A Self-Certifying Compilation Framework for WebAssembly

Kedar S. Namjoshi[1(✉)] and Anton Xue[2]

[1] Nokia Bell Labs, Murray Hill, NJ 07974, USA
kedar.namjoshi@nokia-bell-labs.com
[2] University of Pennsylvania, Philadelphia, PA 19104, USA
antonxue@seas.upenn.edu

Abstract. A *self-certifying* compiler is designed to generate a correctness proof for each optimization performed during compilation. The generated proofs are checked automatically by an independent proof validator. The outcome is formally verified compilation, achieved *without* formally verifying the compiler. This paper describes the design and implementation of a self-certifying compilation framework for WebAssembly, a new intermediate language supported by all major browsers.

1 Introduction

Compiling is everywhere, in astonishing variety. A compiler systematically transforms a source program into an executable program through a series of "optimizations" — program transformations that improve run-time performance by, for instance, rewriting instructions or compacting memory use. It is vital that each transform preserves input-output behavior, so that the behavior of the final executable is identical to that of the original source program. Compiler writers put considerable care into programming these transforms, but when mistakes happen they are often difficult to detect.

The obvious importance of correct compilation has prompted decades of research on compiler verification. The gold standard is a *verified* compiler, where each transform is formally proved correct. Originally proposed in the 1960s [21], verified compilers have been constructed for Lisp (the CLI stack [4]) and for C (CompCert [18,19]). The proofs require considerable mathematical expertise and substantial effort, of the order of multiple person-years. As an illustration, a proof of the key Static Single Assignment (SSA) transform required about a person-year of effort and approximately 10,000 lines of Coq proof script [36].

Our research goal is to "democratize" this process by making it feasible for compiler developers who are not also theorem-proving experts to build a provably correct compiler. This requires relaxing the notion of correctness. Rather than establish that a transform is correct for *all* programs, we establish that each *specific application* of the transform is correct. We do so by instrumenting every transform to additionally generate a proof object that (if valid) guarantees that the original and transformed programs have the same input-output behavior.

Thus, the work of formal verification is divided between a compiler writer, who writes auxiliary code to generate proofs, and an automated validation program, which checks each proof. We call such a compiler *self-certifying*, for it justifies its own correctness.

An invalid proof exposes either an error in the transform code or a gap in the compiler-writer's understanding of its correctness argument, which are both valuable outcomes. It is important to note that a self-certifying compiler may still contain latent errors: its guarantee applies only to an instance of compilation. Yet, in practice, that is what is desired: a programmer cares (selfishly) only about the correct compilation of their own program.

Self-certification may seem unfamiliar, but it is a recurring concept. A model checker is self-certifying, as a counterexample trace certifies a negative result. A SAT solver is also self-certifying, as an assignment certifies a positive result. For the other outcomes, a deductive proof acts as a certificate: cf. [23,29] for model checking and [35] for SAT. Self-certification for parsing is described in [15]. In each case, the certificate is easy to check, while it justifies the outcome of a rather complex calculation.

This paper describes the design and implementation of a self-certifying optimization framework for WebAssembly, a recently-introduced intermediate language that is supported by all major browsers [14]. Programs in C, C++, Rust, and LLVM IR can be compiled to WebAssembly and run within a browser. We choose to focus on WebAssembly for two main reasons: it is an open and widely-adopted standard, and it has a compact, well-designed instruction set with a precisely defined semantics.

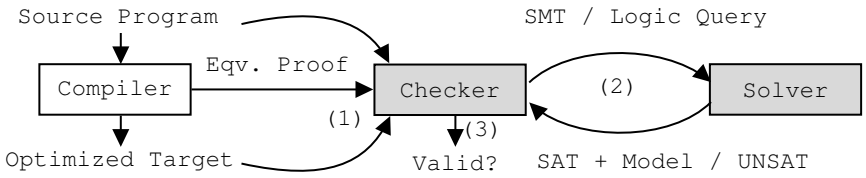

Fig. 1: Self-certification Overview. Trusted components are shaded gray.

Figure 1 illustrates the framework. Its core is the proof checker. That takes as input two WebAssembly programs (the source and target of a transform) and a purported equivalence proof, and uses SMT methods to check the validity of this proof. Fortunately, correctness proofs for many standard optimizations can be expressed in logics that are well-supported by SMT solvers.

Armed with a proof validator, one can proceed to augment each optimization with a proof generator. A typical generator records auxiliary information during the transformation and uses it to produce an equivalence proof. In our experience, it is an enjoyable exercise to design proof generators and straightforward to implement them. For instance, proof generation for SSA (described

later) requires only about 100 lines of code, and the entire certifying transform was written in three person-weeks.

Self-certification for compilers was originally proposed by Rinard [31] as "credible compilation" and rediscovered in [27] as "witnessing." It is closely related to Translation Validation (TV) [32,30,28,6] but with crucial differences. In TV, a validator has access to only the source and target programs. As program equivalence is undecidable, heuristics are necessary to show equivalence. This has drawbacks: heuristics differ across transforms, and each must be separately verified. In a self-certifying compiler, although the content of a proof depends on the transform, all proofs are checked by the same validator.

Self-certification also has a close relationship to deductive proof. In a deductive proof, the correctness of a transform τ is established by proving that for all programs P, there exists an input-output-preserving simulation relation R such that $\tau(P)$ refines P via R. Through Skolemization, this is equivalent to the existence of a function G such that for all programs P, $\tau(P)$ refines P via $G(P)$. In a self-certifying compiler, the mathematical object G is turned into a computational proof-generator for the transform τ. The generated proof object is the relation $G(P)$, and a validator is thus a generic refinement checker.

While conceptually simple, self-certification is challenging to implement. The first implementations were for a textbook-style language [20]. The most advanced implementation is Crellvm [16], for LLVM. In Crellvm, proofs are syntax-directed, based on relational Hoare logic. While this suffices for many transforms, the authors note in [16] that it cannot support transformations such as loop unrolling that make large alterations to control structure. Our proof format can handle those transforms. An in-depth comparison is given later in the paper.

The central contribution of this work is in defining and implementing a self-certifying compilation framework for a widely-used language.[3] The current system is best thought of as a fledgling compiler for WebAssembly. We have implemented a variety of optimizations, among them SSA, dead store removal, constant propagation and loop unrolling. Experience shows that proof generation imposes only a small programming burden: the typical generator is about a hundred lines of code. Experiments show that the run-time overhead of proof generation is small, under 20%. Proof-checking, on the other hand, may take substantial time (though it is easily parallelized). That is not caused by logical complexity, it is due to the sheer number of lemmas that must be discharged.

The current system has some limitations. Proof-checking is slow, but we believe that that can be improved through careful engineering, as each proof is logically simple. A technical limitation is that the proof-checker does not support transforms with unbounded instruction reordering (such as loop tiling), or inter-procedural transforms. This is because refinement relations for both require quantification over unbounded auxiliary state. Those transforms *can* be validated with specialized rules that have simpler hypotheses (cf. [37,1,34,25]). Integrating those transforms into the system is a direction for future work.

[3] The implementation is available as open source at https://github.com/nokia/web-assembly-self-certifying-compilation-framework.

2 Overview

We illustrate self-certification with the loop unrolling transform. A loop is given as "`loop B`". For instance, the sum of the first N natural numbers may be expressed as follows. (For readability, this is in pseudo-code, not WebAssembly.)

```
sum := 0; i := 0;
loop {if i >= N then goto Exit; sum := sum + i; i := i+1;}
Exit:
```

The unrolling transformation simply changes the program to "`loop (B;B)`". This may appear to be of little use. However, unrolling facilitates further analysis and transformation. For the example, assuming N is even, an analysis phase computes invariants such as the assertion below. Then, control-flow simplification applies this invariant to eliminate the second copy of the conditional. The resulting loop executes only half as many conditional tests as the original.

```
assume (N is even); sum := 0; i := 0;
loop {if i >= N then goto Exit; sum := sum + i; i := i+1;
      assert (i < N and i is odd); sum := sum+i; i := i+1;}
Exit:
```

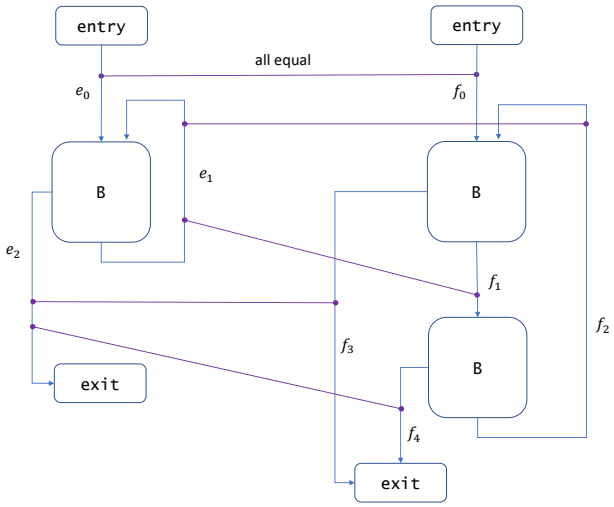

Fig. 2: Loop Unrolling, with refinement relation shown by horizontal lines.

Figure 2 illustrates the template for loop unrolling and its refinement relation. A program is expressed here in the familiar control-flow graph (CFG) form. The refinement relation connects the two state spaces, in this case it is simply the identity relation. It is the responsibility of a compiler writer to think carefully

about the correctness argument for a transform and to program a proof generator that produces the right refinement relation and any additional hints.

The proof validator tests the inductiveness of the refinement relation over loop-free path segments by generating lemmas in SMT form. In this example, assuming that states are identical at edges e_0 and f_0, they must be identical after the path segments $f_0; B; f_1$ and $e_0; B; e_1$. Continuing from that point, the segment $f_1; B; f_2$ is matched by $e_1; B; e_1$; segment $f_1; B; f_4$ is matched by $e_1; B; e_2$; and so forth. The proof-generator suggests the segment matches as hints, while the validator ensures that all path segments in the target CFG are covered.

The inductiveness checks combined with segment coverage ensure that validation is sound; i.e., it never accepts an incorrect proof. The use of SMT-supported logics to define the refinement relation ensures that a wide range of proofs can be checked automatically. Fortunately, logical theories that SMT solvers handle well — equality, uninterpreted functions, arrays and integer arithmetic — suffice for many intra-procedural optimizations.

3 WebAssembly: Syntax and Semantics

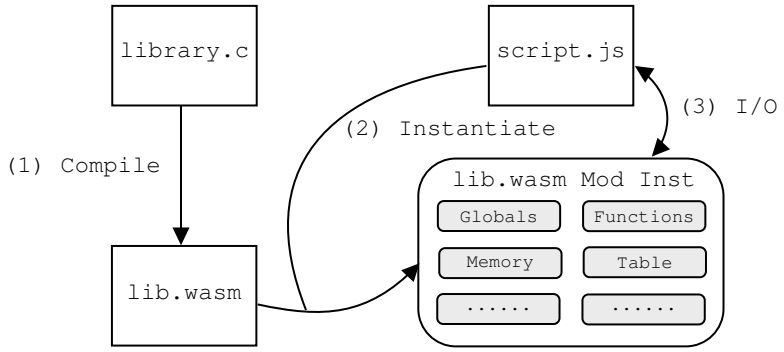

Fig. 3: WebAssembly Usage

We summarize the structure and notable features of WebAssembly[4]. A typical setup is shown in Figure 3. A C program `library.c` is (1) compiled using tools such as LLVM to produce a WebAssembly module, `lib.wasm`. This module is (2) loaded by a browser via JavaScript to create a sandboxed instance. Communication is bidirectional (3): the JavaScript code may invoke functions of `library.wasm`, and the WebAssembly code invokes JavaScript functions through a foreign-function interface for input and output actions.

τ ::= `i32` | `i64` | `f32` | `f64`
$\tau^{\rightarrow}$ ::= $\tau^\star \rightarrow \tau^\star$ (Function Type)
τ_p ::= `i8` | `i16` | `i32` (Pack Size)
x ::= i_{32} (Variable)
c ::= i_{32} | i_{64} | f_{32} | f_{64} (Constant)
a ::= i (Address)
o ::= i (Offset)
sx ::= `U` | `S` (Signage)

$unop_i$ ::= `Clz` | `Ctz` | `Popcnt`
$unop_f$ ::= `Neg` | `Abs` | `Ceil` | `Floor` | `Trunc` | `Nearest` | `Sqrt`

$binop_i$ ::= `Add` | `Sub` | `Mul` | `Div_`sx | `Rem_`sx | `And` | `Or` | `Xor` | `Shl` | `Shr_`sx | `Rotl` | `Rotr`
$binop_f$ ::= `Add` | `Sub` | `Mul` | `Div` | `Min` | `Max` | `CopySign`

$testop$ ::= `Eqz`

$relop_i$ ::= `Eq` | `Ne` | `Lt_`sx | `Gt_`sx | `Le_`sx | `Ge_`sx
$relop_f$ ::= `Eq` | `Ne` | `Lt` | `Gt` | `Le` | `Ge`

$cvtop$ ::= `ReinterpretFloat` | `ReinterpretInt` | `ExtendSI32` | ...

e ::= (Numeric Instructions)
`Const` c | `Unary` $unop$ | `Binary` $binop$ | `Test` $testop$ | `Compare` $relop$ | `Convert` $cvtop$ |

(Type-Parametric Instructions)
`Nop` | `Drop` | `Select` |

(Variable Instructions)
`LocalGet` x | `LocalSet` x | `LocalTee` x | `GlobalGet` x | `GlobalSet` x |

(Memory Instructions)
`Load` τ a o $\tau_p^?$ | `Store` τ a o $(\tau_p\ sx)^?$ | `MemorySize` | `MemoryGrow` |

(Control Flow Instructions)
`Block` $\tau^{\rightarrow}$ $e^\star$ | `Loop` $\tau^{\rightarrow}$ $e^\star$ | `If` $\tau^{\rightarrow}$ $e^\star$ | `Br` x | `BrIf` x | `BrTable` $x^\star$ x | `Return` | `Call` x | `CallIndirect` x | `Unreachable`

tys ::= `types` $\{x \rightarrow \tau^{\rightarrow}\}$
f ::= `function` $x\,\tau^\star$ $e^\star$
$funcs$ ::= `functions` $\{x \rightarrow f\}$
$glob$ ::= `global` x c
mem ::= `memory` $\{a \rightarrow i\}$
tab ::= `table` $\{x \rightarrow a\}$
mod ::= `module` tys $funcs$ $glob$ tab mem

Fig. 4: The WebAssembly instruction set of the reference interpreter: ($\star$) denotes zero or more occurrences, (?) denotes zero or one occurrence. The SSA transform introduces a "phi" assignment instruction, described in Section 5.

3.1 The WebAssembly Instruction Set

The core WebAssembly instruction set is shown in Figure 4. Our presentation is closely based on that in the reference interpreter[5]. The instruction set is a rather standard set of instructions designed for a stack machine with auxiliary local and global memory. A module instance contains a stack of values, local variable stores, a global variable store, a linear memory array, and a function table. These spaces are disjoint from each other, a security feature.

WebAssembly programs operate on a small set of basic data types: integer data (`i32`, `i64`) and floating point data (`f32`, `f64`). Booleans are `i32` values. The sign interpretation for integer values is determined by each operator in its own way (e.g., `LeU` and `LeS`). The memory model is simple. Each module has a

[4] The full specification is at https://webassembly.github.io/spec/
[5] https://github.com/WebAssembly/spec

global memory store. Each function has its own local memory store. The global *linear memory* map is a contiguous, byte-addressable memory from index 0 to an (adjustable) `MemorySize`. A function table stores references to functions, which are invoked via the `CallIndirect` instruction. All input or output is carried out through indirect calls to JavaScript functions.

Evaluation is stack based. *Numeric instructions* operate over values obtained from the top of the evaluation stack and return the result to the top of the evaluation stack. Many instructions (e.g., `add`) are polymorphic and could be partially defined (e.g., no divide by 0). Execution halts ("*traps*") if the next instruction is undefined at the current state. *Type-parametric instructions* are used to modify the evaluation stack. For instance, `drop` pops the topmost value on the stack, while `select` pops the three topmost values and pushes back one chosen value. *Variable instructions* are used to access the local variables of a function (which include its parameters), and the global store. *Memory instructions* are used to access the linear memory, and are parametrized with the index, offset, size, and type of the data to be stored or retrieved

Unusually for a low-level language, control flow in WebAssembly is structured. Break (jump) instructions like `Br` transfer control only to labels defined in the surrounding scope. A branch instruction is parametrized by an `i32` value that determines the number of nested levels to break out of. The `Block`, `Loop`, and `If` instructions generate new, well-nested labels. A function invocation via `Call` pops the appropriate number of arguments from the top of the stack and pushes back the return values when completed via `Return`. A trapping state is entered with the `Unreachable` instruction, which aborts execution.

3.2 WebAssembly Semantics

The standard semantics of WebAssembly [14] is syntax-directed, combining control and execution state in a single stack. While this is mathematically convenient, compiler optimizations are typically framed in a control-flow graph representation that separates control from execution: control is represented by the edges and execution by operations labeling vertices. The translation of a WebAssembly program to a control-flow graph is illustrated in Figure 5.

A *control-flow graph* (CFG) is a labeled directed graph $G = (V, E, \mathsf{label})$, where V is the set of vertices, $E \subseteq V \times V$ is the set of edges, and label is a labeling function. It has a single *entry* vertex with indegree 0 and outdegree 1 (the entry edge), and a single *exit* vertex with indegree 1 (the exit edge) and outdegree 0. A *path* is a sequence of vertices where each adjacent pair is an edge. Vertex m is reachable from vertex n if there is a path where the initial vertex is n and the final vertex is m. Every vertex is reachable from the entry vertex and reaches the exit vertex.

The labeling function associates each vertex with a list of basic instructions, called a *basic block*; the entry vertex is mapped to the empty list and the exit vertex to a single return instruction (the only block with this instruction). Each function call is its own block, labeling a node with indegree and outdegree 1.

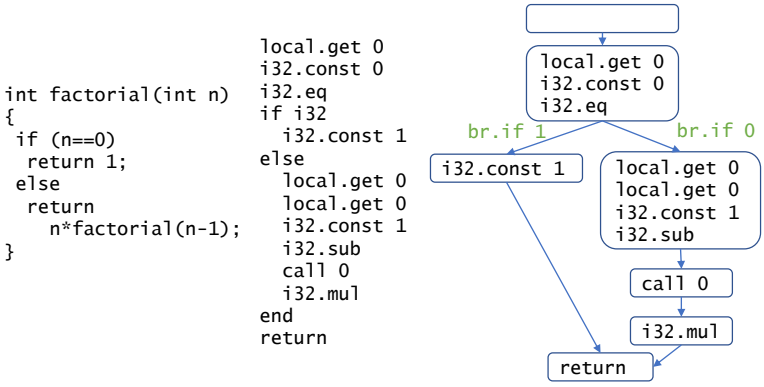

Fig. 5: Conversion from C to WebAssembly to a CFG. The `factorial` function is invoked recursively as (`call 0`) in WebAssembly.

Every edge is labeled by a Boolean-valued function; the entry edge is labeled *true*. These structural conditions simplify the validation of witnesses.

A *labeled transition system* (LTS for short) is defined by a tuple of the form (S, I, T, Σ) where S is a set of states, $I \subseteq S$ is a non-empty set of initial states, Σ is an alphabet, and $T \subseteq S \times (\Sigma \cup \{\tau\}) \times S$ is a (labeled) transition relation, where τ is a symbol not in Σ. An (infinite) *execution* from a state s is an infinite sequence of alternating states and labels $s_0 = s, a_0, s_1, a_1, s_2, \ldots$ such that for each i, the triple (s_i, a_i, s_{i+1}) is in T. The *observation* sequence of this execution is the projection of the sequence $a_0, a_1, \ldots$ on Σ. A *computation* is an execution from an initial state. The *language* of an LTS is the set of observation sequences produced by its computations.

An *evaluation context* for a function, denoted cxt, is a tuple (K, L, G, M) where K is an evaluation stack for that function, L maps local variables (including parameters) to values, G maps global variables to values, and M maps natural numbers to values, representing the linear memory.

WebAssembly Semantics The semantics of a program P is given as an LTS $\mathsf{lts}(P)$, defined as follows. The set of states S consists of tuples of the form (C, G, M) where C is a *call stack* (defined next) and G and M represent the global and linear memory maps, respectively. There is a special trap state, with a self-loop. A call stack is a sequence of *frames* (also called "activation records"). A frame is a tuple $(f, K, L, (e, k))$ where f is a function name; K is an evaluation stack (a list of values); L is a local variable map whose domain includes the function parameters; and (e, k) is a *location* within the CFG for f, where e is a CFG edge and k is an index into the basic block associated with the vertex that the target of e.

The initial state is $(\hat{C}, \hat{G}, \hat{M})$. The initial call stack $\hat{C}$ contains a single activation record for the start function of a WebAssembly module. The local, global

and memory maps are initialized as defined by the WebAssembly specification, edge e is the entry edge of the start function CFG, and $k = 0$.

A transition is either a local transition modifying only the top frame on the call stack; a function call, adding a frame; or a function return, removing the top frame. Undefined behavior (e.g., a division by 0) results in a transition to the trap state. The precise definition of transitions is in the full paper.

Input and output in WebAssembly is via the foreign-function interface; the browser also has access to the global and linear memories. The only observable transitions are therefore foreign function invocations and the final transition returning from the start function with its associated global and linear memories.

Definition 1 (Transformation Correctness). *A program transformation modifying program P to program Q is correct if the language of the transition system for Q is a subset of the language of the transition system for P.*

Transformation correctness therefore requires that calls to foreign functions are carried out in the same order and with the same actual arguments and memory maps in both source and target programs, and that the memory values upon termination of the WebAssembly program are identical. In our proof validator, we strengthen the foreign call requirement to apply to all function calls.

4 Witness Structure and Validation

We consider a program transformation that changes the structure and labeling of a CFG for a single function, keeping parameters and entry and exit nodes unchanged. Let $\mathcal{G}$ denote the source CFG and $\mathcal{H}$ the modified CFG for that function. A *proof witness* is defined by a correspondence between the execution contexts of $\mathcal{H}$ and $\mathcal{G}$ that meets certain inductiveness conditions. We prove that these conditions suffice to establish transformation correctness. The structure of a witness is defined by the following components:

A witness identifies a subset of the edges of each graph, referred to as the *checkpoint* edges and denoted $\mathsf{ckpt}(X)$ for graph X. This set must include the entry and exit edge of the graph and form a feedback-edge set (i.e., every cycle in the graph contains a checkpoint edge). For each node labeled with a function call, its adjacent edges must be checkpoint edges.

From these structural conditions, it follows that every path from a checkpoint edge must eventually cross another checkpoint edge. Let $\mathsf{frontier}(X, e)$ be the set of finite paths in graph X that start at edge e and end at a checkpoint edge with no checkpoint edges in between.

A witness specifies a partial function $W : \mathsf{ckpt}(\mathcal{H}) \times \mathsf{ckpt}(\mathcal{G}) \to (\mathsf{cxt}(\mathcal{H}) \to \mathsf{cxt}(\mathcal{G}) \to \mathsf{Bool})$. This defines the relationship between source and target states that holds on the given pair of edges: concretely, the value of $W(f, e)$ for an edge pair (f, e) is a predicate defined on target and source contexts. The entry edges of $\mathcal{H}, \mathcal{G}$ must be in the domain of W.

The final component of a witness is a function choice that maps a pair of edges (f, e) in the domain of W and a path q in $\mathsf{frontier}(\mathcal{H}, f)$ to a path p in $\mathsf{frontier}(\mathcal{G}, e)$. This relates paths in the target to paths in the source.

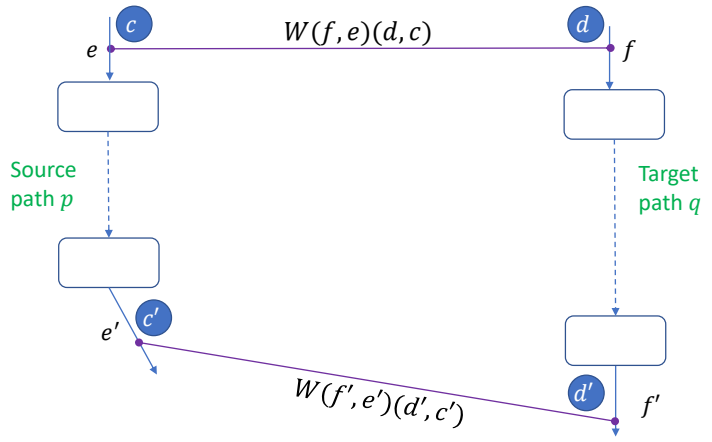

Fig. 6: Path matching and inductiveness. Note the mnemonic patterns: edges (e/f), paths (p/q), and contexts (c/d) for the source/target programs.

Valid Witnesses As illustrated in Figure 6, let (f, e) be an edge pair such that $W(f, e)$ is defined. Let q be a path in $\mathsf{frontier}(\mathcal{H}, f)$ and let $p = \mathsf{choice}((f, e), q)$ be its matching path in $\mathsf{frontier}(\mathcal{G}, e)$. Let f', e' be the final edges on the paths q, p respectively.

A *basic* path is one where no vertex is labeled with a function call or return. Every basic path q induces a sequence denoted $\mathsf{actions}(q)$ that contains only non-call-or-return instructions labeling the vertices and Boolean guards labeling the edges on the path. The semantics of each action a in context d is specified by (1) predicate $\mathsf{def}(a, d)$ which is true if a is well-defined at d; (2) predicate $\mathsf{en}(a, d)$ which is true (assuming a is well-defined) if a is enabled at d; and (3) a partial function $\mathsf{eval}(a, d)$ (defined if a is well-defined and enabled) which produces a new context. For WebAssembly instructions, eval is defined in Figure 8; def and en are defined in the full version of this paper.

For an action sequence σ, the predicate $\mathsf{enabled}(\sigma, d)$ is recursively defined as follows, where $d' = \mathsf{eval}(a, d)$:

$$\begin{aligned} \mathsf{enabled}(\epsilon, d) &= \mathit{true} \\ \mathsf{enabled}(a : x, d) &= \mathsf{def}(a, d) \wedge \mathsf{en}(a, d) \wedge \mathsf{enabled}(x, d'). \end{aligned}$$

Similarly, if σ is enabled at d, then $\mathsf{exec}(\sigma, d)$ and $\mathsf{trapped}(\sigma, d)$ are defined by

$$\begin{aligned} \mathsf{exec}(\epsilon, d) &= d \\ \mathsf{exec}(a : x, d) &= \mathsf{exec}(x, d') \\ \mathsf{trapped}(\epsilon, d) &= \mathit{false} \\ \mathsf{trapped}(a : x, d) &= \neg\mathsf{def}(a, d) \vee (\mathsf{en}(a, d) \wedge \mathsf{trapped}(x, d')). \end{aligned}$$

We define $\mathsf{enabled}(q, d)$ for a path q as $\mathsf{enabled}(\mathsf{actions}(q), d)$, and similarly define $\mathsf{eval}(q, d)$ and $\mathsf{trapped}(q, d)$. The witness validity conditions are defined as follows:

Initiality For the entry edges $\hat{f}, \hat{e}$ of $\mathcal{H}$ and $\mathcal{G}$, respectively, $W(\hat{f}, \hat{e})$ must be the identity relation.

Path Matching q and p are either basic paths; or both have only a single vertex that is labeled either with return, or with identical function calls.

Enabledness Path p is enabled if q is enabled. That is[6],

$$[W(f,e)(d,c) \wedge \mathsf{enabled}(q,d) \Rightarrow \mathsf{enabled}(p,c)]$$

Trapping If path q leads to a trap, so does path p.

$$[W(f,e)(d,c) \wedge \mathsf{trapped}(q,d) \Rightarrow \mathsf{trapped}(p,c)]$$

Non-blocking: Basic Path Execution of q cannot be blocked. Precisely, for a basic path q, the Boolean guard on edges on q other than the initial and final edge must be true for every initial context d such that $W(f,e)(c,d)$ holds.

Inductiveness: Basic Path The context obtained by executing the instructions on path q is related by W to the context obtained by executing instructions on path p.

$$\begin{aligned}[W(f,e)(d,c) \wedge \mathsf{enabled}(q,d) \wedge d' = \mathsf{exec}(q,d) \\ \wedge\ c' = \mathsf{exec}(p,c) \Rightarrow W(f',e')(d',c')].\end{aligned}$$

Inductiveness: Function Call The paths have identical call instructions. The call state (global, linear memory, and parameter values) should be identical prior to the call. Assuming the call requires k parameters, the length k stack prefixes $d(K)[0..k)$ and $c(K)[0..k)$, which represent the top k values on the respective stacks, must be identical.

$$[W(f,e)(d,c) \Rightarrow d(G,M,K[0..k)) = c(G,M,K[0..k))]$$

W must also hold of the stacks obtained after removing the call arguments:

$$\begin{aligned}[W(f,e)(d,c) \wedge d'(L,G,M) = d(L,G,M) \wedge d'(K) = d(K[k..]) \\ \wedge\ c'(L,G,M) = c(L,G,M) \wedge c'(K) = c(K[k..]) \Rightarrow W(f,e)(d',c')]\end{aligned}$$

These conditions ensure that corresponding calls behave identically. After the calls, $W(f',e')$ must hold with the (unspecified but same) result v:

$$\begin{aligned}[W(f,e)(d,c) \wedge d'(G,M) = c'(G,M) \\ \wedge\ d'(K) = v : d(K[k..]) \wedge c'(K) = v : c(K[k..]) \\ \wedge\ c'(L) = c(L) \wedge d'(L) = d(L) \Rightarrow W(f',e')(d',c')]\end{aligned}$$

Inductiveness: Function Return The values at the top of the stack must be identical, as must the global and linear memories. I.e.,

$$[W(f,e)(d,c) \Rightarrow d(G,M) = c(G,M) \wedge d(K)[0] = c(K)[0]]$$

[6] The formal statements follow Dijkstra-Scholten convention [9], where $[\varphi]$ indicates that the expression φ is valid. We use $d(X,Y,\ldots)$ to abbreviate $(d(X), d(Y), \ldots)$.

Theorem 1. *(**Soundness**) If there is a valid proof witness for a transformation from program P (CFG $\mathcal{G}$) to program Q (CFG $\mathcal{H}$), the transformation from P to Q is correct.*

Proof. (Sketch) We have to show that the language of $\mathsf{lts}(Q)$ is a subset of the language of $\mathsf{lts}(P)$. This is done by setting up a simulation relation between the (unbounded) state spaces of these transition systems. Roughly speaking, this relation matches the sequence of frames on the source call stack with those on the target call stack, relating frames for unmodified functions with the identity relation, and relating frames for the modified function by W. The complete proof (in the full paper) establishes that under the witness validity conditions, this relation is a stuttering simulation that preserves observations. □

5 Proof Generation

We illustrate, using the example of the SSA transform, how a compiler writer programs a proof generator by weaving it into the optimization algorithm. Besides SSA, the system includes proof generating versions of several common optimizations, such as dead store elimination, loop unrolling, constant propagation and folding, a "compress-locals" transform peculiar to WebAssembly which compacts the local memory array, removing unused entries and renaming the others, and finally the unSSA transformation that takes a program out of SSA form. In each case, the proof generator is approximately 100 lines of code; the actual transform is between 500-700 lines of code.

The safety net provided by the validator is analogous to the safety guarantee provided by a strong type system. Programming can proceed as usual, with the reassurance that validation will not allow incorrect compilation. Indeed, we have occasionally made mistakes in programming optimizations (common mistakes such as cut-and-paste errors and missing cases), which have been caught by the proof validator.

SSA is a key transformation in modern compilers. It ensures that in the target program, every variable appears on the left hand side of at most one assignment statement (hence the name). The transformation does not improve performance; instead, it essentially builds definition-use chains into the program text. This structural property considerably simplifies follow-on transformations that do optimize performance, such as dead store elimination (DSE).

For WebAssembly, we apply the SSA transformation to local memory, accessed via `LocalGet`, `LocalSet` and `LocalTee` operations. An example of the SSA transform is shown in Figure 7. The source program is on the left, the target on the right. Notice that the two assignments to index 2 at node n_1 have been replaced with assignments to fresh indexes 4 and 5 in the target program.

SSA introduces the new, so-called "phi" assignment statement. There are two distinct paths in the source program that reach the node n_2. The value of $L[2]$ differs along those paths: in the SSA version it is represented as $L'[5]$ for the left-hand path and as $L'[2]$ for the right-hand path. Those values must necessarily

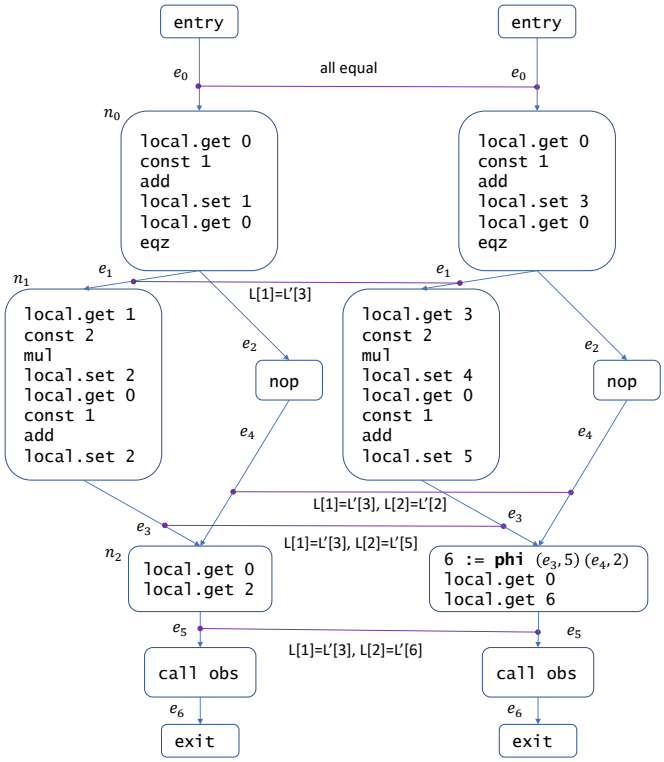

Fig. 7: Example SSA transform. Source on the left, result on the right.

be merged to correctly represent the value of $L[2]$ at source node n_2. This is the role of the phi assignment. The syntax $6 := \mathsf{phi}\,(e_3, 5)(e_4, 2)$ represents that $L'[6]$ should get the value of $L'[5]$ on an execution that follows edge e_3 and the value of $L'[2]$ on an execution that follows edge e_4.

The witness validation conditions are easily adapted to take phi-instructions into account: phi-instructions on a node of a source or target path are resolved to a simple assignment based on the path edge that enters the node.

As should be clear from this example, the SSA transformation is in essence a renaming of variables — but with a twist, in that the renaming is not uniform over the program. For example, $L[2]$ is represented at various points in the target program by $L'[2]$, $L'[4]$, $L'[5]$, and $L'[6]$. The correspondence between source and target program must reflect this fact. To avoid clutter, the figure only shows the important portions of the correspondence[7]. It should be easy to check that the full correspondence is inductive and that (as the stack and global memories

[7] The full correspondence for edge e_3 is that $K, G, M = K', G', M'$ (stack, global, and main memories are identical) and that $L[0] = L'[0]$, $L[1] = L'[3]$, and $L[2] = L'[5]$.

are identical) the call to the `obs` function (short for "observable") must obtain identical actual parameters and thus produce identical results.

A proof-generator must generate such a correspondence for the SSA transformation on *any* program. We explain next how this is done. Proof generation is necessarily dependent on the algorithm used for SSA conversion. We base the explanation on the well known algorithm of Cytron et al [7], which is implemented in our framework.

SSA algorithm The SSA conversion algorithm operates in two stages. Our description must necessarily be brief; for more detail, please refer to the original article [7]. In Stage 1, the location and form of the necessary phi-assignments is determined, while Stage 2 fills in the details of those assignments.

The first stage is technically complex. First, for each local index k, the set of nodes $\mathsf{asgn}(k)$ is determined, this is the set of nodes that contain an assignment to k (through `LocalSet` or `LocalTee`). Then the *iterated dominance frontier* of $\mathsf{asgn}(k)$ is determined; those are precisely the nodes that must have a phi-assignment for k. Dominance is a standard notion in program analysis. In short, node n *dominates* node m if every path in the CFG from the entry node to m must pass through node n. A node m is in the *dominance frontier* of node n if m is *not* strictly dominated by n but some predecessor of m is dominated by n. The dominance frontier of a set X of nodes (denoted $\mathsf{DF}(X)$) is the union of the individual dominance frontiers. The *iterated* dominance frontier (IDF) of a set X is defined as the least fixed point of the function $(\lambda Z : \mathsf{DF}(X \cup Z))$.

For our source program, $\mathsf{asgn}(2) = \{\mathsf{entry}, n_1\}$; it includes the entry node as all variables are initialized. The IDF of this set is just the singleton $\{n_2\}$. Thus there must be a phi-assignment for $L[2]$ at node n_2. However, the details of this assignment; in particular, which renamed versions of $L[2]$ reach this node, is not yet known. That information is filled in by Stage 2.

The second stage does a depth-first traversal of the CFG. For each original index, the traversal carries a stack of fresh index values that are used to rename it. For instance the stack for $L[2]$ on entering edge e_3 is $[5; 4; 2]$ with index 5 being the top entry, while the stack for $L[2]$ on entering edge e_4 is just $[2]$. In processing the instructions at a node, a (`LocalGet` k) instruction is replaced with (`LocalGet` k') where k' is the index at the *top* of the stack for k. A (`LocalSet` k) instruction is replaced with (`LocalSet` k') where k' is a *fresh* index, which is pushed on to the stack for $L[k]$. (A similar replacement occurs for instances of `LocalTee`.)

For the example program, the phi-assignment $6 := \mathsf{phi}\,(e_3, 5)(e_4, 2)$ at node n_2 is filled in by taking the indexes $5, 2$ at the top of the stacks for $L[2]$ for edges e_3 and e_4, respectively, and generating a fresh index 6.

Proof Generation We now turn to proof generation. First, note that the SSA transformation does not alter CFG structure. Thus, the correspondence relates identical edges in source and target. Moreover, the contents of the value stack K and the memories G and M are uninfluenced by this transformation. Thus, the

focus is entirely on the local variables. The key to defining the relation on each edge is knowing which fresh local index represents an original local index k on that edge. Fortunately, this information is easy to obtain. In the second stage above, the fresh index corresponding to original index k at edge e is precisely the index at the top of the stack for $L[k]$ when edge e is traversed (each edge is traversed exactly once). Thus, the template for the full correspondence at edge e is that $K, G, M = K', G', M'$ and for each original index k, $L[k] = L'[k']$ where k' is the fresh index at the top of the stack for $L[k]$ at edge e.

We have implemented the SSA transformation in about 700 lines of OCaml code (including comments). That includes the iterated dominance frontier calculation but not the calculation of the base dominance relationship, which is done separately in about 300 lines of OCaml. Proof generation is implemented in an additional 130 lines of OCaml (including comments). The implementation of the SSA algorithm and the proof generator took (we estimate) about 3 person-weeks.

6 Validator Implementation

We have so far laid out the design of the self-certifying framework and shown how to write proof generators. In this section, we describe the implementation of the validator, which builds on the reference WebAssembly implementation. It is about 6300 source lines of OCaml source code[8], which includes the proof checking algorithm, an interface to the Z3 SMT solver [22], code for manipulating control flow graphs, and utility functions. The code has substantial explanatory comments. It was developed in roughly 7 person-months of effort.

The method is defined as Algorithm 1. It receives as input two CFGs for the same WebAssembly function (the source CFG $\mathcal{G}$ and the target CFG $\mathcal{H}$), and a candidate witness object $(\mathsf{ckpt}, W, \mathsf{frontier}, \mathsf{choice})$. The algorithm then checks the witness for validity against $\mathcal{G}$ and $\mathcal{H}$, through a simple workset algorithm that repeatedly invokes the back-end SMT solver to check the validity of the given formula; the witness check fails if the formula is invalid.

The witness conditions defined in Section 4 and checked by Algorithm 1 ultimately depend on the semantics of individual actions. We supply this semantics in Figure 8 for the context $c = (K, L, G, M)$; this defines the resulting context $c' = \mathsf{eval}(a, c)$. Instruction semantics is defined in the top group, branch conditions at the bottom. `MemoryGrow` has no effect; the memory is assumed to be of a large fixed size. Label-context pairs not listed here are undefined. The bulk of the implementation effort is in the encoding this semantics in SMT terms.

6.1 Encoding into SMT

We now describe how the action semantics can be encoded into appropriate first-order logical theories. Conceptually, the process is straightforward; nevertheless, an actual implementation must resolve or work around several complexities.

[8] Excluding comments. Measured with `cloc`: https://github.com/AlDanial/cloc

Context c	Label a		$\mathsf{eval}(a, c)$	Condition
(K, L, G, M)	`Const` c	$\hookrightarrow$	$(c : K, L, G, M)$	
$(v_1 : K, L, G, M))$	`Unary` op	$\hookrightarrow$	$(v : K, L, G, M)$	if $v = op(v_1)$
$(v_1 : v_2 : K, L, G, M)$	`Binary` bop	$\hookrightarrow$	$(v : K, L, G, M)$	if $v = op(v_1, v_2)$
$(v_1 : K, L, G, M)$	`Test` op	$\hookrightarrow$	$(v : K, L, G, M)$	if $v = op(v_1)$
$(v_1 : v_2 : K, L, G, M)$	`Compare` op	$\hookrightarrow$	$(v : K, L, G, M)$	if $v = op(v_1, v_2)$
$(v_1 : v_2 : K, L, G, M)$	`Convert` op	$\hookrightarrow$	$(v : K, L, G, M)$	if $v = op(v_1, v_2)$
(K, L, G, M)	`Nop`	$\hookrightarrow$	(K, L, G, M)	
$(v : K, L, G, M)$	`Drop`	$\hookrightarrow$	(K, L, G, M)	
$(i : v_1 : v_2 : K, L, G, M)$	`Select`	$\hookrightarrow$	$(v_1 : K, L, G, M)$	if $i = 0_{\texttt{i32}}$
$(i : v_1 : v_2 : K, L, G, M)$	`Select`	$\hookrightarrow$	$(v_2 : K, L, G, M)$	if $i \neq 0_{\texttt{i32}}$
(K, L, G, M)	`LocalGet` x	$\hookrightarrow$	$(v : K, L, G, M)$	if $v = L(x)$
$(v : K, L, G, M)$	`LocalSet` x	$\hookrightarrow$	(K, L', G, M)	$L' = L[x \mapsto v]$
$(v : K, L, G, M)$	`LocalTee` x	$\hookrightarrow$	$(v : K, L', G, M)$	$L' = L[x \mapsto v]$
(K, L, G, M)	`GlobalGet` x	$\hookrightarrow$	$(v : K, L, G, M)$	if $v = G(x)$
$(v : K, L, G, M)$	`GlobalSet` x	$\hookrightarrow$	(K, L, G', M)	$G' = G[x \mapsto v]$
$(x : K, L, G, M)$	`Load` o	$\hookrightarrow$	$(v : K, L, G, M)$	if $v = M(x + o)$
$(v : x : K, L, G, M)$	`Store` o	$\hookrightarrow$	(K, L, G, M')	$M' = M[x + o \mapsto v]$
(K, L, G, M)	`MemorySize`	$\hookrightarrow$	$(v : K, L, G, M)$	$v =$ `MEMORY_SIZE`
$(v : K, L, G, M)$	`MemoryGrow`	$\hookrightarrow$	$(v : K, L, G, M)$	$v =$ `MEMORY_SIZE`
(K, L, G, M)	`Br`	$\hookrightarrow$	(K, L, G, M)	
$(v : K, L, G, M)$	`If` *true*	$\hookrightarrow$	(K, L, G, M)	$v \neq 0_{\texttt{i32}}$
$(v : K, L, G, M)$	`If` *false*	$\hookrightarrow$	(K, L, G, M)	$v = 0_{\texttt{i32}}$
$(v : K, L, G, M)$	`BrIf` *true*	$\hookrightarrow$	(K, L, G, M)	$v \neq 0_{\texttt{i32}}$
$(v : K, L, G, M)$	`BrIf` *false*	$\hookrightarrow$	(K, L, G, M)	$v = 0_{\texttt{i32}}$
$(v : K, L, G, M)$	`BrIndex` d	$\hookrightarrow$	(K, L, G, M)	$v = d$
$(v : K, L, G, M)$	`BrDefault` l	$\hookrightarrow$	(K, L, G, M)	$l \leq v$

Fig. 8: The definition of $c' = \mathsf{eval}(a, c)$, where c is the current evaluation context, a is a local transition action label, and c' is the following context. Basic instructions are in the top group; branch conditions are in the bottom group; transitions not defined are assumed to `trap` by default. The notation $L[x \mapsto v]$ denotes a map identical to L except at element x where its value is v.

The *fully interpreted* encoding must represent the `i32`, `i64`, `f32` and `f64` datatypes precisely. Integer types are represented with bitvectors to properly account for low-level bit manipulation with `Xor` and `Rotr` instructions. However, encoding floating point types is a challenge. The current fully interpreted encoding applies only to programs over `i32` values, that do not use `MemoryGrow`, correctly specify `MEMORY_SIZE`, and where load and store memory operations are `i32`-aligned. This encoding is used to check proofs for constant propagation and folding on `i32` values.

On the other hand, proofs of several transformations (including all other implemented transformations and others such as loop peeling and common subexpression elimination) amount to reasoning about substitution under equality. For such proofs, a *fully uninterpreted* encoding suffices to check refinement. A signif-

Algorithm 1 Witness Checking Algorithm

```
1: procedure RefinementCheck(𝒢, ℋ, witness = (ckpt, W, frontier, choice))
2:     Initialize workset to {(f̂, ê)}, the entry edges
3:     while workset is not empty do
4:         remove checkpoint edge pair (f, e) from workset, mark it as visited
5:         if (f, e) is not in domain(W), abort (bad witness structure)
6:         for all paths q in frontier(ℋ, f) do
7:             let f′ be the final edge of q
8:             let p = choice((f, e), q) be the corresponding source path, final edge e′
9:             invoke an SMT solver to check witness conditions from Section 4 on q, p
10:            add (f′, e′) to the workset if not visited
11:        end for
12:    end while
13: end procedure
```

icant advantage of the uninterpreted encoding is that the validator can handle *all* WebAssembly programs, without restrictions. Proof witnesses also specify the encoding that is to be used to check their validity.

These two options naturally suggest a third, a *partially interpreted* SMT encoding where, say, all `int32` and `int64` operations are fully interpreted in the theory of bitvectors, while floating point operations are uninterpreted. We are in the process of developing such an interpretation; it would remove many of the restrictions currently placed by the fully interpreted encoding.

6.2 Evaluation

The goal of our evaluation was to test how well our prototype implementation scales on real programs. To do this, we ran our checker against the proofs generated by proof-generating optimizations on two benchmarks: the WebAssembly reference interpreter's test suite (https://github.com/WebAssembly/spec/) and the WebP image library (https://github.com/webmproject/libwebp). We found that nearly all proofs are easily verified, although a small percentage of checks fail because the SMT solver is a bottleneck.

Procedure We first gathered WebAssembly S-expression (WAST) files from each benchmark. This was either already provided in the case of the reference interpreter (73 files, 3036 functions, 49113 LoC, total 2.6 MB), or in the case of WebP's C implementation, can be compiled to WebAssembly using Emscripten[9] (1 file, 953 functions, 328780 LoC, total 6.8 MB).

Next, for each function of each module of each file we ran the following:

(1) Convert the function into a source CFG.
(2) Run an optimization (either SSA, SSA + unSSA, SSA + DSE, or Loop unroll), which generates a target CFG and a proof witness to be checked.

[9] https://developer.mozilla.org/en-US/docs/WebAssembly/C_to_wasm

(3) For each witness, generate SMT lemmas as in Algorithm 1 and pass those to the Z3 SMT solver if they meet a heuristic size restriction. If the lemma is too large or if the solver times out, the check is considered unsuccessful.

All experiments were run with Z3 4.8.7 on a machine with 30 GB RAM and an AMD Ryzen 7 PRO 3700U CPU. The proof validation process is parallelizable: each function of each module can be checked separately. All proof lemmas associated with a source-target pair of CFGs can also be checked separately. We do not, however, use parallelization in this evaluation, and this is reflected in the relatively low CPU (typically $< 30\%$) and RAM (typically $< 25\%$) usage throughout the experiments.

Reference Interpreter (49113 LoC)					
	SMT/Total Time (s)		**Checked/Total SMT**		**Faulty**
SSA	1412.4/1415.5	$\approx 99.8\%$	45376/45380	$\approx 99.9\%$	0
SSA + unSSA	2374.2/2377.7	$\approx 99.8\%$	77584/77592	$\approx 99.9\%$	0
SSA + DSE	1547.9/1551.3	$\approx 99.8\%$	49596/49604	$\approx 99.9\%$	0
Loop Unroll	12.6/14.6	$\approx 86.3\%$	488/488	$= 100\%$	0
WebP Image Library (328780 LoC)					
	SMT/Total Time (s)		**Checked/Total SMT**		**Faulty**
SSA	13593.4/13617.5	$\approx 99.8\%$	135088/135156	$\approx 99.9\%$	0
SSA + unSSA	27339.3/27364.4	$\approx 99.9\%$	237460/266492	$\approx 89.1\%$	101?
SSA + DSE	21068.5/21095.8	$\approx 99.9\%$	231068/266116	$\approx 86.8\%$	0
Loop Unroll	3589.4/3606.0	$\approx 99.5\%$	38036/38036	$= 100\%$	0

Fig. 9: We examine four optimizations on two different benchmarks. The total number of SMT lemmas is a multiple of four because we check that the source-target paths are (1) inductive, (2) enabled, (3) non-trapping, and (4) non-blocking. A checked lemma is correct if the solver (Z3 with timeout = 2 sec) returns `Unsat`; it is considered faulty if the solver returns `Sat` and potentially faulty if the solver returns `Unknown`. We do not check lemmas that are too large with respect to a size heuristic.

Results and Discussion Our results are summarized in Figure 9. First, 101/237460 ($\approx 0.04\%$) of SSA + unSSA's lemmas are *potentially* faulty, as the solver returns `Unknown` on these instances rather than `Unsat` (correct) or `Sat` (faulty). However, upon isolating several of these cases and re-running the solver with longer timeouts, the sampled `Unknown`s were in fact `Unsat`, and therefore correct. In a similar vein, some lemmas are unchecked because of heuristic size restrictions. Thus, although we have not completely verified the optimizations on the reference interpreter and WebP, we have, however, succeeded in verifying a significant portion. Furthermore, every skipped lemma that we have manually extracted and checked has also been valid.

Second, the solver calls dominate runtime in all experiments, which is expected in part due to the sheer amount of queries. Fortunately the check of each

lemma is usually fast, on average we check about 11 lemmas per second, but without timeout settings we have observed exceptional outliers. For simplicity, SMT lemmas are written out as SMT-LIB2 strings that are piped into Z3 rather than via Z3's direct OCaml bindings; this reduces performance somewhat.

Finally, an obvious point is that additional machine resources would improve the evaluation results. For one, increasing the size restriction allows more lemmas to be checked. Additionally, Z3 and the overall pipeline would also be faster — all this without parallelizing proof checking. In summary, the evaluation results here give us confidence that self-certification can be feasibly adopted in practice.

7 Related Work and Conclusions

This work is inspired by and builds upon a large body of prior work on compiler verification. We highlight the most closely related work below.

Mechanized Proof The seminal work on mechanized proof of compiler optimizations is by McCarthy and Painter from 1960s [21]. Mechanized proofs have been carried out in several settings, notable ones are for the Lisp compiler in the CLI stack [4] and for the C compiler CompCert [18,19]. Such proofs require enormous effort and considerable mathematical expertise — the CompCert and CLI proofs each required several person-years. A proof of a roughly 800-line SSA transformation needed nearly a person-year and over 10,000 lines of Coq proof script [36], illustrating the difficulty of the problem. As explained in the Introduction, there are close connections between deductive proof methods and self-certification.

Translation Validation Translation Validation (TV) [32,30,28,37,1,8,6] is a form of result checking [5]. Compilation is treated as a black-box process; the validator has access only to the input and output programs. As explained in the Introduction, specialized heuristics must be crafted for each optimization. Incompleteness of these heuristics shows up in missed equivalences, for instance [8] report that about 25% of equivalences were not detected on a particular test suite. The complexity of some TV validators raises the question of whether the validators are themselves correct. Unfortunately, verifying a TV validator is difficult. For instance, the verification of a 1000-line TV validator for SSA [2] needed over a person-year of effort and 15,000 lines of Coq proof script.

Self-certification avoids the introduction of transformation-specific heuristics. In principle, self-certification is complete. In practice, it is possible for SMT solvers to run out of time or memory and thus produce an "unknown" result.

Self certification We discuss prior work on compiler self-certification in the Introduction; we do so now in more detail. Credible compilation was first implemented in [20] for a basic textbook-style intermediate language. It has proved to be challenging to implement self-certification for languages used in practice. The implementation of witnessing for LLVM in [26,11] handles only a small subset of LLVM IR and simple optimizations. Validation of the LLVM SSA transform

is shown in [24], but that validator uses a simplified LLVM semantics and proof generation is somewhat incomplete.

The most thorough implementation of certification to date is in the Crellvm system for LLVM [16]. A Crellvm proof consists of Extended Relational Hoare Logic (ERHL) assertions (cf. [3]) that connect corresponding source and target program points, together with hints for instantiating inference rules. The validator applies the given hints to check the inductiveness of the supplied ERHL assertions. The limitations arise from (1) the ERHL logic, which is syntax-driven, and thus cannot be used to witness the correctness of transformations which modify control structure, such as loop unrolling; and from (2) the large collection of custom-built inference rules (221 in the current system), each of which must be formally verified. In contrast, our WebAssembly validator is based on a small set of refinement proof rules, with all of the detailed logical and arithmetic reasoning left to a generic SMT solver. This modular design simplifies the validator implementation, while the proof format is sufficiently expressive to support all of the Crellvm optimizations and more, including loop unrolling.

Regression verification A related line of work is that of regression verification [12,17,13], which establishes the equivalence of structurally similar recursive programs. Each procedure body is loop-free (loops are converted to recursion), simplifying equivalence checking through SMT encoding. The original work bases equivalence on a fixed relation with identical parameter values. Some of these limitations have been overcome in later work [10,33,6] through stronger program equivalence heuristics. The key difference is that self-certification, by design, involves the compiler writer in the process and thus does not require heuristics.

Several enhancements are of interest. One is the extension of self-certification to complex transformations that require specialized proof methods. Rules for validating loop transformations were developed and implemented in the TVOC project [37,1] and re-implemented for LLVM [25]. Rules for validating interprocedural transformations such as tail-recursion elimination and inlining are developed in [34]. A second interesting project is to produce a formally verified validator, mechanizing the soundness proof of Theorem 1.

Acknowledgments. This material is based upon work supported by the National Science Foundation under Grant CCF-1563393. Any opinions, findings, and conclusions or recommendations expressed in this material are those of the author(s) and do not necessarily reflect the views of the National Science Foundation.

References

1. Barrett, C.W., Fang, Y., Goldberg, B., Hu, Y., Pnueli, A., Zuck, L.D.: TVOC: A translation validator for optimizing compilers. In: CAV. pp. 291–295 (2005)
2. Barthe, G., Demange, D., Pichardie, D.: Formal Verification of an SSA-Based Middle-End for CompCert. ACM Trans. Program. Lang. Syst. **36**(1), 4 (2014)
3. Benton, N.: Simple relational correctness proofs for static analyses and program transformations. In: POPL. pp. 14–25 (2004)

4. Bevier, W.R., Hunt, W.A., Moore, J.S., Young, W.D.: An approach to systems verification. J. Autom. Reasoning **5**(4), 411–428 (1989)
5. Blum, M., Kannan, S.: Designing programs that check their work. J. ACM **42**(1), 269–291 (1995)
6. Churchill, B.R., Padon, O., Sharma, R., Aiken, A.: Semantic program alignment for equivalence checking. In: PLDI. pp. 1027–1040 (2019)
7. Cytron, R., Ferrante, J., Rosen, B.K., Wegman, M.N., Zadeck, F.K.: Efficiently computing static single assignment form and the control dependence graph. ACM Trans. Program. Lang. Syst. **13**(4), 451–490 (1991)
8. Dahiya, M., Bansal, S.: Modeling undefined behaviour semantics for checking equivalence across compiler optimizations. In: HVC. pp. 19–34 (2017)
9. Dijkstra, E.W., Scholten, C.S.: Predicate Calculus and Program Semantics. Texts and Monographs in Computer Science, Springer (1990)
10. Fedyukovich, G., Gurfinkel, A., Sharygina, N.: Automated discovery of simulation between programs. In: LPAR. LNCS, vol. 9450, pp. 606–621. Springer (2015)
11. Gjomemo, R., Namjoshi, K.S., Phung, P.H., Venkatakrishnan, V., Zarzani, N., Zuck, L.D.: From Verification to Optimizations. In: VMCAI 2015. Springer (2015)
12. Godlin, B., Strichman, O.: Regression verification. In: Proceedings of the 46th Design Automation Conference, DAC 2009, San Francisco, CA, USA, July 26-31, 2009. pp. 466–471 (2009)
13. Godlin, B., Strichman, O.: Regression verification: proving the equivalence of similar programs. Softw. Test., Verif. Reliab. **23**(3), 241–258 (2013)
14. Haas, A., Rossberg, A., Schuff, D.L., Titzer, B.L., Holman, M., Gohman, D., Wagner, L., Zakai, A., Bastien, J.F.: Bringing the web up to speed with WebAssembly. In: PLDI. pp. 185–200 (2017)
15. Jourdan, J., Pottier, F., Leroy, X.: Validating LR(1) parsers. In: ESOP. pp. 397–416 (2012)
16. Kang, J., Kim, Y., Song, Y., Lee, J., Park, S., Shin, M.D., Kim, Y., Cho, S., Choi, J., Hur, C., Yi, K.: Crellvm: verified credible compilation for LLVM. In: PLDI. pp. 631–645 (2018)
17. Lahiri, S.K., Hawblitzel, C., Kawaguchi, M., Rebêlo, H.: SYMDIFF: A language-agnostic semantic diff tool for imperative programs. In: CAV. pp. 712–717 (2012)
18. Leroy, X.: Formal certification of a compiler back-end or: programming a compiler with a proof assistant. In: POPL. pp. 42–54. ACM (2006)
19. Leroy, X.: Formal verification of a realistic compiler. Commun. ACM **52**(7), 107–115 (2009)
20. Marinov, D.: Credible Compilation. Master's thesis, Massachusetts Institute of Technology (2000)
21. McCarthy, J., Painter, J.: Correctness of a compiler for arithmetic expressions. pp. 33–41. American Mathematical Society (1967)
22. de Moura, L.M., Bjørner, N.: Z3: an efficient SMT solver. In: TACAS. LNCS, vol. 4963, pp. 337–340. Springer (2008), https://github.com/Z3Prover/z3
23. Namjoshi, K.S.: Certifying model checkers. In: CAV. pp. 2–13 (2001)
24. Namjoshi, K.S.: Witnessing an SSA transformation. In: VeriSure Workshop, CAV (2014), https://kedar-namjoshi.github.io/papers/Namjoshi-VeriSure-CAV-2014.pdf
25. Namjoshi, K.S., Singhania, N.: Loopy: Programmable and formally verified loop transformations. In: SAS. pp. 383–402 (2016)
26. Namjoshi, K.S., Tagliabue, G., Zuck, L.D.: A witnessing compiler: A proof of concept. In: RV. pp. 340–345 (2013)

27. Namjoshi, K.S., Zuck, L.D.: Witnessing Program Transformations. In: SAS 2013. pp. 304–323 (2013)
28. Necula, G.: Translation Validation of an Optimizing Compiler. In: (PLDI) 2000. pp. 83–95 (2000)
29. Peled, D.A., Pnueli, A., Zuck, L.D.: From falsification to verification. In: FSTTCS. pp. 292–304 (2001)
30. Pnueli, A., Shtrichman, O., Siegel, M.: The Code Validation Tool (CVT)- Automatic Verification of a Compilation Process. Software Tools for Technology Transfer **2**(2), 192–201 (1998)
31. Rinard, M.: Credible Compilation. Tech. Rep. MIT-LCS-TR-776, MIT (1999)
32. Samet, H.: Automatically proving the correctness of translations involving optimized code - research sponsored by Advanced Research Projects Agency, ARPA order no. 2494. Ph.D. thesis, Stanford University (1975)
33. Strichman, O., Veitsman, M.: Regression verification for unbalanced recursive functions. In: FM 2016. LNCS, vol. 9995, pp. 645–658 (2016)
34. Zaks, A., Pnueli, A.: Program analysis for compiler validation. In: PASTE. pp. 1–7 (2008)
35. Zhang, L., Malik, S.: Validating SAT solvers using an independent resolution-based checker: Practical implementations and other applications. In: DATE. pp. 10880–10885 (2003)
36. Zhao, J., Nagarakatte, S., Martin, M.M.K., Zdancewic, S.: Formal verification of SSA-based optimizations for LLVM. In: PLDI '13. pp. 175–186 (2013)
37. Zuck, L.D., Pnueli, A., Goldberg, B., Barrett, C.W., Fang, Y., Hu, Y.: Translation and run-time validation of loop transformations. Formal Methods in System Design **27**(3), 335–360 (2005)