Covering the Last Mile in Trustworthy Automated Reasoning with CakeML

Yong Kiam Tan

Workshop for HOL4 users, 25 Jun 2024

Outline

1. CakeML: A Verified Language Ecosystem
2. Automated Reasoning, Proof Logging, and Proof Checking
3. The Next 700 Verified Proof Checkers

This Talk: An overview of CakeML with pointers to recent developments, and some exciting applications in trustworthy automated reasoning.

* I will highlight work by various CakeML contributors (many in audience!).

Outline

1. CakeML: A Verified Language Ecosystem
2. Automated Reasoning, Proof Logging, and Proof Checking
3. The Next 700 Verified Proof Checkers

This Talk: An overview of CakeML with pointers to recent developments, and some exciting applications in trustworthy automated reasoning.

* I will highlight work by various CakeML contributors (many in audience!).

CakeML: A Verified Implementation of ML

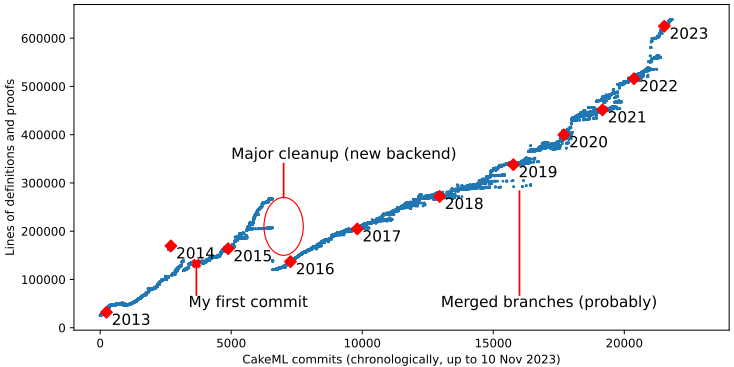
CakeML, circa [POPL'14], is:

- A programming language in the style of Standard ML,
- With formally specified semantics,
- And a formally verified **bootstrapping** compiler,
- Targeting x64 machine code,
- All formalized in the HOL4 theorem prover.

Verified bootstrapping: verified compiler compiles its own verified CakeML implementation (details in appendix).

CakeML: A Verified Implementation of ML

The project has grown significantly over ≈ 10 years:



Plot does not include automation libraries, contributions to HOL4, and developments hosted outside the main CakeML repository.

Jun 25 2024: `wc -l ‘find . -type f -name *Script.sml‘` gives 698590 LoC.

CakeML: A Verified Language Ecosystem

CakeML, today:

- Various additional language frontends & semantics:
  - Candle (theorem proving)
  - Kalas (choreographies)
  - Pancake (systems programming language)
  - PureCake (Haskell-like language)
  - RealCake (real-valued & floating-point semantics)
- Frontends either compile to CakeML, or one of its compiler's internal languages.

CakeML: A Verified Language Ecosystem

CakeML, today:

- New (lightly) optimizing compiler backend
  - Supporting foreign function calls (FFI) with customizable semantics
  - Optimizations for functional languages, register allocation, tail call optimization, …
  - Verified bignum library & (generational) garbage collection
  - Targeting multiple architectures 32-/64-bit (x64, **ARMv7/8**, MIPS64, RISC-V)
    + Silver (custom ISA with verified hardware)
    + **Authoritative ARMv8 specification**
- Still capable of verified **bootstrap** and (new) verified runtime “eval”

CakeML: A Verified Language Ecosystem

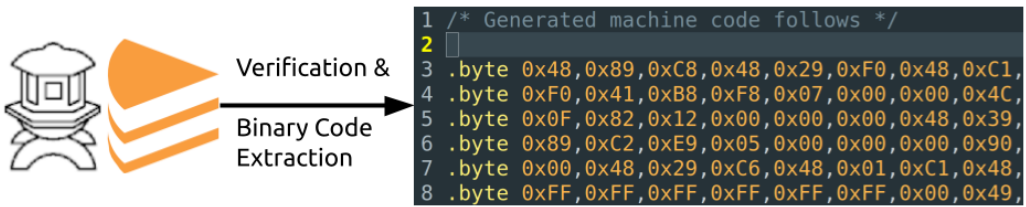
CakeML, today:

- Tools for building and verifying CakeML programs
  - Verified space cost semantics for CakeML
  - Proof-producing translation of (monadic) HOL functions into CakeML
  - Characteristic formulae, with support for imperative CakeML, exceptions, FFIs, liveness properties of diverging programs
- Various end-to-end verified applications **(later in this talk)**

CakeML and Binary Code Extraction

CakeML bootstrapping is an instance of **binary code extraction** [ITP'18]

Software is verified and compiled **entirely** within the theorem prover, giving proven correctness guarantees at the level of **machine-code binaries**.



Eliminates parts of the usual trusted base for ITP-verified code:

- No unverified exporters (e.g., Coq to OCaml, Isabelle/HOL exporters).
- No unspecified/unverified "external" code (e.g., parsing, file I/O, FFI).
- No unverified compilers/assemblers ran on the exported source code.

Outline

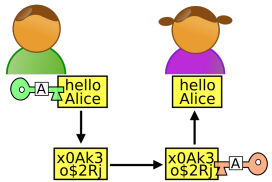
1. CakeML: A Verified Language Ecosystem
2. Automated Reasoning, Proof Logging, and Proof Checking
3. The Next 700 Verified Proof Checkers

This Talk: An overview of CakeML with pointers to recent developments, and some exciting applications in trustworthy automated reasoning.

* I will highlight work by various CakeML contributors (many in audience!).

Automated Reasoning

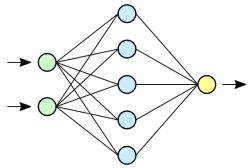
Automated reasoning is used in a wide variety of domains.

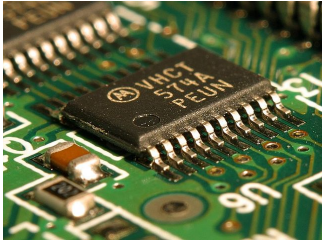


Two-hundred-terabyte maths proof is largest ever

Evelyn Lamb

Nature **534**, 17–18 (2016) | Cite this article



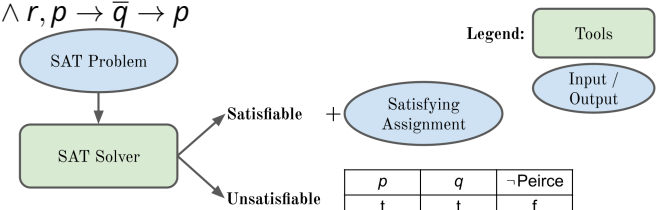


Automated Reasoning

Automated reasoning is used in a wide variety of domains.

Example: SAT solvers decide **satisfiability** for propositional formulas.

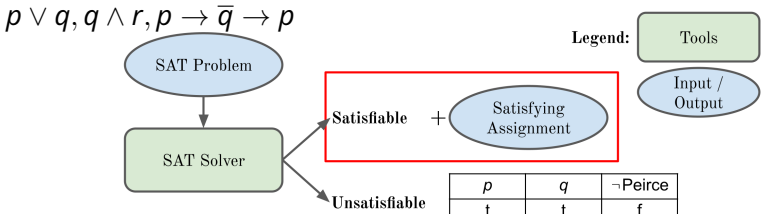
$p \vee q, q \wedge r, p \rightarrow \overline{q} \rightarrow p$



p	q	¬Peirce
t	t	f
t	f	f
f	t	f
f	f	f

Automated Reasoning

Automated reasoning is used in a wide variety of domains.
Example: SAT solvers decide **satisfiability** for propositional formulas.

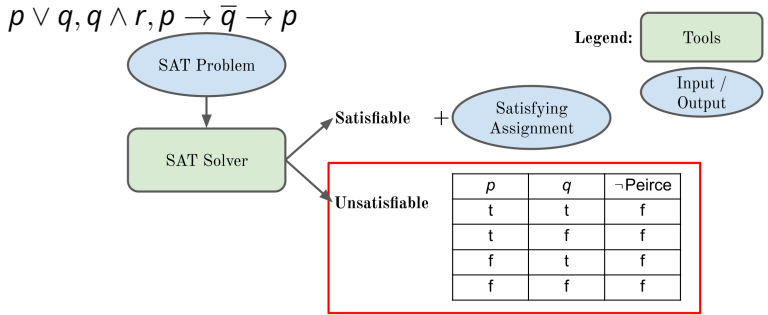


p	q	¬Peirce
t	t	f
t	f	f
f	t	f
f	f	f

Application: Used as part of CEGAR, DPLL(T), ...
Satisfying assignments can be easily checked for correctness by evaluation.

Automated Reasoning

Automated reasoning is used in a wide variety of domains.
Example: SAT solvers decide **satisfiability** for propositional formulas.

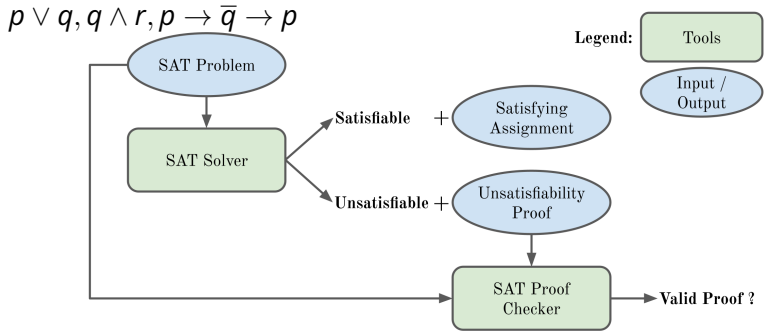


Application: Circuit equivalence, SMT (unsat), mathematics, ...

Problem: How to certify a solver's "UNSAT" claim?
Theory: SAT $\in$ NP (efficient witness checks) but UNSAT $\in$ co-NP.

(UN)SAT Proof Checking

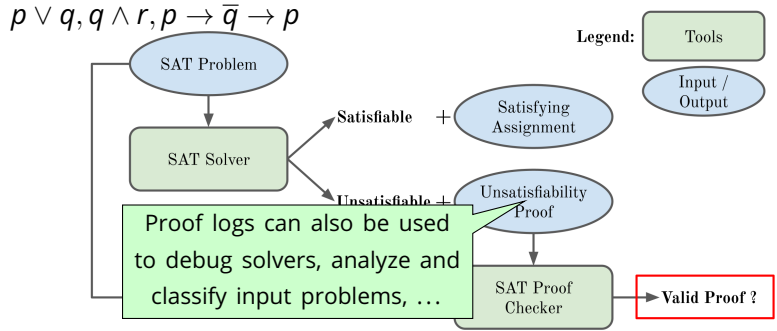
Automated reasoning is used in a wide variety of domains.
Example: SAT solvers decide **satisfiability** for propositional formulas.



Idea: Have solvers efficiently log proofs for unsatisfiability claims.

(UN)SAT Proof Checking

Automated reasoning is used in a wide variety of domains.
Example: SAT solvers decide **satisfiability** for propositional formulas.

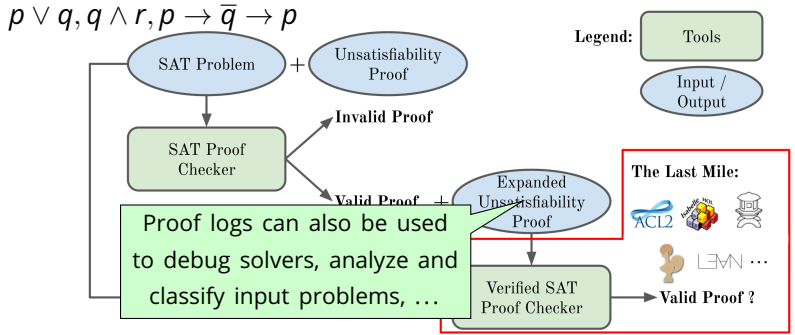


Idea: Have solvers efficiently log proofs for unsatisfiability claims.

Caveat: Trusted base contains unverified (but often simple) proof checker.

(UN)SAT Proof Checking

Automated reasoning is used in a wide variety of domains.
Example: SAT solvers decide **satisfiability** for propositional formulas.



Goal: Covering the last mile with CakeML and binary code extraction.

Crash Course: Clausal Formulas

Propositional formulas normalize to Conjunctive Normal Form (CNF), e.g.:

$$\neg(((p \to q) \to p) \to p)$$

$$\Downarrow \text{ CNF conversion}$$

$$\underbrace{(p \vee p)}_{\text{Clause}_1} \quad \wedge \quad \underbrace{(\overline{q} \vee p)}_{\text{Clause}_2} \quad \wedge \quad \underbrace{(\overline{p})}_{\text{Clause}_3}$$

- **Literals** refer to propositional variables p or their negation $\overline{p}$.
- **Clauses** are a disjunction over a set of literals.
- **Formulas** are a conjunction over a set of clauses.

Conflict-Driven Clause Learning: Core algorithm in modern SAT solvers.
Efficiently derive implied clauses until contradiction (or sat. witness found).

Crash Course: Conflict-Driven Clause Learning (CDCL)

Example:

$$\{x_1, x_2, x_4\}$$
$$\{x_1, \overline{x_2}, x_4\}$$
$$\{\overline{x_1}, x_4\}$$
$$\{\overline{x_4}, x_5\}$$
$$\{\overline{x_4}, \overline{x_5}\}$$

1. Branch on a variable
2. Unit propagation
3. On conflict, back jump (not shown), learn conflict clause
4. Repeat

Crash Course: Conflict-Driven Clause Learning (CDCL)

Example:

$$\{x_1, \mathtt{t}, x_4\}$$
$$\{x_1, \mathtt{f}, x_4\}$$
$$\{\overline{x_1}, x_4\}$$
$$\{\overline{x_4}, x_5\}$$
$$\{\overline{x_4}, \overline{x_5}\}$$

1. Branch on a variable $x_2 = \mathtt{t}$
2. Unit propagation
3. On conflict, back jump (not shown), learn conflict clause
4. Repeat

Crash Course: Conflict-Driven Clause Learning (CDCL)

Example:

$$\{x_1, x_4\}$$
$$\{\overline{x_1}, x_4\}$$
$$\{\overline{x_4}, x_5\}$$
$$\{\overline{x_4}, \overline{x_5}\}$$

1. Branch on a variable $x_2 = \mathtt{t}$
2. Unit propagation
3. On conflict, back jump (not shown), learn conflict clause
4. Repeat

Crash Course: Conflict-Driven Clause Learning (CDCL)

Example:

$$\{x_5\}$$
$$\{\overline{x_5}\}$$

1. Branch on a variable $x_2 = \mathtt{t}$ $x_4 = \mathtt{t}$
2. Unit propagation
3. On conflict, back jump (not shown), learn conflict clause
4. Repeat

Crash Course: Conflict-Driven Clause Learning (CDCL)

Example:

$\{x_5\}$

$\{\ \}$ conflict, learn $\{\overline{x_2}, \overline{x_4}\}$

1. Branch on a variable $x_2 = \mathtt{t}\ x_4 = \mathtt{t}$
2. Unit propagation $\{x_5\}$
3. On conflict, back jump (not shown), learn conflict clause
4. Repeat

Crash Course: Conflict-Driven Clause Learning (CDCL)

Example:

$$\{x_1, x_2, x_4\}$$
$$\{x_1, \overline{x_2}, x_4\}$$
$$\{\overline{x_1}, x_4\}$$
$$\{\overline{x_4}, x_5\}$$
$$\{\overline{x_4}, \overline{x_5}\}$$
$$\{\overline{x_2}, \overline{x_4}\} \quad \{\overline{x_2}\} \quad \{\} \leftarrow \text{Learned clauses}$$

1. Branch on a variable
2. Unit propagation
3. On conflict, back jump (not shown), learn conflict clause
4. Repeat

Observation: Sequence of learned clauses and unit propagation steps provide a proof log of reasoning steps taken by the solver.

Propositional (UN)SAT Proof Checking

Example: DRAT/LRAT proof logs (standard for SAT competitions)

DIMACS file

```
p cnf 12 22
  1  2  3 0
  4  5  6 0
  7  8  9 0
 10 11 12 0
    -1 -4 0
    -2 -5 0
    -3 -6 0
    -1 -7 0
    -2 -8 0
    -3 -9 0
    ...
```

DRAT

```
-11 -12 0
 -7 -12 0
    -12 0
     -8 0
     -7 0
      ...
      2 0
     -5 0
    -11 0
      4 0
     10 0
        0
```

LRAT

```
24 -7 -12 0 8 14 13 19 1 2 6 0
25 -12 0 13 19 22 24 3 9 15 1 2 5 0
25 d 13 19 22 24 0
26 -8 0 25 9 15 21 4 11 17 1 2 7 0
26 d 9 15 21 0
27 -7 0 25 8 14 20 4 12 18 1 2 7 0
27 d 8 14 20 7 0
31 -1 0 25 26 27 3 16 5 2 11 4 18 0
31 d 5 11 18 0
38 0 25 26 27 3 10 16 31 1 6 12 2 4 17 0
```

DRAT files are dumped by solvers; an unverified (and untrusted) tool elaborates them into LRAT format for efficient verified proof checking.

Demo: `cake_lpr`, a CakeML-based proof checker, verified with binary code extraction & used in recent SAT competitions.

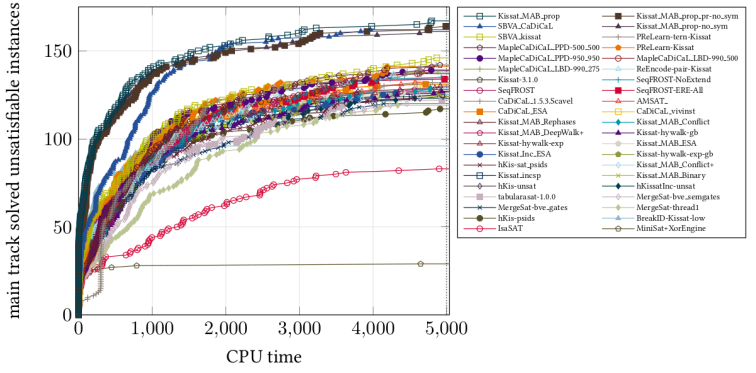
End-to-End Correctness Theorem

```
⊢ cake_lpr_run cl fs mc ms ⇒
  machine_sem mc (basis_ffi cl fs) ms ⊆
   extend_with_resource_limit { Terminate Success (cake_lpr_io_events cl fs) } ∧
  ∃ out err.
   extract_fs fs (cake_lpr_io_events cl fs) = Some (add_stdout (add_stderr fs err) out) ∧
   if length cl = 2 then ...
   else if length cl = 3 then
    if out = «s VERIFIED UNSAT\n» then
     inFS_fname fs (el 1 cl) ∧
     ∃ fml.
      parse_dimacs (all_lines fs (el 1 cl)) = Some fml ∧
      unsatisfiable (interp fml)
    else out = «»
   else ...
```

If `cake_lpr` prints "s VERIFIED UNSAT" to standard output, then the input DIMACS file parsed[a] to a formula that is unsatisfiable.

[a] Parsing is also verified to be left-inverse to printing.

Slide from SAT Competition 2023



- Beyond Resolution Strikes Back (in all top-tier solvers)

Modern solving techniques: symmetry breaking, preprocessing, SDCL
$\Rightarrow$ Demand for new verified proof checkers supporting stronger proof systems

Other CakeML-based Proof Checkers

- Candle (verified clone of HOL Light)
- OpenTheory (format for HOL proofs)
- FloVer/Daisy (error-bound certificates for FP arithmetic)
- Pseudo-Boolean satisfiability/optimization proofs
- Mixed integer program optimization proofs
- CNF-XOR unsatisfiability proofs

Observation: Some common verified optimizations needed across proof checker setups, but many theory-specific ones needed for performance.

Examples:
(Common) Throwing exceptions on error instead of boxing all results
(Specific) Caching information for heavyweight pseudo-Boolean proof steps

Outline

This Talk: An overview of CakeML with pointers to recent developments, and some exciting applications in trustworthy automated reasoning.

* I will highlight work by various CakeML contributors (many in audience!).

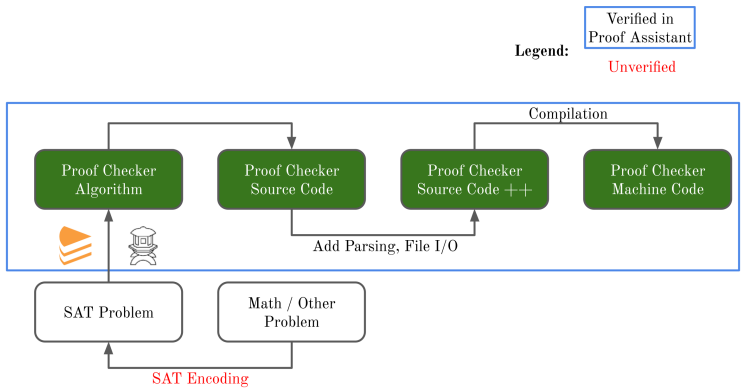
Lessons Learned

- Verified proof checking is a great **stress test** and **use case** for software verification in foundational proof assistants.
  $\Rightarrow$ Intersection between the PL/ITP and ATP communities.
- Ease of use of verified artifacts is important.
  $\Rightarrow$ `cake_lpr` has pre-compiled verified binaries for x64 and ARMv8.
- Focus on optimizing (and verifying) the common cases
  $\Rightarrow$ Careful profiling and/or working with the tool developers

New CakeML tools, e.g., proof automation, libraries, and compiler optimizations, will be needed for the next 700 verified proof checkers.

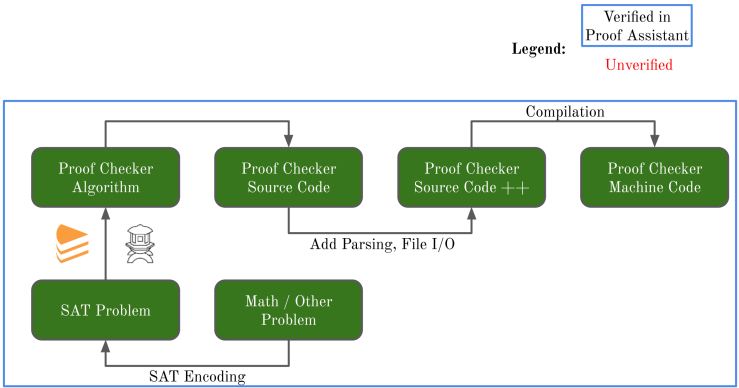
Next Steps: End-to-End Verification with Encodings

Formalize various problem encoders and connect to verified proof checkers



Next Steps: End-to-End Verification with Encodings

Formalize various problem encoders and connect to verified proof checkers

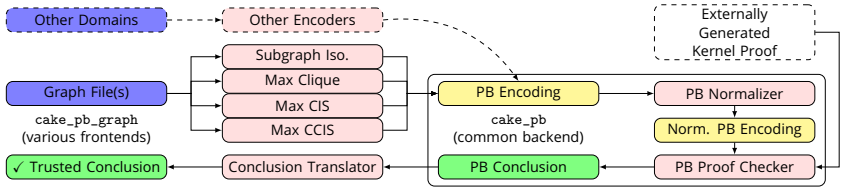


Challenge: Building reusable encoder libraries across formal developments

Next Steps: Extensible Proof Checking Frameworks

Going from individual verified checkers to efficient, reusable frameworks

Example: Common, verified pseudo-Boolean proof checking backend for various combinatorial solvers (at AAAI'24).



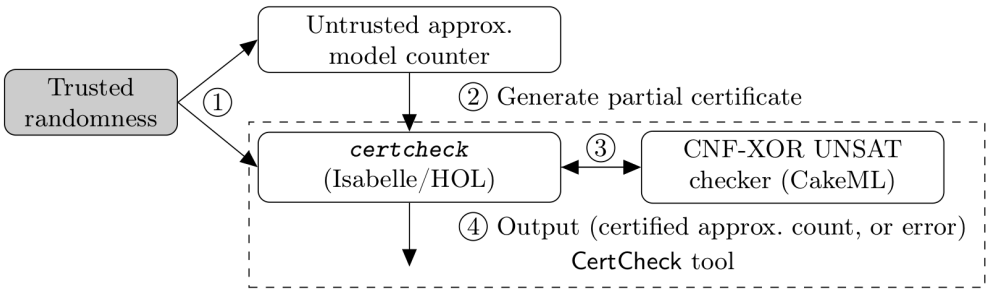
Various potential framework-building scenarios:

- Same theory, customized proof rules for different solving techiques
- Similar theories, single general-purpose backend framework (above)
- Multiple theories, framework supporting extensible theory combination

Next Steps: Novel Proof Checker Guarantees

Verifying novel proof checking guarantees for automated reasoning tools: concurrent, distributed, probabilistic, etc.

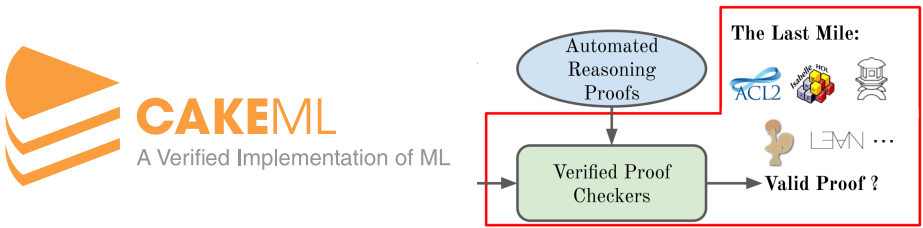
Example: Formally certified approximate model counting (at CAV'24)



Challenge: Need language support to take, e.g., probabilistic verification guarantees to the level of machine code.

Summary

The CakeML ecosystem has grown substantially over ≈ 10 years, but there remains much more to be done.



Verified proof checking for automated reasoning is giving us a wealth of ideas for advancing CakeML (Join: `https://discord.gg/a8UUs6Ce6m`).

Advert: I have open PhD/postdoc positions in Singapore, please do get in contact if you know anyone who might be interested.

References

[1] Kumar, R., Mullen, E., Tatlock, Z., and Myreen, M. O. (2018). Software verification with ITPs should use binary code extraction to reduce the TCB - (short paper). In Avigad, J. and Mahboubi, A., editors, ITP, volume 10895 of Lecture Notes in Computer Science, pages 362–369. Springer.

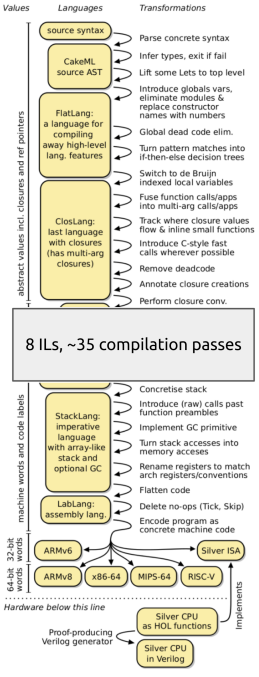
[2] Kumar, R., Myreen, M. O., Norrish, M., and Owens, S. (2014). CakeML: a verified implementation of ML. In Jagannathan, S. and Sewell, P., editors, POPL, pages 179–192. ACM.

[3] Sewell, T., Myreen, M. O., Tan, Y. K., Kumar, R., Mihajlovic, A., Abrahamsson, O., and Owens, S. (2023). Cakes that bake cakes: Dynamic computation in CakeML. Proc. ACM Program. Lang., 7(PLDI):1121–1144.

Compiler Bootstrap (intuition)

① The CakeML compiler (shown abridged) $\Longrightarrow$ is implemented and verified in HOL:

$$\begin{aligned}&\text{comp } (\text{Inst } x) = \ldots\\&\text{comp } (\text{Seq } x\ y) = \text{Seq } (\text{comp } x)\ (\text{comp } y)\\&\ldots\\&\vdash \text{comp } e_{src} = e_{tar} \Rightarrow [\![e_{src}]\!] = [\![e_{tar}]\!]\end{aligned}$$



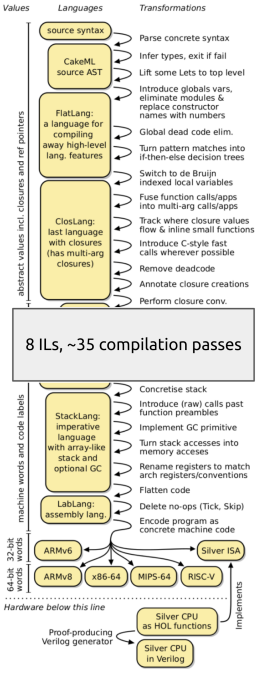
Compiler Bootstrap (intuition)

① The CakeML compiler (shown abridged) $\Longrightarrow$ is implemented and verified in HOL:

$$\begin{aligned}&\text{comp } (\text{Inst } x) = \ldots\\&\text{comp } (\text{Seq } x\ y) = \text{Seq } (\text{comp } x)\ (\text{comp } y)\\&\ldots\\&\vdash \text{comp } e_{src} = e_{tar} \Rightarrow [\![e_{src}]\!] = [\![e_{tar}]\!]\end{aligned}$$

② Proof-producing translation generates a CakeML implementation comp_{src} and proves:

$$\vdash \text{comp} = [\![\text{comp}_{src}]\!]$$



Compiler Bootstrap (intuition)

① The CakeML compiler (shown abridged) $\Longrightarrow$ is implemented and verified in HOL:

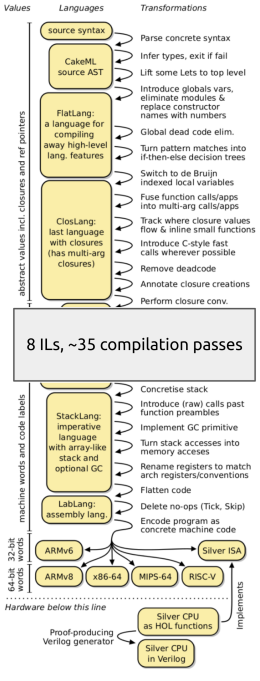
$$\begin{aligned}&\text{comp } (\text{Inst } x) = \ldots\\&\text{comp } (\text{Seq } x\ y) = \text{Seq } (\text{comp } x)\ (\text{comp } y)\\&\ldots\\&\vdash \text{comp } e_{src} = e_{tar} \Rightarrow [\![e_{src}]\!] = [\![e_{tar}]\!]\end{aligned}$$

② Proof-producing translation generates a CakeML implementation comp_{src} and proves:

$$\vdash \text{comp} = [\![\text{comp}_{src}]\!]$$

③ Evaluation in HOL4 (see `cv_compute` talk):

$$\vdash \text{comp } \text{comp}_{src} = \text{comp}_{tar}$$



Compiler Bootstrap (intuition)

① The CakeML compiler (shown abridged) $\Longrightarrow$ is implemented and verified in HOL:

$$\begin{aligned}&\text{comp } (\text{Inst } x) = \ldots\\&\text{comp } (\text{Seq } x\ y) = \text{Seq } (\text{comp } x)\ (\text{comp } y)\\&\ldots\\&\vdash \text{comp } e_{src} = e_{tar} \Rightarrow [\![e_{src}]\!] = [\![e_{tar}]\!]\end{aligned}$$

② Proof-producing translation generates a CakeML implementation comp_{src} and proves:

$$\vdash \text{comp} = [\![\text{comp}_{src}]\!]$$

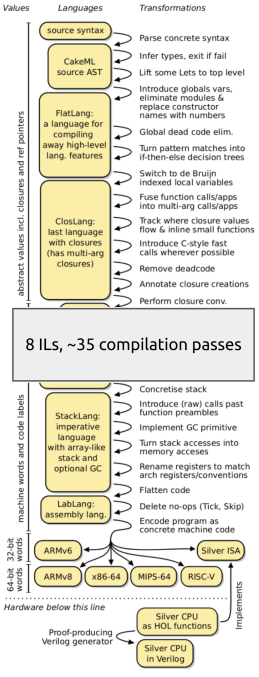
③ Evaluation in HOL4 (see `cv_compute` talk):

$$\vdash \text{comp } \text{comp}_{src} = \text{comp}_{tar}$$

④ Bootstrapping theorem (from ①–③):

$$\vdash \text{comp} = [\![\text{comp}_{tar}]\!]$$

i.e., comp_{tar} is a verified compiler binary



Compiler Bootstrap (reality)

- Most equalities ($=$) on prev. slide are actually appropriate relations ($\sim$)
- Compiler phases maintain state & proofs usually involve designing suitable invariants

$$\vdash \mathrm{comp}\ st\ e_{src} = (st', e_{tar}) \wedge \mathrm{inv}\ st \ldots \Rightarrow \\ [\![e_{src}]\!] \sim [\![e_{tar}]\!] \wedge \mathrm{inv}\ st' \ldots$$

- Translation comp_{src} needs manual tweaks to use efficient imperative code, e.g., for register allocation & assembly
- ~~Compiler bootstrap is expensive~~ ~~$\approx$24 hours on server with 64GB RAM~~ (see `cv_compute` talk)
- Bootstrapped comp_{tar} can be used for dynamic computation, see [PLDI'23]

Regular releases at `https://cakeml.org/`