



CAKEML

A Verified Implementation of ML

Ramana Kumar, Magnus Myreen, Michael Norrish, Scott Owens



Dimensions of Compiler Verification

Dimensions of Compiler Verification

Source Code

Abstract Syntax

Intermediate Language

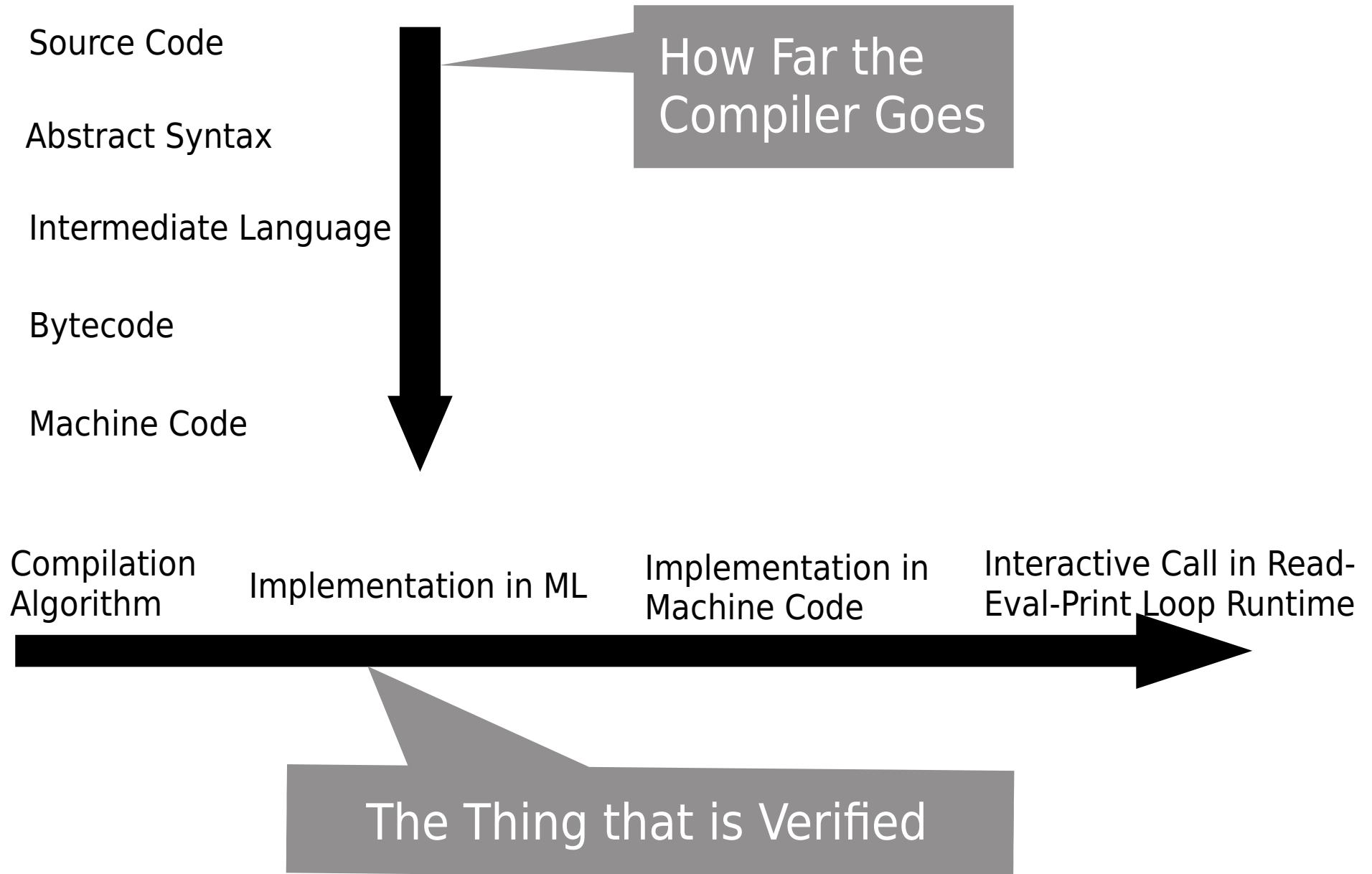
Bytecode

Machine Code

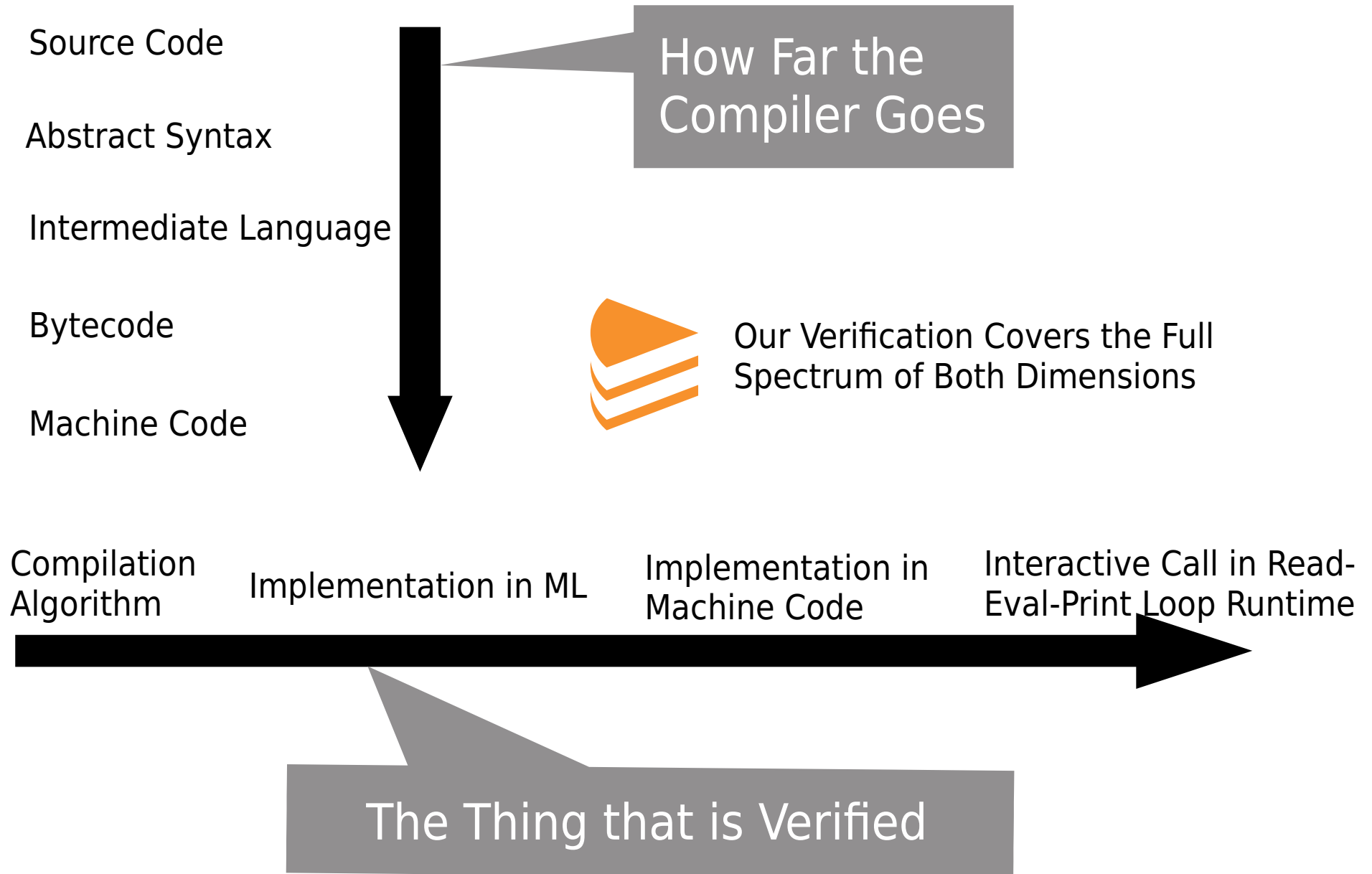


How Far the
Compiler Goes

Dimensions of Compiler Verification



Dimensions of Compiler Verification



The Size of the Language

CakeML, the language

= Standard ML without I/O, floats, functors

The Size of the Language

CakeML, the language

= Standard ML without I/O, floats, functors

i.e., most everything else:

- ✓ higher-order functions,
- ✓ mutual recursion and polymorphism,
- ✓ datatypes and pattern matching,
- ✓ references and exceptions,
- ✓ modules and signatures

Contributions

Artefacts

Proof Techniques

Contributions

Artefacts

Specifications

Verified Algorithms

Verified Executable

Proof Techniques

Contributions

Artefacts

Specifications

Verified Algorithms

Verified Executable

Proof Techniques

Interactive Proof

x86 Synthesis

ML Synthesis

Bootstrapping

Contributions

Artefacts

Specifications

Verified Algorithms

Verified Executable

including new inductive
approach to **divergence**
preservation

Proof Techniques

Interactive Proof

x86 Synthesis

ML Synthesis

Bootstrapping

Contributions

Artefacts

Specifications

Verified Algorithms

Verified Executable

including new inductive approach to divergence preservation

Proof Techniques

Interactive Proof

x86 Synthesis

ML Synthesis

Bootstrapping

main new technique: use verified compiler to produce verified implementation

Approach

Proof by refinement:

Step 1: specification of CakeML language

- big-step and small-step operational semantics

Step 2: functional implementation in logic

- read-eval-print loop as a verified function

Step 3: production of verified x86-64 code

- produced mainly by bootstrapping the compiler

Operational Semantics

Operational Semantics

Big-step evaluation relation

environment semantics, **closures**

metatheory: deterministic, total, and type sound
stuck means divergence, runtime type errors for bad input

Operational Semantics

Big-step evaluation relation

environment semantics, **closures**

metatheory: deterministic, total, and type sound
stuck means divergence, runtime type errors for bad input

Small-step CEK machine

for type soundness proof, and defining divergence explicitly

Operational Semantics

Big-step evaluation relation

environment semantics, **closures**

metatheory: deterministic, total, and type sound
stuck means divergence, runtime type errors for bad input

Small-step CEK machine

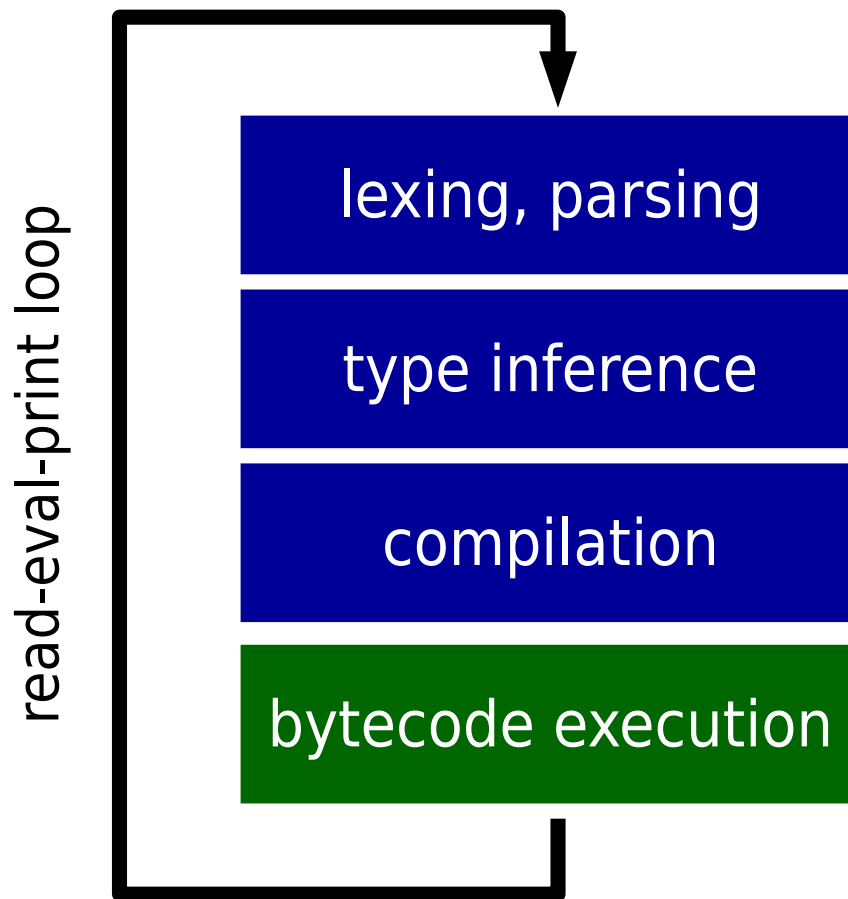
for type soundness proof, and defining divergence explicitly

Read-eval-print loop semantics

relates **input string** to list of **output strings**
ending in termination or divergence

Functional Implementation

Defined as a function in the logic of HOL4



and verified to
implement the
read-eval-print
loop semantics

lexing, parsing

Specification

Context-Free Grammar (CFG) for significant subset of SML

Executable Lexer

Implementation

Parsing-Expression-Grammar (PEG) Parser

inductive evaluation relation

executable interpreter for PEGs

Correctness

Soundness and Completeness

induction on length of token list/parse tree and non-terminal rank

type inference

Specification

Declarative type system

Implementation

Based on Algorithm J

Purely functional (state-exception monad)

Correctness

Proved soundness: every inferred type is derivable

Re-used existing work on verified unification [ITP 2010]

compilation

Purpose

Translate type-checked CakeML to CakeML Bytecode

Implementation

Translation via an intermediate language (IL).
Big-step operational semantics.

CakeML to IL: make language more uniform, de Bruijn indices

IL to IL: remove pattern matching, free-variable analysis

IL to Bytecode: closure conversion, data refinement, tail-call opt.

Semantics of bytecode execution

Instructions:

bc_inst ::= Stack bc_stack_op | PushExc | PopExc
| Return | CallPtr | Call loc
| PushPtr loc | Jump loc | JumpIf loc
| Ref | Deref | Update | Print | PrintC char
| Label n | Tick | Stop

bc_stack_op ::= Pop | Pops n | Shift n n | PushInt int
| Cons n n | El n | TagEq n | IsBlock n
| Load n | Store n | LoadRev n
| Equal | Less | Add | Sub | Mult | Div | Mod

Small-step semantics. Values and States:

bc_value ::= Number int | RefPtr n | Block n bc_value^*
| CodePtr n | StackPtr n

bc_state ::= { stack : bc_value^* ; refs : $n \mapsto bc_value$;
code : bc_inst^* ; pc : n ; handler : n ;
output : string; names : $n \mapsto$ string;
clock : $n^?$ }

Semantics of bytecode execution

Sample rules:

$$\frac{\text{fetch}(bs) = \text{Stack } (\text{Cons } t \ n) \quad bs.\text{stack} = vs @ xs \quad |vs| = n}{bs \rightarrow (\text{bump } bs)\{\text{stack} = \text{Block } t \ (\text{rev } vs) :: xs\}}$$

$$\frac{\text{fetch}(bs) = \text{Return} \quad bs.\text{stack} = x :: \text{CodePtr } ptr :: xs}{bs \rightarrow bs\{\text{stack} = x :: xs; \text{pc} = ptr\}}$$

$$\frac{\text{fetch}(bs) = \text{CallPtr} \quad bs.\text{stack} = x :: \text{CodePtr } ptr :: xs}{bs \rightarrow bs\{\text{stack} = x :: \text{CodePtr } (\text{bump } bs).\text{pc} :: xs; \text{pc} = ptr\}}$$

compilation

Correctness

Proved in the **direction of compilation**.

Data refinement for a closure uses label from closure conversion. Inductive proof assumes this points to the compilation of the body.

Shape of correctness theorem:

$$env \vdash exp \Downarrow val \implies$$

“the code for *exp* is installed in *bs* etc.” \implies

$$\exists bs'. bs \rightarrow^* bs' \wedge \text{“}bs' \text{ contains } val\text{”}$$

compilation

Correctness

Proved in the **direction of compilation**.

Data refinement for a closure uses label from closure conversion. Inductive proof assumes this points to the compilation of the body.

Shape of correctness theorem:

IL big-step eval

$env \vdash exp \Downarrow val \implies$

“the code for exp is installed in bs etc.” \implies

$\exists bs'. bs \rightarrow^* bs' \wedge$ “ bs' contains val ”

bytecode next step reln.

What about divergence?

Idea: add logical clock

Big-step semantics:

- has an optional **clock** component
- **clock 'ticks'** decrement every function application
- when **clock hits zero**, execution stops with a **TimeOut**

Idea: add logical clock

Big-step semantics:

- has an optional **clock** component
- **clock 'ticks'** decrement every function application
- when **clock hits zero**, execution stops with a **TimeOut**

Why do this?

because now big-step semantics describes both terminating and non-terminating evaluations

Idea: add logical clock

Big-step semantics:

- has an optional **clock** component
- **clock 'ticks'** decrement every function application
- when **clock hits zero**, execution stops with a **TimeOut**

Why do this?

because now big-step semantics describes both terminating and non-terminating evaluations

$$\forall clock\ env\ exp. \exists res. (\text{Some } clock, env) \vdash exp \Downarrow res$$

Idea: add logical clock

Big-step semantics:

- has an optional **clock** component
- **clock 'ticks'** decrement every function application
- when **clock hits zero**, execution stops with a **TimeOut**

Why do this?

because now big-step semantics describes both terminating and non-terminating evaluations

for every clock env exp

there is a result

either: Result
or TimeOut

$\forall \text{clock env exp}. \exists \text{res}. (\text{Some clock, env}) \vdash \text{exp} \Downarrow \text{res}$

produced by the semantics

Divergence

Evaluation diverges if

$\forall clock. (\text{Some } clock, env) \vdash exp \Downarrow \text{TimeOut}$

Divergence

Evaluation diverges if

$\forall clock. (\text{Some } clock, env) \vdash exp \Downarrow \text{TimeOut}$

for all clock values

TimeOut happens

Divergence

Evaluation diverges if

$\forall clock. (\text{Some } clock, env) \vdash exp \Downarrow \text{TimeOut}$

for all clock values

TimeOut happens

Compiler correctness, proved forwards:

$env \vdash exp \Downarrow val \implies$

“the code for exp is installed in bs etc.” \implies

$\exists bs'. bs \rightarrow^* bs' \wedge$ “ bs' contains val ”

Divergence

Evaluation diverges if

$\forall clock. (\text{Some } clock, env) \vdash exp \Downarrow \text{TimeOut}$

for all clock values

TimeOut happens

Compiler correctness, proved forwards:

$env \vdash exp \Downarrow val \implies$

“the code for exp is installed in bs etc.” \implies

$\exists bs'. bs \rightarrow^* bs' \wedge$ “ bs' contains val ”

bytecode has clock

... that stays in sync with CakeML clock

Divergence

Evaluation diverges if

$\forall clock. (\text{Some } clock, env) \vdash exp \Downarrow \text{TimeOut}$

for all clock values

TimeOut happens

Compiler correctness, proved forwards:

$env \vdash exp \Downarrow val \implies$

“the code for exp is installed in bs etc.” \implies

$\exists bs'. bs \rightarrow^* bs' \wedge$ “ bs' contains val ”

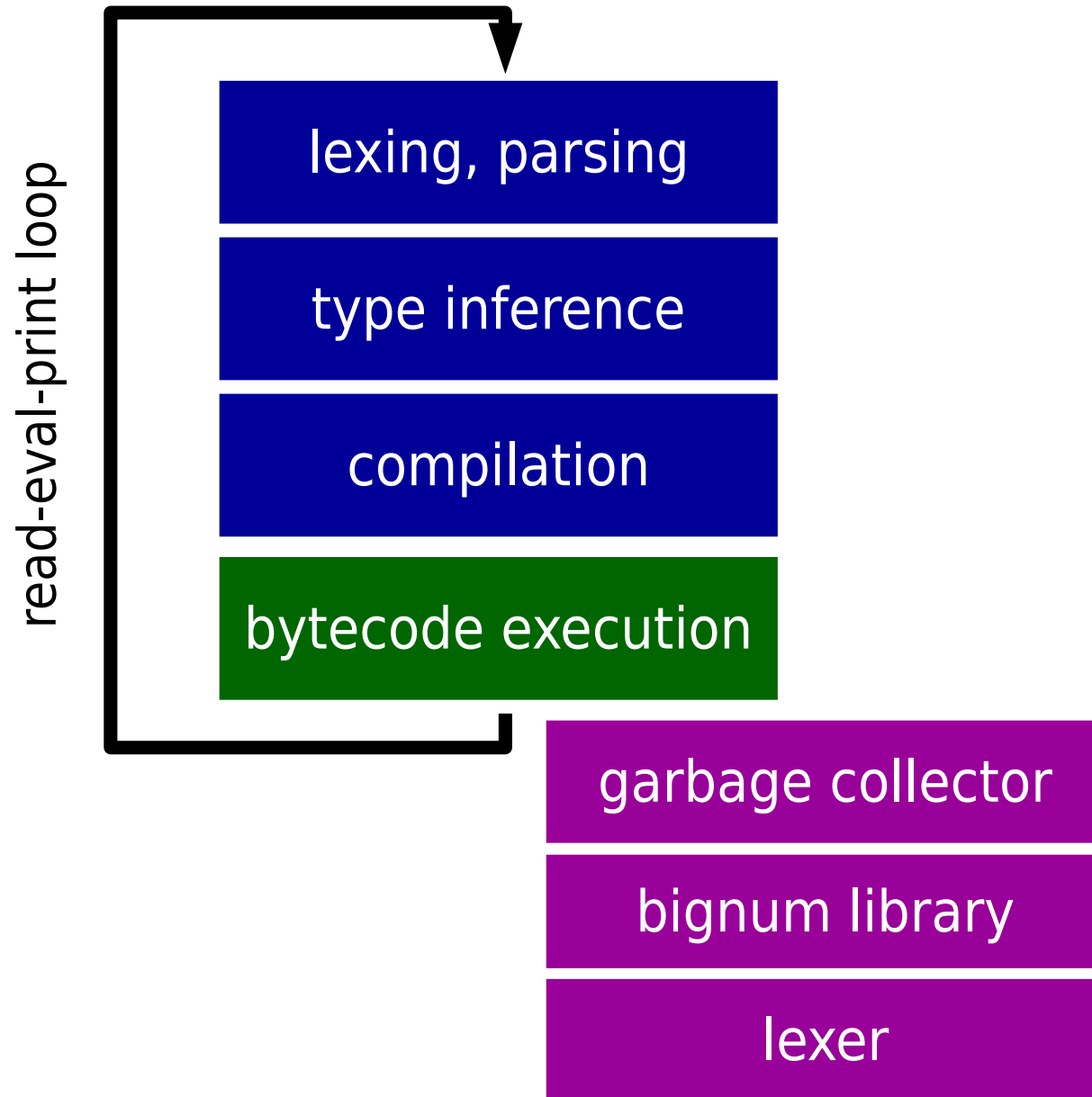
bytecode has clock

... that stays in sync with CakeML clock

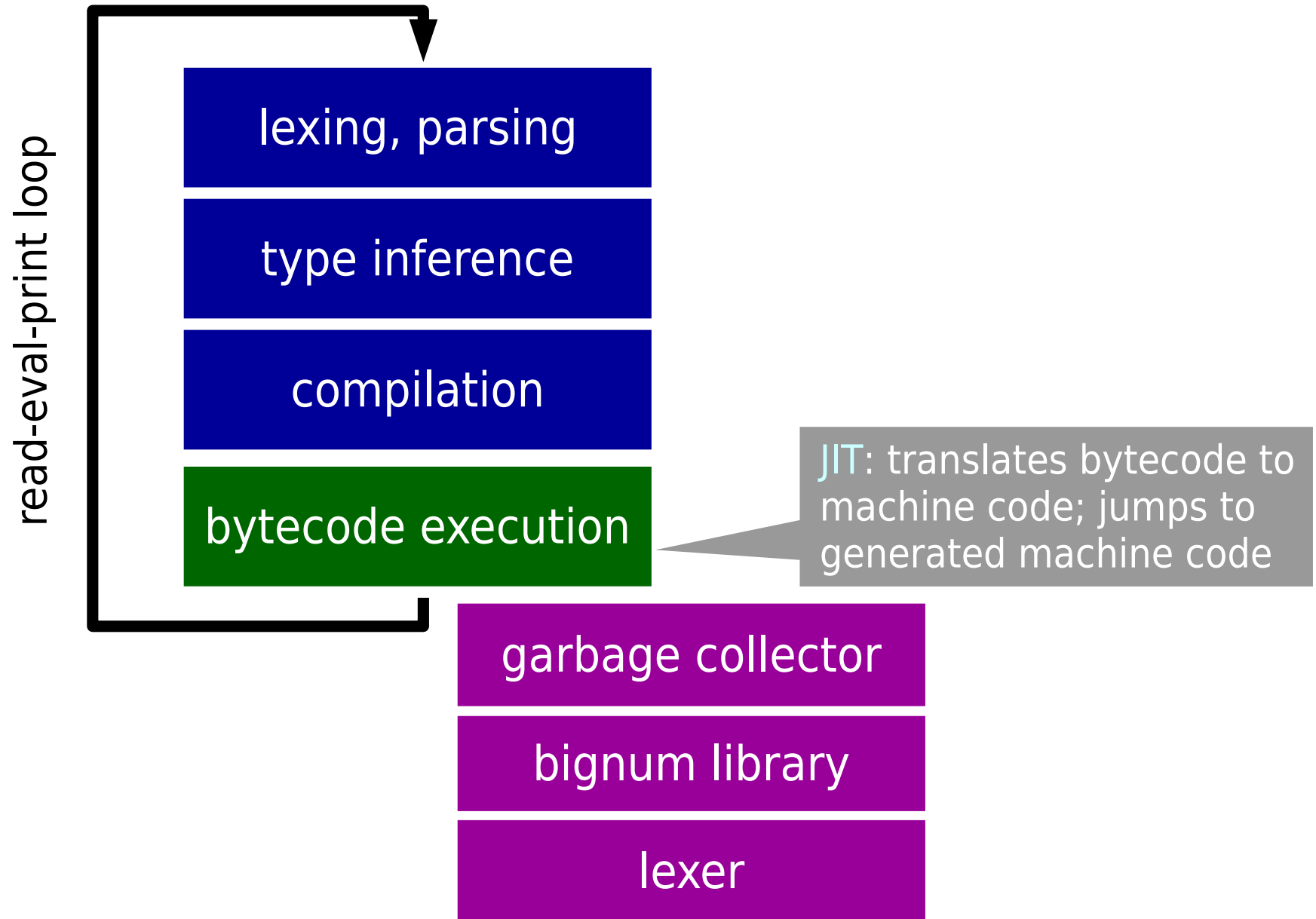
Theorem: **bytecode diverges** if and only if **CakeML evaluation diverges**

Step 3: production of **verified x86-64 code**

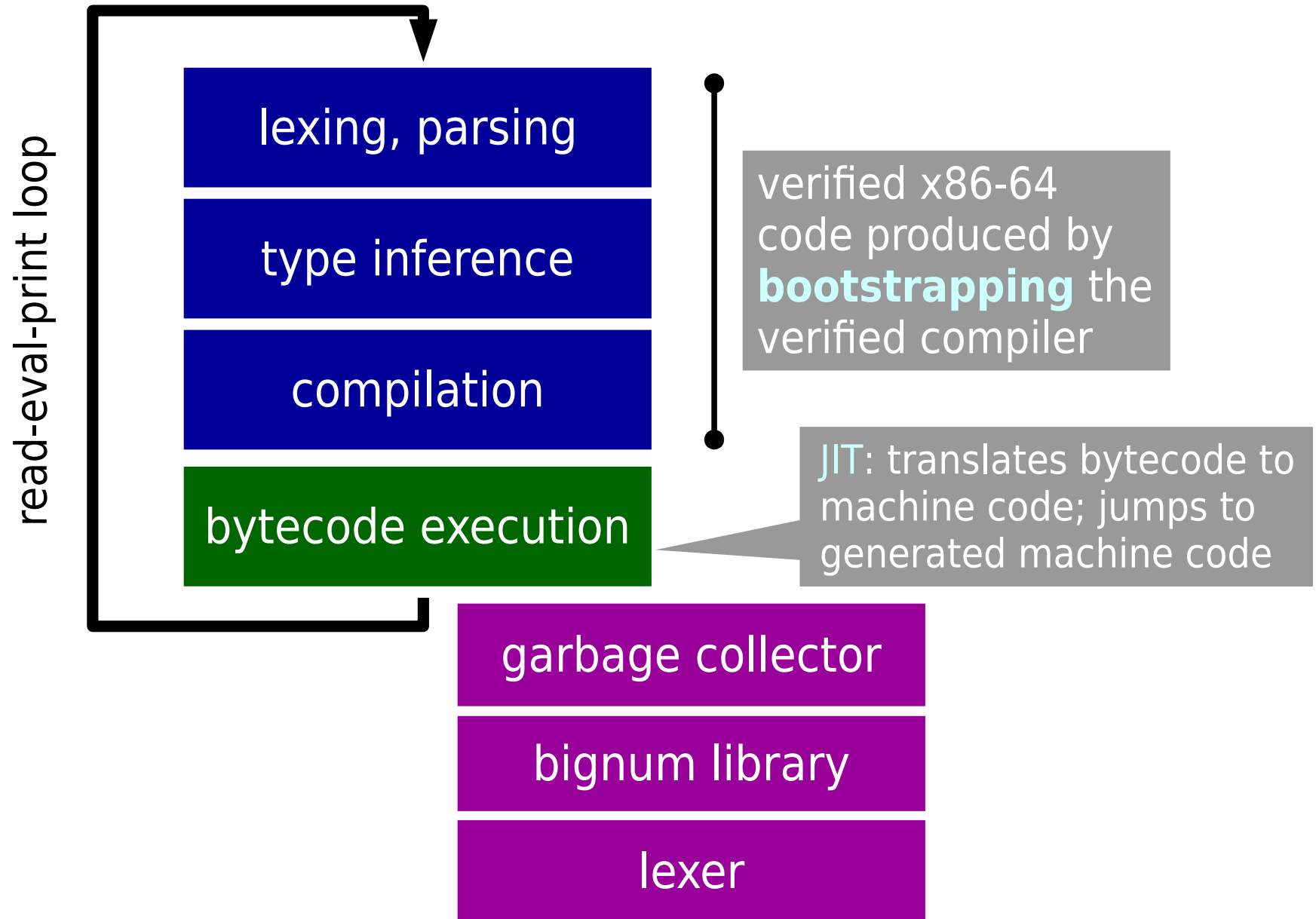
Verified Implementation in x86-64



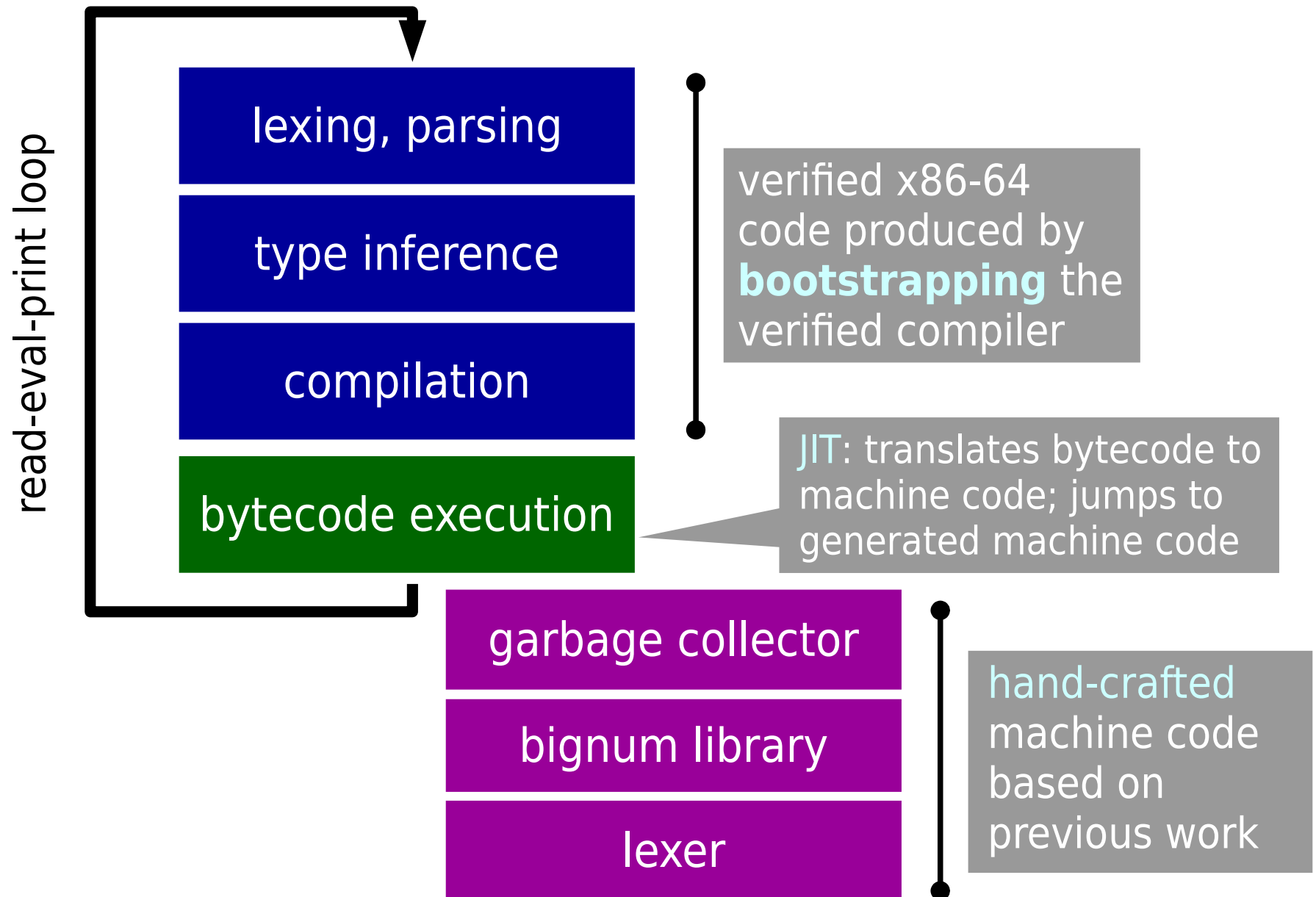
Verified Implementation in x86-64



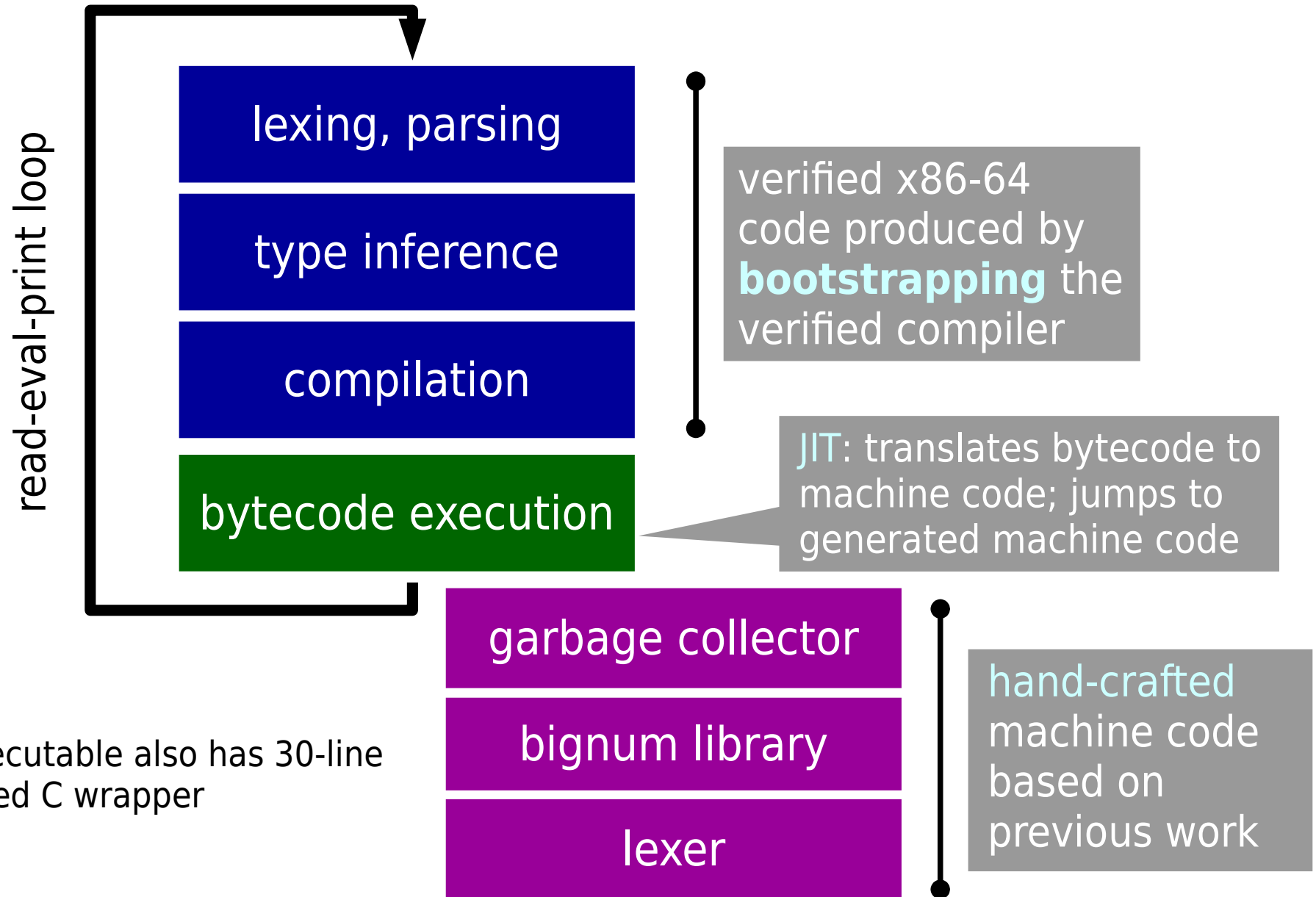
Verified Implementation in x86-64



Verified Implementation in x86-64



Verified Implementation in x86-64



JIT Translation of Bytecode

Specification

Implement each bytecode instruction in x86-64.

x86 semantics from previous work [POPL 2009, 2010].

Implementation

x64 i Pop = [0x48, 0x58], ..., x64 i Return = ...,

x64_code i (x::xs) = x64 i x @ x64_code (i+...) xs

Correctness

$bs \rightarrow bs' \implies$

temporal $\{(base, \text{x64_code } 0 \text{ } bs.\text{code})\}$

$(\text{now } (bc_heap \text{ } bs \text{ } (base, aux))) \implies$

$\text{later } (\text{now } (bc_heap \text{ } bs' \text{ } (base, aux)))$

$\vee \text{now } (\text{out_of_memory_error } aux))$

JIT Translation of Bytecode

Specification

Implement each bytecode instruction in x86-64.

x86 semantics from previous work [POPL 2009, 2010].

Implementation

x64 i Pop = [0x48, 0x58], ..., x64 i Return = ...,

x64_code i (x::xs) = x64 i x @ x64_code (i+...) xs

synthesised definition
from assembly [SBMF
2008]

Correctness

$bs \rightarrow bs' \implies$

temporal $\{(base, x64_code\ 0\ bs.code)\}$

$(now\ (bc_heap\ bs\ (base, aux)) \implies$

later $(now\ (bc_heap\ bs'\ (base, aux))$

$\vee\ now\ (out_of_memory_error\ aux)))$

JIT Translation of Bytecode

Specification

Implement each bytecode instruction in x86-64.

x86 semantics from previous work [POPL 2009, 2010].

Implementation

x64 i Pop = [0x48, 0x58], ..., x64 i Return = ...,

x64_code i (x::xs) = x64 i x @ x64_code (i+...) xs

synthesised definition
from assembly [SBMF
2008]

Correctness

$$bs \rightarrow bs' \implies$$
$$\text{temporal } \{(base, \text{x64_code } 0 \text{ } bs.\text{code})\}$$
$$(\text{now } (\text{bc_heap } bs \text{ } (base, aux)) \rightarrow$$
$$\text{later } (\text{now } (\text{bc_heap } bs' \text{ } (base, aux))$$
$$\vee \text{now } (\text{out_of_memory_error } aux)))$$

bytecode heap
invariant / memory
abstraction

Bootstrapping the verified compiler

Bootstrapping

lexing, parsing

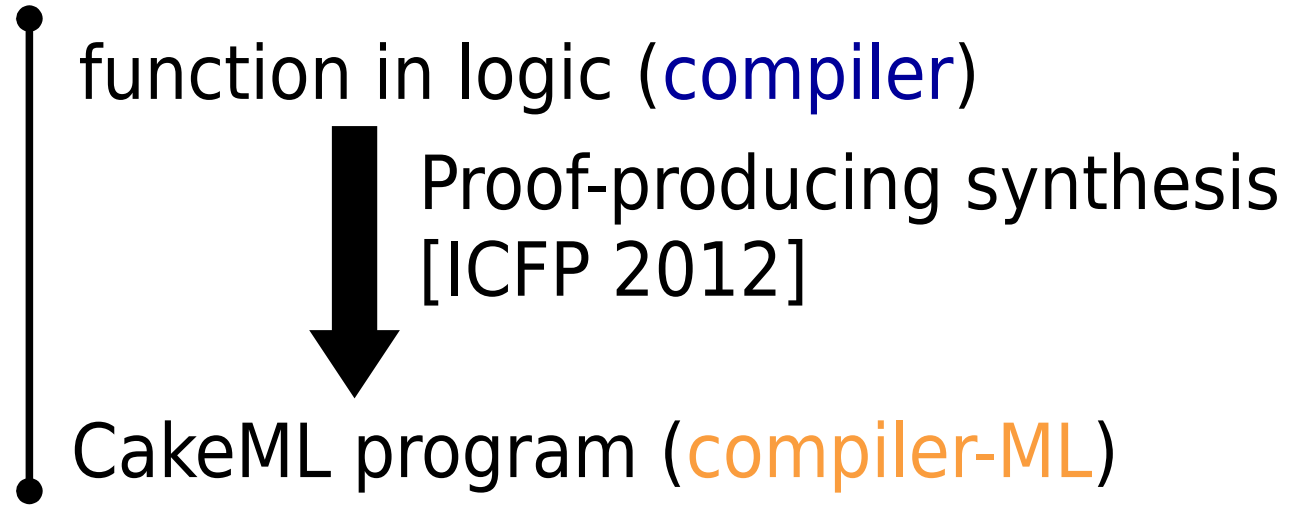
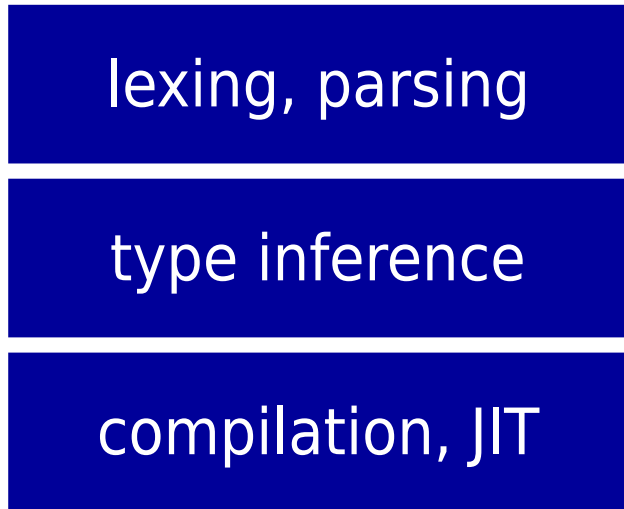
type inference

compilation, JIT

function in logic (**compiler**)

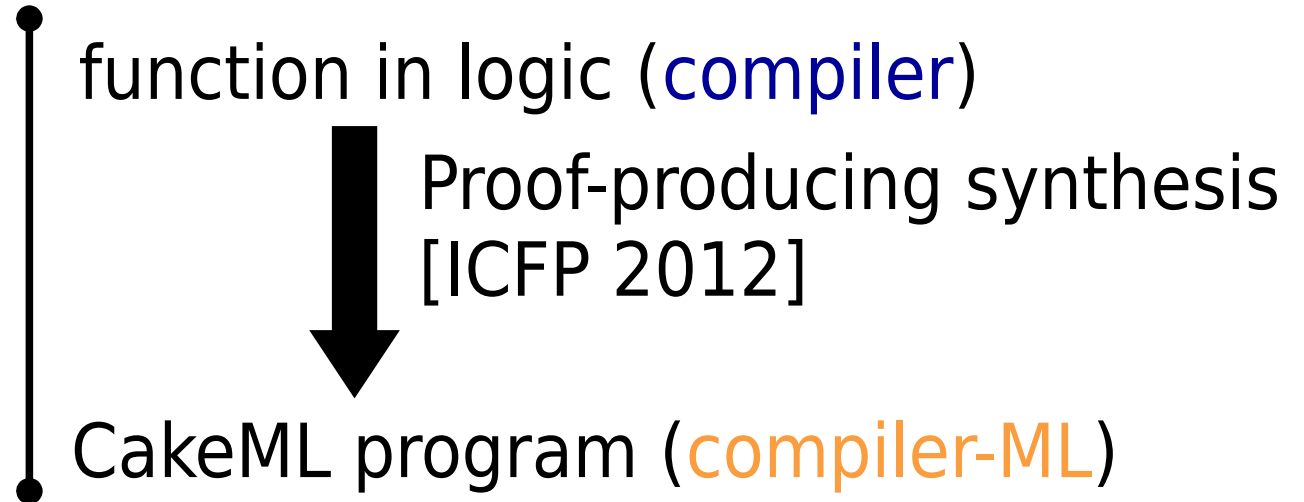
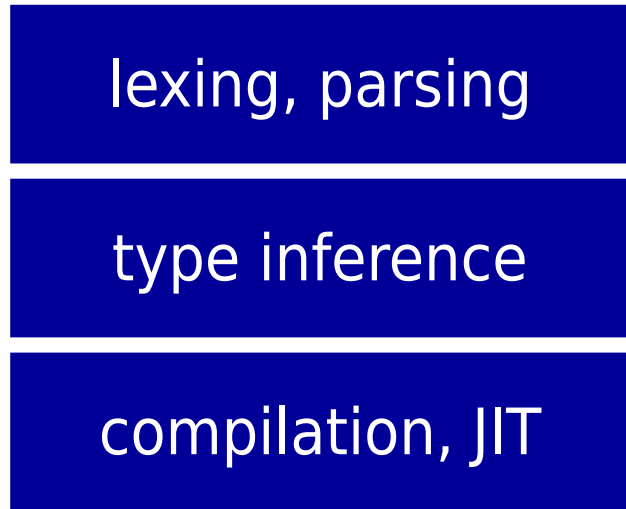


Bootstrapping



⊢ **compiler-ML** implements **compiler**

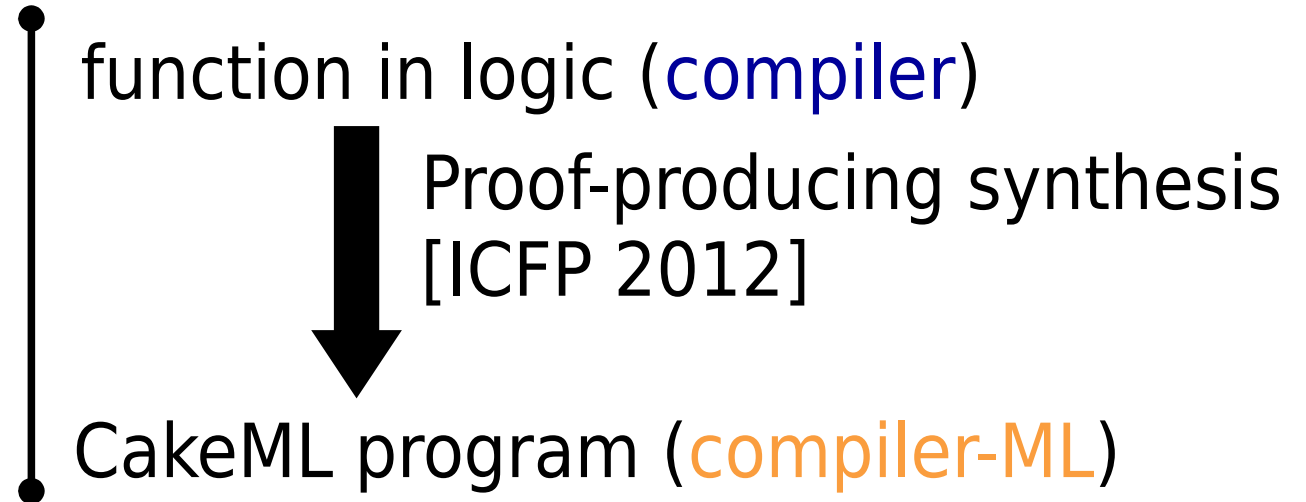
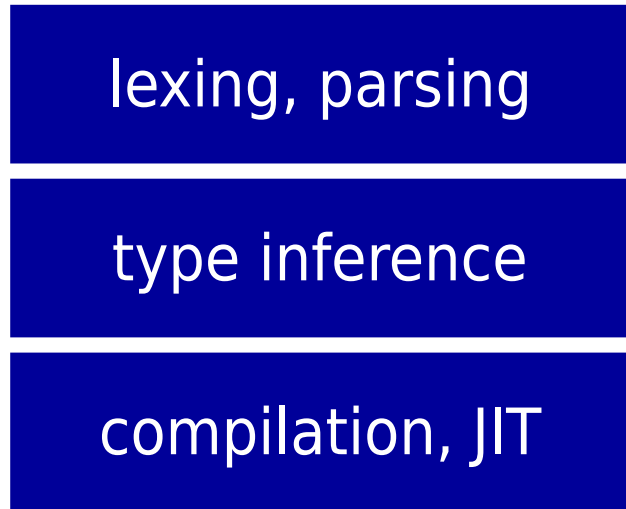
Bootstrapping



⊢ **compiler-ML** implements **compiler**

⊢ **compiler** (**compiler-ML**) = **compiler-x86** by evaluation in the logic

Bootstrapping

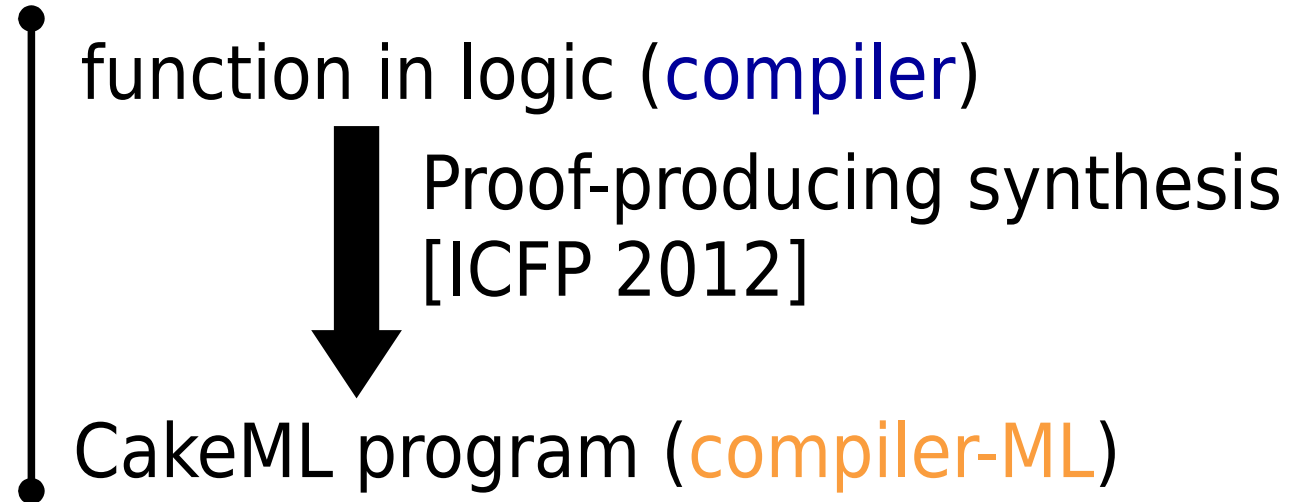
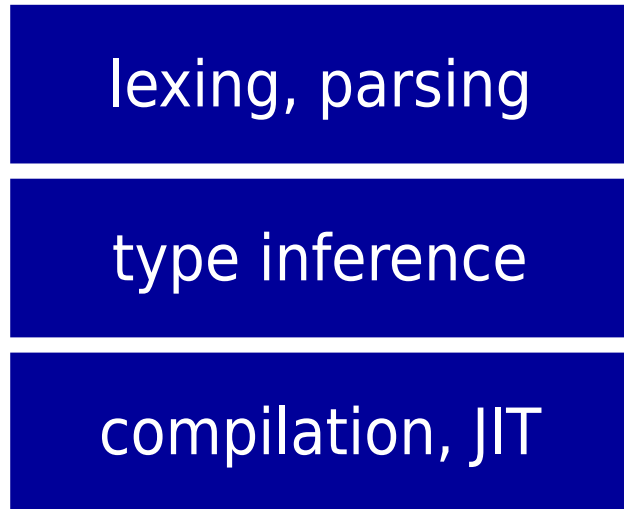


\vdash **compiler-ML** implements **compiler**

\vdash **compiler** (**compiler-ML**) = **compiler-x86** by evaluation in the logic

$\vdash \forall \text{code. (compiler code) implements code}$ by compiler correctness

Bootstrapping



\vdash **compiler-ML** implements **compiler**

\vdash **compiler** (**compiler-ML**) = **compiler-x86** by evaluation in the logic

$\vdash \forall \text{code}. (\text{compiler code})$ implements code by compiler correctness

Theorem: \vdash **compiler-x86** implements **compiler**

Top-Level Correctness Theorem

Specification

$\text{REPL}_s l \text{ inp } res$

Correctness

temporal entire_machine_code_implementation

(now (init *inp aux*) \Rightarrow

later (($\exists l \text{ res. repl_returns (out } res) \text{ aux} \wedge$
($\text{REPL}_s l \text{ inp } res \wedge \text{terminates } res$))

\vee

($\exists l \text{ res. repl_diverged (out } res) \text{ aux} \wedge$
($\text{REPL}_s l \text{ inp } res \wedge \neg \text{terminates } res$))

\vee

now (out_of_memory_error *aux*)))

Top-Level Correctness Theorem

Specification

$\text{REPL}_s l \text{ inp } res$

list of output strings ending in termination or divergence

input string

Correctness

temporal entire_machine_code_implementation

$(\text{now } (\text{init } \text{inp } aux) \Rightarrow$

$\text{later } ((\exists l \text{ res. repl_returns } (\text{out } res) aux \wedge$
 $(\text{REPL}_s l \text{ inp } res \wedge \text{terminates } res)))$

\vee

$(\exists l \text{ res. repl_diverged } (\text{out } res) aux \wedge$
 $(\text{REPL}_s l \text{ inp } res \wedge \neg \text{terminates } res)))$

\vee

$\text{now } (\text{out_of_memory_error } aux)))$

Numbers and Summary

Numbers

Performance

Slow compared with state-of-the-art, but not hopeless:

	compiled OCaml	Poly/ML	CakeML	
Fibonacci	7.9	4.6	2.2	
Quicksort	3.1	10.0	0.6	
Batched queue	2.0	12.9	0.4	
Binary tree	4.3	5.6	0.6	... details in paper

Effort

~70,000 lines of HOL4 proof script.

Biggest slice for typechecked-to-bytecode compiler (30%).

A couple of man-years, but builds on lots of previous work.

Result

??? instructions of verified x86-64 machine code implementing a CakeML interactive compiler and read-eval-print loop.

Summary

Contributions:

A verified implementation of ML.

- a development where everything fits together.

New lightweight method for **divergence preservation**.

First **bootstrapping** of a formally **verified compiler**.

For more:

<https://cakeml.org>

Together we can build a fully verified ecosystem.

Questions?