Towards a Self-Certifying Compiler for WebAssembly

Anton Xue <antonxue@seas.upenn.edu> and Kedar Namjoshi <kedar.namjoshi@nokia-bell-labs.com>

What is WebAssembly?

- Instruction set for a browser-native stack-based VM
- Designed with formal semantics and simple type system
- Compilation target for your favorite language
- Supported on Chrome, Firefox, Safari, and Edge

Why WebAssembly?

- Replaces JavaScript as a browser compilation target
- Offers better speed and portability over JavaScript
- Fairly small and simple instruction set (see right)

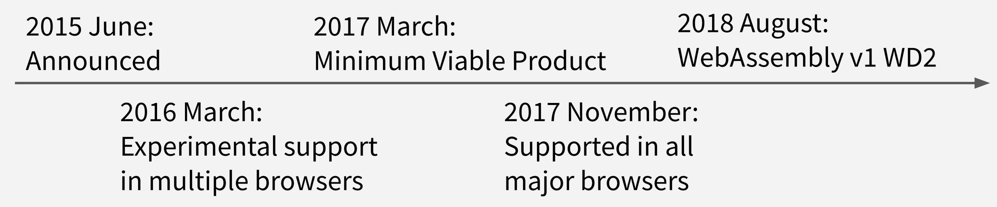
How to Run WebAssembly?

- Can be invoked from JavaScript
- Future: support for <script type="module"> in HTML

```
(* WASM instruction set (from reference interpreter) *)
type instr = instr' Source.phrase
and instr' =
  (* Numeric instructions *)
  | Const of literal | Test of testop | Compare of relop
  | Unary of unop | Binary of binop | Convert of cvtop
  (* Control flow instructions *)
  | Unreachable | Nop | Drop | Select
  | Block of stack_type * instr list
  | Loop of stack_type * instr list
  | If of stack_type * instr list * instr list
  | Br of var | BrIf of var | BrTable of var list * var
  | Return | Call of var | CallIndirect of var
  (* Variable instructions *)
  | LocalGet of var | LocalSet of var | LocalTee of var
  | GlobalGet of var | GlobalSet of var
  (* Memory instructions *)
  | Load of loadop | Store of storeop
  | MemorySize | MemoryGrow

(* WASM value types (from reference interpreter) *)
type value_type = I32Type | I64Type | F32Type | F64Type
type stack_type = value_type list
type func_type = FuncType of stack_type * stack_type
```

WebAssembly in the World Wild Web



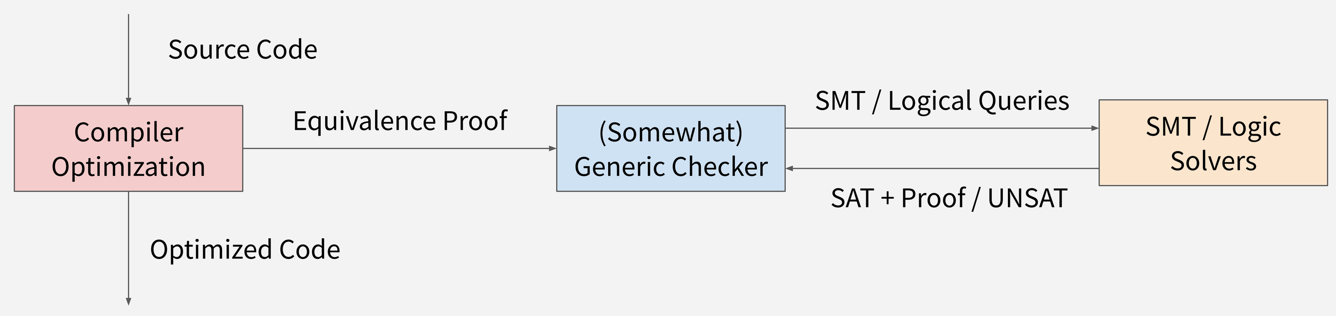
The State of the Tooling

- Most browser implementations written in C / C++ for performance
- Compiler toolchains such as Binaryen also heavily use C / C++

Problem: Are the tooling for WebAssembly … safe?

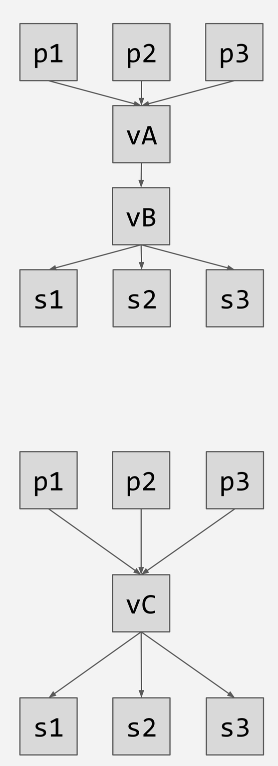
- Tension in what developers like vs formal methods best practice:
  - Developers like writing language tools in C / C++ …
  - … but this does not play well with formal methods tooling
  - Hard to yield formal guarantees on big C / C++ code bases
- Ideal trade-offs:
  - Let developers write in their preferred languages
  - Let developers choose which parts of the code to "verify"
  - Let developers easily call formal methods machinery

Self-Certifying Compiler Optimizations: The Vision



- High-level goal: smoothly integrate formal methods into software development
- Insight: easier for developers to leverage techniques they are familiar with
- Idea: SMT-backed checker API that exposes familiar features is likely easier to use

Case Study: Block Merging in Control Flow Graphs




```
graph merge_blocks(src_cfg, vA, vB) {
  // ...
  if (can_merge(src_cfg, vA, vB)) {
    // v_C is the new vertex that replaces v_A and v_B
    let (opt_cfg, vC) = merge(src_cfg, vA, vB);
    // Check that the incoming edges are preserved
    check_eq(src_cfg.preds(vA), opt_cfg.preds(vC));
    // Check that the outcoming edges are preserved
    check_eq(src_cfg.succs(vB), opt_cfg.succs(vC));
   // Iterate through all pred / succ pairings ...
    for p in src_cfg.preds(vA) {
      for s in src_cfg.succs(vB) {
       // ... and check that opt_cfg's path are equiv
       check_exec_equiv(src_cfg.path([p, vA, vB, s]),
                        opt_cfg.path([p, vC, s]));
      }
    }
    // Return the optimized cfg after the checks
    return opt_cfg;
  }
  // ...
}
```

Where are We Now?

- Proof checker backend is implemented
- Leverages reference interpreter written in OCaml for parsing and printing
- Basic interfacing with Z3
- Primitive pipeline written with self-certifying optimizations on the way
- Works on hard-coded source CFG, optimized CFG, and proof relations

Challenge 1: WebAssembly's Type System

- WebAssembly's type system is restrictive
- Some "obviously correct" programs are invalid since they fail to type check

```
; Valid program
(block (i32_const 111) (i32_const 222) (add))
; Invalid program
(block (i32_const 111) (block (i32_const 222) (block (add))))
```

- Converting from CFG back to WASM can be a challenge

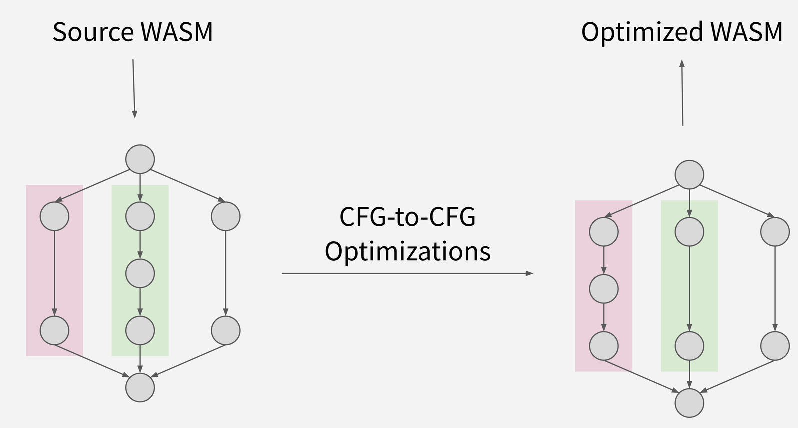
Challenge 2: Integration with Existing WASM Compiler Tooling

- Hacking existing WebAssembly toolchains' optimizations to generate proofs is hard
- Requires understanding other people's software architecture
  - Eg: Binaryen has wild C++ inheritance structures for its optimizations

Next Steps

- Writing self-certifying optimizations is independent of generating valid WASM
- Write out simple optimizations like block merging and constant propagation
- Being able to generate valid WASM is still key to making this technique useful
- Testing against real-world WASM code

Self-Certifying Compiler Optimizations: The Engineering



Optimizations Done:
Magenta path elongated
Green path shortened

Relations Between CFGs:
Magenta paths should be equivalent
Green paths should be equivalent

Proof (as list of triplets):

```
[(src_magenta, opt_magenta, magentas_eqv),
 (src_green, opt_green, greens_eqv)]
```

High-level Proof Checker Algorithm

1. Convert source WASM into CFG representation
   a. Each vertex (basic block) of the CFG is "straight-line" code (no control flow)
   b. Each edge (jump) of the CFG denotes how control flow changes
2. Optimizer's generated proof should identify correspondences between paths and their relations
   a. Proofs can be thought of as a list of triples `(src_path, opt_path, path_rels)`
3. For each `(src_path, opt_path, path_rels)` in the proof:
   a. Encode `src_path`, `opt_path`, and `path_rels` into a logical formulas (theory of arrays + bit vectors)
   b. Query an SMT solver with the formulas' conjunction
   c. Proof is valid iff SAT

Converting CFG into SMT formulas

```
; state 0
(i32_const 111) ; push 111 on stack
; state 1
(i32_const 222) ; push 222 on stack
; state 2
(add) ; pop, pop, push 333 on stack
; state 3
```

- Each vertex is a control flow-free sequence of instructions
- States of program execution occur between instructions
- A state can be characterized by its (1) stack, (2) stack pointer, (3) memory, (3) local variables, (5) global variables
- Each non-control flow instruction maps a state to another state

```
state1_stack == state0_stack[state0_pointer + 1 <- 111]
state1_pointer == state0_pointer + 1;
state1_locals == state0_locals
state1_globals == state0_globals
state1_memory == state0_memory

state2_stack == state1_stack[state1_pointer + 1 <- 222]
state2_pointer == state1_pointer + 1;
state2_locals == state1_locals
state2_globals == state1_globals
state2_memory == state1_memory

state3_stack ==
  state2_stack
    [state2_pointer - 1
       <- state2_stack(state2_pointer)
          + state2_stack(state2_pointer - 1);
     state2_pointer <- 0 ] ; add and clear top with 0
state3_pointer == state2_pointer - 1;
state3_locals == state2_locals
state3_globals == state2_globals
state3_memory == state2_memory
```

Source Path and Optimized Path Equivalence

- Different notions of program equivalence exist
- Observational equivalence:
  - Stack, locals, globals, and memory the same
  - Useful for block merging, for instance
- More generally:
  - State-state relation holds at start of execution
  - State-state relation holds at end of execution
  - Require developer to explicitly tell us what it is

```
src_statei_stack = opt_statej_stack
src_statei_locals = opt_statej_locals
src statei_globals = opt_statej_globals
src_statei_memory = opt_statej_memory
```

```
state_relation(src_state0, opt_state0)
state_relation(src_statei, opt_statej)
```

Conclusion

We present a work-in-progress of bringing **self-certifying compiler optimizations** to WebAssembly. In addition to the goal of making WebAssembly tooling more robust, self-certification as a framework has potential to tighten the gap between theory and practice: to **increase adoption of formal methods** in real-world software engineering. By allowing developers to work with familiar techniques while preserving formal rigor, it becomes easier -- and practical -- to write correct code.