Towards a Self-Certifying Compiler for WebAssembly

Anton Xue (University of Pennsylvania)
Kedar Namjoshi (Nokia Bell Labs)

IBM PL Day 2019
2019 December 09

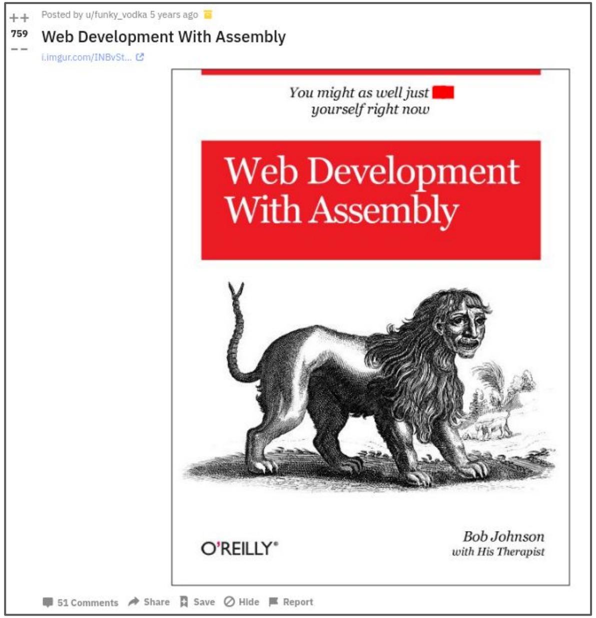
What is WebAssembly?

A binary instruction set for a **stack-based virtual machine**

A widely adopted **browser-native language**

A **compilation target** for your favorite language

A language designed with **formal semantics**



Posted by u/funky_vodka 5 years ago

Web Development With Assembly

i.imgur.com/INBvSt...

https://www.reddit.com/r/ProgrammerHumor/comments/1ykdi6/web_development_with_assembly/

A new compile target ... so what?

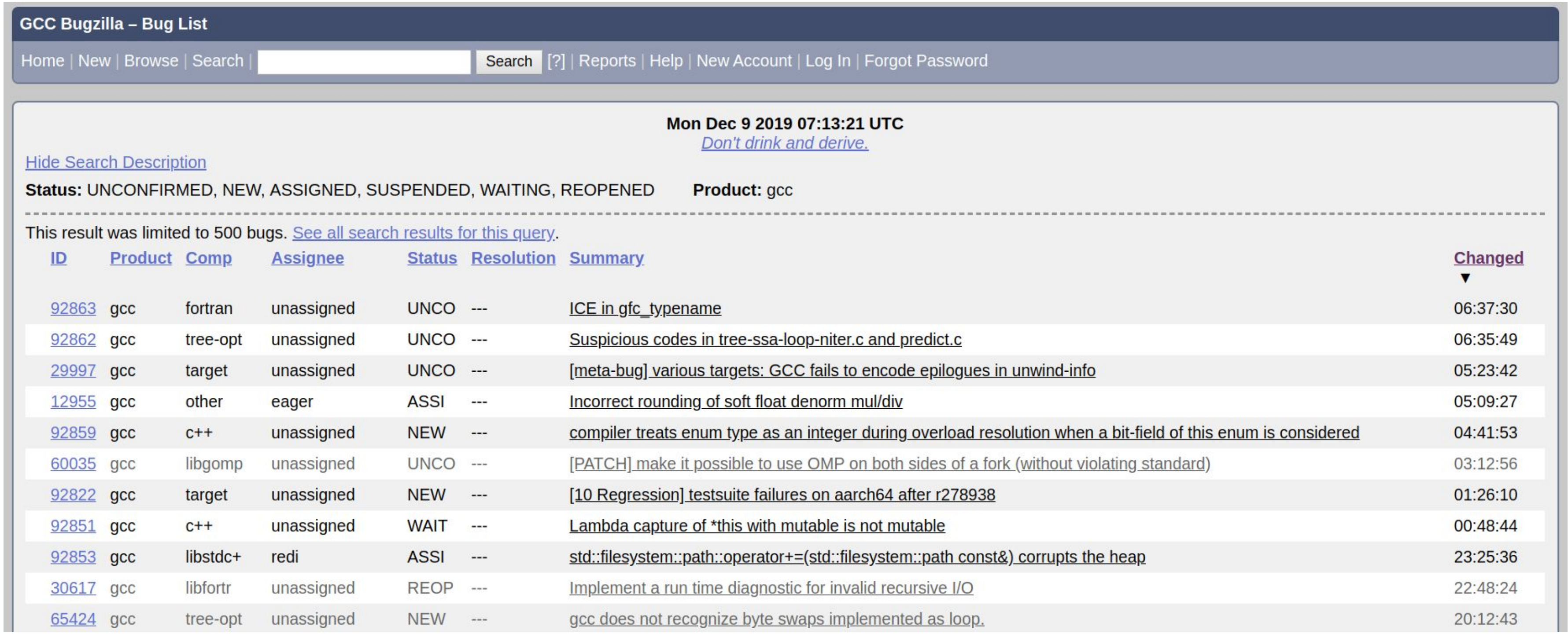
Compiled code that runs in the browser

… with relatively new technology (released in 2017)

… and a lot of deployed and unverified C/C++ code

= **Potential bugs** in the compiler toolchain

Even (reliable) compilers like GCC have bugs



Mon Dec 9 2019 07:13:21 UTC
Don't drink and derive.

Hide Search Description

Status: UNCONFIRMED, NEW, ASSIGNED, SUSPENDED, WAITING, REOPENED **Product:** gcc

This result was limited to 500 bugs. See all search results for this query.

ID	Product	Comp	Assignee	Status	Resolution	Summary	Changed ▼
92863	gcc	fortran	unassigned	UNCO	---	ICE in gfc_typename	06:37:30
92862	gcc	tree-opt	unassigned	UNCO	---	Suspicious codes in tree-ssa-loop-niter.c and predict.c	06:35:49
29997	gcc	target	unassigned	UNCO	---	[meta-bug] various targets: GCC fails to encode epilogues in unwind-info	05:23:42
12955	gcc	other	eager	ASSI	---	Incorrect rounding of soft float denorm mul/div	05:09:27
92859	gcc	c++	unassigned	NEW	---	compiler treats enum type as an integer during overload resolution when a bit-field of this enum is considered	04:41:53
60035	gcc	libgomp	unassigned	UNCO	---	[PATCH] make it possible to use OMP on both sides of a fork (without violating standard)	03:12:56
92822	gcc	target	unassigned	NEW	---	[10 Regression] testsuite failures on aarch64 after r278938	01:26:10
92851	gcc	c++	unassigned	WAIT	---	Lambda capture of *this with mutable is not mutable	00:48:44
92853	gcc	libstdc+	redi	ASSI	---	std::filesystem::path::operator+=(std::filesystem::path const&) corrupts the heap	23:25:36
30617	gcc	libfortr	unassigned	REOP	---	Implement a run time diagnostic for invalid recursive I/O	22:48:24
65424	gcc	tree-opt	unassigned	NEW	---	gcc does not recognize byte swaps implemented as loop.	20:12:43

https://gcc.gnu.org/bugzilla/buglist.cgi?bug_status=__open__&no_redirect=1&order=changeddate%20DESC%2Cpriority%2Cbug_severity&product=gcc&query_format=specific

How does one write “correct” software?

Unit tests

Assert statements

Bugs = features

Continuous integration

Code review

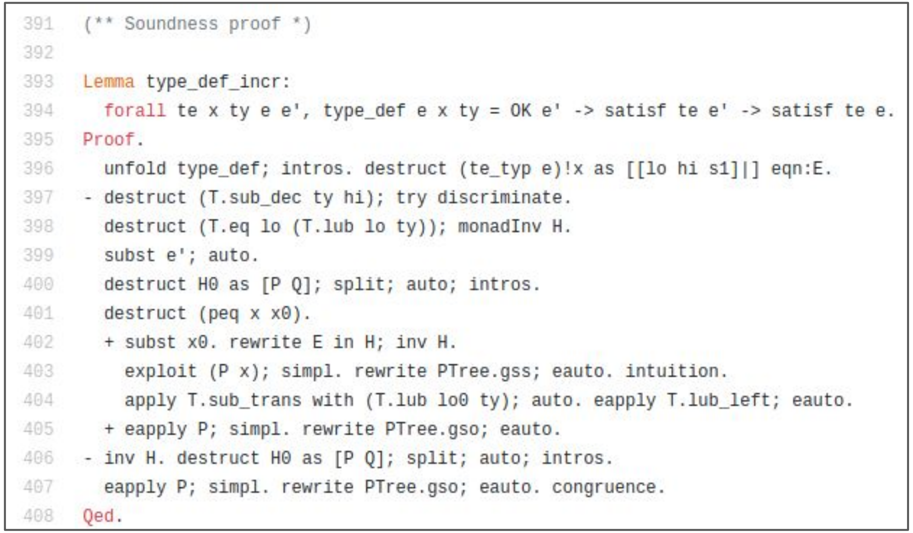
How does one write CORRECT software?

FORMAL logic

FORMAL specification

FORMAL proofs

FORMAL verification



```
(** Soundness proof *)

Lemma type_def_incr:
  forall te x ty e e', type_def e x ty = OK e' -> satisf te e' -> satisf te e.
Proof.
  unfold type_def; intros. destruct (te_typ e)!x as [[lo hi s1]|] eqn:E.
- destruct (T.sub_dec ty hi); try discriminate.
  destruct (T.eq lo (T.lub lo ty)); monadInv H.
  subst e'; auto.
  destruct H0 as [P Q]; split; auto; intros.
  destruct (peq x x0).
  + subst x0. rewrite E in H; inv H.
    exploit (P x); simpl. rewrite PTree.gss; eauto. intuition.
    apply T.sub_trans with (T.lub lo0 ty); auto. eapply T.lub_left; eauto.
  + eapply P; simpl. rewrite PTree.gso; eauto.
- inv H. destruct H0 as [P Q]; split; auto; intros.
  eapply P; simpl. rewrite PTree.gso; eauto. congruence.
Qed.
```

https://github.com/AbsInt/CompCert/blob/ec49c7b8bd450
2c380b88c78baa674000db109fd/common/Subtyping.v#L393

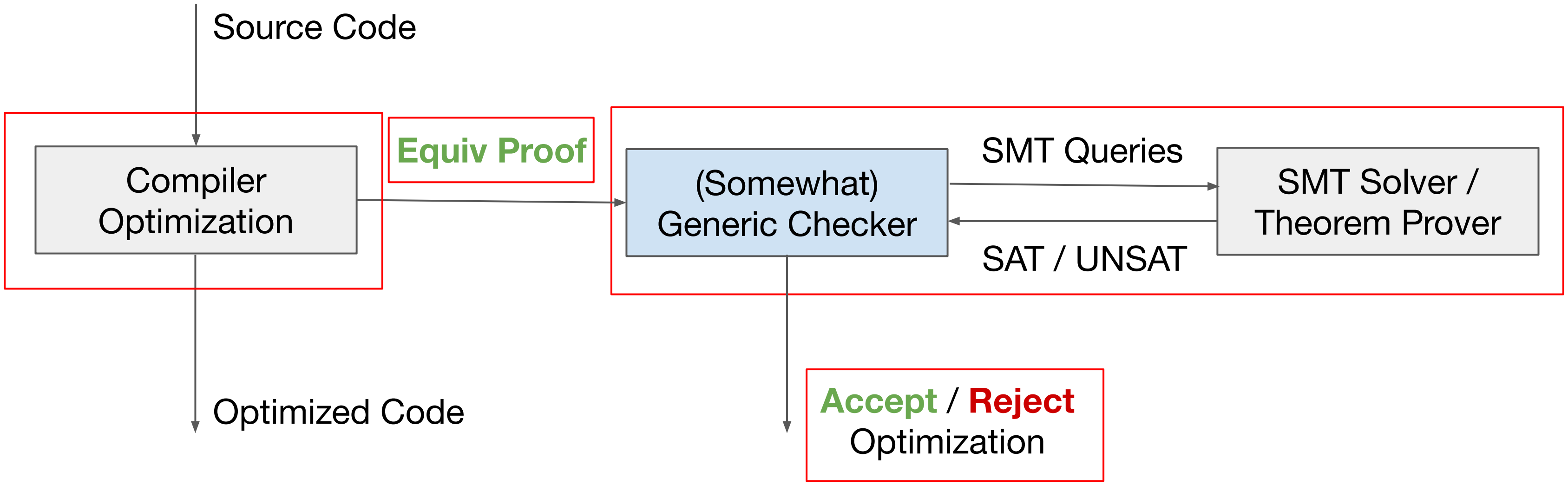
Is Classical Verification Easily Adoptable?

Requires user expertise in formal logic and theorem proving

Learning curve is a deterrence to adoption

But what if we’re willing to trade expressive power for ease-of-use?

Idea: Self-Certified Compilers



Source Code

Compiler Optimization

Equiv Proof

(Somewhat) Generic Checker

SMT Queries

SAT / UNSAT

SMT Solver / Theorem Prover

Optimized Code

Accept / **Reject** Optimization

Example: Dead Code Elimination

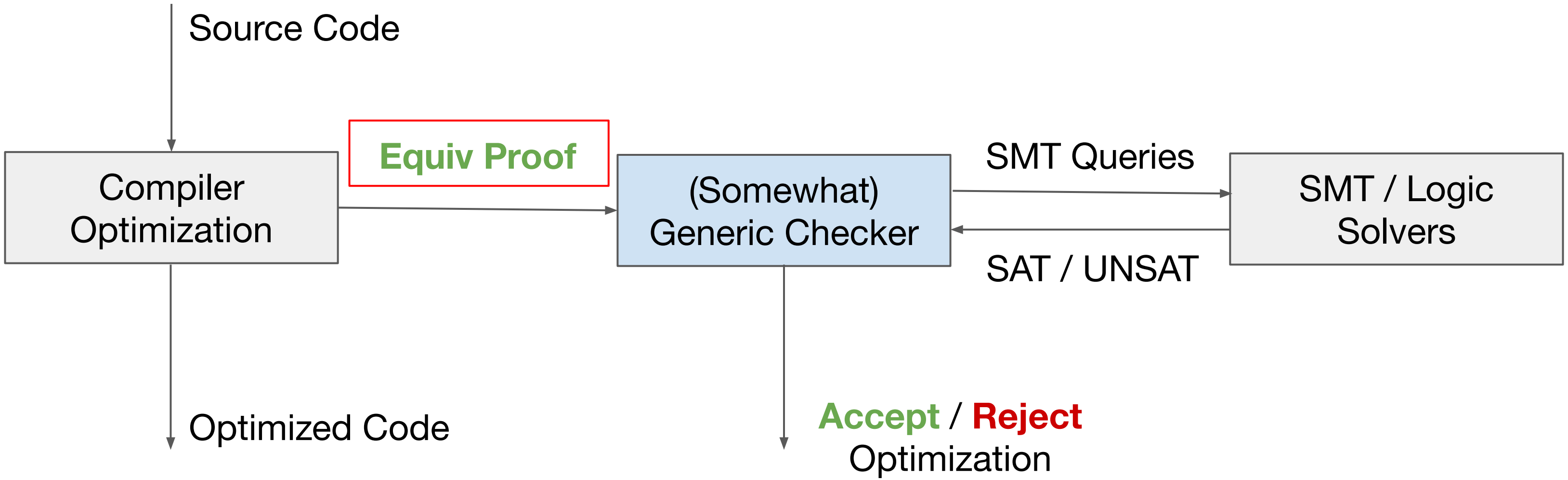
```
(local.set 1 (i32.const 617))                           (local.set 1 (i32.const 617))
                               Loc[1] == Loc'[1]

(local.set 2 (i32.const 212))                           ;; (local.set 2 (i32.const 212))


(local.set 3 (i32.const 781))                           (local.set 3 (i32.const 781))
                               Loc[1] == Loc'[1]
                               Loc[3] == Loc'[3]

                               Loc[1] == Loc'[1]
(local.set 2 (i32.const 267))  Loc[3] == Loc'[3]        (local.set 2 (i32.const 267))
                               Loc[2] == Loc'[2]

(add (local.get 1)                                      (add (local.get 1)
  (add (local.get 2)                                      (add (local.get 2)
       (local.get 3)))                                         (local.get 3)))
```


Source Code

Compiler Optimization

Equiv Proof

(Somewhat) Generic Checker

SMT Queries

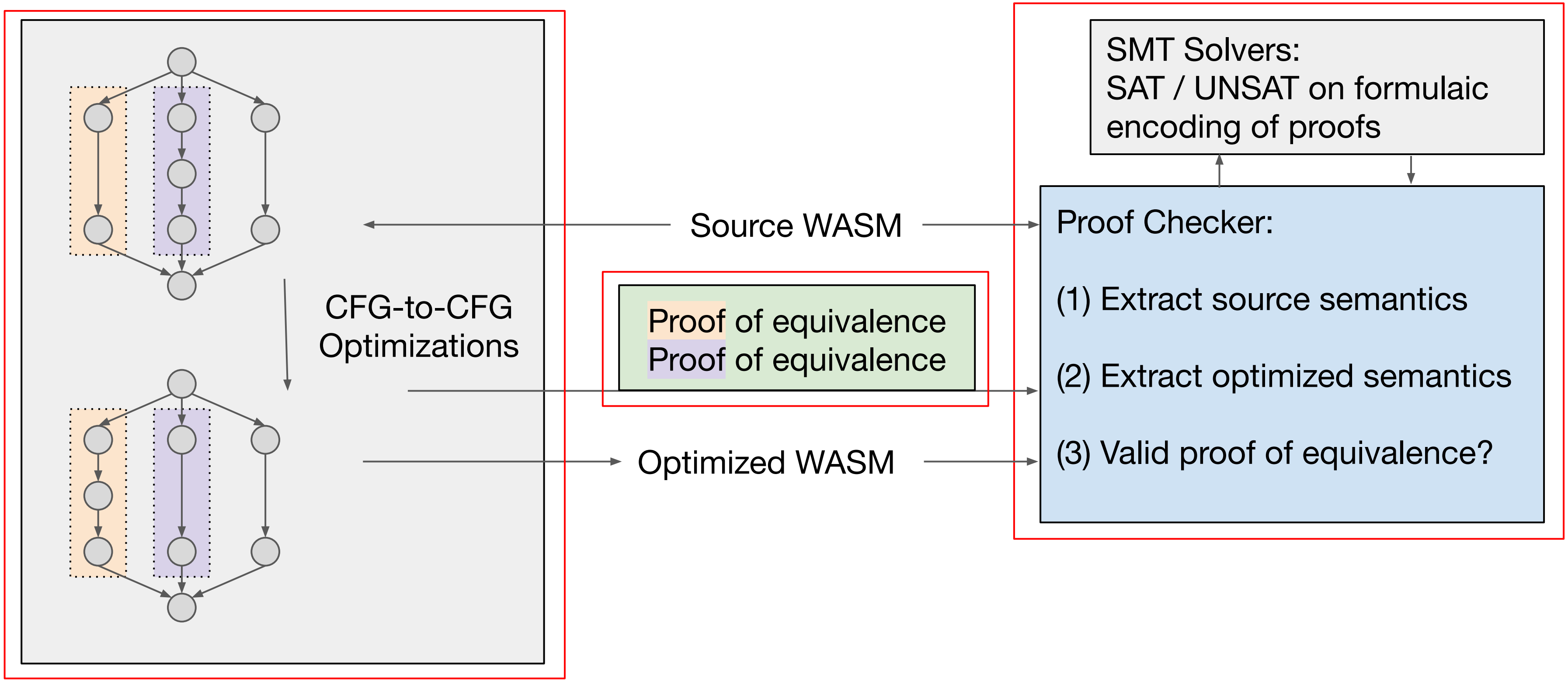
SAT / UNSAT

SMT / Logic Solvers

Optimized Code

Accept / Reject Optimization

Self-Certified Compiler Optimizations: The Vision



CFG-to-CFG
Optimizations

Source WASM

Proof of equivalence
Proof of equivalence

Optimized WASM

SMT Solvers:
SAT / UNSAT on formulaic encoding of proofs

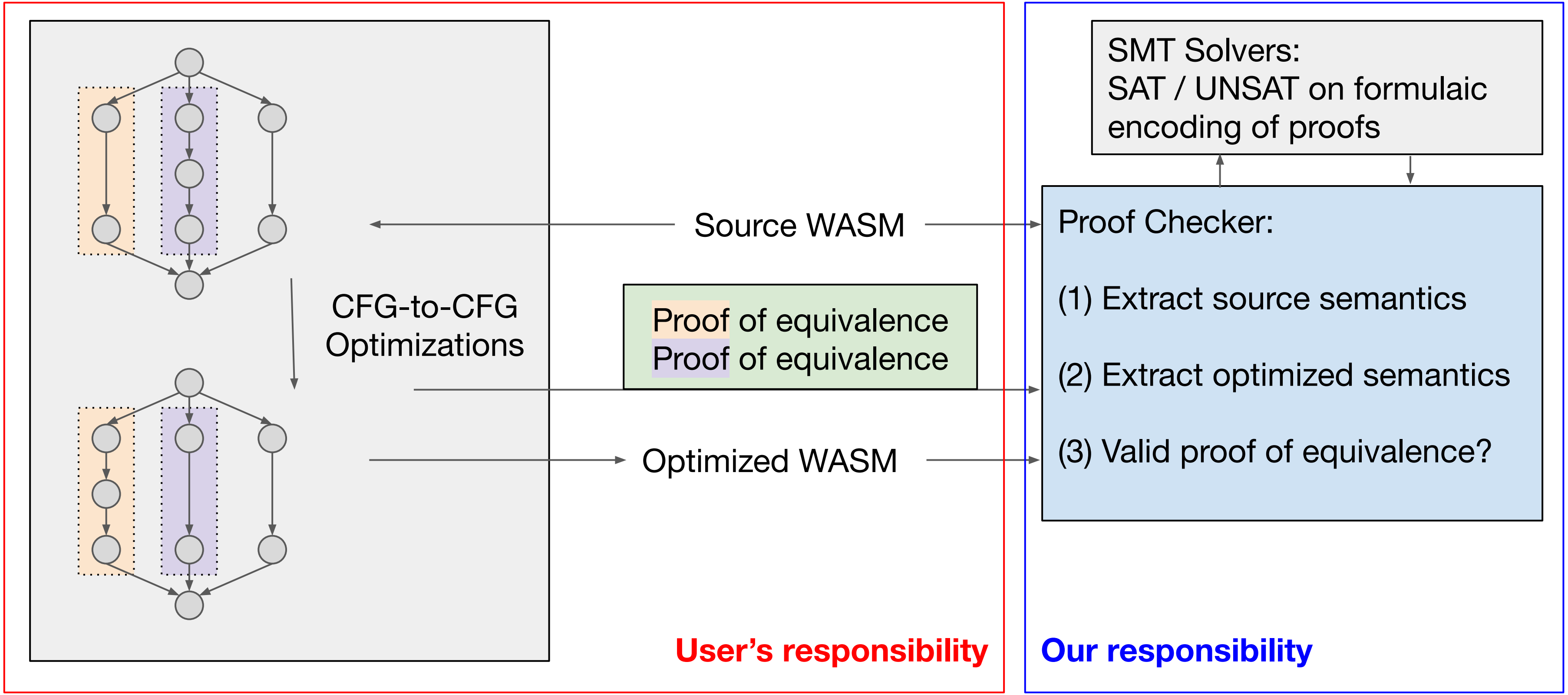
Proof Checker:

(1) Extract source semantics

(2) Extract optimized semantics

(3) Valid proof of equivalence?

Self-Certified Compiler Optimizations: The Vision



CFG-to-CFG Optimizations

Source WASM

Proof of equivalence
Proof of equivalence

Optimized WASM

User's responsibility

SMT Solvers:
SAT / UNSAT on formulaic encoding of proofs

Proof Checker:

(1) Extract source semantics

(2) Extract optimized semantics

(3) Valid proof of equivalence?

Our responsibility

Self-Certification and Classical Verification

Self-certification: proves correctness ***per every*** execution

Classical verification: proves correctness ***once*** before shipping

Self-certification: requires the ***user*** to annotate code

Classical verification: requires the Coq ***expert*** to theoremize and prove

Self-certification: should be designed with ***ease of use***

Classical verification: requires more expertise and ***power tools***

Self-certification is ***complementary*** to classical verification!

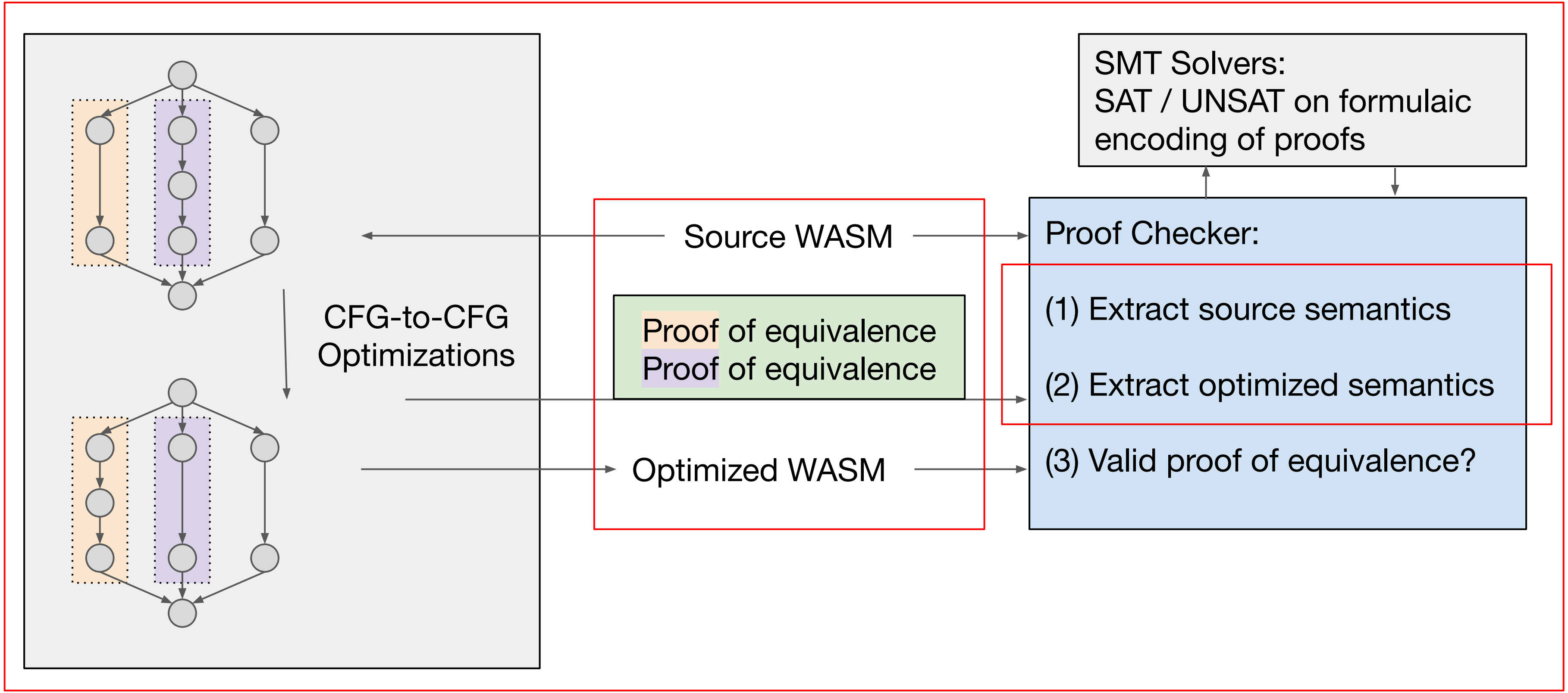
What needs to be done?

Goal 1: (our/back-end) analyzing source and optimized WASM for equivalence
Challenge 1: encoding execution semantics into logical formulas

Goal 2: (user/front-end) generating proofs of behavioral equivalence
Challenge 2: what should a proof of equivalence look like?

Goal 3: putting everything together
Challenge 3: engineering and more engineering

Self-Certified Compiler Optimizations: The Vision



CFG-to-CFG Optimizations

Source WASM

Proof of equivalence
Proof of equivalence

Optimized WASM

SMT Solvers:
SAT / UNSAT on formulaic encoding of proofs

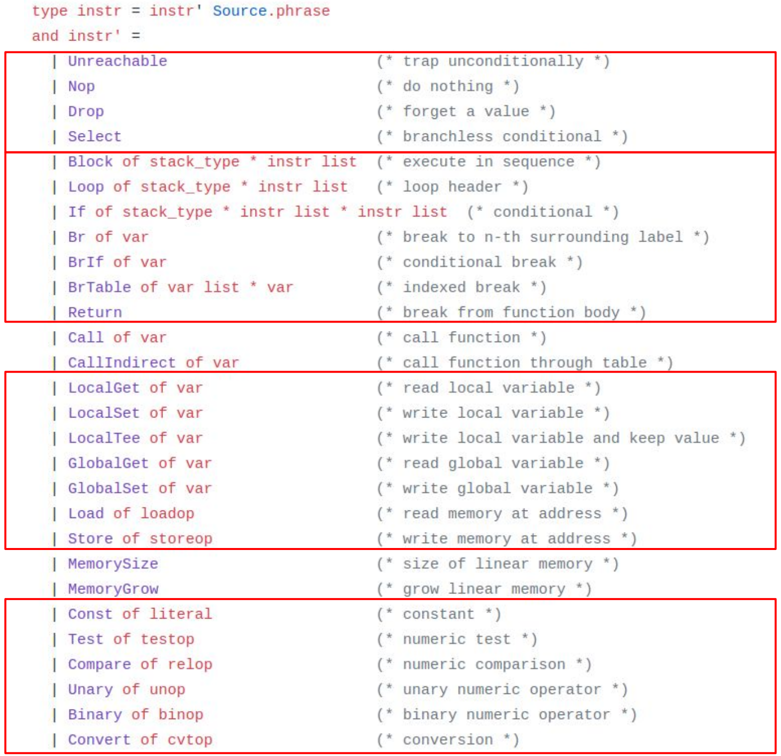
Proof Checker:

(1) Extract source semantics

(2) Extract optimized semantics

(3) Valid proof of equivalence?

Goal 1: WebAssembly Execution Semantics



```
type instr = instr' Source.phrase
and instr' =
  | Unreachable                       (* trap unconditionally *)
  | Nop                               (* do nothing *)
  | Drop                              (* forget a value *)
  | Select                            (* branchless conditional *)
  | Block of stack_type * instr list  (* execute in sequence *)
  | Loop of stack_type * instr list   (* loop header *)
  | If of stack_type * instr list * instr list  (* conditional *)
  | Br of var                         (* break to n-th surrounding label *)
  | BrIf of var                       (* conditional break *)
  | BrTable of var list * var         (* indexed break *)
  | Return                            (* break from function body *)
  | Call of var                       (* call function *)
  | CallIndirect of var               (* call function through table *)
  | LocalGet of var                   (* read local variable *)
  | LocalSet of var                   (* write local variable *)
  | LocalTee of var                   (* write local variable and keep value *)
  | GlobalGet of var                  (* read global variable *)
  | GlobalSet of var                  (* write global variable *)
  | Load of loadop                    (* read memory at address *)
  | Store of storeop                  (* write memory at address *)
  | MemorySize                        (* size of linear memory *)
  | MemoryGrow                        (* grow linear memory *)
  | Const of literal                  (* constant *)
  | Test of testop                    (* numeric test *)
  | Compare of relop                  (* numeric comparison *)
  | Unary of unop                     (* unary numeric operator *)
  | Binary of binop                   (* binary numeric operator *)
  | Convert of cvtop                  (* conversion *)
```

https://github.com/WebAssembly/spec/blob/master/interpreter/syntax/ast.ml

WebAssembly Execution Semantics

```
type value =
  int32 | int64 | float32 | float64

type state =
  { id      : int32;
    values  : values stack;
    locals  : int32 -> value;
    globals : int32 -> value;
    memory  : int32 -> value }

type instr = state -> state

type step = (state * instr * state)

type formula_of_step = step -> formula
```

Values are one of 4 types

States tell us the program’s variable values

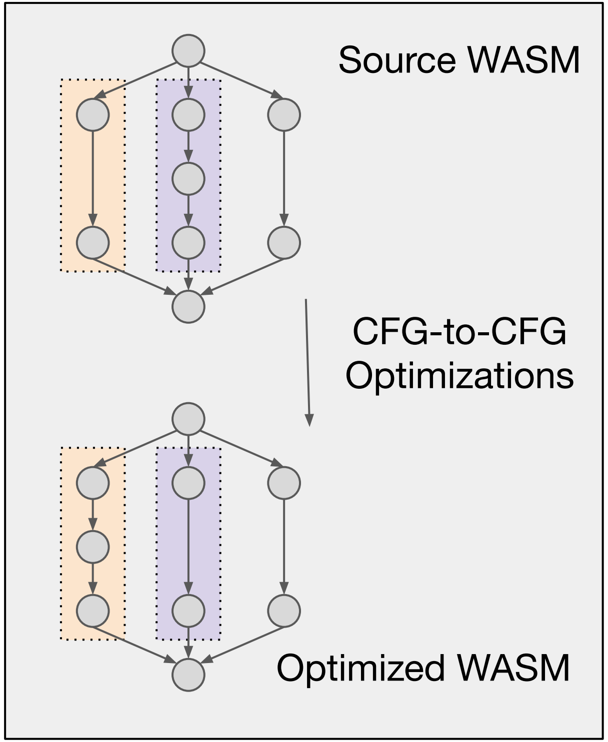
… and identify values / locals / globals / memory

Instructions map states to states

Step is a triple `(state0, instr, state1)`

Formulas in **theory of arrays** and **bit vectors**

Goal 2: Equivalence Proofs of Program Traces



The user is responsible for:

- + Instrumenting their own code to generate proofs
- + Deciding what equivalences on CFGs to prove

Proofs should:

- + Identify **path pairs** in the source and target CFGs
- + Identify **equivalences** on source and target states

Goal 3: Engineering

Where are we now?

+ Much of the backend proof checker component is written

+ Can validate simple optimizations like local CFG block merging

TODO:

+ Handle more optimizations

+ Generating WASM code (a few difficulties in the type system)

+ Instrumenting existing optimizations

Summary

Self-certification as an approach to writing correct software

+ Complementary to classical formal verification

Self-certified compiler optimizations for WASM

+ Extract execution semantics to theory of arrays and bit vectors

+ Optimization proofs identify CFG path pairings and equivalence relations used

FIN