CAKEML

A Verified Implementation of ML

Ramana Kumar, Magnus Myreen, Michael Norrish, Scott Owens

CakeML Logo: Trần Tiến Dũng

Dimensions of Compiler Verification

Dimensions of Compiler Verification

Source Code

Abstract Syntax

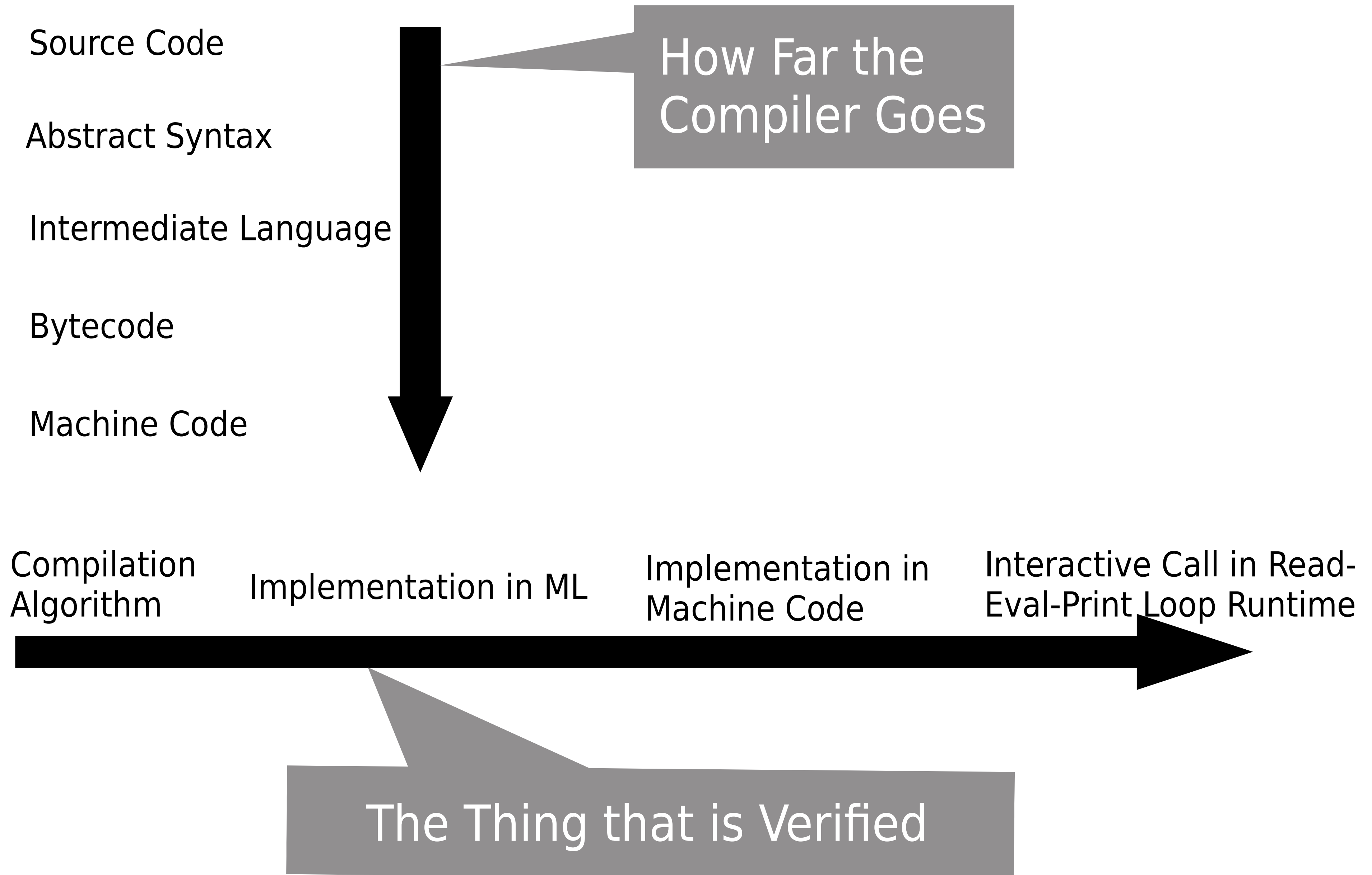
Intermediate Language

Bytecode

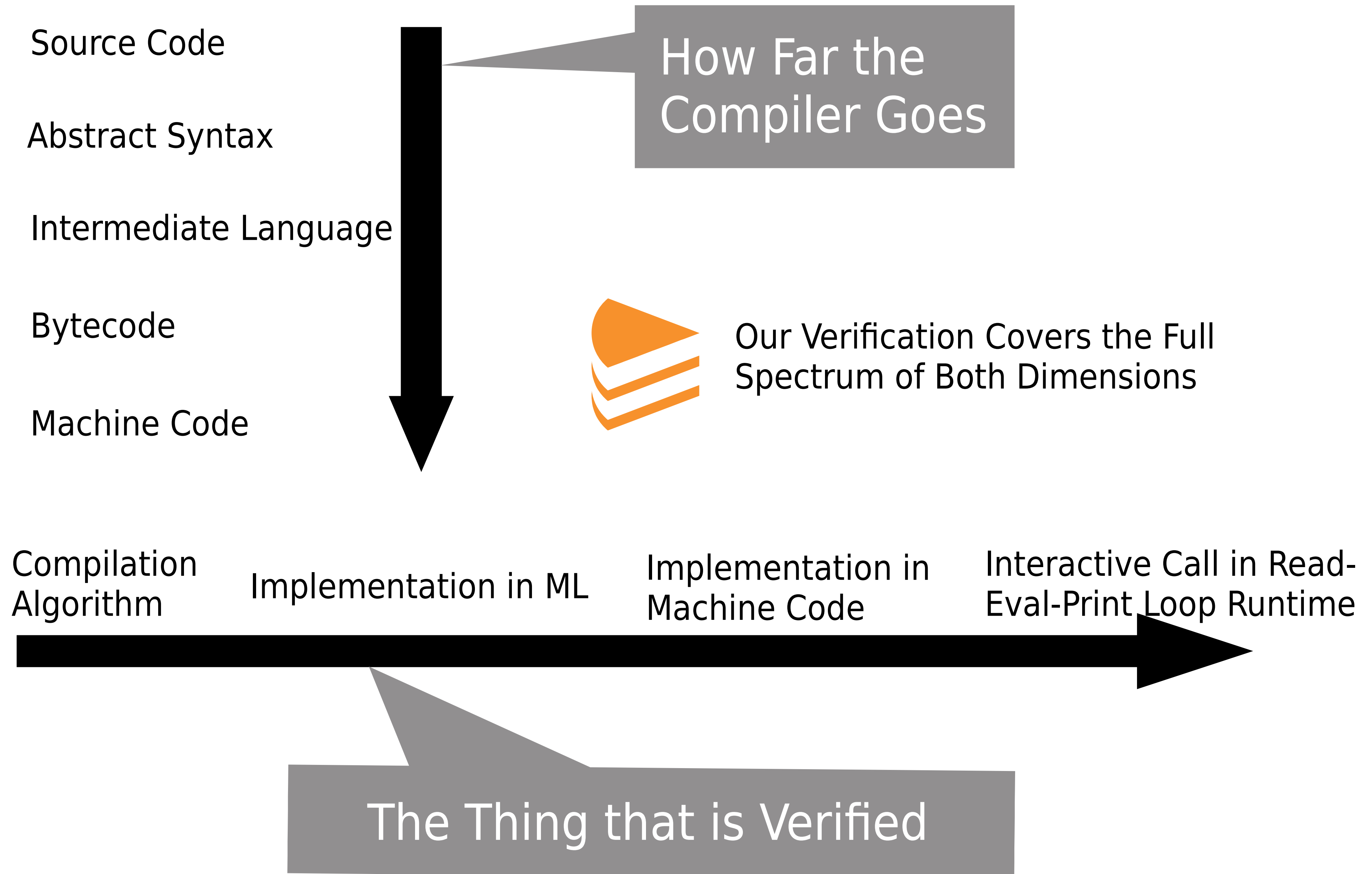
Machine Code

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 or functors

The Size of the Language

CakeML, the language

= Standard ML without I/O or 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

Proof Techniques

Interactive Proof

x86 Synthesis

ML Synthesis

Bootstrapping

including new inductive approach to divergence preservation

Contributions

Artefacts

Specifications

Verified Algorithms

Verified Executable

Proof Techniques

Interactive Proof

x86 Synthesis

ML Synthesis

Bootstrapping

including new inductive approach to divergence preservation

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

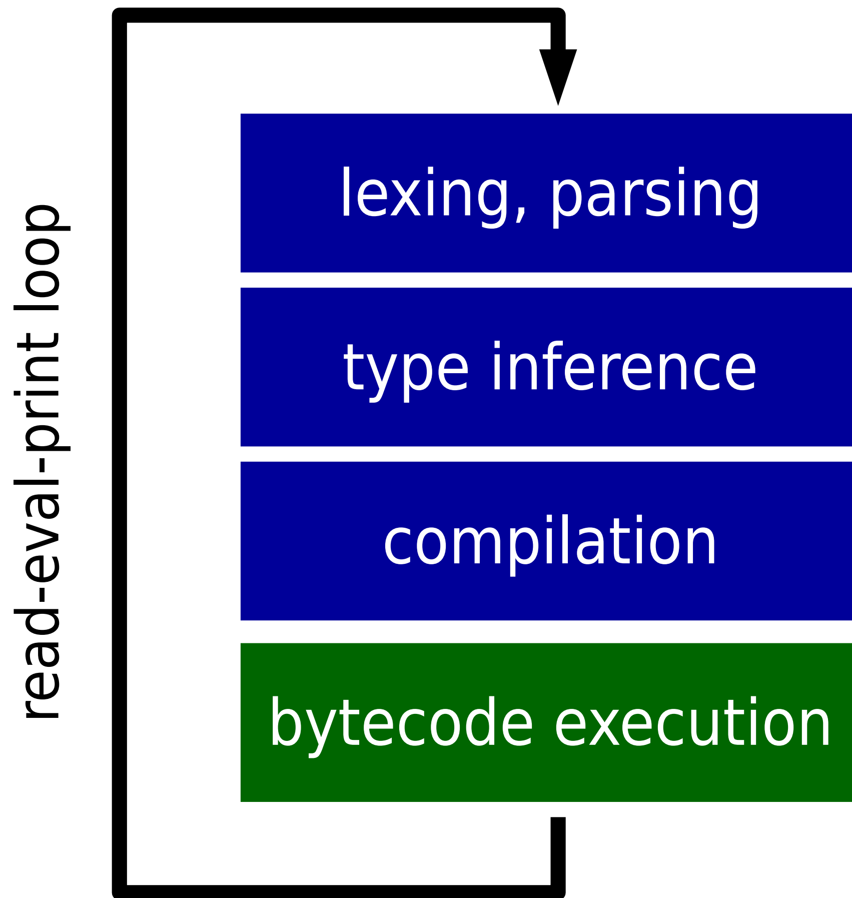
Small-step CEK machine

for type soundness proof, and defining divergence explicitly

Read-eval-print loop semantics

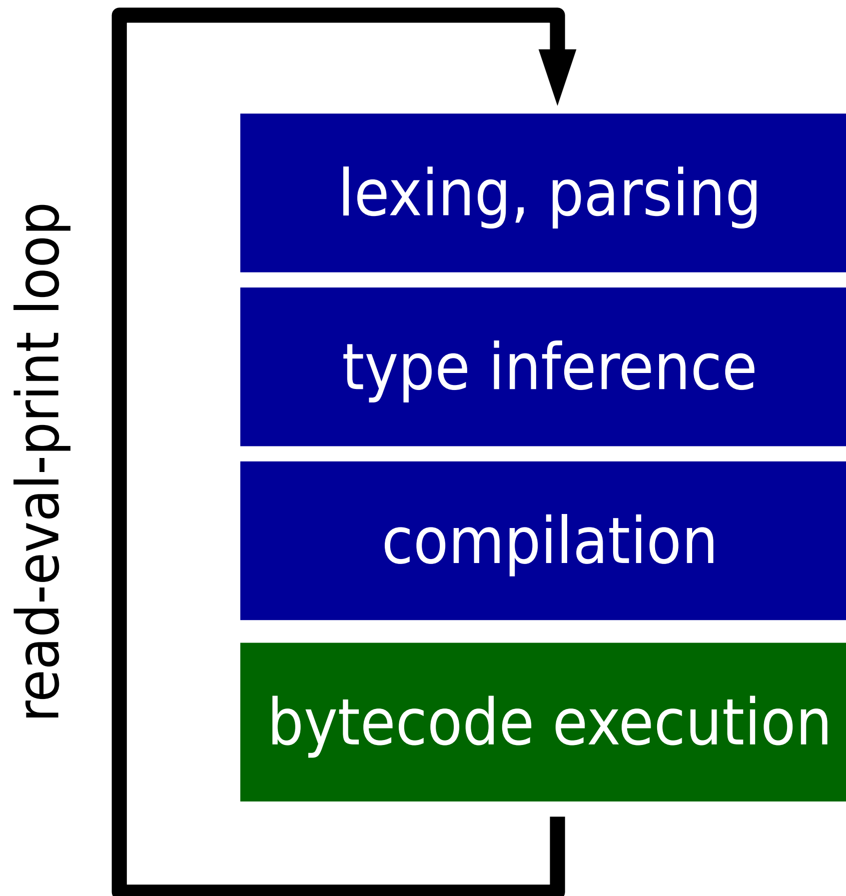
relates **input string** to list of **output strings**
ending in termination or divergence

Functional Implementation



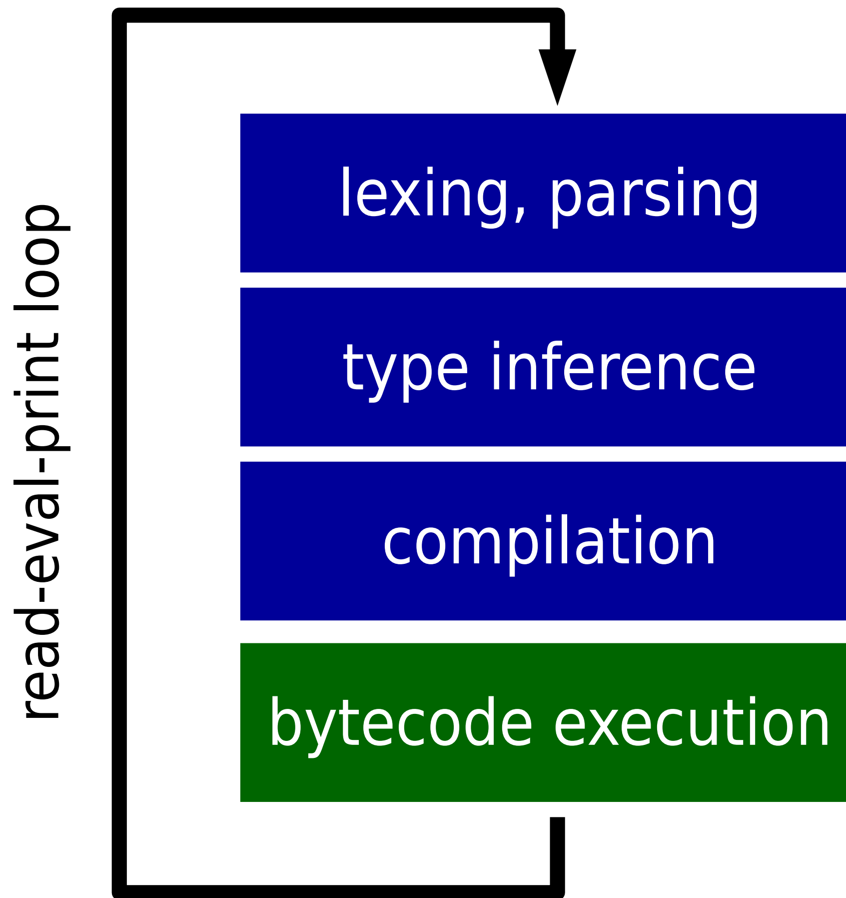
Functional Implementation

Defined as a function in the logic of HOL4



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:

$$
\begin{array}{lcl}
bc_inst & ::= & \mathsf{Stack}\ bc_stack_op \mid \mathsf{PushExc} \mid \mathsf{PopExc} \\
& \mid & \mathsf{Return} \mid \mathsf{CallPtr} \mid \mathsf{Call}\ loc \\
& \mid & \mathsf{PushPtr}\ loc \mid \mathsf{Jump}\ loc \mid \mathsf{JumpIf}\ loc \\
& \mid & \mathsf{Ref} \mid \mathsf{Deref} \mid \mathsf{Update} \mid \mathsf{Print} \mid \mathsf{PrintC\ char} \\
& \mid & \mathsf{Label}\ n \mid \mathsf{Tick} \mid \mathsf{Stop} \\
bc_stack_op & ::= & \mathsf{Pop} \mid \mathsf{Pops}\ n \mid \mathsf{Shift}\ n\ n \mid \mathsf{PushInt\ int} \\
& \mid & \mathsf{Cons}\ n\ n \mid \mathsf{El}\ n \mid \mathsf{TagEq}\ n \mid \mathsf{IsBlock}\ n \\
& \mid & \mathsf{Load}\ n \mid \mathsf{Store}\ n \mid \mathsf{LoadRev}\ n \\
& \mid & \mathsf{Equal} \mid \mathsf{Less} \mid \mathsf{Add} \mid \mathsf{Sub} \mid \mathsf{Mult} \mid \mathsf{Div} \mid \mathsf{Mod}
\end{array}
$$

Small-step semantics. Values and States:

$$
\begin{array}{lcl}
bc_value & ::= & \mathsf{Number\ int} \mid \mathsf{RefPtr}\ n \mid \mathsf{Block}\ n\ bc_value^* \\
& \mid & \mathsf{CodePtr}\ n \mid \mathsf{StackPtr}\ n \\
bc_state & ::= \{ & \mathsf{stack} : bc_value^*; \mathsf{refs} : n \mapsto bc_value; \\
& & \mathsf{code} : bc_inst^*;\ \mathsf{pc} : n; \mathsf{handler} : n; \\
& & \mathsf{output} : \mathsf{string};\ \mathsf{names} : n \mapsto \mathsf{string}; \\
& & \mathsf{clock} : n^? \quad \}
\end{array}
$$

Semantics of bytecode execution

Sample rules:

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

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

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

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:

$$env \vdash exp \Downarrow val \implies$$
$$\text{"the code for } exp \text{ is installed in } bs \text{ etc."} \implies$$
$$\exists bs'.\ bs \to^* bs' \wedge \text{"}bs' \text{ contains } val\text{"}$$

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

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

$$\text{“the code for } exp \text{ is installed in } bs \text{ etc.”} \implies$$

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

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

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

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

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

produced by the semantics

Divergence

Evaluation diverges if

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

Divergence

Evaluation diverges if

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

for all clock values

TimeOut happens

Divergence

Evaluation diverges if

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

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

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

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." $\implies$

$\exists bs'.\ bs \to^* bs' \wedge$ "bs' contains val"

bytecode has clock

... that stays in sync with CakeML clock

Divergence

Evaluation diverges if

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

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." $\implies$

$\exists bs'.\ bs \to^{*} bs' \wedge$ "bs' contains val"

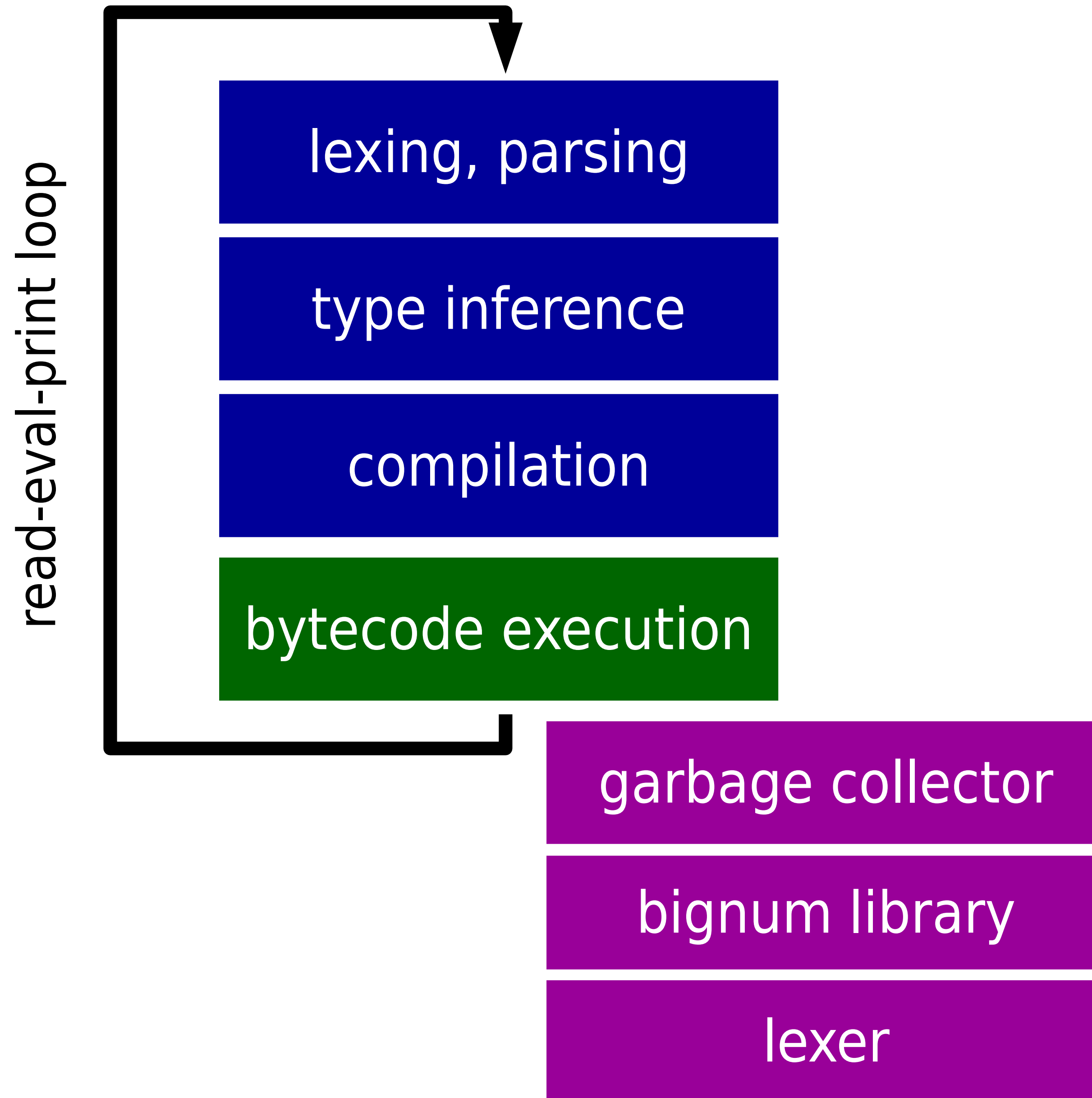
bytecode has clock

... that stays in sync with CakeML clock

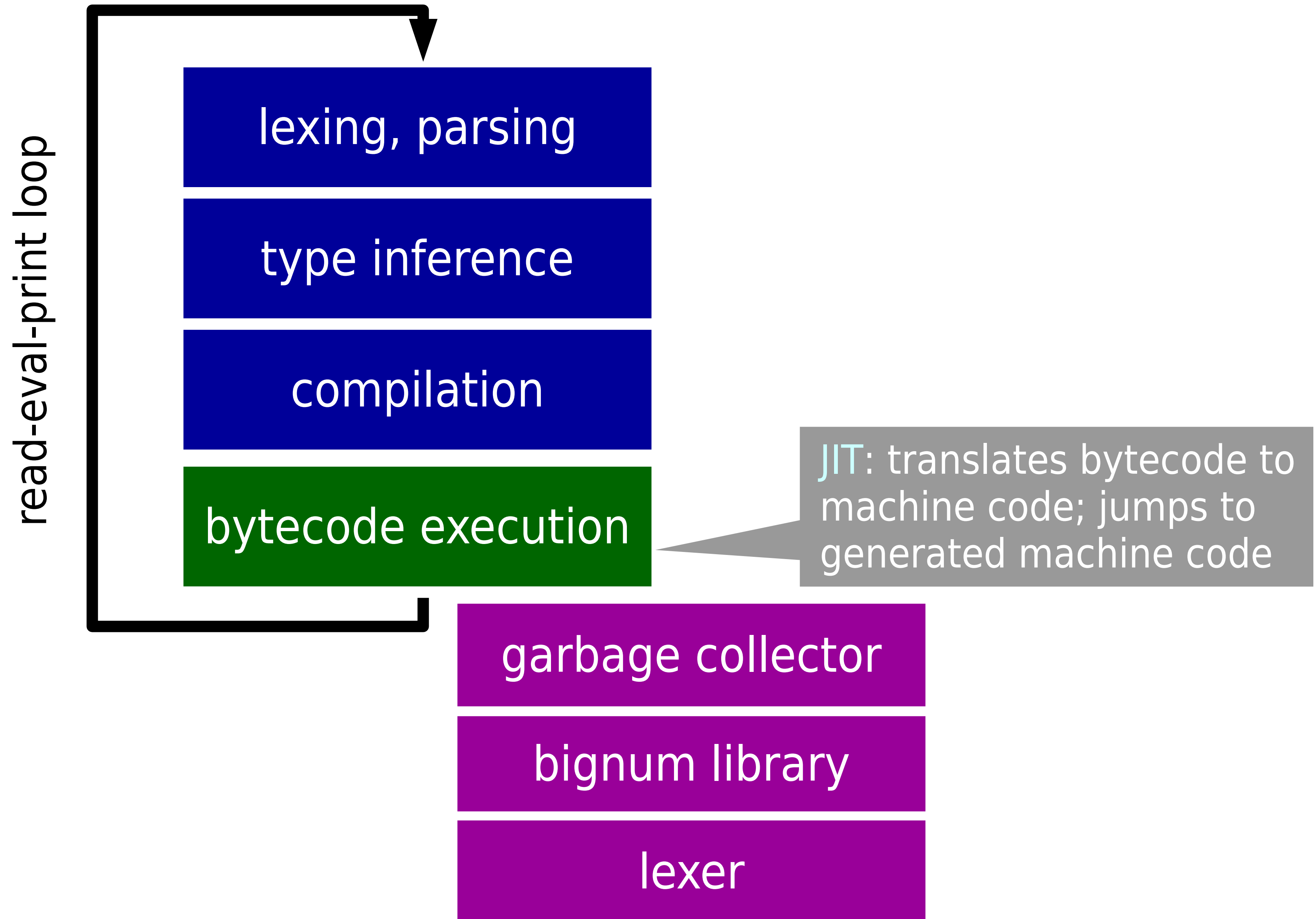
Theorem: bytecode diverges if and only if CakeML evaluation diverges

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

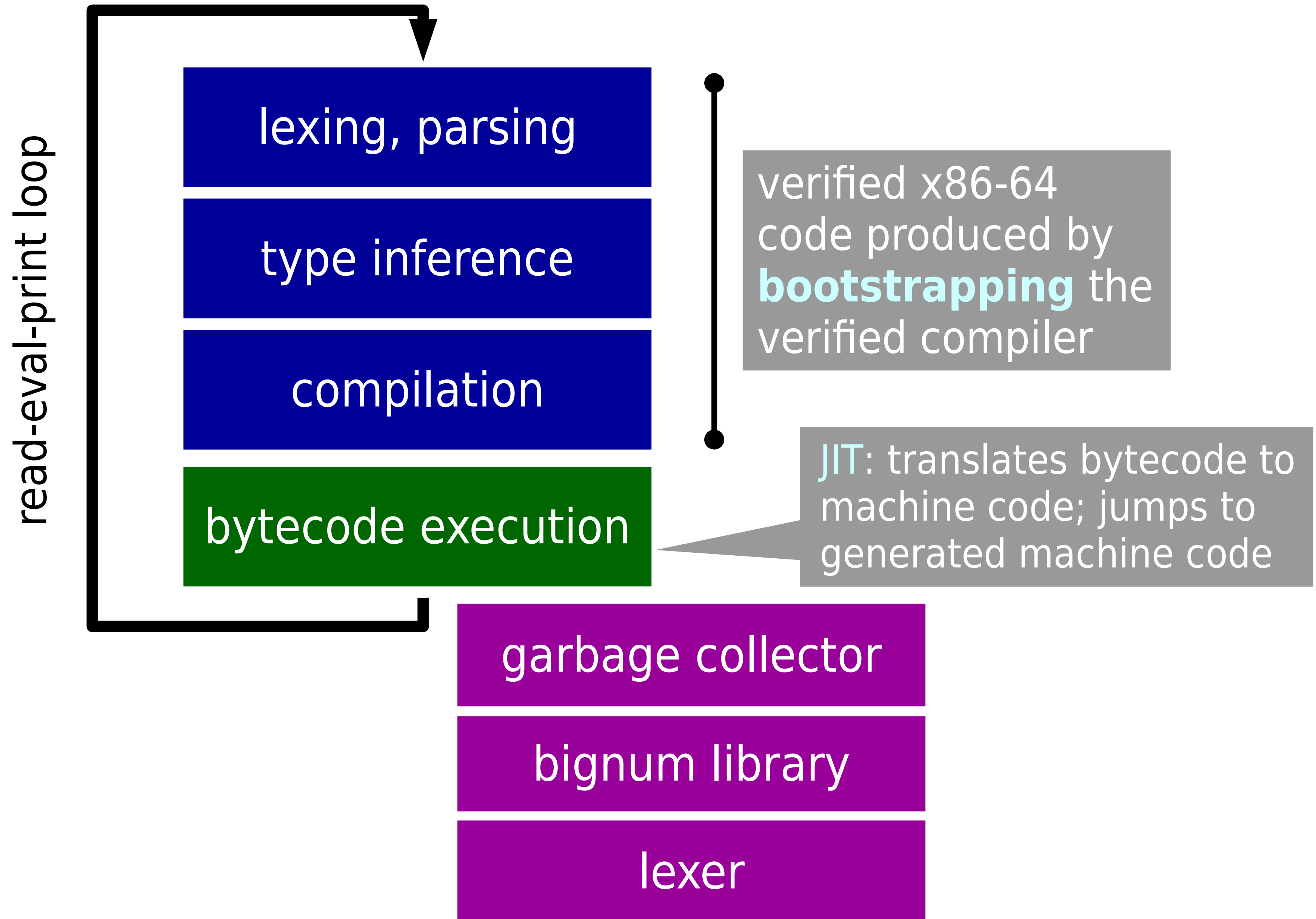
Verified Implementation in x86-64



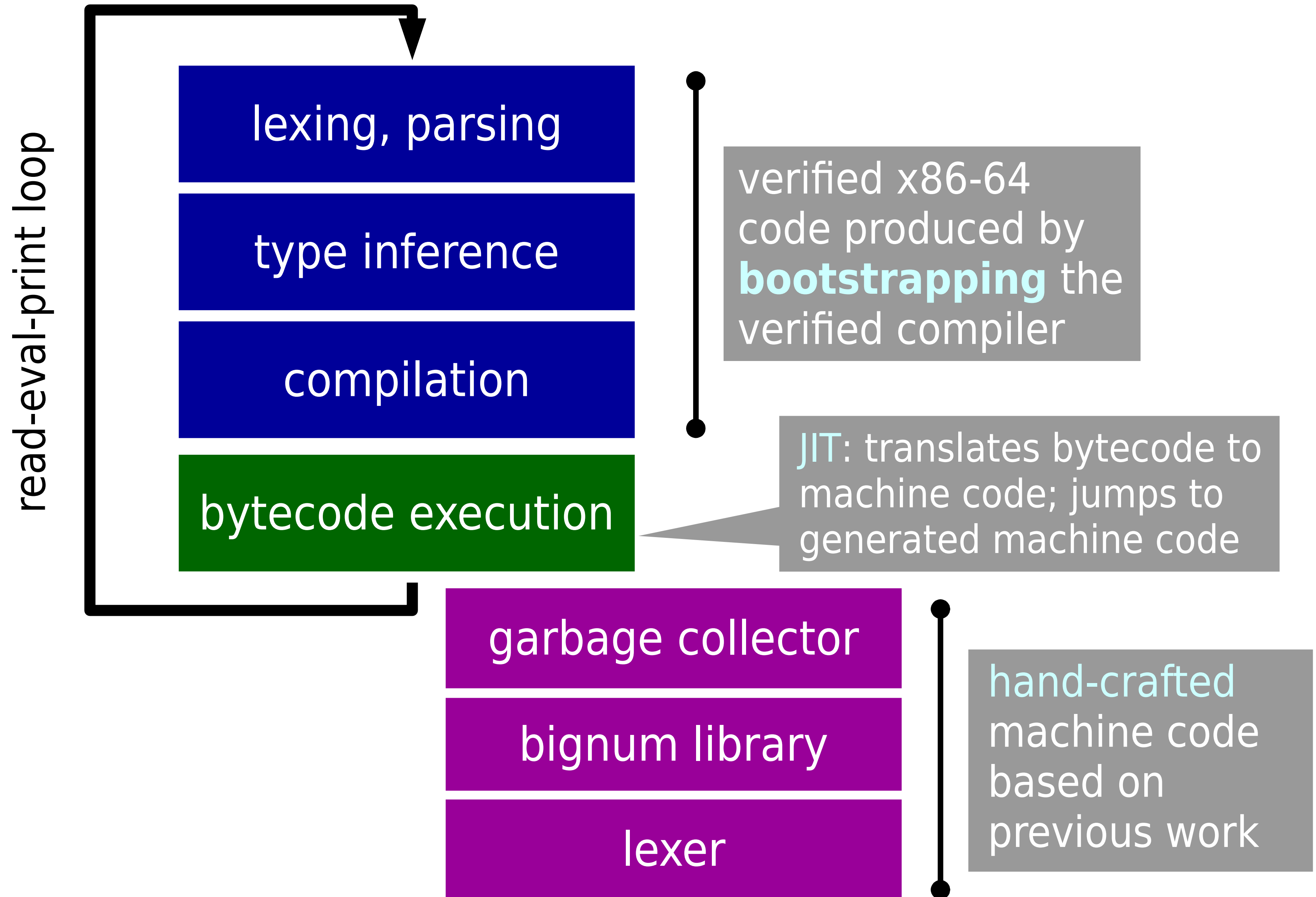
Verified Implementation in x86-64



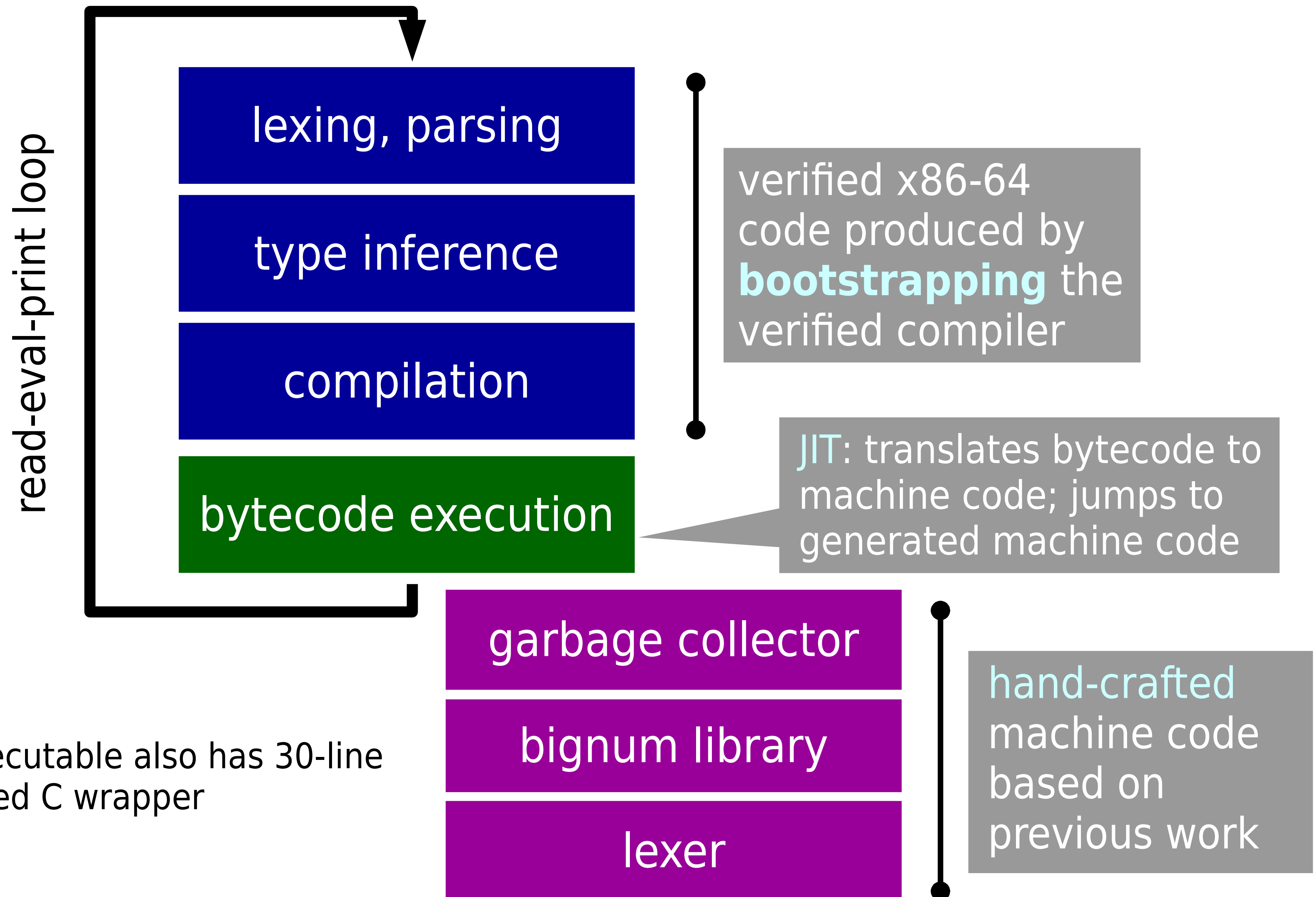
Verified Implementation in x86-64



Verified Implementation in x86-64



Verified Implementation in x86-64



Real executable also has 30-line unverified C wrapper

JIT

Bootstrapping

Top-level Correctness Theorem

Numbers and Summary

Performance and Effort

<conclusion slide>
https://cakeml.org