Is the CakeML compiler secure?

Dagstuhl Seminar on Secure Compilation, May 2018

Active developers: Oskar Abrahamsson, Johannes Åman Pohjola, Hugo Férée, Anthony Fox, **Ramana Kumar**, Andreas Lööw, Alexander Mihajlovic, **Magnus Myreen**, **Michael Norrish**, **Scott Owens**, **Yong Kiam Tan**

Is the CakeML compiler secure?

This talk:

1. What is the CakeML compiler?
2. What has been proved about it? Is it secure?
3. Discussion: how could it be made more secure?

What is the CakeML compiler?

Aim: To be a realistic ML compiler that can be used for research and teaching (e.g. MSc students).

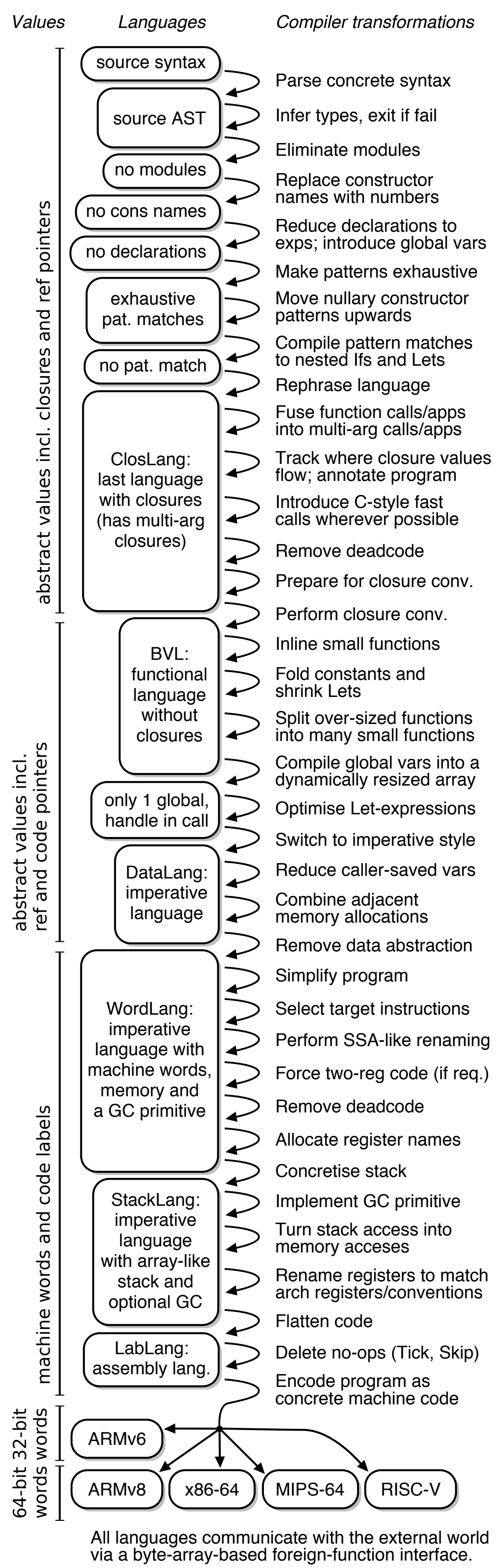
Realistic end-points:

no undefined behaviour

High-level ML source language that is convenient for program verification: no bounds on sizes and mathematical integers as default int type; type safe.

Output: concrete machine code (not assembly) with semantics that is as accurate as possible for commercial ISAs (but no concurrency).

Acceptable performance, bootstrapped compiler etc.



Implementation

Latest version:

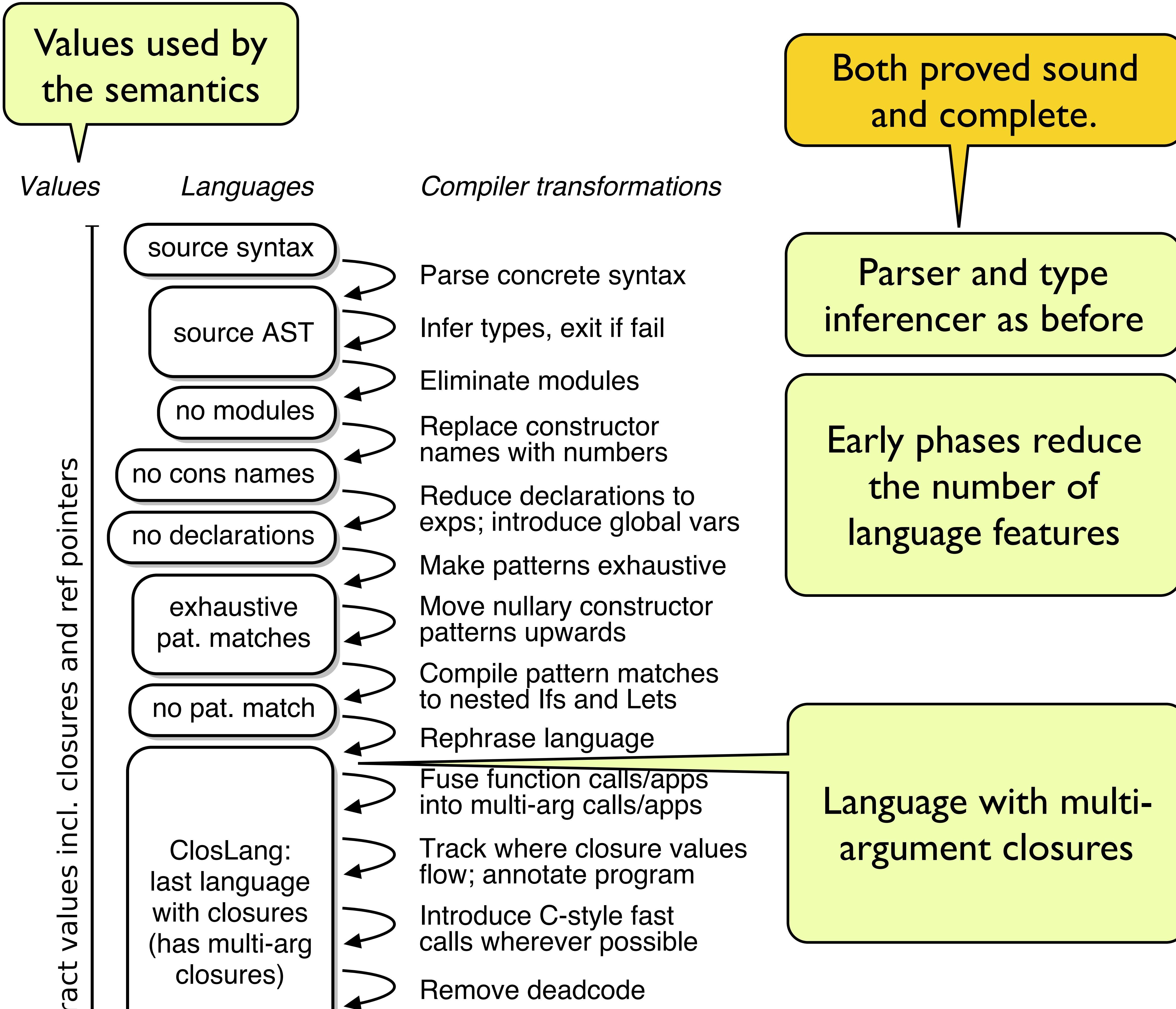
12 intermediate languages (ILs)

and many within-IL optimisations

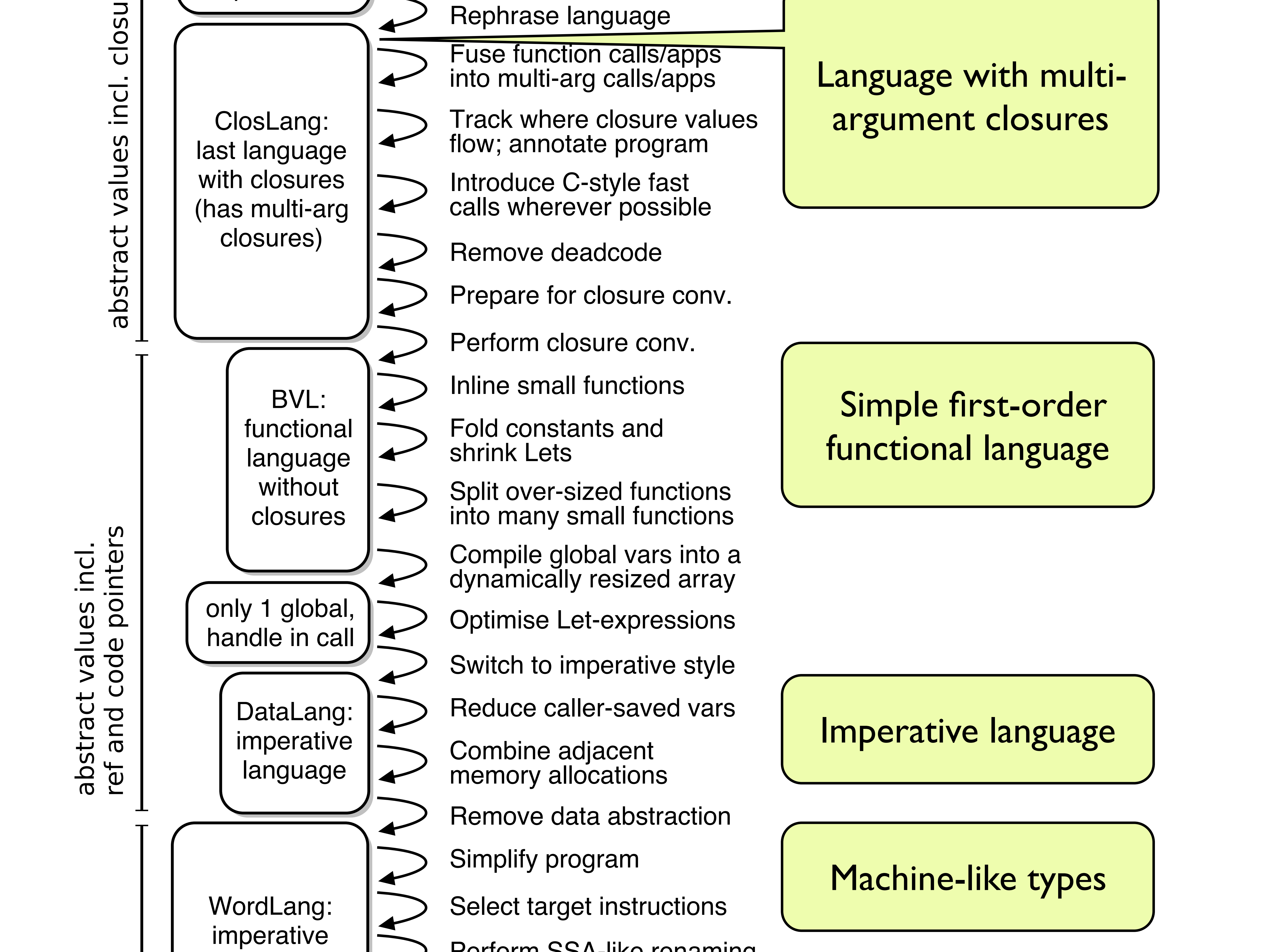
each IL at the right level of abstraction

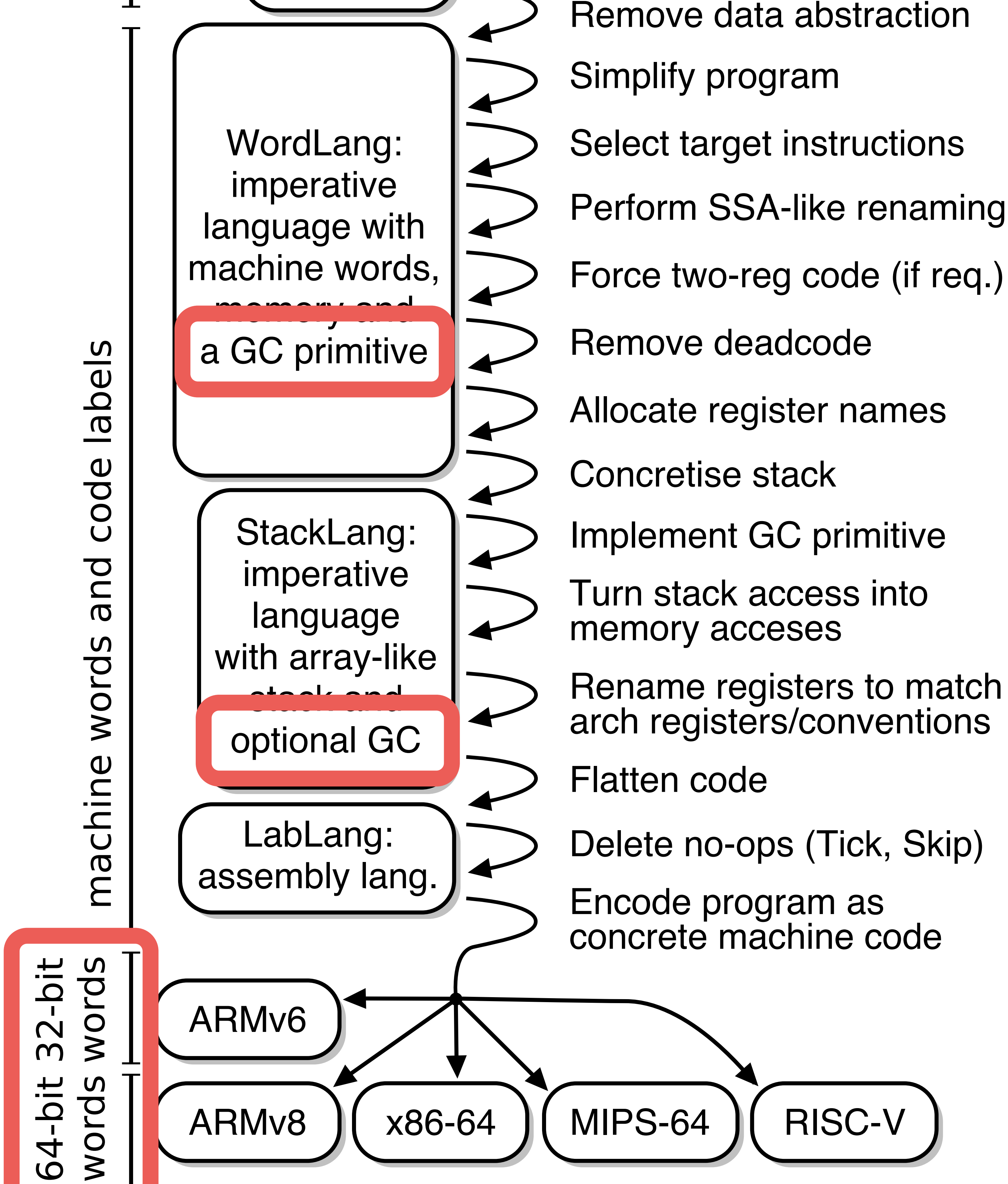
for the benefit of proofs and compiler implementation

Next slide zooms in



Values used by the semantics
Both proved sound and complete.
Values
Languages
Compiler transformations
ract values incl. closures and ref pointers
source syntax
Parse concrete syntax
source AST
Infer types, exit if fail
Eliminate modules
no modules
Replace constructor names with numbers
no cons names
Reduce declarations to exps; introduce global vars
no declarations
Make patterns exhaustive
exhaustive pat. matches
Move nullary constructor patterns upwards
Compile pattern matches to nested Ifs and Lets
no pat. match
Rephrase language
Fuse function calls/apps into multi-arg calls/apps
ClosLang: last language with closures (has multi-arg closures)
Track where closure values flow; annotate program
Introduce C-style fast calls wherever possible
Remove deadcode
Parser and type inferencer as before
Early phases reduce the number of language features
Language with multi-argument closures


abstract values incl. closu
ClosLang: last language with closures (has multi-arg closures)
Rephrase language
Fuse function calls/apps into multi-arg calls/apps
Track where closure values flow; annotate program
Introduce C-style fast calls wherever possible
Remove deadcode
Prepare for closure conv.
Perform closure conv.
Language with multi-argument closures
abstract values incl. ref and code pointers
BVL: functional language without closures
Inline small functions
Fold constants and shrink Lets
Split over-sized functions into many small functions
Simple first-order functional language
Compile global vars into a dynamically resized array
only 1 global, handle in call
Optimise Let-expressions
Switch to imperative style
DataLang: imperative language
Reduce caller-saved vars
Combine adjacent memory allocations
Remove data abstraction
Imperative language
WordLang: imperative
Simplify program
Select target instructions
Machine-like types



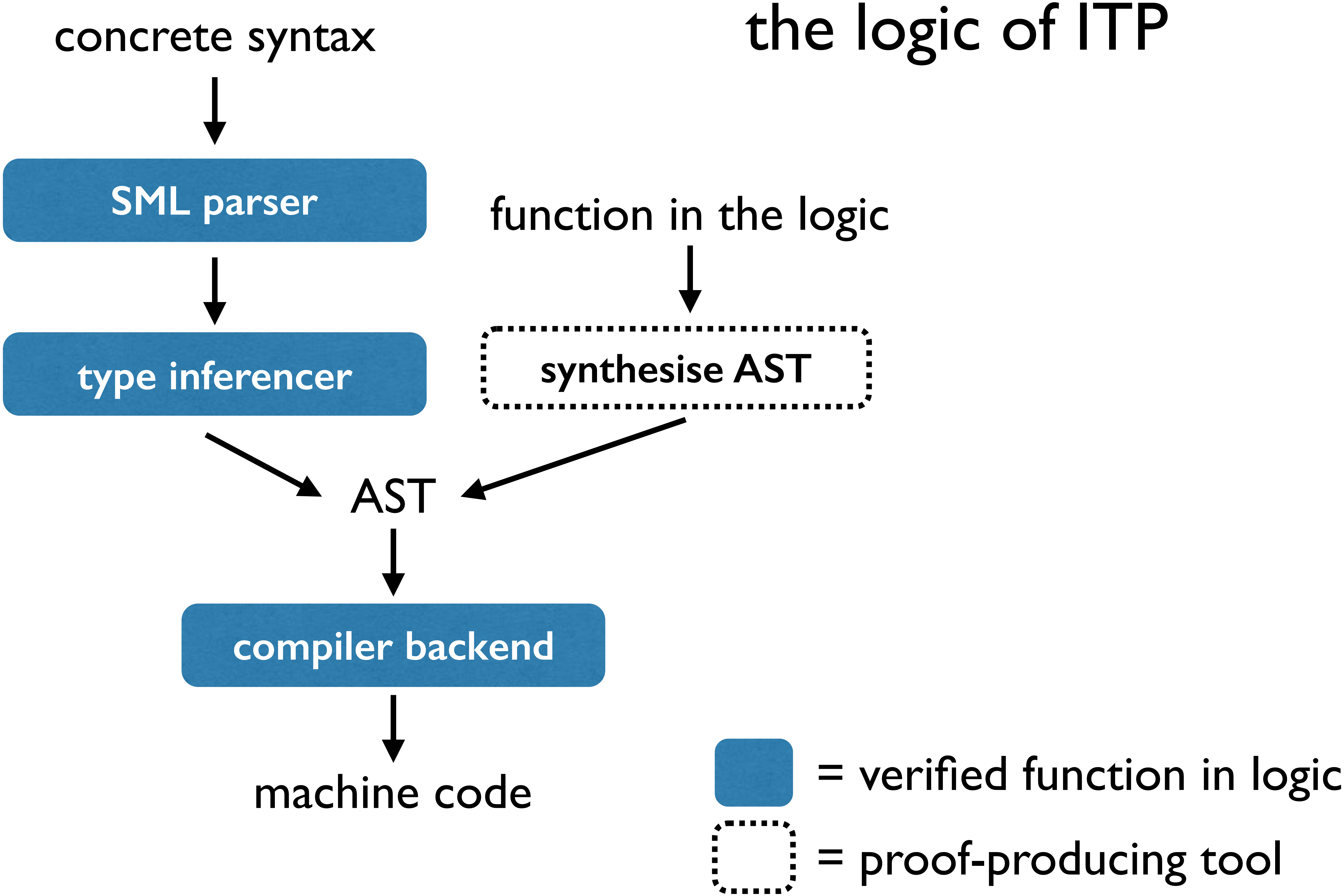
All languages communicate with the external world via a byte-array-based foreign-function interface.

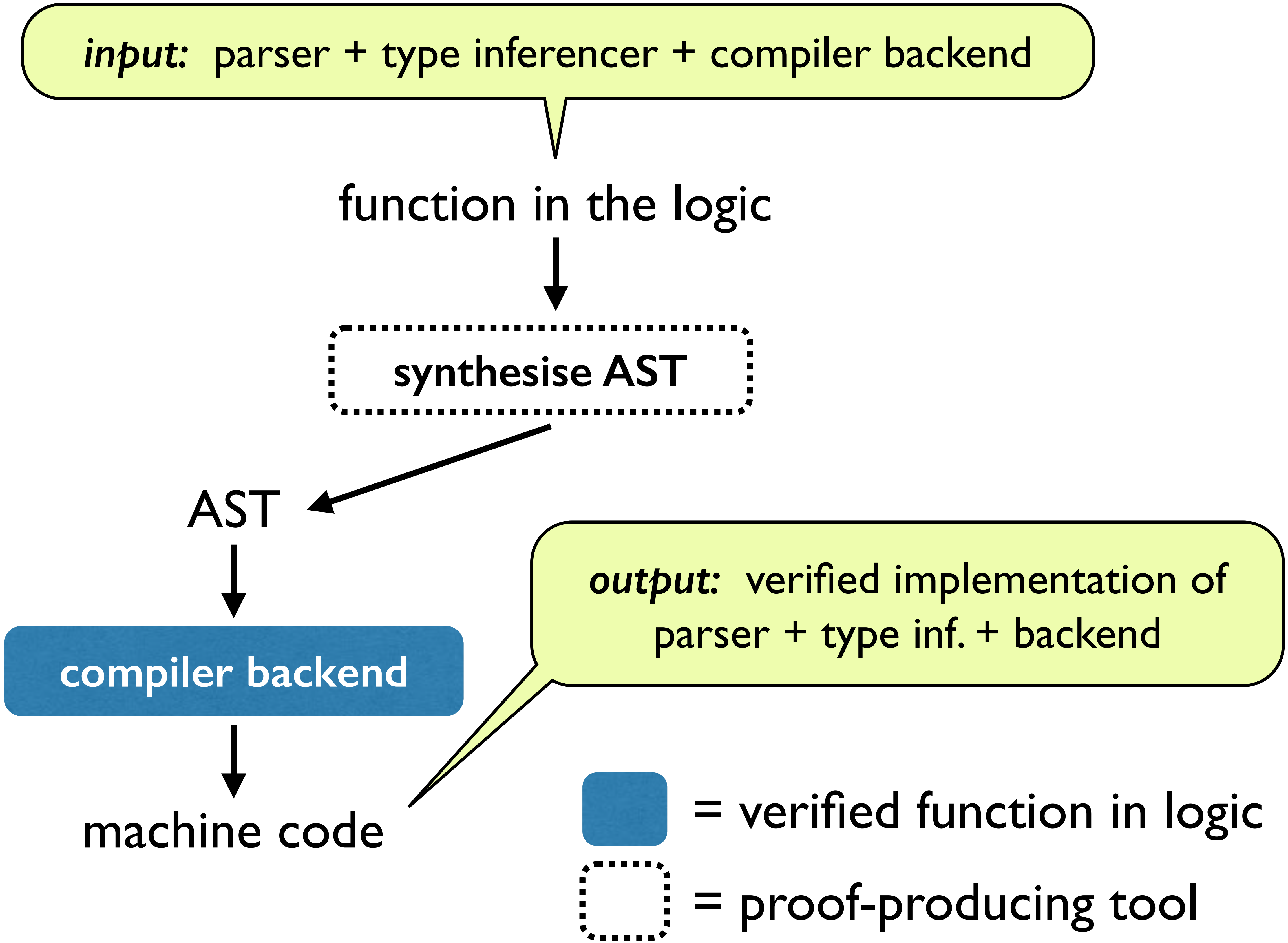
Machine-like types

Imperative compiler ***with an FP twist:*** garbage collector, live-var annotations, fast exception mechanisms (for ML)

Targets 5 architectures

Bootstrapping inside the logic of ITP





input: parser + type inferencer + compiler backend
function in the logic
synthesise AST
AST
output: verified implementation of parser + type inf. + backend
compiler backend
machine code
= verified function in logic
= proof-producing tool

Part 2:

This talk:

1. What is the CakeML compiler?
2. What has been proved about it? Is it secure?
3. Discussion: how could it be made more secure?

Compilation cannot be perfect

Source code:

- no bounds on length of lists, trees etc.
- mathematical integers

bignum lib
& GC

source programs cannot always be flawlessly represented in the target languages

Target languages:

- 32-bit and 64-bit architectures
- finite memory

Top-level correctness theorem

(next slides will explain more)

$$
\begin{array}{l}
\vdash \textsf{inf\_conf\_ok}\ cc.\textsf{inferencer\_config} \land \neg cc.\textsf{input\_is\_sexp} \land \\
\quad \textsf{backend\_config\_ok}\ cc.\textsf{backend\_config} \land \textsf{mc\_conf\_ok}\ mc \land \\
\quad \textsf{mc\_init\_ok}\ cc.\textsf{backend\_config}\ mc \Rightarrow \\
\quad\ \ \textsf{case cakeml\_compile}\ cc\ prelude\ input\ \textsf{of} \\
\quad\quad \textsf{Success}\ (code,data,cc') \ \Rightarrow \\
\quad\quad\ \ \exists behaviours. \\
\quad\quad\quad \textsf{cakeml\_semantics}\ \mathit{ffi}\ prelude\ input = \textsf{Execute}\ behaviours \land \\
\quad\quad\quad \forall ms. \\
\quad\quad\quad\quad \textsf{installed}\ code\ data\ cc'.\textsf{ffi\_names}\ \mathit{ffi} \\
\quad\quad\quad\quad\ \ (\textsf{heap\_regs}\ cc.\textsf{backend\_config.stack\_conf.reg\_names})\ mc\ ms \Rightarrow \\
\quad\quad\quad\quad\ \ \textsf{machine\_sem}\ mc\ \mathit{ffi}\ ms \subseteq \textsf{extend\_with\_resource\_limit}\ behaviours \\
\quad\ \ \mid \textsf{Failure ParseError} \ \Rightarrow\ \textsf{cakeml\_semantics}\ \mathit{ffi}\ prelude\ input = \textsf{CannotParse} \\
\quad\ \ \mid \textsf{Failure (TypeError \_)} \ \Rightarrow\ \textsf{cakeml\_semantics}\ \mathit{ffi}\ prelude\ input = \textsf{IllTyped} \\
\quad\ \ \mid \textsf{Failure CompileError} \ \Rightarrow\ \textsf{true} \\
\quad\ \ \mid \textsf{Failure (ConfigError \_)} \ \Rightarrow\ \textsf{true}
\end{array}
$$

Top-level correctness theorem

if the compiler returns success then the source code has well-defined behaviour

$\vdash$ inf_conf_ok *cc*.inferencer_config $\wedge$ $\neg$*cc*.input

backend_config_ok *cc*.backend_config $\wedge$ mc

mc_init_ok *cc*.backend_config *mc* $\Rightarrow$

case cakeml_compile *cc prelude input* of

Success (*code*,*data*,*cc′*) $\Rightarrow$

$\exists$*behaviours*.

cakeml_semantics *ffi prelude input* $=$ Execute *behaviours* $\wedge$

$\forall$*ms*.

installed *code data cc′*.ffi_names *ffi*

(heap_regs *cc*.backend_config.stack_conf.reg_names) *mc ms* $\Rightarrow$

machine_sem *mc ffi ms* $\subseteq$ extend_with_resource_limit *behaviours*

| Failure ParseError $\Rightarrow$ cakeml_semantics *ffi prelude input* $=$ CannotParse

| Failure (TypeError _) $\Rightarrow$ cakeml_semantics *ffi prelude input* $=$ IllTyped

| Failure CompileError $\Rightarrow$ true

| Failure (ConfigError _) $\Rightarrow$ true

Error cases

$$
\begin{array}{l}
\vdash \mathsf{inf\_conf\_ok}\ cc.\mathsf{inferencer\_config} \wedge \neg cc.\mathsf{input\_is\_sexp} \wedge \\
\quad \mathsf{backend\_config\_ok}\ cc.\mathsf{backend\_config} \wedge \mathsf{mc\_conf\_ok}\ mc \wedge \\
\quad \mathsf{mc\_init\_ok}\ cc.\mathsf{backend\_config}\ mc \Rightarrow \\
\quad\ \ \mathsf{case}\ \mathsf{cakeml\_compile}\ cc\ prelude\ input\ \mathsf{of} \\
\quad\quad \mathsf{Success}\ (code,data,cc')\ \Rightarrow \\
\quad\quad\ \ \exists behaviours. \\
\quad\quad\quad \mathsf{cakeml\_semantics}\ ffi\ prelude\ input = \mathsf{Execute}\ behaviours \wedge \\
\quad\quad\quad \forall ms. \\
\quad\quad\quad\quad \mathsf{installed}\ code\ data\ cc'.\mathsf{ffi\_names}\ ffi \\
\quad\quad\quad\quad\ \ (\mathsf{heap\_regs}\ cc.\mathsf{backend\_config.stack\_conf.reg\_names})\ mc\ ms \Rightarrow \\
\quad\quad\quad\quad\ \ \mathsf{machine\_sem}\ mc\ ffi\ ms \subseteq \mathsf{extend\_with\_resource\_limit}\ behaviours \\
\quad\ \ \mid \mathsf{Failure\ ParseError}\ \Rightarrow\ \mathsf{cakeml\_semantics}\ ffi\ prelude\ input = \mathsf{CannotParse} \\
\quad\ \ \mid \mathsf{Failure}\ (\mathsf{TypeError}\ \_)\ \Rightarrow\ \mathsf{cakeml\_semantics}\ ffi\ prelude\ input = \mathsf{IllTyped} \\
\quad\ \ \mid \mathsf{Failure\ CompileError}\ \Rightarrow\ \mathsf{true} \\
\quad\ \ \mid \mathsf{Failure}\ (\mathsf{ConfigError}\ \_)\ \Rightarrow\ \mathsf{true}
\end{array}
$$

Partiality I (at compile time)

$\vdash$ inf_conf_ok cc.inferencer_config $\wedge$ $\neg cc$.input_is_sexp $\wedge$
backend_config_ok cc.backend_config $\wedge$ mc_conf_ok mc $\wedge$
mc_init_ok cc.backend_config mc $\Rightarrow$
case cakeml_compile cc $prelude$ $input$ of
Success $(code, data, cc')$ $\Rightarrow$
$\exists behaviours.$
cakeml_semantics ffi $prelude$ $input$ $=$ Execute $behaviours$ $\wedge$
$\forall ms.$
installed $code$ $data$ cc'.ffi_names ffi
(heap_regs cc.backend_config.stack_conf.reg_names) mc ms $\Rightarrow$
machine_sem mc ffi ms $\subseteq$ extend_with_resource_limit $behaviours$
| Failure ParseError $\Rightarrow$ cakeml_semantics ffi $prelude$ $input$ $=$ CannotParse
| Failure (TypeError _) $\Rightarrow$ cakeml_semantics ffi $prelude$ $input$ $=$ IllTyped
| Failure CompileError $\Rightarrow$ true
| Failure (ConfigError _) $\Rightarrow$ true

compiler can exit with error (rare in practice)

Partiality 2 (at execution time)

$$
\begin{array}{l}
\vdash \mathsf{inf\_conf\_ok}\ cc.\mathsf{inferencer\_config} \wedge \neg cc.\mathsf{input\_is\_sexp} \wedge \\
\quad \mathsf{backend\_config\_ok}\ cc.\mathsf{backend\_config} \wedge \mathsf{mc\_conf\_ok}\ mc \wedge \\
\quad \mathsf{mc\_init\_ok}\ cc.\mathsf{backend\_config}\ mc \Rightarrow \\
\qquad \mathsf{case\ cakeml\_compile}\ cc\ prelude\ input\ \mathsf{of} \\
\qquad\quad \mathsf{Success}\ (code,data,cc')\ \Rightarrow \\
\qquad\qquad \exists behaviours. \\
\qquad\qquad\quad \mathsf{cakeml\_semantics}\ \mathit{ffi}\ prelude\ input = \mathsf{Execute}\ behaviours \wedge \\
\qquad\qquad\quad \forall ms. \\
\qquad\qquad\qquad \mathsf{installed}\ code\ data\ cc'.\mathsf{ffi\_names}\ \mathit{ffi} \\
\qquad\qquad\qquad\quad (\mathsf{heap\_regs}\ cc.\mathsf{backend\_config.stack\_conf.reg\_names})\ mc\ ms \Rightarrow \\
\qquad\qquad\qquad \mathsf{machine\_sem}\ mc\ \mathit{ffi}\ ms \subseteq \mathsf{extend\_with\_resource\_limit}\ behaviours \\
\qquad\ \mid \mathsf{Failure\ ParseError}\ \Rightarrow\ \mathsf{cakeml\_semantics}\ \mathit{ffi}\ prelude\ input = \mathsf{CannotParse} \\
\qquad\ \mid \mathsf{Failure}\ (\mathsf{TypeError}\ \_)\ \Rightarrow\ \mathsf{cakeml\_semantics}\ \mathit{ffi}\ prelude\ input = \mathsf{IllTyped} \\
\qquad\ \mid \mathsf{Failure\ CompileError}\ \Rightarrow\ \mathsf{true} \\
\qquad\ \mid \mathsf{Failure}\ (\mathsf{ConfigError}\ \_)\ \Rightarrow\ \mathsf{true}
\end{array}
$$

executable can give up with out-of-memory (OOM) error

Information leakage?

Behaviour can differ

$\exists\, behaviours.$
$\quad \text{cakeml\_semantics}\ \mathit{ffi}\ prelude\ input = \text{Execute}\ behaviours\ \wedge$
$\quad \ldots$
$\quad \text{machine\_sem}\ mc\ \mathit{ffi}\ ms \subseteq \text{extend\_with\_resource\_limit}\ behaviours$

under cost limit implies no OOM

One could have a ***cost semantics***

… but this would clutter the source semantics

How would one reason about the cost semantics? (not local)

Would we need to preserve OOM?

Timing

Nothing is proved about timing.

Constant-time code impossible to write in CakeML.

Reasons:

- GC,
- bignums,
- function call optimisations,
- unpredictable timing in target ISAs

Assumptions about exec. env.

⊢ inf_conf_ok *cc*.inferen
backend_config_ok *cc*.
mc_init_ok *cc*.backend_config *mc* ⇒
case cakeml_compile *cc* p
Success (*code*,*data*,*cc*′)
∃*behaviours*.
cakeml_semantics *ffi* p
∀*ms*.
installed *code data cc*′.ffi_names *ffi*
(heap_regs *cc*.backend_config.stack_conf.reg_names) *mc ms* ⇒
machine_sem *mc ffi ms* ⊆ extend_with_resource_limit *behaviours*
| Failure ParseError ⇒ cakeml_semantics *ffi prelude input* = CannotParse
| Failure (T
| Failure C
| Failure (C

we assume that the machine code is present in the machine state

… and that each foreign function interface (FFI) call only touches FFI relevant state and behaves according to *ffi* oracle model.

we allow the operating system and other processes run between each instruction execution ***and assume that other processes do not alter CakeML's memory***

Reasonable assumptions?

What are reasonable assumptions about the execution environment?

Answer(?): assumptions that could be discharged if the proof is composed with verified OS, verified linker etc.

… but where do we stop?

Part 3:

This talk:

1. What is the CakeML compiler?
2. What has been proved about it? Is it secure?
3. Discussion: how could it be made more secure?

Summary

CakeML is safe

safe control-flow (no return-oriented programming), no buffer overflows, …

but we could have a cost model

CakeML is only secure to <u>weak</u> attacker model

attacker needs to be blind to OOM

attacker has very coarse sense of time

realistic only for some applications

attacker only has access through CakeML's FFI

reasonable assumption?