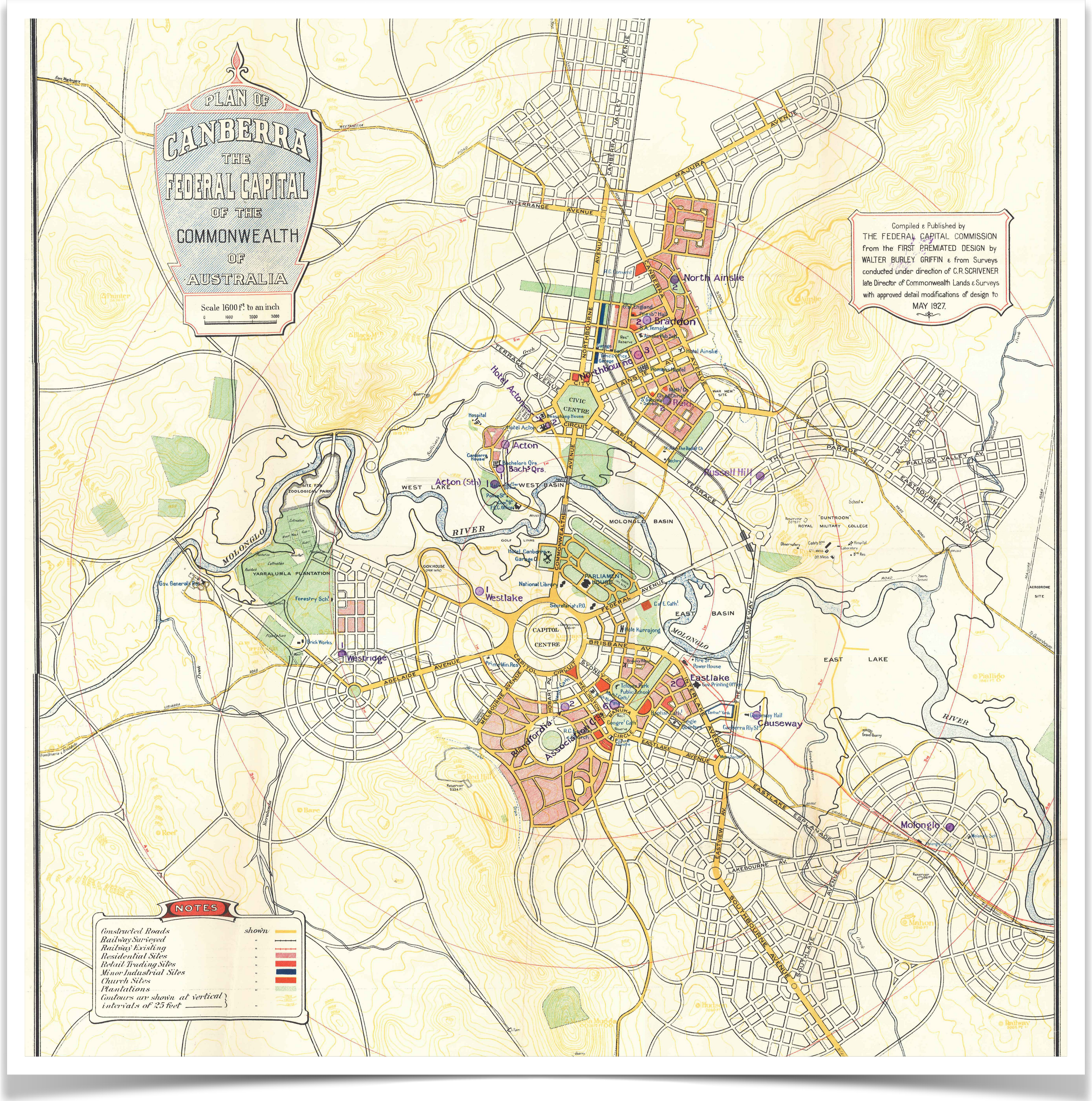
CakeML: Verified Computation/ Compilation Stories

\+ 

=

Michael Norrish, School of Computing, ANU
23 May 2023

Itinerary



Canberra Plan, 1927. Archives of the ACT Government *via* flickr.com

- Interactive theorem-proving
- CakeML
- Rendering super-standard formal language theory with maximum cleanness—puzzle included.

“Solving the world’s problems with theorem-proving.”

–me, in moments of delusion

Interactive Theorem-Proving

Interactive theorem-proving is core to 99% of what I do.

Example systems: ACL2, Coq, HOL4, HOL Light, Isabelle and PVS

Proof in Action

```
> g ‘∀n a b c. 2 < n ⇒ a ** n + b ** n ≠ c ** n’;
val it =
   Proof manager status: 1 proof.
1. Incomplete goalstack:
     Initial goal:

     ∀n a b c. 2 < n ⇒ a ** n + b ** n ≠ c ** n

:
   proofs
> e Induct;
OK..
2 subgoals:
val it =

∀a b c. 2 < SUC n ⇒ a ** SUC n + b ** SUC n ≠ c ** SUC n
------------------------------------
  ∀a b c. 2 < n ⇒ a ** n + b ** n ≠ c ** n


∀a b c. 2 < 0 ⇒ a ** 0 + b ** 0 ≠ c ** 0
```

- Human guided proof
- Posit the goal; provide the proof
- Machine helps:
  - by checking validity of steps
  - with some automatic tools
- Thus: “*proof assistant*” term

HOL4

- Under development since the mid 1980s
- Engineered in the "LCF tradition": a small kernel minimises the TCB
- Similar logic to Isabelle/HOL and HOL Light
- Simpler logic than (e.g.) Coq's.

CakeML

What?

CakeML

What?

1. A programming language in the style of Standard ML and OCaml.

CakeML

What?

strict evaluation, stateful

1. A programming language in the style of Standard ML and OCaml.

CakeML

What?

strict evaluation, stateful

1. A programming language in the style of Standard ML and OCaml.
2. An ***ecosystem*** of proofs and verification tools

CakeML

What?

strict evaluation, stateful

1. A programming language in the style of Standard ML and OCaml.
2. An ***ecosystem*** of proofs and verification tools
3. A ***verified, end-to-end compiler***

What Do These Words Mean? (I)

“A programming language in the style of SML and OCaml…”

(functional, has pattern-matching, nice recursion, nice datatypes)

But:

Strict: arguments are evaluated before being passed to functions. (*Unlike* Haskell.)

Stateful: CakeML supports variables that you can update by *assigning* to them. (Again, *unlike* Haskell.)

What Do These Words Mean? (II)

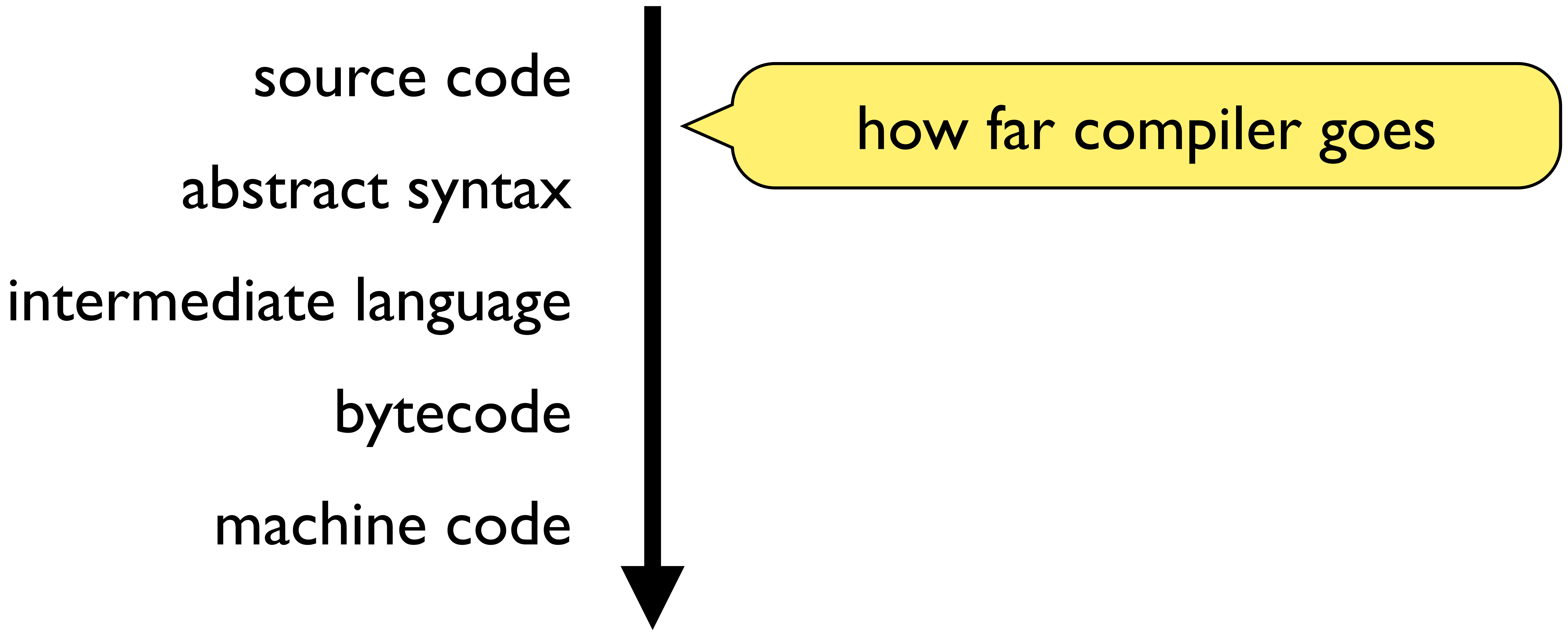
Verified: CakeML has proofs that guarantee it will behave correctly

End-to-end: The proofs are about "all of it":

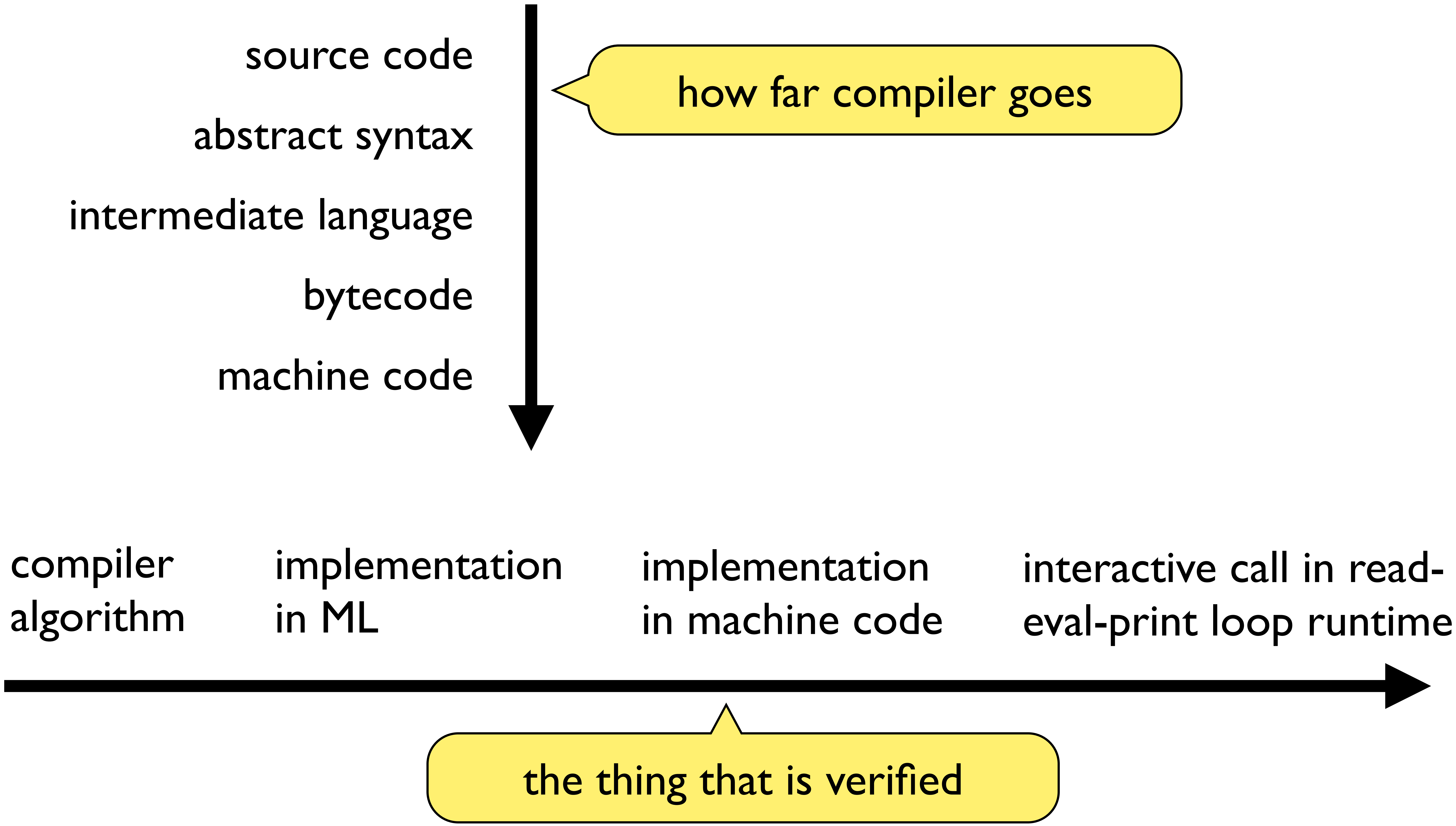
- From: reading in the input file (a stream of characters)
- To: actual machine code for the CPU (x86, ARM etc)

Dimensions of Compiler Verification

Dimensions of Compiler Verification



Dimensions of Compiler Verification



Dimensions of Compiler Verification

source code

abstract syntax

intermediate language

bytecode

machine code

how far compiler goes

Our verification covers the full spectrum of **both** dimensions.

compiler algorithm

implementation in ML

implementation in machine code

interactive call in read-eval-print loop runtime

the thing that is verified

Goal: End-to-end

$$\vdash P(\texttt{val x = ...})$$

 $[\![\cdot]\!]$

Ingredients

$$[\![\cdot]\!] : \texttt{string} \to v$$

Verified Compilation

$$[\![\cdot]\!] : \texttt{int list} \to s \to s$$

Bootstrapping

parsing

type inference

compilation

function in logic (compiler)

Bootstrapping

- parsing
- type inference
- compilation

function in logic (compiler)

↓ Proof-producing synthesis (ICFP'12)

CakeML program (compiler-ML)

⊢ compiler-ML implements compiler

Bootstrapping

parsing

type inference

compilation

function in logic (compiler)

Proof-producing synthesis (ICFP'12)

CakeML program (compiler-ML)

$\vdash$ compiler-ML implements compiler

by evaluation in the logic

$\vdash$ compiler (compiler-ML) = compiler-x86

Bootstrapping

parsing

type inference

compilation

function in logic (compiler)

Proof-producing
synthesis (ICFP'12)

CakeML program (compiler-ML)

$\vdash$ compiler-ML implements compiler

by evaluation
in the logic

$\vdash$ compiler (compiler-ML) = compiler-x86

by compiler
correctness

$\vdash \forall c.$ (compiler c) implements c

Bootstrapping

- parsing
- type inference
- compilation

function in logic (compiler)

↓ Proof-producing synthesis (ICFP'12)

CakeML program (compiler-ML)

$\vdash$ compiler-ML implements compiler

by evaluation in the logic

$\vdash$ compiler (compiler-ML) = compiler-x86

by compiler correctness

$\vdash \forall c.$ (compiler c) implements c

Theorem: $\vdash$ compiler-x86 implements compiler

What Do These Words Mean?(III)

Photo by Kohei314, *via* `flickr.com`

- **Ecosystem:** Not only is the CakeML compiler "verified" (as before), but we also have a variety of methods for proving (other) CakeML programs correct.
- When your Haskell program misbehaves, who/what do you blame?
  - Your program (you)? GHC? The OS? The hardware? Cosmic rays?

Hmm, Can This Possibly Be True?

- Scepticism is fair.
- Must ask:
  - *"What are your assumptions?"*
- (Correct) Proofs are only as good as the assumptions behind them!

Assumption 1: our logic is sound.

- Any attempt to prove this would in turn depend on knowing that the logic being used to prove this was sound,
    - which would require another proof of soundness, carried out in yet another logical system…
        - this makes for an infinite regress…

- See also: Gödel.

from Wikipedia

Assumption 2: Our *implementation* of the logic is correct.

- HOL4 is *not* verified…
- The language it's written in hasn't been verified either
- *But*:
  - The *Trusted Code Base* in HOL4 is small by design
  - It's been eyeballed for many decades by experts
  - It can export proof logs for independent checking

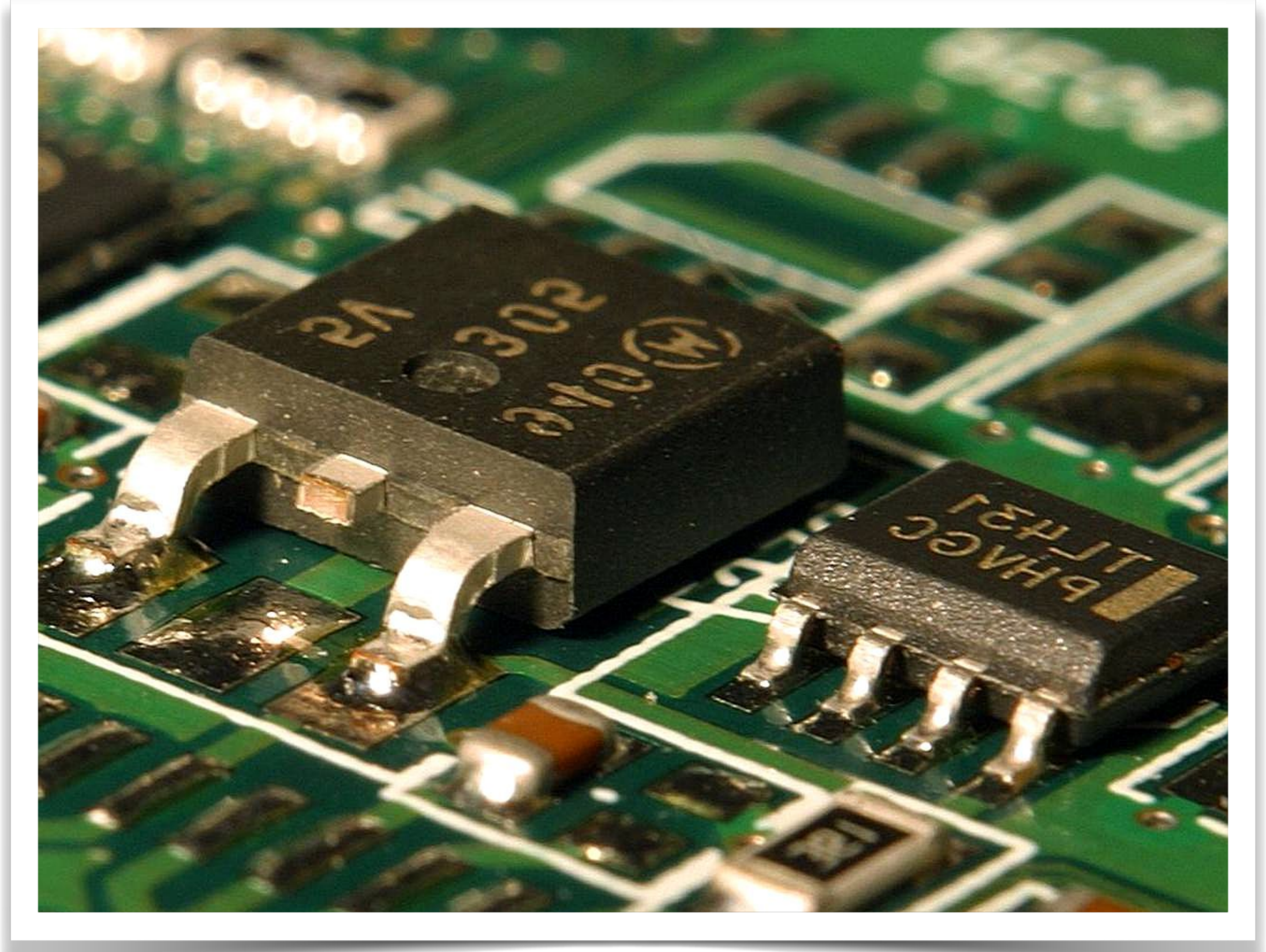
Assumption 3: Our correctness theorem says what we think it says

- Complicated logical statements are easy to misinterpret
- Luckily, our correctness statement is not *so* bad:

```
⊢ config_ok cc mc ⇒
   case compile cc prelude input of
    Success (bytes,ffi_limit) ⇒
     ∃ behaviours.
      cakeml_semantics ffi prelude input =
       Execute behaviours ∧
      ∀ ms.
       code_installed (bytes,cc,ffi,ffi_limit,mc,ms) ⇒
        machine_sem mc ffi ms ⊆
        extend_with_resource_limit behaviours
   | Failure ParseError ⇒
     cakeml_semantics ffi prelude input = CannotParse
   | Failure TypeError ⇒
     cakeml_semantics ffi prelude input = IllTyped
   | Failure CompileError ⇒ true
```

Assumption 4: Our logical model of the real world is accurate

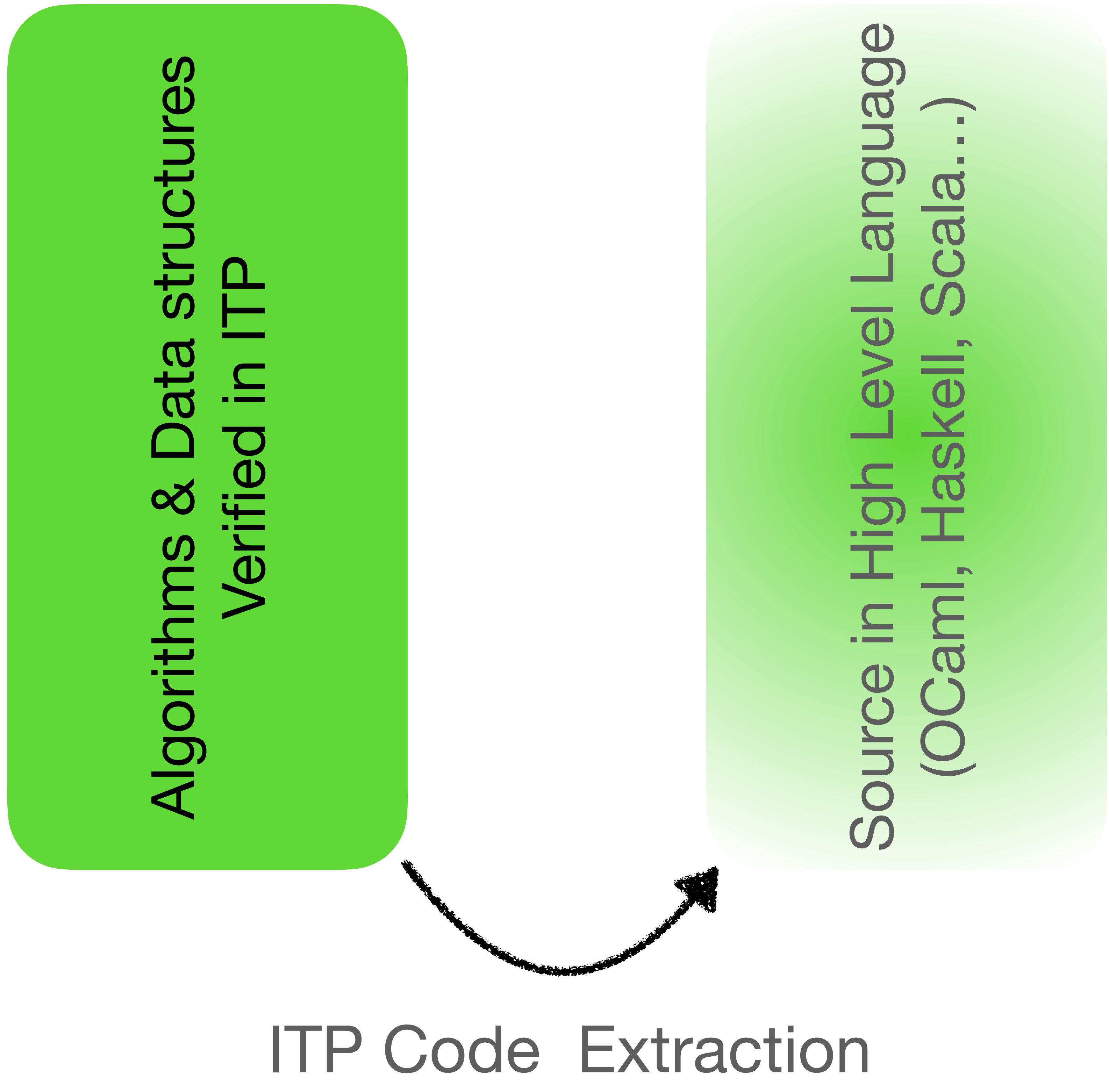
- We assume that x86 (ARM, RISCV,…) chips really do behave according to the logical spec we have for them.
- We assume that the OS implements its various system calls in accordance with our spec.



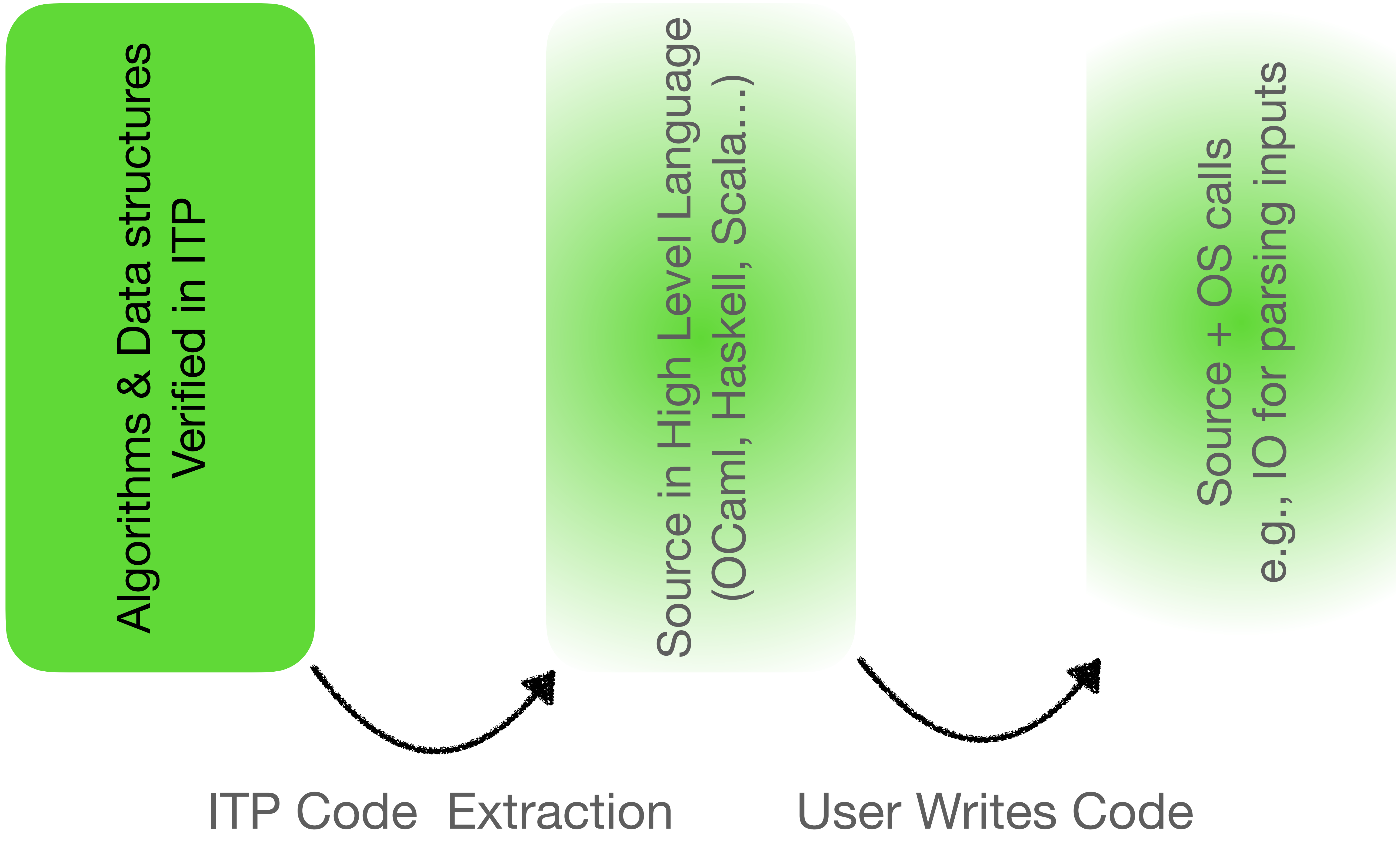
State-of-the-Art Assurance

Algorithms & Data structures
Verified in ITP

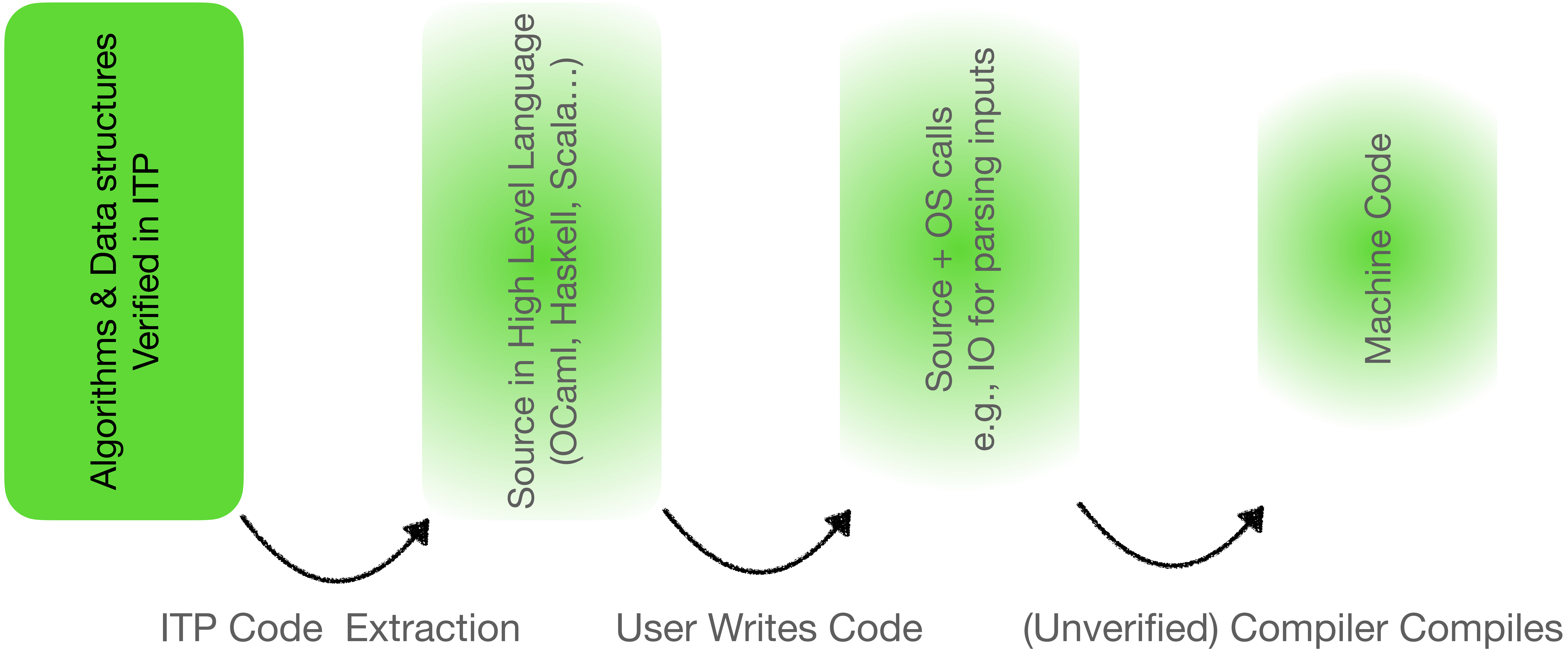
State-of-the-Art Assurance



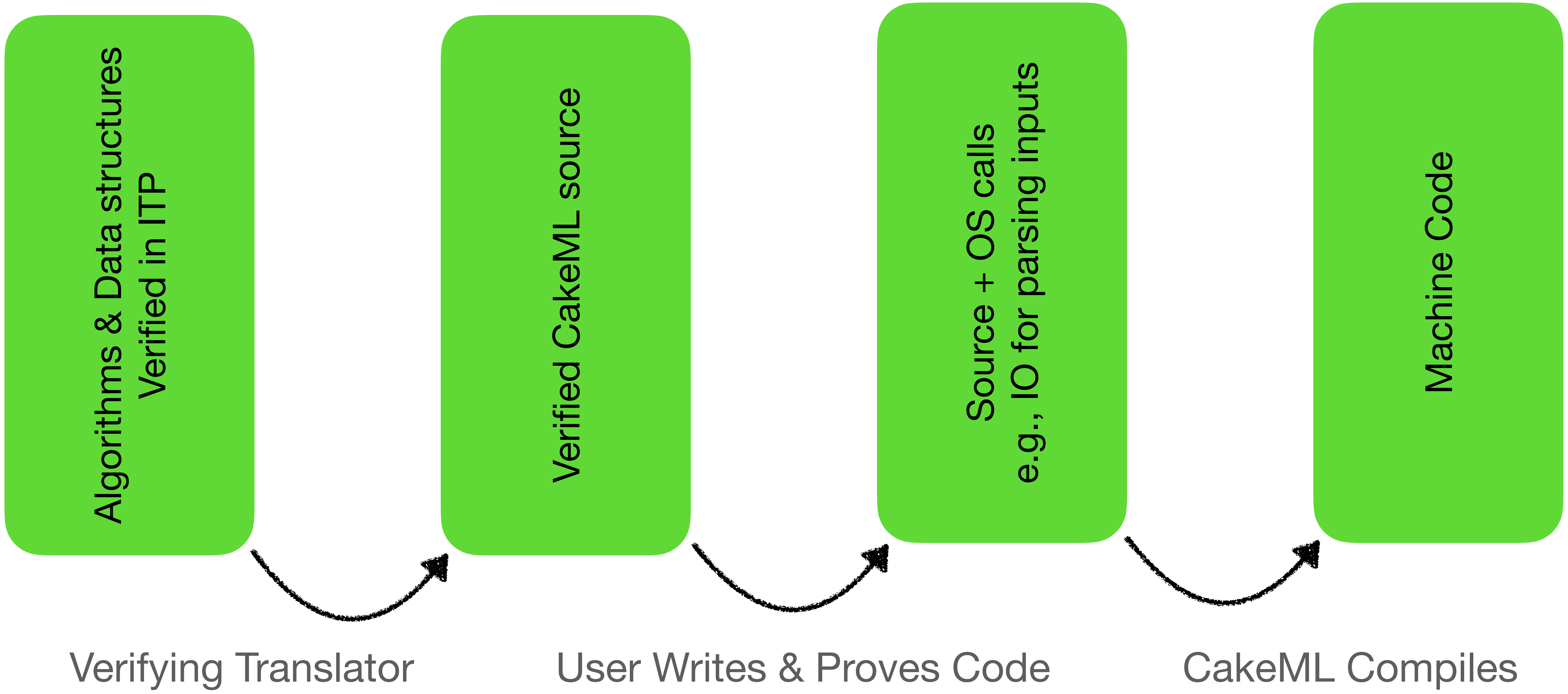
State-of-the-Art Assurance



State-of-the-Art Assurance



CakeML Assurance



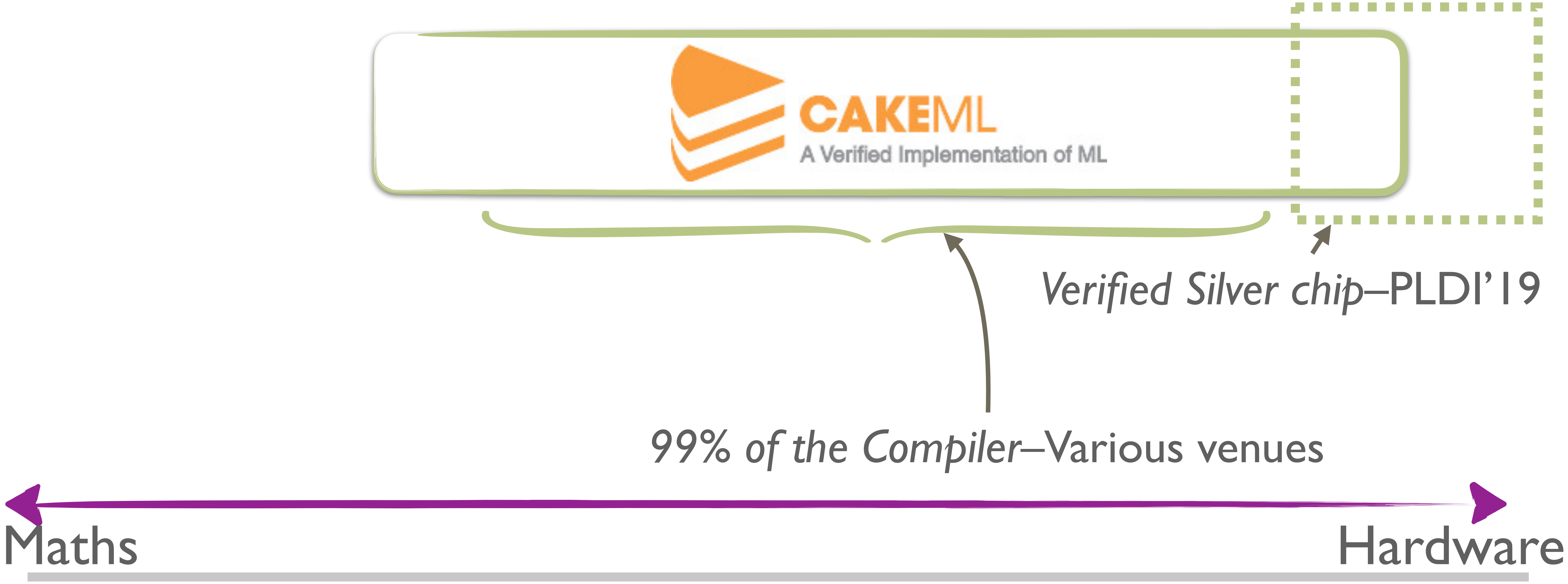
CakeML Projects at Many Levels

CAKEML
A Verified Implementation of ML

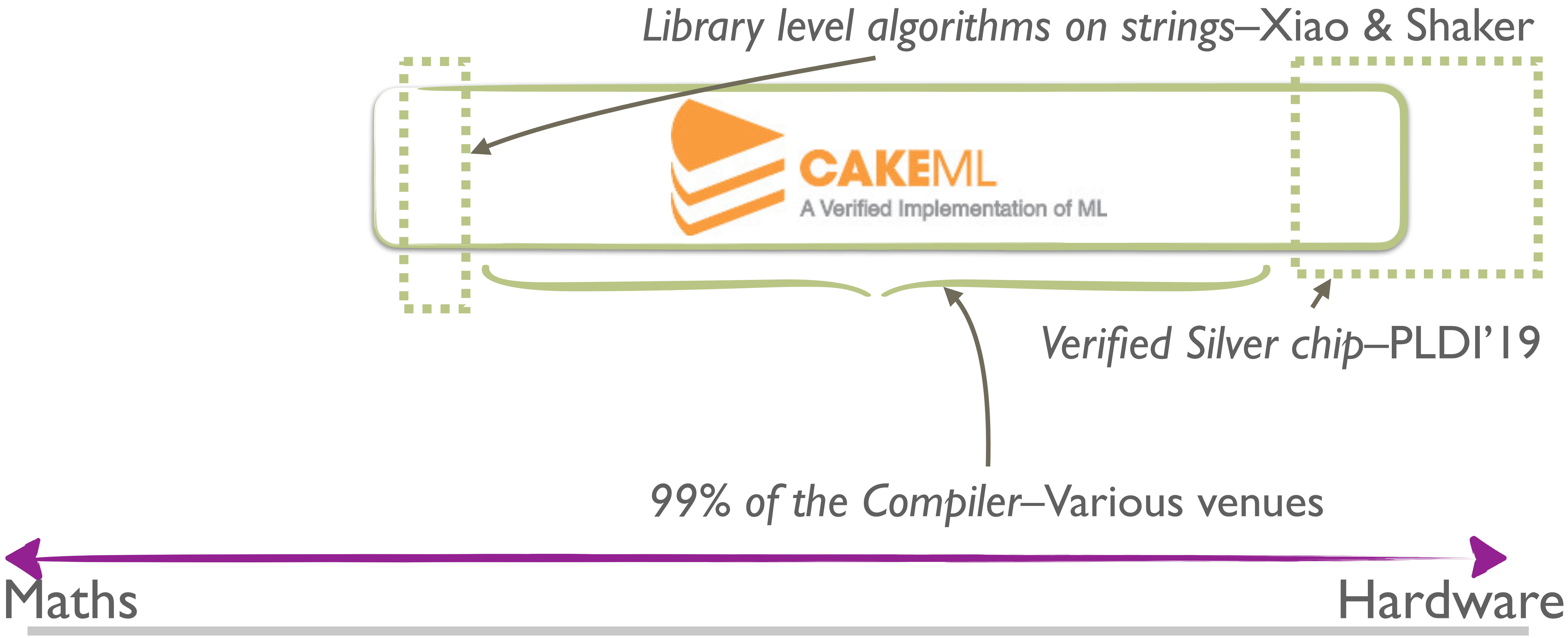
Maths ⟷ Hardware

CakeML Projects at Many Levels

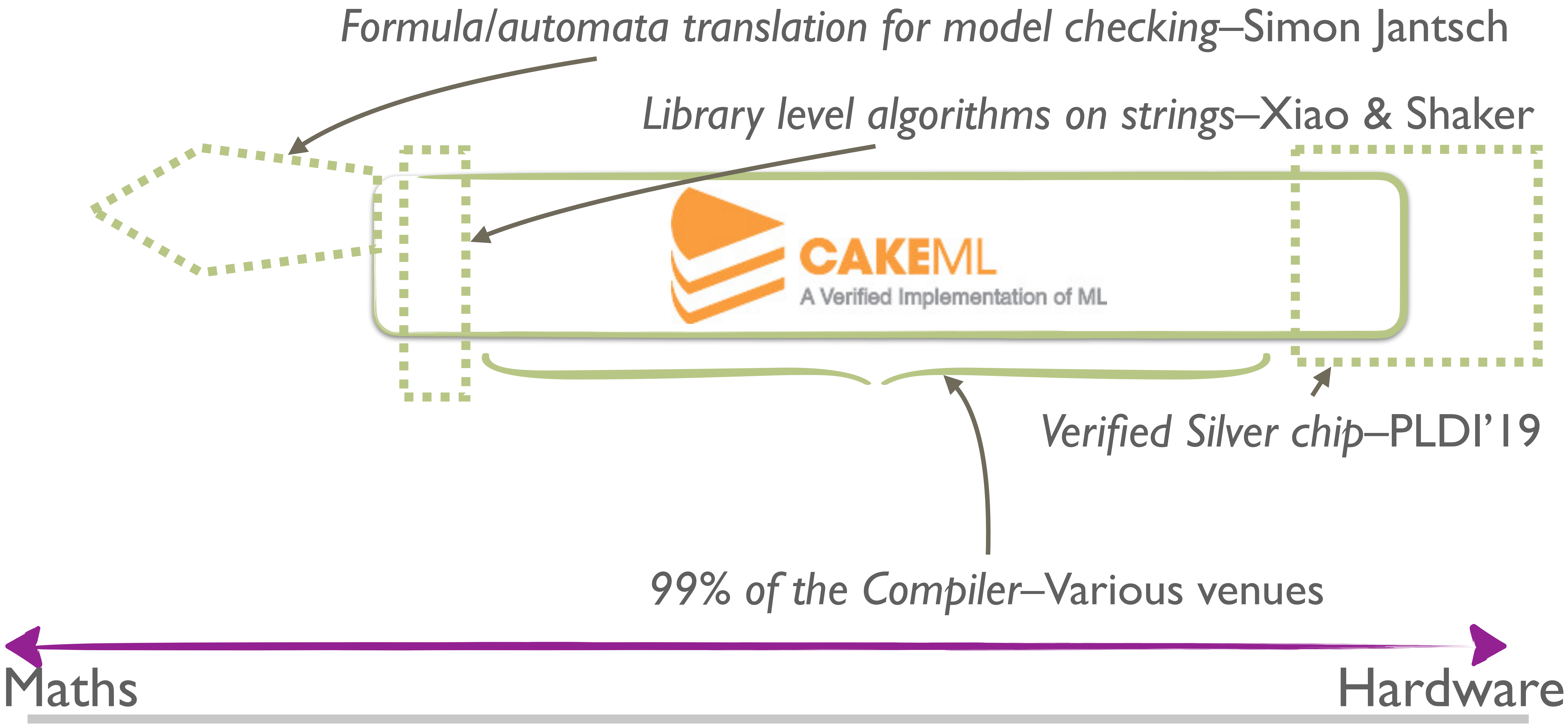
CakeML Projects at Many Levels



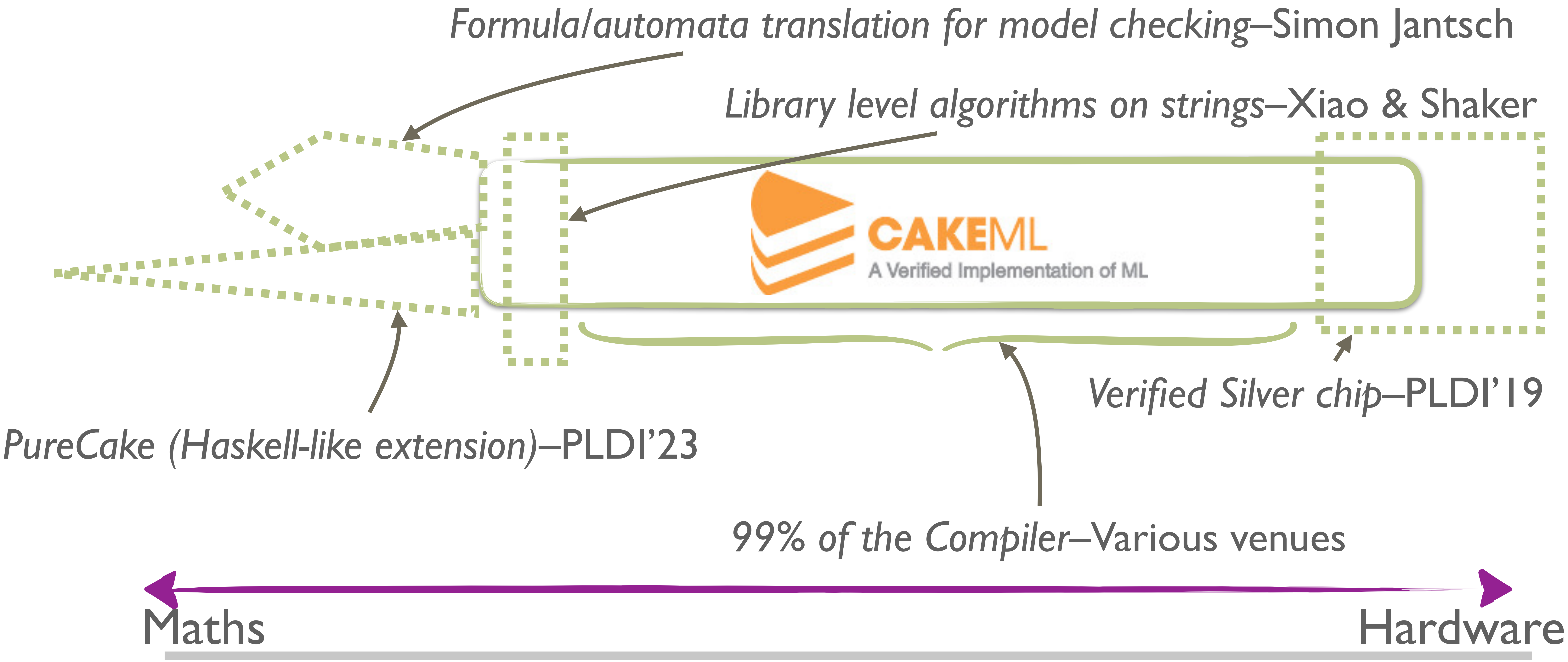
CakeML Projects at Many Levels



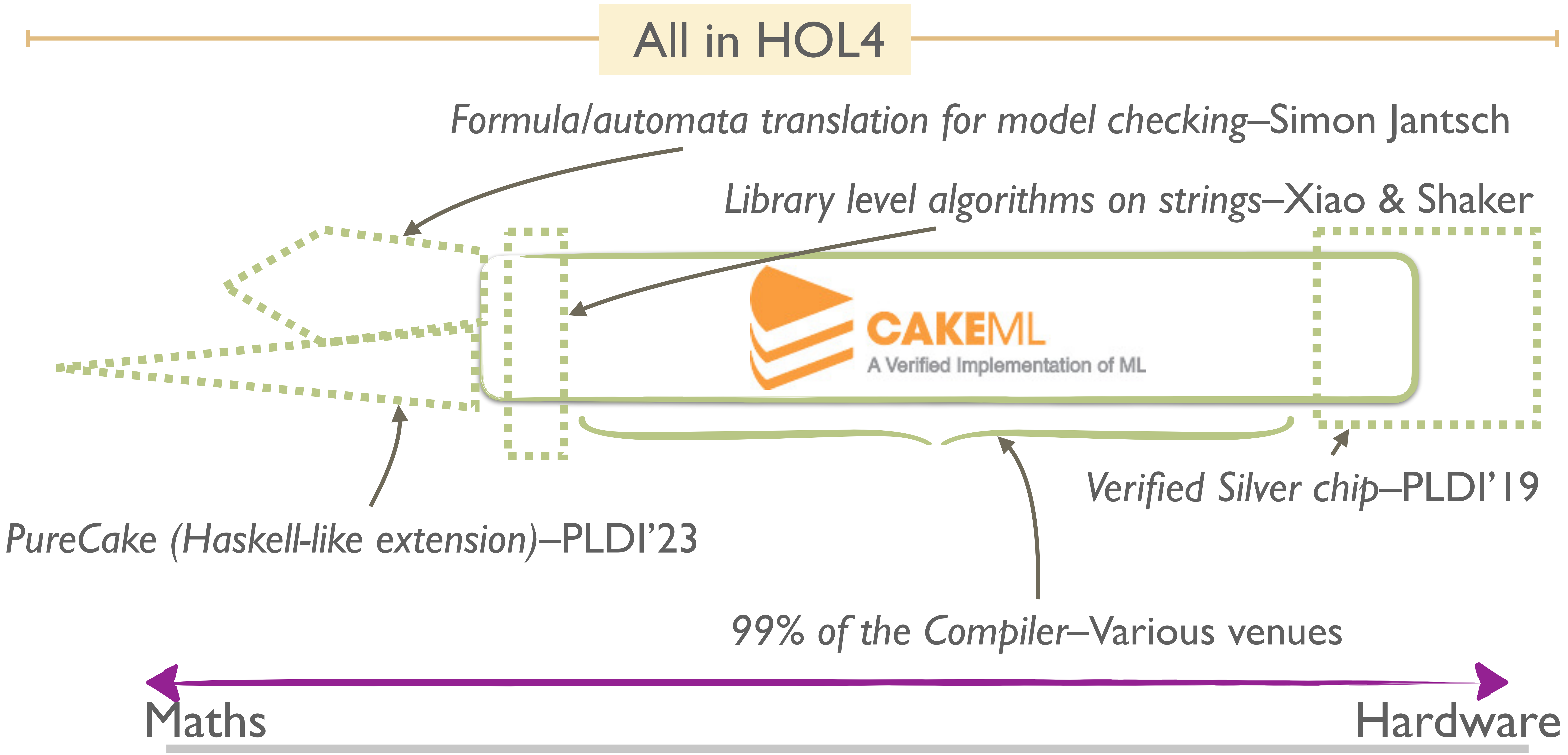
CakeML Projects at Many Levels



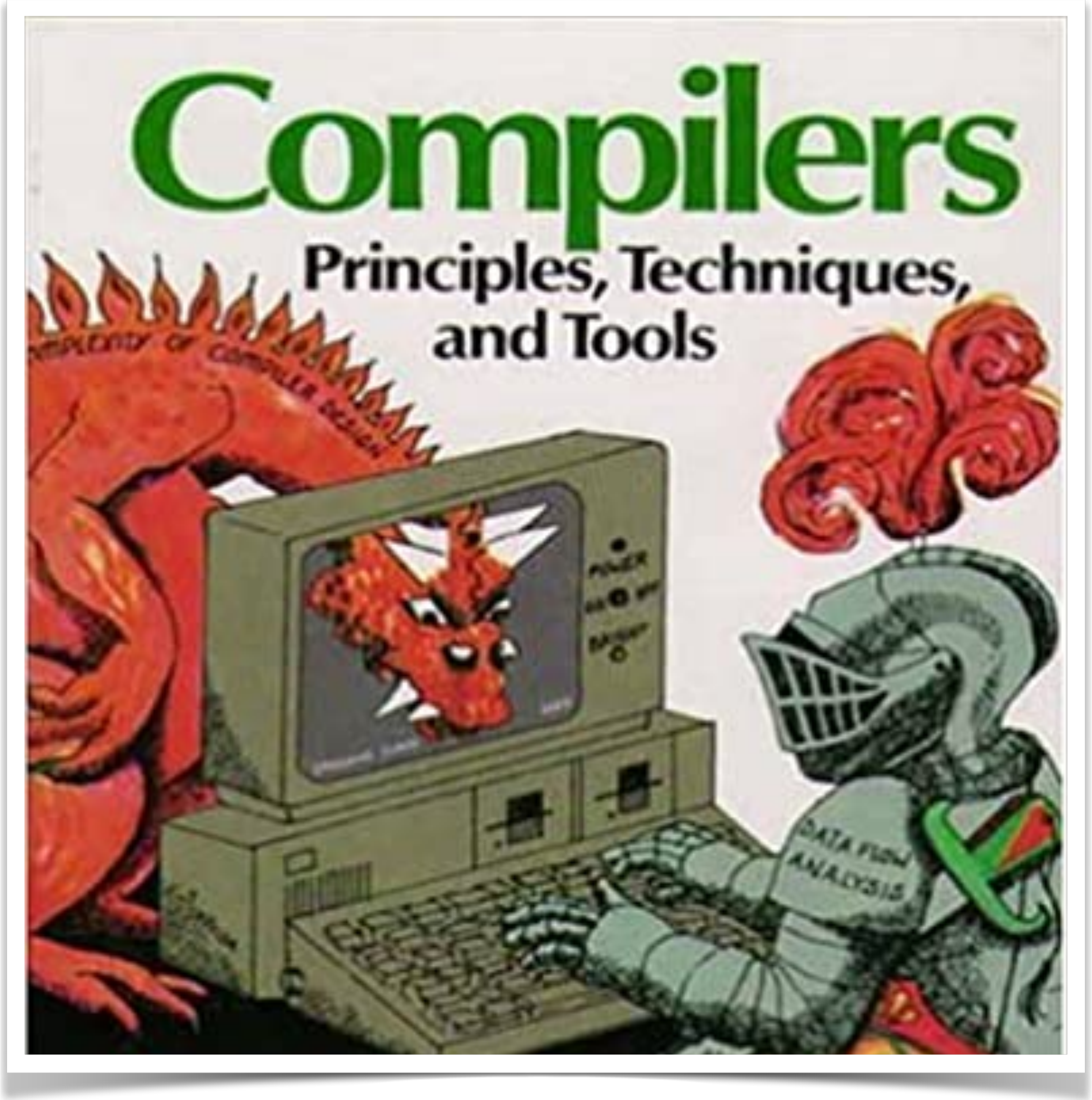
CakeML Projects at Many Levels



CakeML Projects at Many Levels

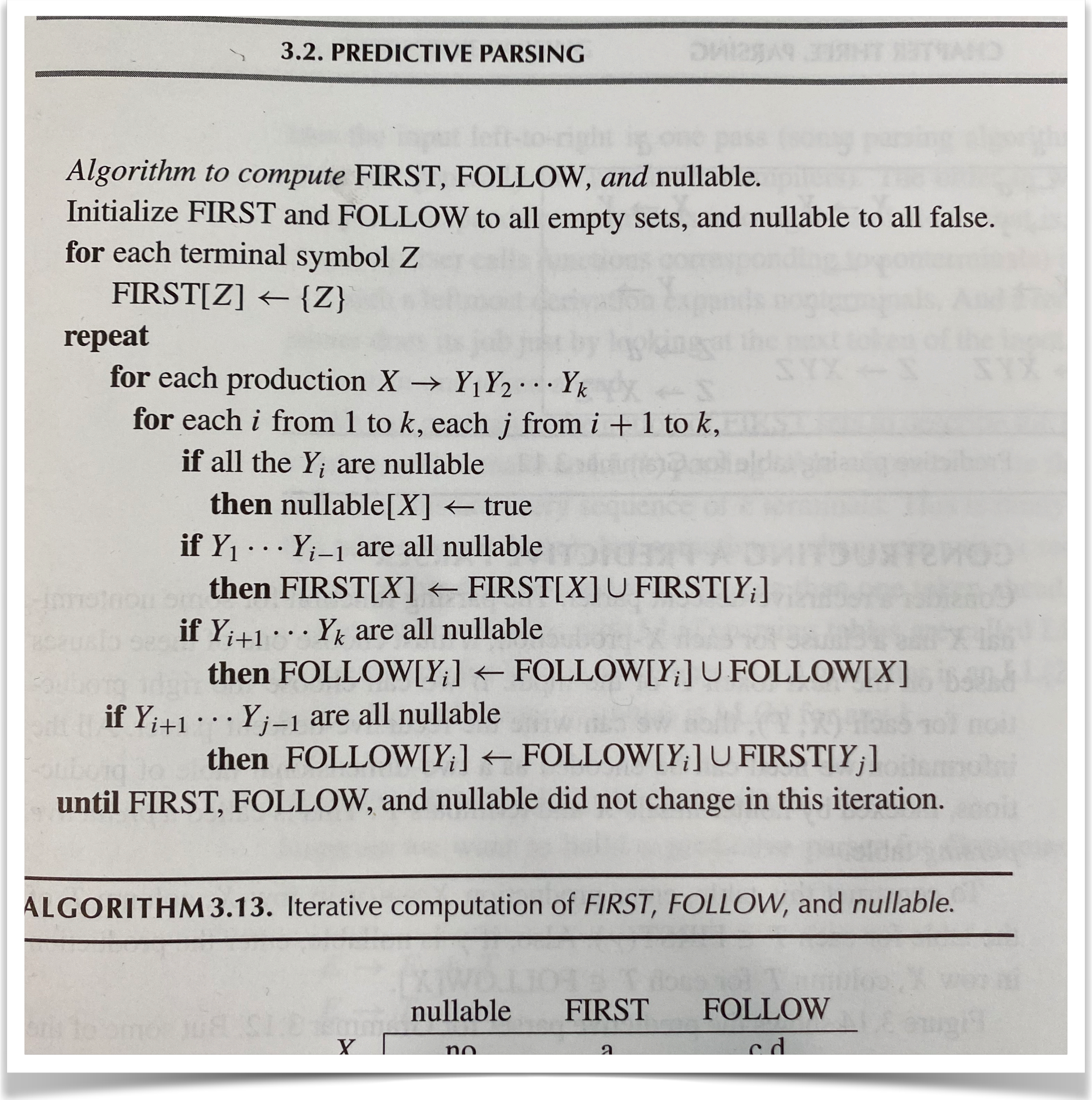


Parsing: an Application for the Ecosystem



- A verified, general-purpose, parser-construction tool is very appealing
- Applications (not just compilers) often need to parse input formats.
- “Verify Once, Run Ever-after”
  - Strong work in this area does already exist

Parsing: an Application for the Ecosystem


3.2. PREDICTIVE PARSING

Algorithm to compute FIRST, FOLLOW, *and* nullable.
Initialize FIRST and FOLLOW to all empty sets, and nullable to all false.
for each terminal symbol Z
    FIRST$[Z] \leftarrow \{Z\}$
repeat
    for each production $X \rightarrow Y_1 Y_2 \cdots Y_k$
        for each i from 1 to k, each j from $i+1$ to k,
            if all the Y_i are nullable
                then nullable$[X] \leftarrow$ true
            if $Y_1 \cdots Y_{i-1}$ are all nullable
                then FIRST$[X] \leftarrow$ FIRST$[X] \cup$ FIRST$[Y_i]$
            if $Y_{i+1} \cdots Y_k$ are all nullable
                then FOLLOW$[Y_i] \leftarrow$ FOLLOW$[Y_i] \cup$ FOLLOW$[X]$
        if $Y_{i+1} \cdots Y_{j-1}$ are all nullable
                then FOLLOW$[Y_i] \leftarrow$ FOLLOW$[Y_i] \cup$ FIRST$[Y_j]$
until FIRST, FOLLOW, and nullable did not change in this iteration.

ALGORITHM 3.13. Iterative computation of *FIRST, FOLLOW,* and *nullable.*

	nullable	FIRST	FOLLOW
X	no	a	c d

from: Appel, *Modern Compiler Implementation in ML*

- CakeML's existing parser is a custom-built PEG
- Its verification was just as "custom" (*i.e.*, tedious)
- General tools need general treatments of things like *first* and *follow* sets

Grammars, Classically

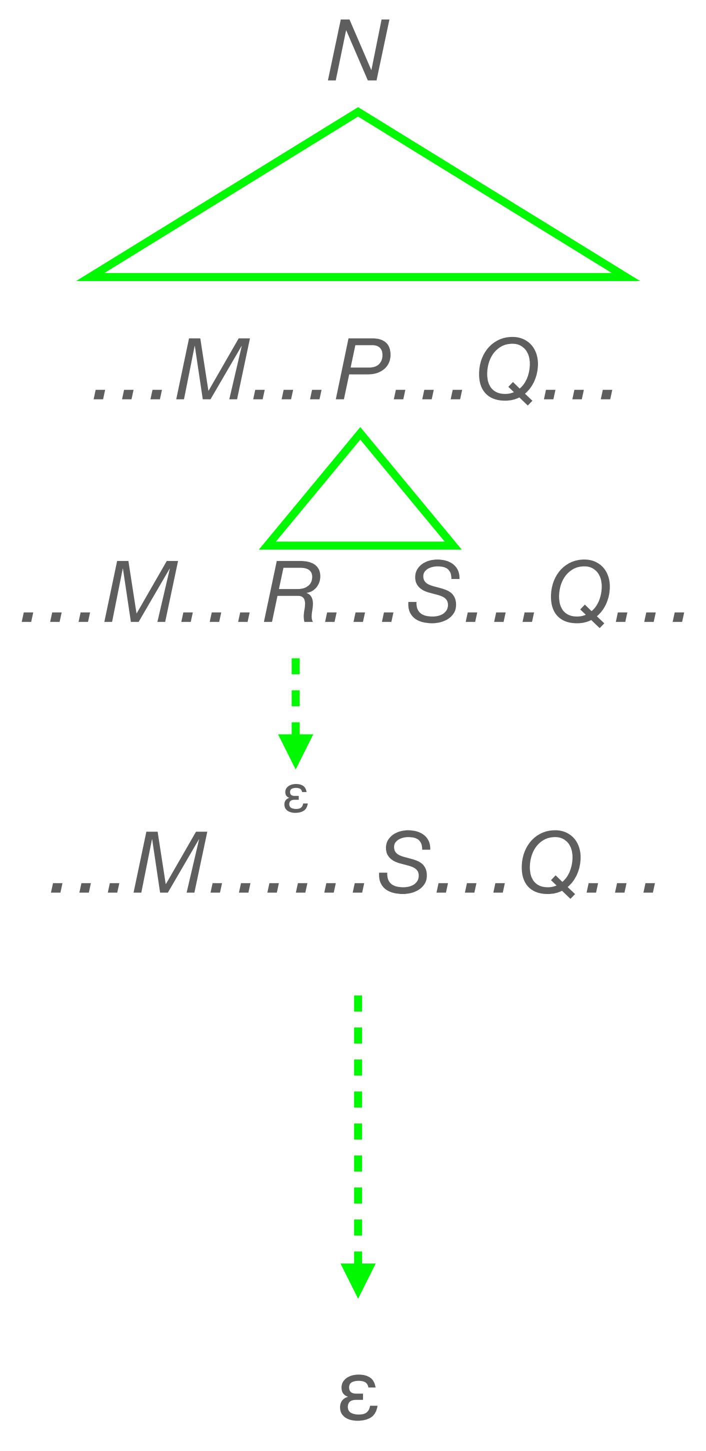
A grammar is a 4-tuple (G, N, T, S), with

- N a finite set of non-terminal symbols;
- T a finite set of terminal symbols;
- $S \in N$ a distinguished non-terminal (the "start symbol");
- G a finite set of production rules, each of the form: $N \rightarrow (N+T)^*$

"I never met a finite set I didn't want to treat as a list"

–Every interactive theorem-proving person ever

Calculating Nullability

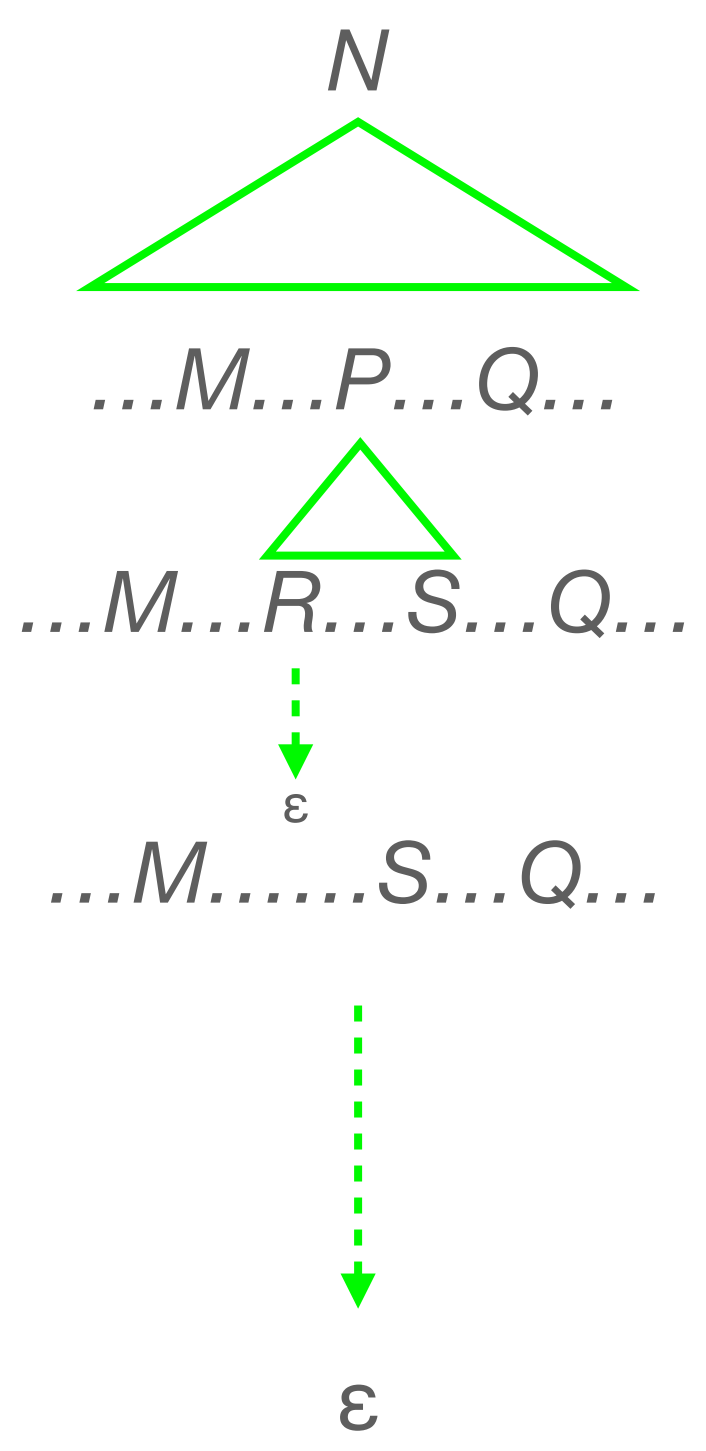


Start with the mathematical definition:

Definition nullable_def:
nullable G sf ⇔ derives G sf [ ]
End

Where *derives* is the reflexive and transitive closure of the relation that expands a non-terminal into a production rule's RHS.

Calculating Nullability



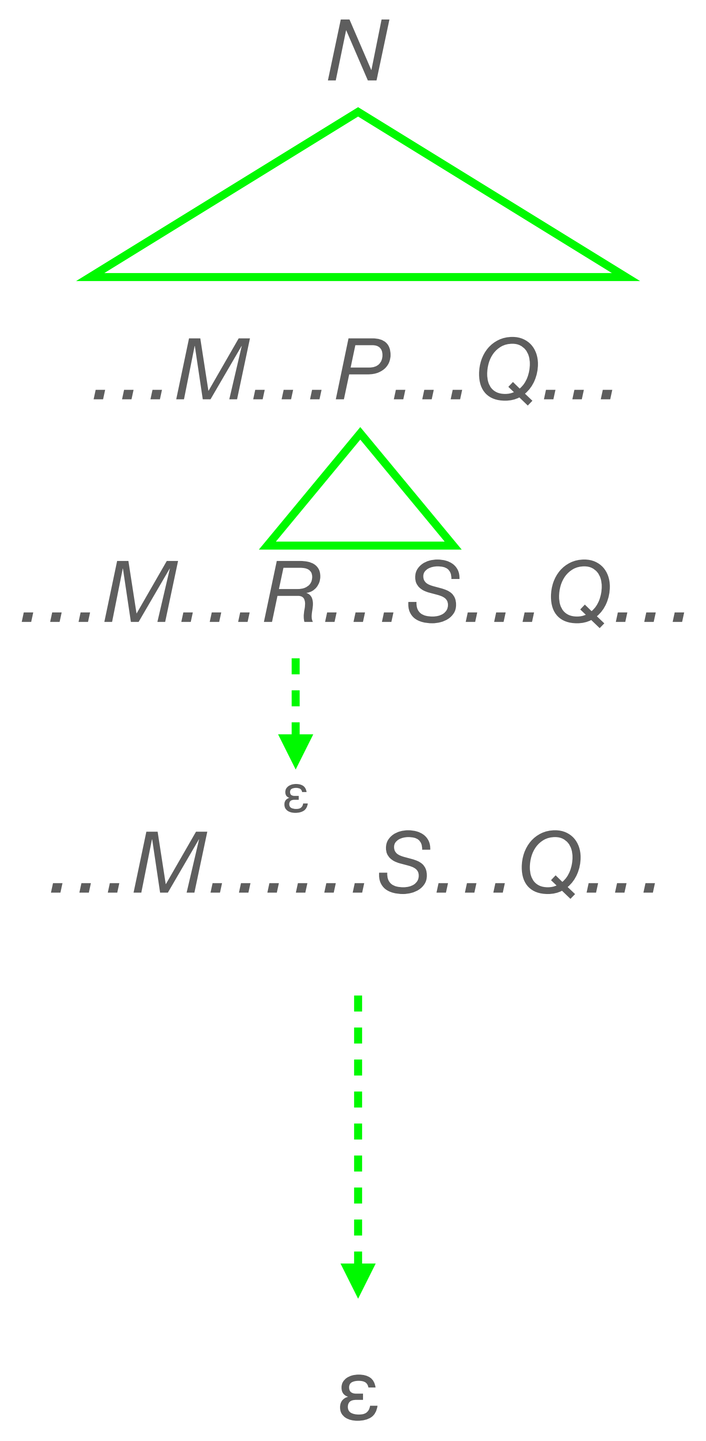
Start with the mathematical definition:

This list is just fine

Definition nullable_def:
 nullable G sf ⇔ derives G sf [ ]
End

Where *derives* is the reflexive and transitive closure of the relation that expands a non-terminal into a production rule’s RHS.

Calculating Nullability



Start with the mathematical definition:

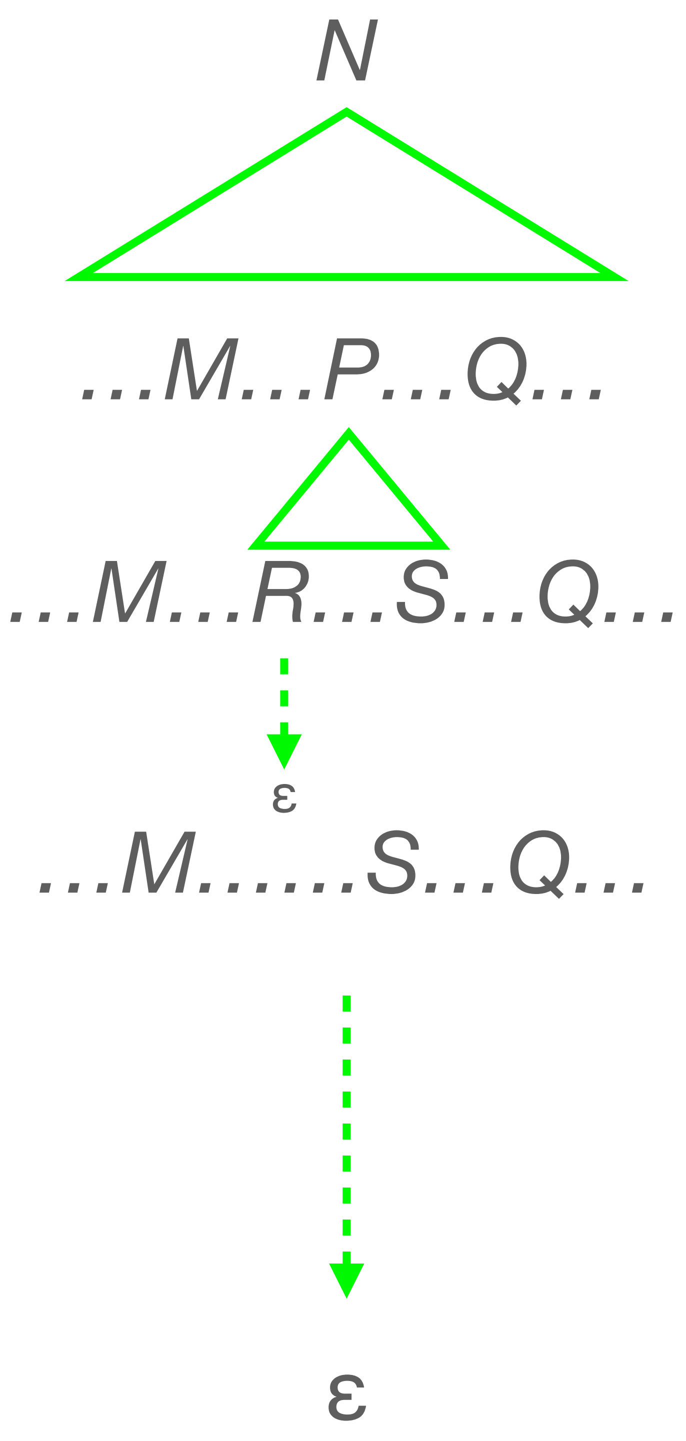
This list is just fine

Definition nullable_def:
nullable G sf ⇔ derives G sf [ ]
End

Where *derives* is the reflexive and transitive closure of the relation that expands a non-terminal into a production rule's RHS.

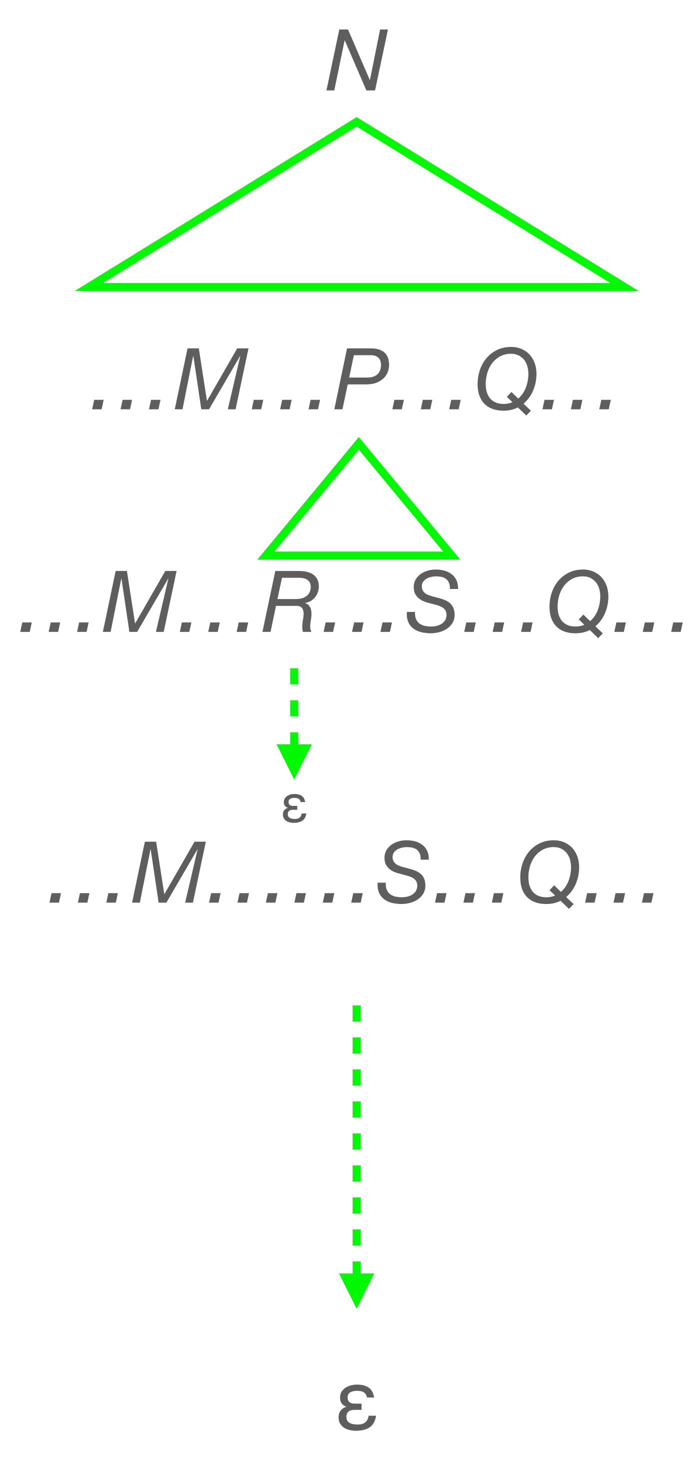
- Terminals are **not** nullable.
- Non-terminals are nullable if any of their RHSs are nullable.
- Critical Realisation: recursive loopbacks can be ignored.

Calculating Nullability



Recursive algorithm:

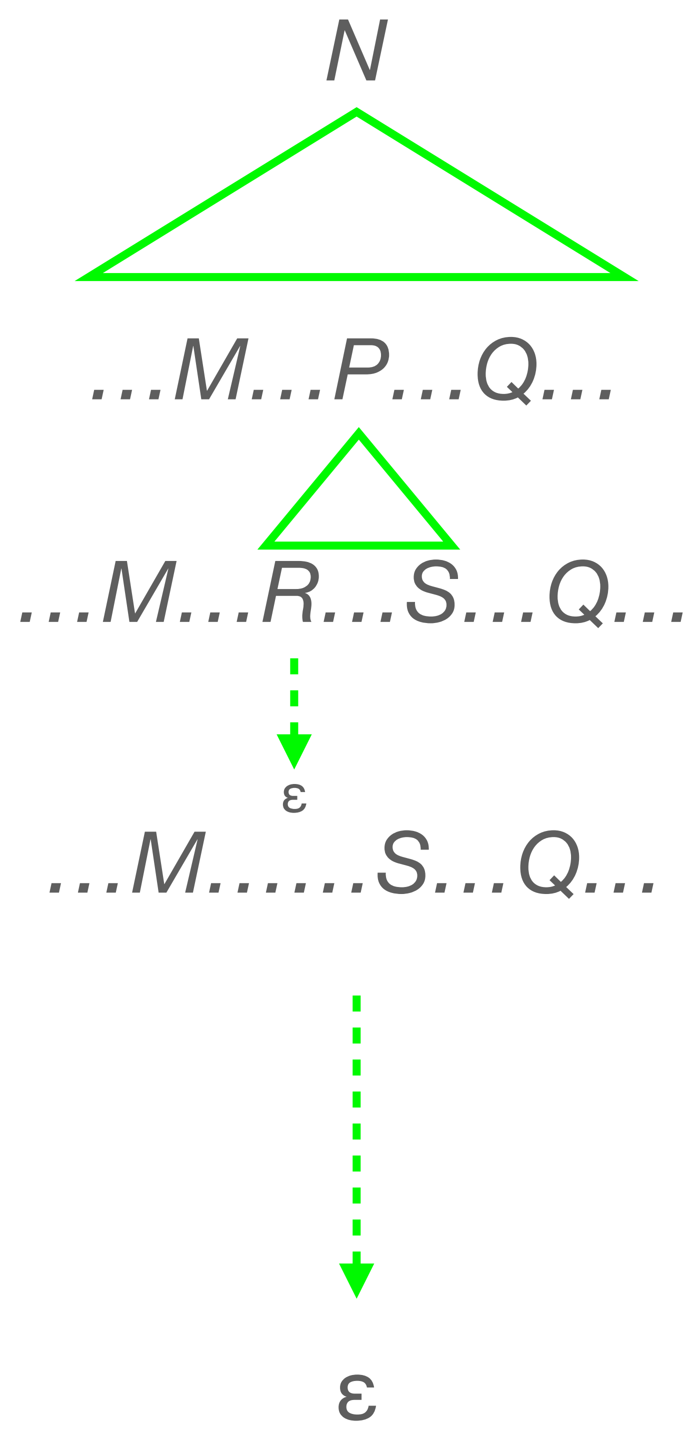
Calculating Nullability



Recursive algorithm:

```
nullableA G s [ ] = T
```

Calculating Nullability

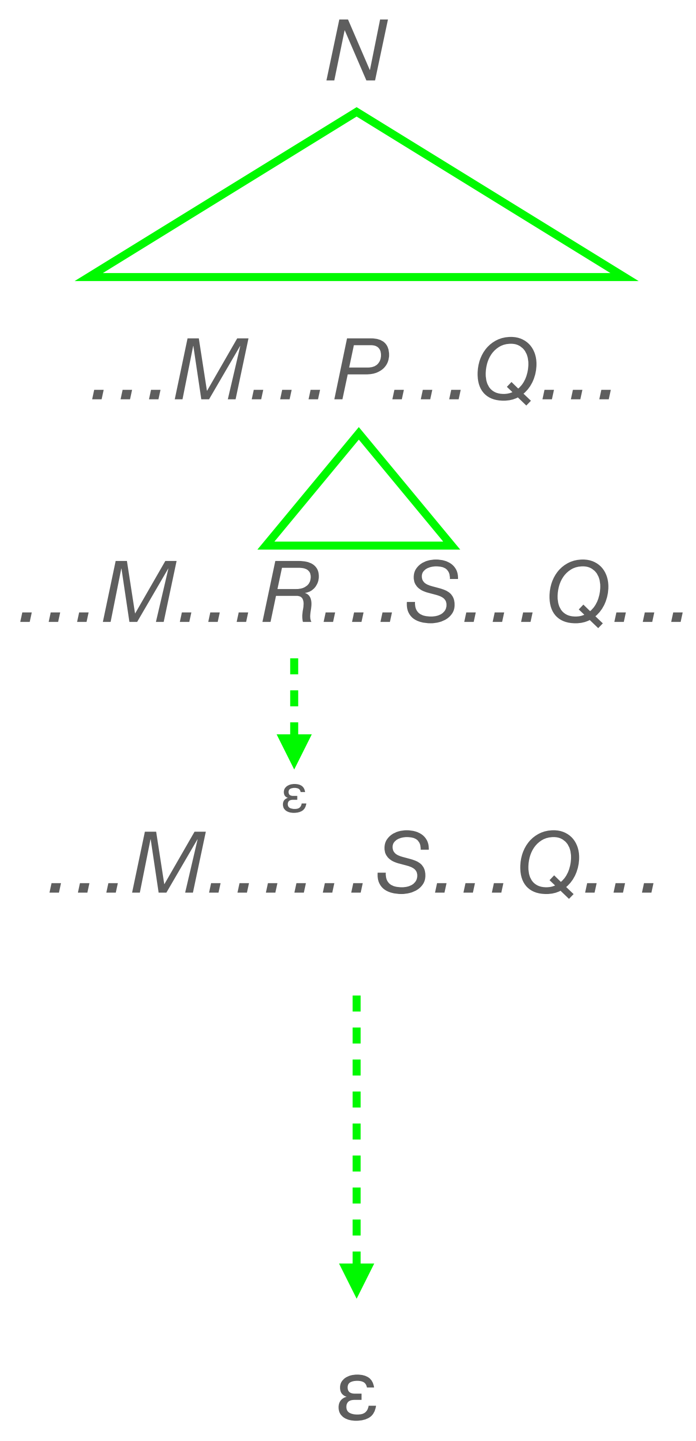


Recursive algorithm:

```
nullableA G s [ ] = T

nullableA G s (TOK _ :: _) = F
```

Calculating Nullability



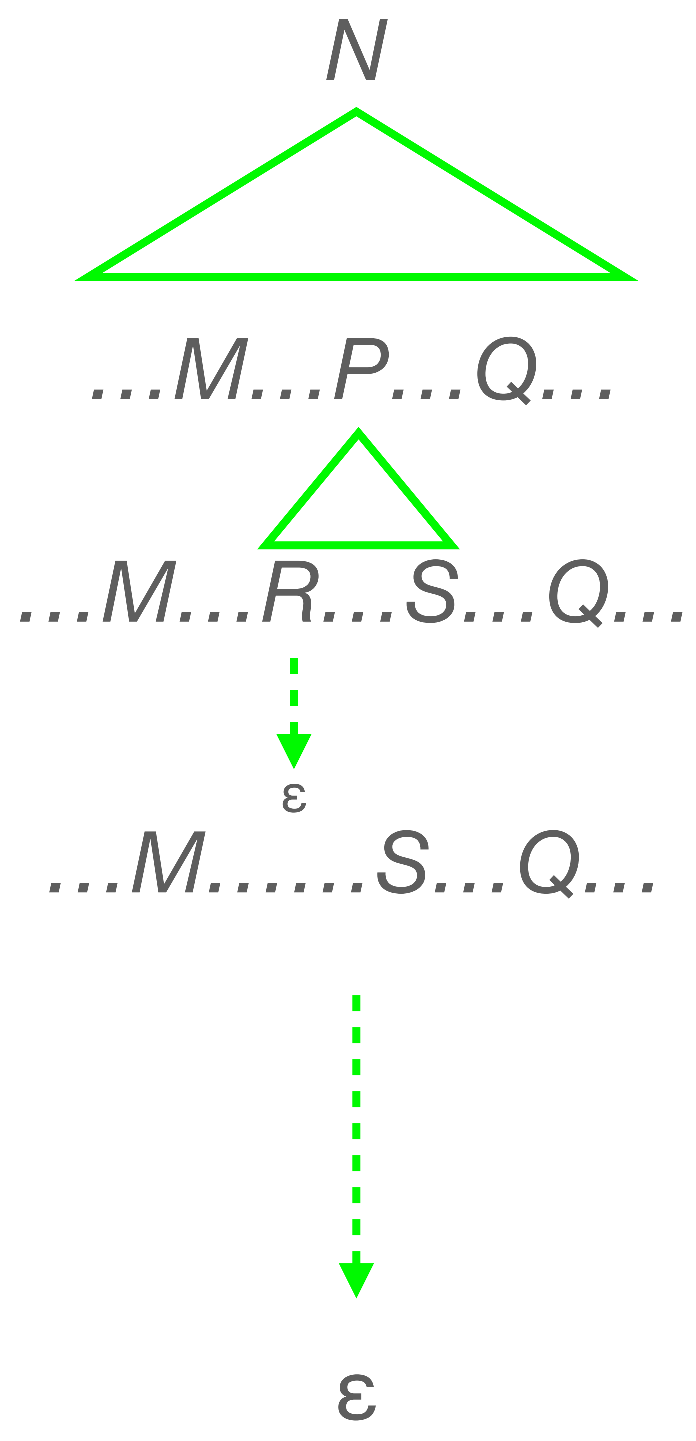
Recursive algorithm:

```
nullableA G s [ ] = T

nullableA G s (TOK _ :: _) = F

nullableA G s (NT n :: rest) =
```

Calculating Nullability



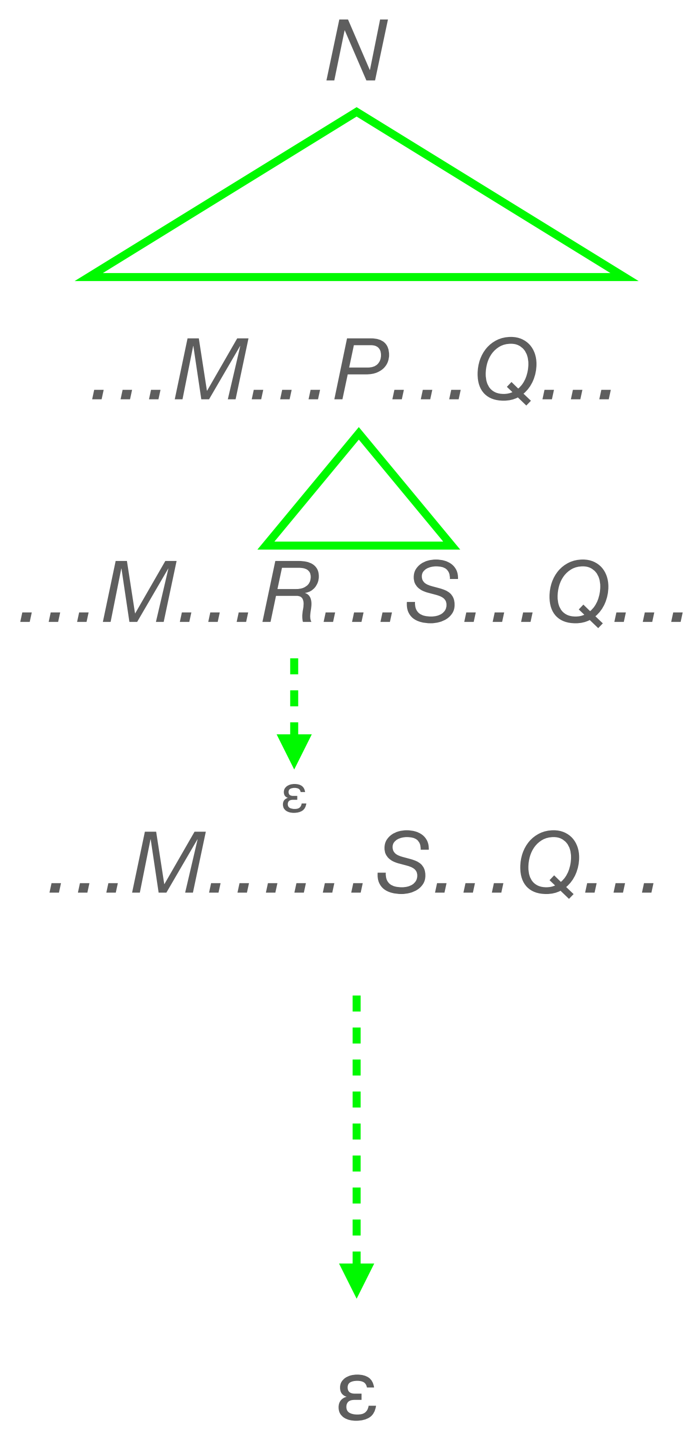
Recursive algorithm:

```
nullableA G s [ ] = T

nullableA G s (TOK _ :: _) = F

nullableA G s (NT n :: rest) =
    nullableA G s rest ∧
```

Calculating Nullability



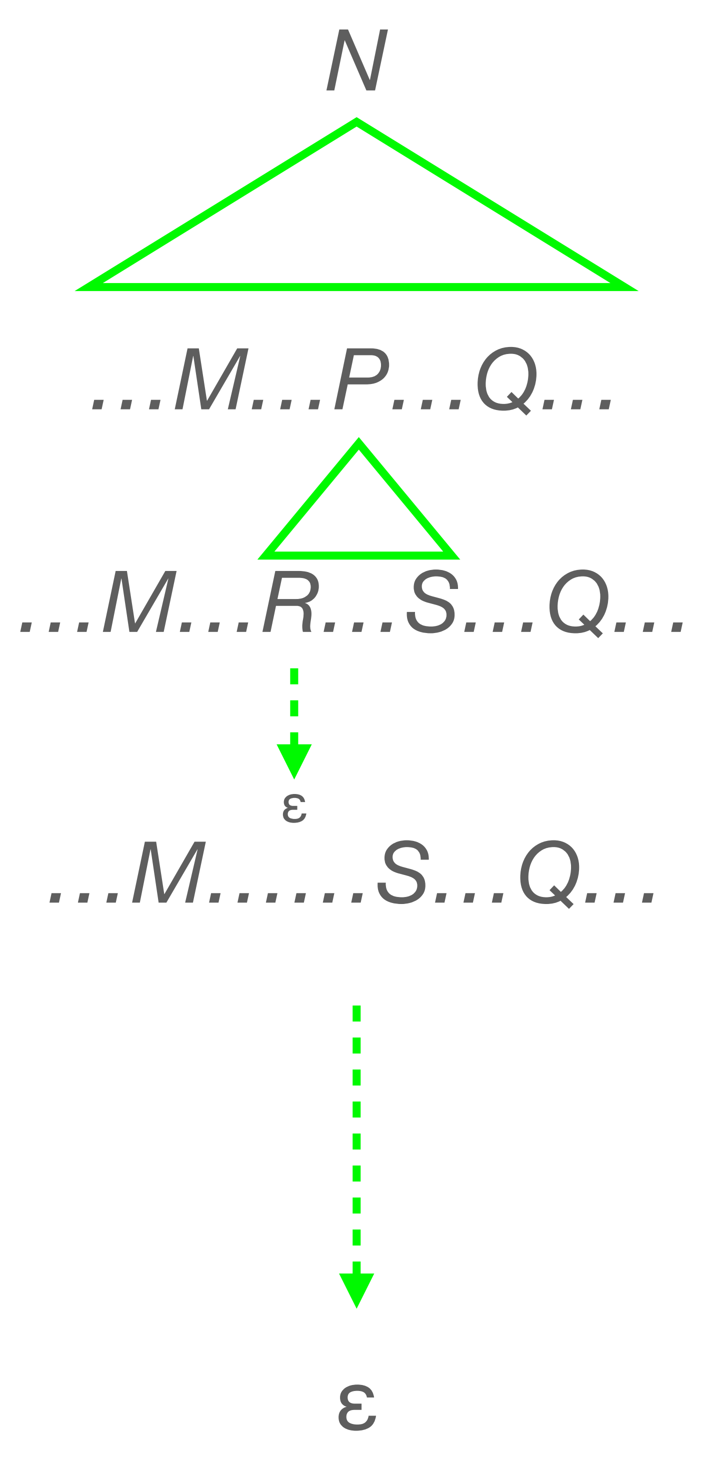
Recursive algorithm:

```
nullableA G s [ ] = T

nullableA G s (TOK _ :: _) = F

nullableA G s (NT n :: rest) =
     nullableA G s rest ∧
     n  is not a member of set s   ∧
```

Calculating Nullability



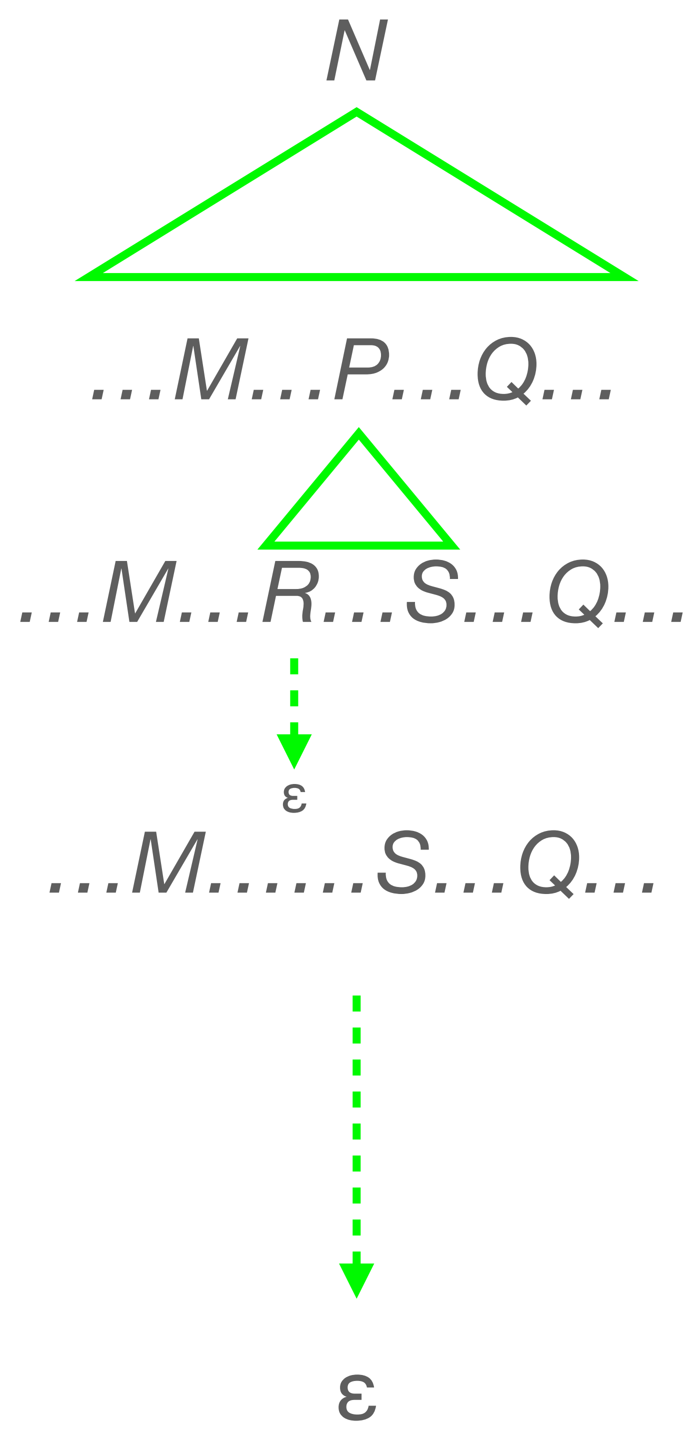
Recursive algorithm:

```
nullableA G s [ ] = T

nullableA G s (TOK _ :: _) = F

nullableA G s (NT n :: rest) =
     nullableA G s rest ∧
     n is not a member of set s   ∧
     nullableA G (n INSERT s) r
          For some r a production for non-terminal n
```

Calculating Nullability



Recursive algorithm:

```
nullableA G s [ ] = T

nullableA G s (TOK _ :: _) = F

nullableA G s (NT n :: rest) =
     nullableA G s rest ∧
     n  is not a member of set s   ∧
     nullableA G (n INSERT s) r
          For some r  a production for non-terminal n
```

Theorem:

nullable G sf ⇔ nullableA G ∅ *sf*

Clean, Mathematical Formulations

Clean, Mathematical Formulations

A high-level property characterising *nullability* can be re-expressed more "algorithmically"

- without using lists!

Clean, Mathematical Formulations

A high-level property characterising *nullability* can be re-expressed more "algorithmically"

- without using lists!

The notion of *first* set can be handled similarly:

- A sentential form has a *first* set (just as an s.f. may be nullable)
- Uses "seen" set of visited non-terminals (recursive calls can be ignored)

Iterating Over all of a Grammar

Formulations of *nullable* and *first* are functions on sentential forms.

Each of a grammar's non-terminals are themselves (short) sentential forms.

Thus: we can take the image of these functions over the non-terminal set, and be done.

- Computationally, this looks bad: calculating e.g., *nullable(N)* will recalculate *nullable* for all non-terminals *N* refers to, and so on, recursively.

Essence of Refinement

CENEX oil refinery, Montana—Greg Goebel *via* flickr.com

- Haven't committed to using lists to represent grammars
- Have separated concerns
- Have deferred other algorithmic decisions
- Have already lost some efficiencies…

The Evil That Is the Follow Set

The "*iterate until result stops changing*" seems unavoidable.

It's also painful:

The Evil That Is the Follow Set

The "*iterate until result stops changing*" seems unavoidable.

It's also painful:

> Because of this algorithm's "iterate until convergence" structure, we need to do some extra work to prove that it terminates. To accomplish this task, we use Coq's Program extension [18], which provides support for defining functions using well-founded recursion. The `Program Fixpoint` command enables the user to define a non-structurally recursive function by providing a measure—a mapping from one or more function arguments to a value in some well-founded relation $\mathcal{R}$—and then showing that the measure of recursive call arguments is less than that of the original arguments in $\mathcal{R}$.

—Lasser, Casinghino, Fisher, Roux (ITP'2019)

The Evil That Is the Follow Set

The "*iterate until result stops changing*" seems unavoidable.

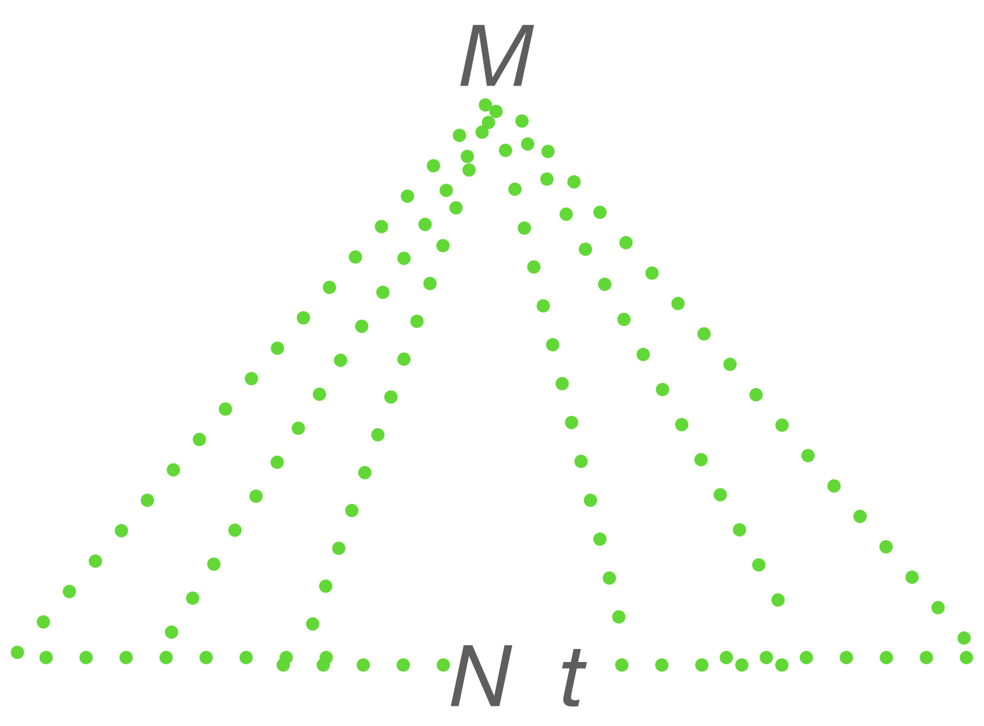
It's also painful:

> Because of this algorithm's "iterate until convergence" structure, we need to do some extra work to prove that it terminates. To accomplish this task, we use Coq's Program extension [18], which provides support for defining functions using well-founded recursion. The **`Program Fixpoint`** command enables the user to define a non-structurally recursive function by providing a measure—a mapping from one or more function arguments to a value in some well-founded relation $\mathcal{R}$—and then showing that the measure of recursive call arguments is less than that of the original arguments in $\mathcal{R}$.

—Lasser, Casinghino, Fisher, Roux (ITP'2019)

(The paper above is following Appel and doing this for all of *nullable*, *first*, and *follow*.)

Follow's Clean Characterisation (I)



- Symbol *t* is in *N*'s follow set if there is a valid derivation from some *M* ending in a sentential form with *t* occurring immediately after *N*.
- The "all at once" view

Follow's Clean Characterisation (II)

The equivalent step-at-a-time view, following Lasser *et al.*:

$$\frac{M \to \alpha N \beta \ \in \ G \qquad a \in \mathsf{first}_G(\beta)}{a \in \mathsf{follow}_G(N)}$$

$$\frac{M \to \alpha N \beta \ \in \ G \qquad \mathsf{nullable}_G(\beta) \qquad a \in \mathsf{follow}_G(M)}{a \in \mathsf{follow}_G(N)}$$

Here, the recursive reference is “backwards” (which rules does *N* appear *in*), and recursions can’t be ignored.

Iterating Over Finite Sets

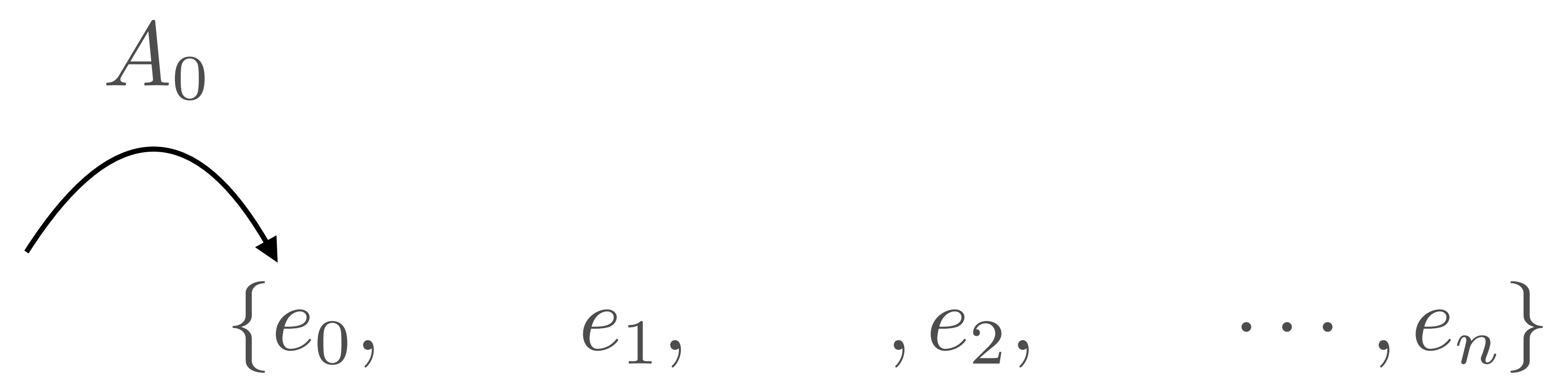
$\{e_0, \quad e_1, \quad , e_2, \quad \cdots, e_n\}$

Iterating Over Finite Sets

$\{e_0, \quad e_1, \quad , e_2, \quad \cdots , e_n\}$

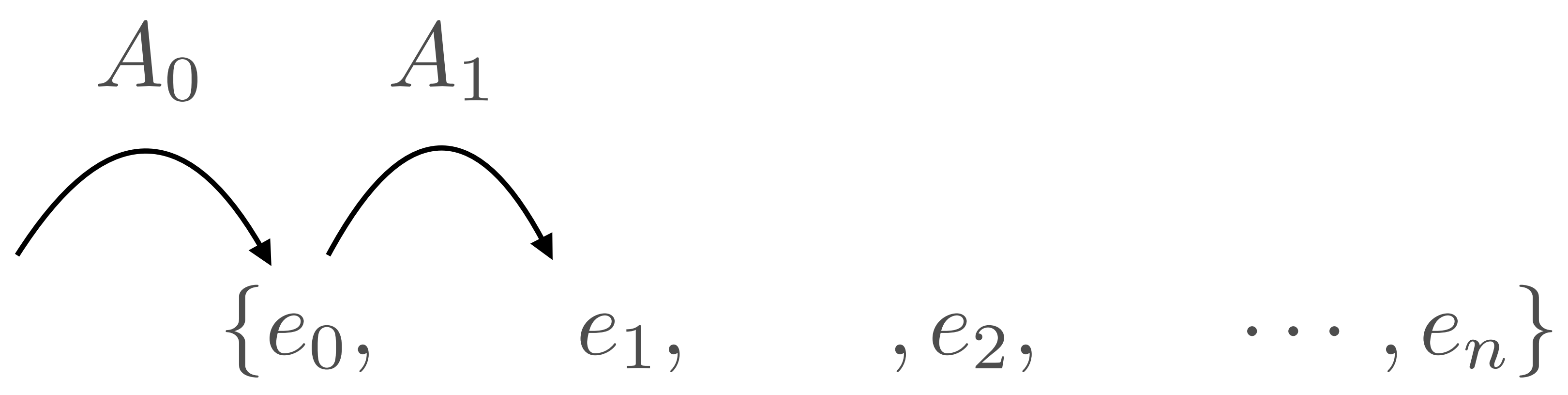
- We want a fold-like way to iterate over the elements of the set (e.g., grammar's rules).
- (Making the set look like a list.)

Iterating Over Finite Sets



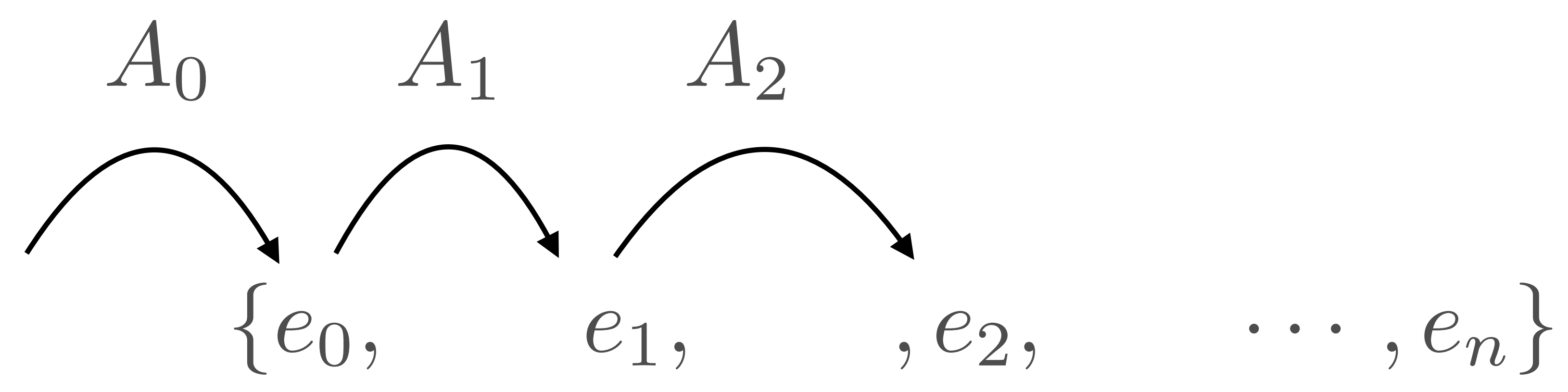
- We want a fold-like way to iterate over the elements of the set (e.g., grammar's rules).
- (Making the set look like a list.)

Iterating Over Finite Sets



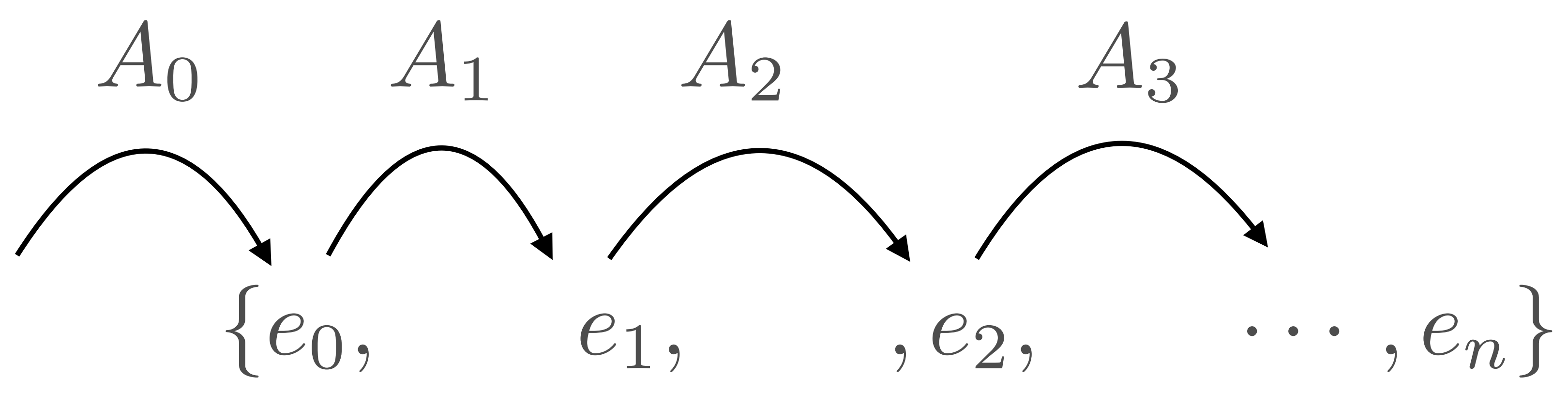
- We want a fold-like way to iterate over the elements of the set (e.g., grammar's rules).
- (Making the set look like a list.)

Iterating Over Finite Sets



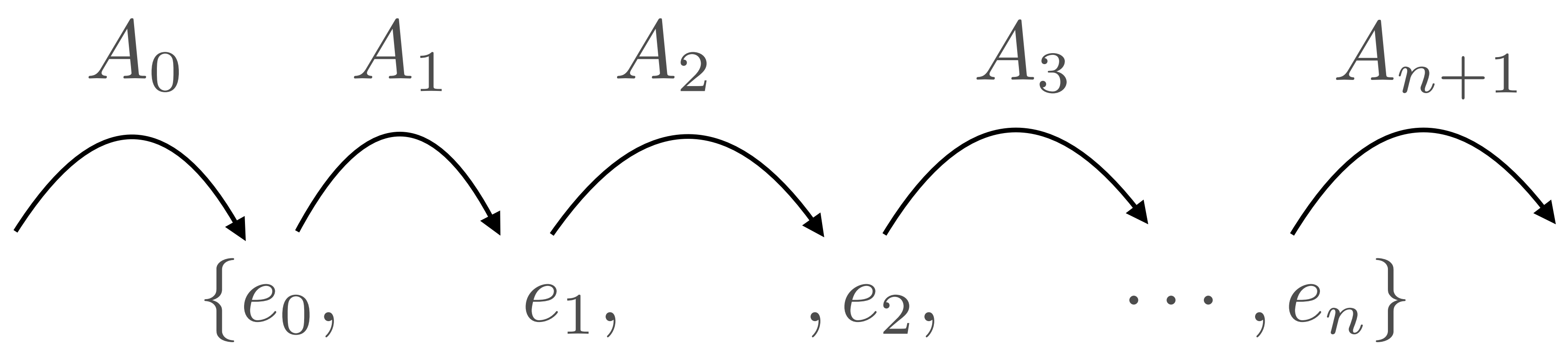
- We want a fold-like way to iterate over the elements of the set (e.g., grammar's rules).
- (Making the set look like a list.)

Iterating Over Finite Sets



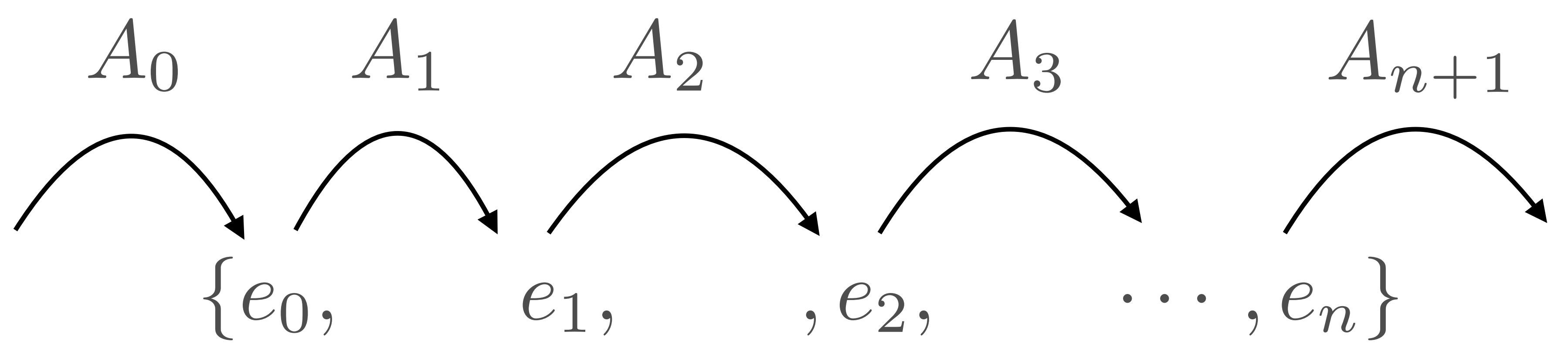
- We want a fold-like way to iterate over the elements of the set (e.g., grammar's rules).
- (Making the set look like a list.)

Iterating Over Finite Sets



- We want a fold-like way to iterate over the elements of the set (e.g., grammar's rules).
- (Making the set look like a list.)

Iterating Over Finite Sets



- We want a fold-like way to iterate over the elements of the set (e.g., grammar's rules).
- (Making the set look like a list.)
- But for soundness, the result cannot depend on the order in which the elements are consumed!

Iterating for Follow Calculation

Complexities:

- Processing one sentential form updates *follow* information for multiple non-terminals at once
  - For example, $N \rightarrow aMbPcQ$ gives partial info for follow sets of M, P and Q
- Recursive calls (to N above, say), fold in yet more partial info

The Challenge

✷✷✷ WANTED ✷✷✷

Clean Solutions to
Annoying Problems

But Which Still Refine
To
Efficient Code

The Challenge

✷✷✷ WANTED ✷✷✷

CLEAN SOLUTIONS TO
ANNOYING PROBLEMS

BUT WHICH STILL REFINE
TO
EFFICIENT CODE

- Iterate ’til convergence (with non-commutative accumulation) *is* possible
- Preserves grammars as finite sets
- Mostly aesthetically pleasing
- How to then memoize and recombine 3 separate functions?

The Challenge

*** WANTED ***

Clean Solutions to
Annoying Problems

But Which Still Refine
To
Efficient Code

- Iterate 'til convergence (with non-commutative accumulation) *is* possible
- Preserves grammars as finite sets
- Mostly aesthetically pleasing
- How to then memoize and recombine 3 separate functions?
- Translation to CakeML will be easy, (and will re-introduce lists…)

Conclusion

Compilers can be made super formal:

- Programming language semantics and interactive theorem-proving combine to create verified compilers (not only CakeML);
- Compilers use a great deal of theory from many different areas to implement their algorithms;
- Even *first* and *follow* set computations present some interesting challenges…