Stack-Driven Program Generation of WebAssembly

Árpád Perényi and Jan Midtgaard ✉

The Maersk Mc-Kinney Moller Institute, University of Southern Denmark
Campusvej 55, 5230 Odense M, Denmark
arpad.perenyi@gmail.com mail@janmidtgaard.dk

Abstract. WebAssembly (Wasm) is a popular portable assembly-like language. Besides browser support in the four most common browsers (Chrome, Firefox, Safari, Edge) a number of standalone Wasm engines are available. With several such independent implementations naturally follows a risk of disagreement between the individual implementations.
To help ensure agreement between Wasm implementations, we develop a stack-directed program generator to drive differential testing of the four browsers' Wasm engines. We describe our experimental setup, our development of a stack-directed shrinker for reducing a generated counterexample program, and finally report on a number of disagreements and bugs found. Surprisingly our black-box generator found 2 crashing bugs, despite browser vendor efforts to fuzz test their Wasm engines using a state-of-the-art fuzzer.

1 Introduction

WebAssembly (Wasm) is a new open web standard [26] for executing low-level code in web pages. In order to succeed, Wasm programs should be interpreted consistently by the four major browsers implementations (Chrome, Firefox, Safari, Edge). To ensure such consistency both a reference interpreter and an extensive test suite is available. Given the incompleteness of testing, one may wonder whether these efforts are sufficient to guarantee consistency. In this paper we present a generator of arbitrary Wasm programs and report on testing for this consistency. Furthermore we present a *shrinker* to automatically reduce a machine-generated counterexample illustrating inconsistency.

```
(module
  (func)
  (start 0)
  (table $0 1 anyfunc)
  (elem 0
    (offset (i32.const 0)) 0))
```

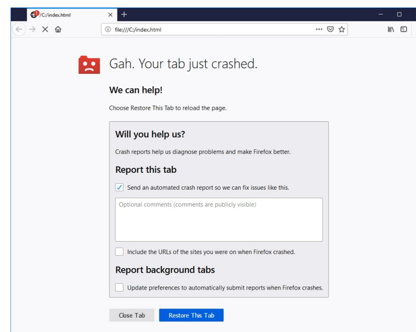

Fig. 1: Wasm program crashing SpiderMonkey

```
(module
  (func (param i32) (param i32) (result i32)
    (get_local 0)
    (get_local 1)
    (i32.add))
  (export "add" (func 0)))
```

Listing 1: A Wasm module in text format with a simple addition function

For example, a Wasm program produced by our generator was able to crash SpiderMonkey, the JavaScript engine inside the Firefox web browser. Fig. 1 illustrates a reduced version of the test case and Firefox's behavior upon attempting to run it.

Overall the contributions of this paper are:

- We suggest the ideas of (backwards) stack-directed program generation and stack-directed shrinking.
- We illustrate the approach with an application to WebAssembly.
- We demonstrate that the approach is both viable and useful as illustrated by a number of real-world Wasm engine bugs found (including crashing bugs).
- We discuss ours findings, documenting real-world bugs that escaped a coverage-guided "gray-box" fuzzer thus questioning the current focus on such generators.

2 Background

We first present background material on Wasm and property-based testing.

2.1 WebAssembly

The Wasm standard defines a low-level programming language for a stack-based virtual machine [12]. For example, Listing 1 shows a simple Wasm module with a function that takes two arguments and returns their sum. The function loads each of the numbered parameters onto the operand stack, adds them, and leaves the result on the stack. Wasm is designed for embedding. This is expressed as *exporting* functions for the surrounding context to call and *importing* functions from the surrounding context for Wasm to call. For example, Listing 1 exports the module's function under the name `"add"`. In a web-embedding context, this means Wasm modules can call imported JavaScript functions and that JavaScript can call the exported Wasm functions. Similarly a Wasm module can import and export functions from other Wasm modules.

Wasm programs can be expressed in both a textual assembly-like format (`.wat`) as in Listing 1 and in a corresponding binary format (`.wasm`). Translators are available to translate between the two. In a web-embedding context a Wasm-module can be loaded from an untrusted source into a browser. A module then has to be validated to ensure that it is well formed and safe to run. The validator is phrased as a type system centered around four *value types*: `i32`, `i64`, `f32` and `f64`, denoting 32-bit and 64-bit integers and 32-bit and 64-bit floating-point numbers, respectively. For the example in Listing 1

```
(module
  (table 10 funcref)
  (func $f)
  (func $f2 (call_indirect 3))
  (elem (i32.const 0) $f)
  (elem (i32.const 3) $f))
```

Listing 2: Wasm table initialization

the validator checks that the two arguments on the stack when performing `i32.add` are indeed `i32`s and that the `i32` result agrees with the function's declared result type.

A Wasm program consists of one or more modules. Besides functions, a module can contain a combination of elements which we now cover.

Global variables A Wasm module can contain global variables. A global variable can be accessed throughout the module using the `get_global` instruction. Each global is declared with a value type and optionally marked as *mutable*. Mutable global variables can be updated using the `set_global` instruction. Global variables can both be imported and exported.

Memories and data segments A Wasm module can also contain a memory which is a mutable array of raw bytes. By default the memory is initialized with zeroes. A module can contain a separate section of *data segments* that each specify the initial memory contents at a specific *offset* and *length*. When a Wasm module is loaded and *instantiated*, the allocated memory is initialized accordingly before Wasm code is run.

Functions A module can contain multiple functions. Each function's parameters are defined as locals and can only be accessed by the defining function. Locals are mutable. They can be read and updated with the `get_local` and `set_local` instructions, which push and pop values to and from the stack. A function can optionally declare a return value type. In the current version of Wasm, a function can return at most one value. A function's body is a possibly empty instruction sequence. The instructions may interact with the stack, locals, globals, memories, or tables. A module can also contain a dedicated *start function*. The start function is executed automatically after the memories and tables have been initialized. The start function cannot take any arguments or return any value. For example, the Wasm module in Fig. 1 designates function `0`, the module's only (empty) function as the start function.

Tables A Wasm module may contain a table of functions. In the current version of Wasm, a module can only contain a single table instance. A table instance can be defined by the module itself or imported from another module, hence a table can also be exported. Tables require a minimum size and can optionally declare a maximum size. A table can also be initialized through *element segments*. The `call_indirect` instruction calls a function through a table. Listing 2 shows an example of a table initialization via element segments. The table is declared to contain 10 elements, with `funcref` (function reference) type.[1] The named function `$f` is then added to the table at indices 0 and 3. The named function `$f2` calls the function `$f` indirectly through the table.

Instructions We summarize the abstract syntax of Wasm instructions and modules in Fig. 2. Instructions can consume multiple arguments and produce a result value by

[1] In revised text format `https://github.com/WebAssembly/spec/issues/884`

$$
\begin{array}{lrl}
\text{(value types)} & t & ::= \texttt{i32} \mid \texttt{i64} \mid \texttt{f32} \mid \texttt{f64} \\
\text{(packed types)} & tp & ::= \texttt{i8} \mid \texttt{i16} \mid \texttt{i32} \\
\text{(function types)} & tf & ::= t^* \rightarrow t^* \\
\text{(global types)} & tg & ::= \texttt{mut}^? \; t \\
\text{(functions)} & f & ::= ex^* \; \texttt{func} \; tf \; \texttt{local} \; t^* \; e^* \\
 & & \mid \; ex^* \; \texttt{func} \; tf \; e^* \; im \\
\text{(globals)} & glob & ::= ex^* \; \texttt{global} \; tg \; e^* \\
 & & \mid \; ex^* \; \texttt{global} \; tg \; im \\
\text{(tables)} & tab & ::= ex^* \; \texttt{table} \; n \; i^* \\
 & & \mid \; ex^* \; \texttt{table} \; n \; im \\
\text{(memories)} & mem & ::= ex^* \; \texttt{memory} \; n \\
 & & \mid \; ex^* \; \texttt{memory} \; n \; im \\
\text{(imports)} & im & ::= \mathsf{import} \; \text{“}\mathit{name}\text{”} \; \text{“}\mathit{name}\text{”} \\
\text{(exports)} & ex & ::= \mathsf{export} \; \text{“}\mathit{name}\text{”} \\
\text{(modules)} & m & ::= \texttt{module} \; f^* \; glob^* \; tab^? \; mem^?
\end{array}
$$

$$
\begin{array}{rl}
unop_{\texttt{i}N} & ::= \texttt{clz} \mid \texttt{ctz} \mid \texttt{popcnt} \\
unop_{\texttt{f}N} & ::= \texttt{neg} \mid \texttt{abs} \mid \texttt{ceil} \\
 & \quad \mid \texttt{floor} \mid \texttt{trunc} \\
 & \quad \mid \texttt{nearest} \mid \texttt{sqrt} \\
binop_{\texttt{i}N} & ::= \texttt{add} \mid \texttt{sub} \mid \texttt{mul} \\
 & \quad \mid \texttt{div}_{sx} \mid \texttt{rem}_{sx} \mid \texttt{and} \\
 & \quad \mid \texttt{or} \mid \texttt{xor} \mid \texttt{shl} \\
 & \quad \mid \texttt{shr}_{sx} \mid \texttt{rotl} \mid \texttt{rotr} \\
binop_{\texttt{f}N} & ::= \texttt{add} \mid \texttt{sub} \mid \texttt{mul} \mid \texttt{div} \\
 & \quad \mid \texttt{min} \mid \texttt{max} \mid \texttt{copysign} \\
testop_{\texttt{i}N} & ::= \texttt{eqz} \\
relop_{\texttt{i}N} & ::= \texttt{eq} \mid \texttt{ne} \mid \texttt{lt}_{sx} \\
 & \quad \mid \texttt{gt}_{sx} \mid \texttt{le}_{sx} \mid \texttt{ge}_{sx} \\
relop_{\texttt{f}N} & ::= \texttt{eq} \mid \texttt{ne} \mid \texttt{lt} \\
 & \quad \mid \texttt{gt} \mid \texttt{le} \mid \texttt{ge} \\
cvtop & ::= \texttt{convert} \mid \texttt{reinterpret} \\
sx & ::= \texttt{s} \mid \texttt{u}
\end{array}
$$

$$
\begin{array}{lrl}
\text{(instructions)} & e & ::= \texttt{unreachable} \mid \texttt{nop} \mid \texttt{drop} \mid \texttt{select} \mid \texttt{block} \; tf \; e^* \; \texttt{end} \mid \texttt{loop} \; tf \; e^* \; \texttt{end} \\
 & & \mid \; \texttt{if} \; tf \; e^* \; \texttt{else} \; e^* \; \texttt{end} \mid \texttt{br} \; i \mid \texttt{br_if} \; i \mid \texttt{br_table} \; i^+ \mid \texttt{return} \\
 & & \mid \; \texttt{call} \; i \mid \texttt{call_indirect} \; tf \mid \texttt{get_local} \; i \mid \texttt{set_local} \; i \mid \texttt{tee_local} \; i \\
 & & \mid \; \texttt{get_global} \; i \mid \texttt{set_global} \; i \mid t.\texttt{load} \; (tp_sx)^? \; a \; o \mid t.\texttt{store} \; tp^? \; a \; o \\
 & & \mid \; \texttt{current_memory} \mid \texttt{grow_memory} \mid t.\texttt{const} \; c \\
 & & \mid \; t.unop_t \mid t.binop_t \mid t.testop_t \mid t.relop_t \mid t.cvtop_t \; t_sx^? \\
\text{(context)} & C & ::= \{ \; \texttt{func} \; tf^*, \; \texttt{table} \; n^?, \; \texttt{memory} \; n^?, \; \texttt{global} \; tg^*, \\
 & & \qquad \texttt{local} \; t^*, \; \texttt{label} \; (t^*)^*, \; \texttt{return} \; (t^*)^? \; \}
\end{array}
$$

Fig. 2: Abstract syntax of modules and contexts

popping and pushing the stack. In the current version of Wasm, instructions can push at most one value to the stack. *Numeric instructions* perform basic operations over numeric values of a specific type, e.g., `i32.add` in Listing 1. *Parametric instructions* operate on operands of any type, e.g., the `select` instruction selects one of its first two operands based on whether its third operand is zero or not. *Variable instructions* get or set the values of local and global variables, e.g., `get_local` in Listing 1. *Memory instructions* query or mutate the memory, e.g., the `memory.grow` instruction extends the size of a module's memory. *Control instructions* affect the flow of control, e.g., the `return` instruction breaks from the current instruction block and returns the current value from the top of the stack.

Module validation A Wasm module is executed in a web browser after validation and instantiation. While validation ensures internal consistency and memory safety of a module, instantiation ensures that the imports and exports are correctly formulated. The validator is phrased as a syntax-directed type system [7] over the abstract syntax of a module. Typing is relative to a context C holding information about the surrounding functions, tables, memories, globals, locals, labels, and return type for a given program point. Fig. 2 recalls the abstract syntax of Wasm modules and contexts, utilizing *extended BNF* grammars for succinctness. We furthermore let t range over value type, tf range over function types, tg range over global types, and n range over numbers.

$$(\textsc{Empty})\ \frac{}{C \vdash \epsilon : [t^*] \to [t^*]} \qquad \boxed{C \vdash e^* : [t^*] \to [t^*]}$$

$$(\textsc{Non-empty})\ \frac{C \vdash e^* : [t_1^*] \to [t_0^*\ t^*] \quad C \vdash e : [t^*] \to [t_3^*]}{C \vdash e^*\ e : [t_1^*] \to [t_0^*\ t_3^*]}$$

$$(\textsc{Cong})\ \frac{C \vdash e^* : [t_1^*] \to [t_2^*]}{C \vdash e^* : [t^*\ t_1^*] \to [t^*\ t_2^*]}$$

$$\boxed{C \vdash e : [t^*] \to [t^*]}$$

$$(\textsc{Const})\ \frac{}{C \vdash t.\texttt{const}\ c : \epsilon \to [t]} \qquad (\textsc{BinOp})\ \frac{}{C \vdash t.\mathit{binop} : [t\ t] \to [t]}$$

$$(\textsc{Drop})\ \frac{}{C \vdash \texttt{drop}\ : [t] \to \epsilon} \qquad (\textsc{Call})\ \frac{C_{\text{func}}(i) = \mathit{tf}}{C \vdash \texttt{call}\ i : \mathit{tf}}$$

$$(\textsc{CallIndir})\ \frac{\mathit{tf} = [t_1^*] \to [t_2^*] \qquad C_{\text{table}} = n}{C \vdash \texttt{call_indirect}\ \mathit{tf} : [t_1^*\ \text{i32}] \to [t_2^*]}$$

$$(\textsc{Loop})\ \frac{\mathit{tf} = [t_1^n] \to [t_2^m] \qquad C, \text{label}(t_1^n) \vdash e^* : \mathit{tf}}{C \vdash \texttt{loop}\ \mathit{tf}\ e^*\ \text{end} : \mathit{tf}} \qquad (\textsc{Br})\ \frac{C_{\text{label}}(i) = [t^*]}{C \vdash \texttt{br}\ i : [t_1^*\ t^*] \to [t_2^*]}$$

Fig. 3: Typing rules for instructions and instruction sequences

Fig. 3 displays a selection of the typing rules. The two judgements are of the form $C \vdash e : [t^*] \to [t^*]$, where e is a single instruction (or an instruction sequence e^*) and $[t^*]$ is a stacktype. The stacktype expresses e's requirement to elements on the stack prior to its execution (a precondition) and the elements on the stack as a result of e (a postcondition). Arrow types $\mathit{tf} = [t^*] \to [t^*]$ double as *function types*, as functions receive parameters and leave their results on the stack.

The rule (EMPTY) says that an empty instruction sequence is valid in any context C and that any value types t^* on the stack will remain unchanged. The rule (NON-EMPTY) for a non-empty instruction sequence e^*e ensures that (parts of) the resulting stack from executing e^* agrees with the stack input expected by e. The congruence rule (CONG) allows one to disregard untouched elements on the stack and thereby apply the instructions rule in an arbitrary context.

The rule (CONST) says that a `const` instruction requires no input from the stack and leaves type t on top of the stack. Similarly the rule (BINOP) for a binary operation requires two elements with type t on top of the stack, and leaves a single element with type t. The rule (DROP) says that a `drop` instruction is valid in any context C with a one-element stack and results in an empty stack ϵ. The rule (CALL) for a function `call` instruction expects the function to have some index i and function type $\mathit{tf} = [t_1^*] \to [t_2^*]$ and requires the parameters to be present on the stack at entry and leaves the (optional) result type on the stack. The rule (CALLINDIR) for a `call_indirect` instruction additionally ensures that a function table is present and that the function's index is available as an `i32` on top on the stack. The rule (LOOP) checks a `loop` instruction's body recur-

sively in a context that records the loop head's label and expected input type. Finally the rule (BR) for a branch instruction `br` i checks agreement between the stack's input types and the expected stack types $[t^*]$ at the target label i.

2.2 Property-based testing

Property-based testing (also known as QuickCheck) is a randomized testing approach introduced by Claessen and Hughes [8]. Originally QuickCheck was phrased as a Haskell library, but the approach has since been ported to over 30 other programming languages. In this paper we will use the QCheck property-based testing library [9] for OCaml. In property-based testing, a test is described by a *generator* and a *property*. The generator delivers randomized test input whereas the property expresses a test specification for each such generated input.[2] As an example, consider the following QCheck test:

```
open QCheck
let t = Test.make (pair pos_int pos_int)
          (fun (a, b) -> a + b >= 0)
```

Here the generator produces pairs of positive integers (including zero). It is phrased by composing QCheck's built-in `pos_int` and `pair` generator combinators. For each such pair (a, b) we wish to test the property $a + b \geq 0$. The generator and the property are passed as arguments to `Test.make` and the resulting test is bound to the name `t`.

We can now provide QCheck with a (singleton) list of tests to run:

```
QCheck_runner.run_tests ~verbose:true [t]
```

This runs a loop for 100 iterations (a configurable number) checking that each generated pair satisfies the specified property. The framework reports a *counterexample* if it finds one, i.e., a generated test input that fails to satisfy the specified property. QCheck quickly finds a counterexample for our example property:

```
generated error fail pass / total     time test name
[✗]    4    0    1    3 /  100     0.0s anon_test_1

--- Failure --------------------------------------------------

Test anon_test_1 failed (22 shrink steps):

(829922565348744309, 3781763453078643595)
```

In this case, the 4th generated pair failed the property. We confirm that the reported pair represents a counterexample, due to integer overflow:

```
# 829922565348744309 + 3781763453078643595;;
- : int = -4611686018427387904
```

Note how this sum coincides with OCaml's `min_int`, the least representable integer within OCaml's 63-bit integer type. In general, a counterexample triggers a second

[2] Other generation approaches exist, e.g., SmallCheck's enumeration up to some bound [24].

shrinking loop, that repeatedly tries to reduce the test input and checks whether the reduced input still fails the property. In the above case, using QCheck's built-in shrinkers for integers and pairs it took 22 shrink steps to reduce the counterexample.

To test more complex systems, custom generators and shrinkers can be developed. Such generators and shrinkers can be used for testing multiple different properties. Since its inception, property-based testing has found bugs missed by hand-written tests across a range of domains, such as telecom protocols [2], data structures [1, 18], election software [15], automotive software [14], and compilers [19, 20].

3 Generating WebAssembly

Generating Wasm programs from a more high-level language, such as C, is a viable solution. In the process of mechanising and verifying the WebAssembly specification, Watt [25] opted for this approach to verify his model. Although this approach produces valid Wasm programs that pass the type-checker, it is not an ideal solution, since the produced programs are confined to the subset of Wasm utilized by the compiler.

To achieve the highest possible coverage of the Wasm language, we instead chose to directly generate Wasm text format programs that can be translated to the binary format and executed in a browser. To ensure this, the generated programs must both be syntactically correct to pass the parser, as well as type-correct to pass the validator. Structuring a generator according to the productions of the grammar ensures the former. To ensure the generated program also passes validation, the generation should follow the typing rules of the language. Pałka et al. [20] suggested to structure such a *type-directed program generator* according to the typing rules. For a functional language with roots in a typed λ-calculus, this means that a typing relation of the form $\Gamma \vdash e : \tau$, is interpreted as a generation procedure with two inputs: the surrounding type environment Γ and the goal type τ. In this way, the generator proceeds to build a term recursively, in each step randomly choosing among the typing rules able to satisfy (unifying with) the goal type.

With Wasm's typing rules tracking value types on the stack, we propose to phrase a *stack-directed program generator*. In the rest of this paper we show that such a generator is both viable and useful, as it has helped locate subtle bugs in major Wasm engines.

3.1 A stack-directed generator

Our generator can generate modules with an arbitrary number of globals and functions and with an optional memory and an optional table, both of arbitrary size. In order for the context to have the right entries in scope, this mandates a certain structure for the generator. Overall our module generator is structured in the following order:

- generate context with an optional memory and an optional table
- generate global types and constant initializers, function signatures, and optional data segments for the memory
- generate optional element segments for the table
- generate function bodies

```
(** instrs_rule : context_ -> value_type list -> int -> (instr list) option Gen.t **)
let rec instrs_rule context output_ts size =
  let recgen con t_opt tr = Gen.(instr_rule con t_opt (size/4) >>= function
    | None              -> return None
    | Some (con', instr', ts')  ->
      instrs_rule con' (ts'@tr) (3*size/4) >>= (function
          | None        -> return None
          | Some instrs -> return (Some (instr'::instrs)))) in
  match output_ts with
  | [] ->
    let empty_gen = recgen context None [] in
    Gen.(oneof [ empty_gen; return (Some []) ])
  | t1::trst  ->
    let empty_gen = recgen context None output_ts in
    let non_empty_gen = recgen context (Some t1) trst in
    Gen.frequency [ 1, empty_gen; 4, non_empty_gen; ]
```

Listing 3: The implementation of `instrs_rule`

By generating the function signatures before the function bodies, we can add them to the context, thus enabling us to generate both recursive and mutually recursive functions. With this order the globals and the optional memory and table are similarly in scope for function bodies. Our generator of function bodies follows the typing rule specification:

$$\frac{tf = [t_1^*] \to [t_2^*] \qquad C, \texttt{local}\ t_1^*, \texttt{label}\ (t_2^*), \texttt{return}\ (t_2^*) \vdash e^* : [] \to [t_2^*]}{C \vdash ex^*\ \texttt{func}\ tf\ e^*}$$

Upon entry to a function, the stack is empty and the actual parameters are available as locals. To generate a body we extend the context accordingly and seek to generate a body with the desired result type $[t_2^*]$. This way we generate Wasm programs backwards in a goal-directed manner.

Our instruction generator performs a back-tracking randomized search. We use option types to distinguish a successful generation attempt from a failed one. The algorithm for generating instructions is phrased as two mutually recursive function `instrs_rule` and `instr_rule` for generating instruction sequences and single instructions, respectively, thereby reflecting the two forms of typing judgments in Figure 3. The two search functions are both parameterized by the context (modeling C) and a *"gas parameter"* to bound the search depth. In addition `instrs_rule` expects a goal stack type matching the resulting stack type in the corresponding typing judgments. Similarly `instr_rule` expects an optional goal type matching the potentially absent type result in the corresponding typing judgments.

Listing 3 contains the implementation of `instrs_rule` which heavily utilizes the monadic interface (`return`, `>>=`) of QCheck generators. It depends on a local function `recgen` that generates the last instruction `instr'` and an instruction list preceding it and then gluing them together. We dedicate $\frac{3}{4}$ of the gas parameter `size` to generating the instruction list, thinking that more gas should be dedicated to generating a sequence than an individual instruction. The `instr_rule` generator performs a weighted shuffle of the compatible instruction rules and then tries them one at a time in the resulting order. When no rules are left to try it returns `None` to signal failure and backtrack.

Our generator produces a single module with three hard-coded export and import functions. The three exported functions return an `i32`, an `f32`, and an `f64` for the surrounding engine to invoke. We omit `i64` as a surrounding JavaScript engine currently

has no way to represent these precisely. We import three printing functions for printing `i32`, `f32`, and `f64` values to increase the chance of some observable program output. Currently our generator does not produce modules that export or import globals, tables, and memories.

Our implementation builds on the reference interpreter for Wasm [23]. This saved us from reimplementing a representation of Wasm modules. On the other hand, the representation is not custom fit for program generation, e.g., with positional information surrounding all internal AST nodes and functions referenced by list index which complicates shrinking (described in Sec. 4).

4 A stack-directed shrinker

As illustrated in Sec. 2.2, a shrinker is useful to automatically reduce a counterexample to help narrow down a potential bug. This is vital as our generator sometimes produces modules with several large data segments and many functions with long and complex bodies. We have therefore implemented a shrinker. Since the generator was carefully engineered to produce modules that pass validation, our shrinker's reductions should preserve this property. We achieve this by stack-type preserving simplifications.

Our shrinker is composed of a number of overall heuristics which attempt the reductions with most impact first. The surrounding QCheck library (like its Haskell ancestor) wraps this shrinker in a loop that repeatedly applies simplifications while still leading to a false property. This way, the individual shrink heuristics complement each other and work together to reduce a counterexample module. To shrink a given Wasm module, the shrinker attempts the following, in order:

- shrink functions and function types simultaneously
- shrink imports and import types simultaneously
- shrink function bodies
- remove unneeded functions
- remove the `start` function
- reduce the exported functions
- reduce the globals
- reduce the declared types
- remove the table
- shrink the element segment
- remove the memory
- shrink the data segment

Few of these rewrite steps are semantics preserving. Functions are shrunk by first attempting to aggressively remove their body or replace it with a constant 0 of the appropriate return type. If this fails, a more complex instruction list shrinker is invoked. The instruction list shrinker pattern matches on either 1, 2, or 3 consecutive instructions and attempts stack-preserving rewrites for each of them. Below we give examples from each category.

One instruction Removing `nop` instructions is the most simple as it has no effect on the stack. Similarly we can remove `tee_local` and unary operations as they leave the

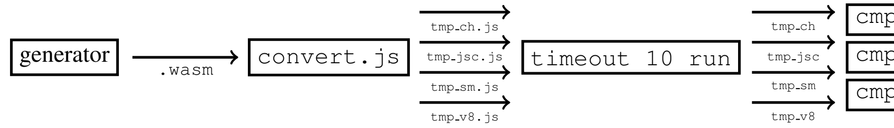

Fig. 4: Experimental setup

same value type on the stack as they consume. Additionally we rewrite global references to a lower index of the same type, e.g., `get_global 321` to `get_global 3` of the same type. Although it hardly represents a reduction in itself, it typically triggers further reductions in the module's list (tail) of global variables. As a final example we rewrite a `call` instruction into a `drop` instruction for each argument, finishing with an optional `const 0` of the appropriate type for non-void functions. Again, locally this may not constitute a reduction. However it may trigger removal of the target function or further reductions involving the `drop` instructions.

Two instructions Motivated by the above we remove consecutive sequences of `const` c `drop`, `get_local` i `drop`, and `get_global` i `drop`. Similarly we remove subsequences of `const` c `br_if` i and of `get_local` i `set_local` j and its variations and combinations involving `globals`. Sequences `const` c testop_{iN} that perform a test on c are replaced with `const 0` and sequences `const` c **`if _`** is_1 **`else`** is_2 is with a two-armed conditional we attempt to rewrite into either is_1@is or is_2@is. Finally, we rewrite two consecutive `unreachable` instructions into a single one. Combined with another heuristic that swaps two instructions if the first is `unreachable`, this has the effect of bubbling `unreachable` instructions last and eliminating duplicates.

Three instructions We rewrite a sub-sequence `const` c `const` c' `compare` into a `const 0` thus removing two instructions. Similarly to the 2 instruction-case, we omit a sequence consisting of `const` c `const` c' `select`. Since `select` expects three value types $[t\ t$ `i32`$]$ on the stack and leaves either the second or the third, the reduction has the effect of leaving a t and thus preserving the types.

The heuristics were inspired by actual counterexample programs. Generally, we found that the shrinker got faster as we added more aggressive heuristics, e.g., removing unused functions saved shrinker time over repeatedly reducing function bodies. We confirmed this observation by rerunning such tests with the same randomization seed with and without the added heuristic. Overall the shrinker fills 535 lines of OCaml code.

5 Testing experiments

We first describe our experimental setup before discussing our findings.

5.1 Experimental setup

We use our generator and accompanying shrinker to test four Wasm engines against the reference interpreter. Concretely we use JSVU [11] to install pre-built command-line versions of Chakra (`ch`) from Edge, JavaScriptCore (`jsc`) from Safari, SpiderMonkey

(`sm`) from Firefox, and V8 (`v8`) from Chrome. This installs nightly builds of each of the four engines. Each JavaScript (JS) engine contains a `WebAssembly` module to test.

The pre-built engines support pure JS and Wasm. As such we cannot run them on a JS-file that requires file-reading or network to load a generated Wasm module. As a workaround we have written a conversion script, `convert.js`, in Node.js which supports file-reading. The script converts a `.wasm`-file into a self-contained JS-file with an embedded `Uint8Array` containing the Wasm module, thus suitable for running in each engine (see Fig. 4). The self-contained JS-file sorts the Wasm-module's exported functions, calls them in sorted order, prints the return value from each, and redirects any output to a temporary file. As we may generate an infinite loop we run each engine with a `timeout` of 10 seconds akin to CSmith [27]. Finally we use the `cmp` command to compare the resulting output files. Overall, our agreement property ensures that the conversions succeed, that the `timeout`s return identical return codes, and that their redirected outputs agree. To further compare the four engines with the reference interpreter, we `fork` a separate process that interprets the module's AST directly, using a `Unix.alarm` to time out. There are more complications however:

Printing across engines Pure JS does not support `console.log`, but 3 out of 4 engines support it. As a further complication we experienced that V8 would buffer output when this was redirected to a file. This would show up as a difference in behavior, e.g., when a generated program `console.log`s one line and then enters an infinite loop: after a 10 second timeout the other three engines would have output, whereas V8 would not. We eventually settled on using `print` which happens to be supported by all four JS engines, despite not being part of the ECMAScript standard.

Host error messages When invoking a generated Wasm module from JS throws an exception, the attached error message varies across the different JS engines. We solved this issue by formulating regular expressions for each engine to catch and normalize engine-specific error messages into comparable ones.

Printing floating point numbers The different JS engines apply different algorithms for printing floating point numbers. For example, one generated Wasm program returned the floating point number `6.980439946950613e+234` to the hosting JS engine. However when invoked as `print(6.980439946950613e+234)` the constant prints as `6.980439946950614e+234` in all 3 engines except Chakra where it prints as `6.980439946950613e+234`. This is a known issue and Chakra's engineers have already adjusted their printer to agree more with the other JS engines.[3] Since we are concerned with testing Wasm engines we did not want such differences to raise any flags. As a first attempt we added logic to only print a certain amount of significant digits, thus checking agreement up to this bound. This left the difficulty of deciding how many significant digits to leave. Eventually we settled on a simpler approach: `(6.980439946950613e+234).toString(2)` instead prints the number in base 2 which agrees across all engines. To compare these outputs with the reference interpreter's output, we then had to implement a compatible base 2 printing for it.

Stack size Our generator has a chance of generating programs that require increasing amounts of stack space and ultimately stack overflow due to excessive (sometimes

[3] `https://github.com/microsoft/ChakraCore/issues/149`

indirect) recursive calls. When such programs have output, the number of written characters may differ across implementations. We solved this problem by comparing only the 5000 first output characters of each JS-embedded implementation. For the reference interpreter with a significantly smaller stack, we compare only its first 300 output characters. Even so, the test setup found a counterexample program where each of the 4 JS engines blew the stack before the 10 second timeout, whereas the reference interpreter did not. For this example, JavaScriptCore would blow the stack after 0.241s, V8 after 6.148s, SpiderMonkey after 0.164s, Chakra after 1.861s, and the OCaml interpreter after 19.664s (all measured with the `time` command), which may indicate either a significantly slower reference interpreter or some tail-call optimization.

Maximum table size The official specification declares the maximum table size to be $2^{32} = 4.294.967.296$ however none of the four JS engines support that value. At first glance all of the four engines allow the maximum table size to be $10.000.000$. Analyzing further, we determined that JavaScriptCore supports a table with a maximum size of $9.999.999$. For a table size set to $10.000.000$ precisely, `jsc` throws the error message `couldn't create Table`. All four engines accepted tables less than $10.000.000$ entries, hence we adjusted the generator accordingly.

Maximum number of parameters During testing, we came across a Wasm module that caused all of the four tested JS engines to err. This happened because the number of function parameters exceeded 1.000. Examining the specification, we did not find any mention of a limitation on the maximum number of function parameters. Subsequently we adjusted the generator to stay below this bound.

Square root non-determinism Our generator found a counterexample calculating the square root of a negative number thus resulting in a `NaN` floating-point value, which would later be reinterpreted as an integer value and eventually printed. Because `NaN`s can carry additional underspecified bits, this also showed up as observable output differences. This constitutes one of the few known sources of Wasm non-determinism [26].

5.2 Testing the generator

The generator is a non-trivial piece of software with a risk of itself containing errors. To reduce these errors and to "take our own medicine" we test the generator using property-based testing. Specifically the generator is engineered to output valid Wasm modules. As there further exists many implementations of the validation algorithm in the reference interpreter and in each of the JS engines, these lend themselves to test the property *each generated Wasm module passes validation*. By testing this property for each of the validation implementations, we effectively test both our own generator as well as each of the validation implementations.

5.3 Testing the shrinker

The shrinker also represents a non-trivial piece of code. To develop and debug it we again property-based tested it. Initially we tested whether the first shrinking candidate would pass validation. This did not find much. Eventually we arrived at a relatively simple property: *for all generated modules m and small natural numbers n, the first n*

shrink candidates of m should all be valid, meaning shrinking should not accidentally turn a valid module into an invalid one while attempting to reduce it in up to n steps.

The refined strategy found multiple bugs as we continued to expand and improve the shrinker: It found problems lifting `If` branches and `Loop` body out which both caused labels to be off. It found another shrinker bug related to reducing functions, types, and imports: These are represented as 3 lists, each containing numbered types and functions. Any reduction in either list therefore means that potentially all function or type indices need to be updated. However the representation has catches we did not anticipate: The imports are present in the type list but not in functions, meaning function indices needed adjusting with ± 3 with 3 hard-coded imports, unless a called function was itself an import.

5.4 Statistics

To ensure that our generator has a reasonable distribution, we have computed statistics across 1000 generated modules. Our statistics covers the number of functions (min: 4, avg: 8.93, max: 14), the total function length (min: 6, avg: 153.18, max: 648), element segment length (min: 0, avg: 1.81, max: 94), number of globals (min: 0, avg: 333.74, max: 9959), data segment length (min: 0, avg: 2.54, max: 87), number of print calls (min 0, avg: 0.96, max 7), as well as percentages of the different instructions. Across the latter, `nop` occurs most often with an average of 12.24% and `callindir` is the most rare occurring with an average of 0.24%. We have added weights to the different instructions in an attempt to even these.

5.5 Bugs found

At the time of writing we have found five bugs of which two were already known. Out of the five bugs three led to a crash of SpiderMonkey and JavaScriptCore. Below we describe the found counterexamples in more detail.

SpiderMonkey crash Our generator found a module which would crash SpiderMonkey with a null pointer de-reference. The hand-shrunk test program is illustrated in Fig. 1.[4] We then created a minimal HTML document encapsulating the test program to investigate how a full Firefox browser would react to it. Upon running the encapsulated counterexample, the released Firefox version crashed the tab as illustrated in Fig. 1. We reported the bug in BugZilla and the error was quickly confirmed and fixed.[5]

Internally, SpiderMonkey's Wasm-engine creates a vector of "exported function" objects, each with a (bit-packed) Boolean, indicating whether a function is marked explicitly as exported. In the test program the same function occurs both as a start function and in a table, causing it to occur twice in the vector, with only one occurrence being marked explicitly. A subsequent removal of duplicates would however eliminate the marked function entry, leaving only an unmarked one. At run-time the JIT-compiler would then expect all explicitly exported functions to have an 'eager stub', which would be `null` in this case and thus cause a crash.

[4] This was found, hand-shrunk, and reported before we developed the automatic shrinker.

[5] `https://bugzilla.mozilla.org/show_bug.cgi?id=1545086`

```
(module
  (type $0 (func))
  (type $1 (func (result f64)))
  (func $0 (type 0))
  (func $1
    (type 1)
    (loop (result f64)
      (f64.const 0.0) (i32.const 0) (br_table 1) (call 0))
    (br 0)
    (unreachable))
  (export "runf64" (func 1)))
```

Listing 4: Shrunk Wasm module causing JavaScriptCore to loop

The bug is particularly interesting, because SpiderMonkey already employs a fuzzer based on libFuzzer to detect such issues. However the above issue had escaped it. We believe this is due to the nature of the bug, being a "logical bug". As such, a coverage-driven fuzzer can visit all branches of the described code to achieve 100% coverage yet still miss the bug. While anecdotal, this represents a real-world bug escaping a state-of-the-art gray-box fuzzer yet being caught by a black-box QuickCheck generator.

JavaScriptCore `br_table` difference Our generator and shrinker automatically found the module in Listing 4 to exhibit different behavior on JavaScriptCore. The other three engines would print `0` when running and printing the result of the exported function, whereas JavaScriptCore would loop. The cause for the difference is the `br_table 1` instruction, which takes a (in this case empty) table of labels and does one of two things: (1) if the value on the stack is a valid index into the table it jumps to that, otherwise (2) it jumps to the provided "fallback" label (`1` above). With label `1` representing the outermost control-context (the surrounding function) this effectively represents a return. JavaScriptCore would instead jump to label `0`, effectively restarting the surrounding `loop`. This was due to a bug in an underlying jump optimizer.

This was reported and quickly acknowledged and fixed.[6] The reported test case was additionally added to the suite of stress tests. Interestingly, multiple Safari users reported this bug as websites using Wasm for font rendering were mis-rendered. While also anecdotal, the example illustrates real-world benefit of our generator: The minimal counterexample enabled developers to quickly identify and fix a real-world problem hitting end users.

JavaScriptCore crash 1 Listing 5 shows another counterexample program we found triggering a segmentation fault in JavaScriptCore. Upon further inspection, this crash was only triggered in the nightly builds and thus the error had not made its way into production. Again we reported the bug along with a sequence of repeatedly smaller counterexamples, also establishing that the error was introduced by a commit between versions 249479 and 250961 of the nightly builds.[7] The error was never confirmed though, and eventually the error was discovered and fixed by other means. We speculate that our ability to file and report bugs has improved since this early bug report.

[6] `https://bugs.webkit.org/show_bug.cgi?id=209333`
[7] `https://bugs.webkit.org/show_bug.cgi?id=202786`

```
(module
  (type $0 (func (result f32)))
  (global $0 i32 (i32.const 1))
  (func $0
    (type 0)
    (f32.const 0.0)
    (f32.const 0.0)
    (i32.const 0)
    (select)
    (loop (result f32)
      (f32.const 0.0) (global.get 0) (br_if 0))
    (drop))
  (export "runf32" (func 0)))
```

Listing 5: Module causing JavaScriptCore to crash

```
(module
  (func (export "run")
    (param i32)
    (unreachable)
    (tee_local 0)
    (drop)))
```

Listing 6: Module erroneously rejected at compile-time by Chakra

JavaScriptCore crash 2 (known) We found another example that would crash JavaScriptCore with the error `FATAL: No color for %ftmp0`, indicating an error in `jsc`'s underlying graph-coloring register allocator. In contrast, the other three engines would all fail with a stack overflow. Again this was reported and acknowledged.[8] This issue was limited to an earlier revision and had since then been resolved.

Chakra compile-time rejection (known) A different mismatch our test setup located involved an `unreachable` and a `tee_local` instruction as illustrated in Listing 6. The module is erroneously rejected at compile-time by Chakra's validator with an error `Can't tee_local unreachable values`, whereas the three other engines throw a run-time error when trying to execute the `unreachable` instruction. Again this was reported[9] but the issue was already known.[10] A fix was merged in Feb. 2019 but still has not made its way into a release.

5.6 Inconsistencies in web-embedding

Imports aside, Wasm programs can only be observed for errors or non-termination. We found three issues related to the web-embedding of Wasm.

Different stack overflow exceptions Our generator found a counterexample program that would blow the call stack by indirectly calling itself. On V8 and JavaScriptCore this would result in an exception instance of `RangeError`, on SpiderMonkey an instance of `InternalError`, and on Chakra an instance of `Error`.

[8] https://bugs.webkit.org/show_bug.cgi?id=209294
[9] https://github.com/microsoft/ChakraCore/issues/6185
[10] https://github.com/microsoft/ChakraCore/pull/5889

Different data segment exceptions Similarly our generator produced an example module with out-of-bounds data segment initializers, which would cause different errors across engines: V8 and SpiderMonkey would throw a `RuntimeError` exception, whereas Chakra and JavaScriptCore would throw a `LinkError` exception.

Different exception name properties JavaScriptCore has inconsistent `name` properties for JavaScript exceptions, which showed up when printing a detailed error for comparison. Consider the following JavaScript program:

```
let e1 = new WebAssembly.CompileError("a␣compile␣error")
let e2 = new WebAssembly.LinkError("a␣link␣error")
let e3 = new WebAssembly.RuntimeError("a␣runtime␣error")
print(e1.name, e1);
print(e2.name, e2);
print(e3.name, e3);
```

On V8, SpiderMonkey, and Chakra this yields:

```
CompileError CompileError: a compile error
LinkError LinkError: a link error
RuntimeError RuntimeError: a runtime error
```

but on JavaScriptCore it yielded:

```
Error Error: a compile error (evaluating 'new [...]')
Error Error: a link error (evaluating 'new [...]')
Error Error: a runtime error (evaluating 'new [...]')
```

The difference was reported but no acknowledgment has been received yet.[11]

5.7 Testing buggy behavior

Chakra's different behavior on an unreachable `tee_local` is relatively often tested, causing our tester to repeatedly rediscover and report it. Despite having its fix merged into the master branch over a year ago, the fix has still not made it into a released version. For this reason, we follow the approach of Hughes in the AUTOSAR project [14] and adjust the test to the documented buggy behavior. We thus consider a Chakra error about unreachable `tee_local` acceptable, despite differing from the other engines.

5.8 A performance experiment

We conducted a small experiment to measure the performance of the generator. The experiment was conducted on a normally loaded MacBook Pro laptop. We invoked the tester 6 times, each generating and comparing the output of 100 Wasm programs. Out of the 6 invocations, 1 exhibited different behavior on the 9th generated program. After 19 shrinking steps and 78.6 sec a counterexample of *'different data segment exceptions'* was reported. For the 5 successful invocations we counted 0–3 timeouts with each invocation taking from 87.8 to 257.1 sec (avg: 158.7). We then reran the experiment with the same randomization seeds, this time excluding a reference interpreter comparison. We observed the same timeouts and the same counterexample, this time taking 79.9 sec. The 5 successful invocations now took from 82.3 to 200.0 sec (avg: 131.7).

[11] `https://bugs.webkit.org/show_bug.cgi?id=204054`

6 Related work

The research literature is rich with contributions within program generation for testing language processors. Purdom [21] originally suggested an algorithm for generating a set of sentences to test parsers and context-free grammars. McKeeman [17] coined the phrase *differential testing* (for software), to characterize his C compiler testing approach. This involved both a *stochastic grammar* associating weights to each production, as well as a test-case reducer repeatedly applying simplifying heuristics.

Our work builds on Palka et al. [20], who tested the GHC Haskell compiler's strictness analyzer by generating random lambda terms. Their generator was structured as a bottom-up reading of the typing rules, thus introducing the idea of using the typing rules as a specification for a generation procedure of well-typed terms. Like our Wasm generator, their generator used backtracking to enable a higher success rate for term generation. Midtgaard et al. [19] also built on Palka et al. [20] in their OCaml program generation approach. To prevent generating programs with evaluation order dependence, they suggest to structure a generator according to a *type and effects system* with dedicated effect indicators. Reading the type and effects system bottom-up, their generator was able to generate evaluation order independent programs and thus find multiple bugs in OCaml compilers. Like us, they also developed a dedicated type-preserving shrinker to shorten counterexample programs.

Alternatives to a randomized recursive generator exist, such as enumeration-based program generation in the style of SmallCheck [22] and Bolzmann samplers to generate typed lambda terms of an approximate size [5]. Both these approaches have currently only been attempted on languages with relatively few language constructs.

Multiple C compilers have been tested by means of randomized testing. Yang et al. introduced CSmith [27], a randomized test-case generator of C program inputs. They used the generator to differentially test each produced program across various C compilers to find differences in their outputs. CSmith generates C programs via a grammar that describes a subset of the C language. It generates a C program with a top-level main function that returns the result of the program via a checksum. The rest of the program is randomly generated. CSmith compares the checksum output across the various compilers. Yang et al. also had to work around the non-deterministic parts of the C language when calculating the checksum. Like our generator, a program from CSmith can loop infinitely and therefore Yang et al. run each program with a timeout.

Barany used differential testing to find missed compiler optimizations in C programs [4]. To do so, he generated random C programs and compared the optimised program code generated by GCC, Clang, and CompCert. For the C program generation he used both CSmith [27] and *ldrgen* [3], a newly developed generator. The ldrgen generator addresses CSmith's tendency to generate dead code by introducing liveness triples in the generation inference rules in addition to the typing context. At each generation step, the liveness of the instruction influences the result. Using this approach, Barany identified multiple missed optimizations in all three tested compilers.

Le at al. [16] introduced *equivalence modulo inputs* (EMI) as an alternative compiler testing approach to differential testing. EMI defines the concept of equivalence of programs on the same input. As a proof of concept they developed Orion to target C compilers. Orion takes a test program as an input. First it extracts coverage information

from the given program, and secondly it then generates EMI variants of the program. Le et al. used the generated EMI variants to test GCC, LLVM and ICC. As a result of their work, Le at al. have found and reported 147 unique bugs in GCC and LLVM.

Donaldson et al. [10] developed GLFuzz, thus using the concept of EMI to test graphics shader compilers in graphics cards. For a given shader input to GLFuzz, GLFuzz repeatedly applies a set of semantics-preserving transformations to the shader. The resulting shader renders a similar image to the original, thereby allowing a comparison between the original and the transformed shader's result. When a significantly different image is rendered, GLFuzz performs reduction (shrinking) to find a minimal set of transformations that lead to a significant difference after rendering. With this approach, Donaldson et al. found defects in all the GPU and driver configurations they tested.

Holler et al. [13] developed LangFuzz, a language-independent program generator. LangFuzz requires a language grammar, sample source code of language implementations, and a test suite. In contrast to CSmith and our own generator which take a generative approach, LangFuzz also utilizes a mutative approach to learn from the provided code samples and produce similar programs. LangFuzz first parses the supplied code samples and builds up code fragments. Afterwards random code fragments are selected and mutated. Finally the mutated program is run against the test suite. As a result of the mutation process, there is a higher chance of finding bugs if the sample source code base contains source code of known bugs. Holler et al. used LangFuzz to generate both JavaScript and PHP programs and found multiple implementation bugs for both.

Watt formalized and verified the Wasm specification within Isabelle [25]. As part of testing his formal model against Wasm engines, he conducted fuzz tests (property-based tests). He used CSmith to generate C programs and then compiled them to Wasm using the Binaryen toolchain [6]. As mentioned, this approach confines tests to the subset of the Wasm language utilized by the Binaryen backend. In contrast, our generator is not limited to such a subset. Consequently we have been able to find errors that span the entire language specification. On the other hand, our generator benefits from both the Wasm specification and Watt's formalization of it to generate valid programs.

7 Conclusion

We have presented a stack-driven generator of WebAssembly programs. For each generated Wasm program we compare the reference interpreter's output against each of the four major browsers WebAssembly engines. In doing so, we have been able to find both major and minor differences, including crashing bugs. To reduce the produced programs, we have developed a stack-driven shrinker. The resulting, minimal counterexample programs allow our bug reports to be short and to the point. With WebAssembly moving beyond client-side web development to new domains such as smart contracts and blockchain, we believe our generator can be a useful tool to ensure agreement across Wasm engines. We have released the source code of the generator under a BSD-license:

`https://github.com/jmid/wasm-prop-tester`

Acknowledgments We are grateful to Andreas Rossberg for suggesting to build a Wasm generator, following a presentation of our OCaml generator [19]. We also thank the APLAS reviewers for their constructive comments.

References

1. Arts, T., Castro, L.M., Hughes, J.: Testing Erlang data types with Quviq Quickcheck. In: Proc. of ERLANG'08. pp. 1–8 (2008)
2. Arts, T., Hughes, J., Johansson, J., Wiger, U.: Testing telecoms software with Quviq QuickCheck. In: Proc. of ERLANG'06 (2006)
3. Barany, G.: Liveness-driven random program generation. In: Proc. of LOPSTR'17. pp. 112–127 (2017)
4. Barany, G.: Finding missed compiler optimizations by differential testing. In: Proc. of CC'18. pp. 82–92 (2018)
5. Bendkowski, M., Grygiel, K., Tarau, P.: Boltzmann samplers for closed simply-typed lambda terms. In: PADL 2017, Proceedings. LNCS, vol. 10137, pp. 120–135 (2017)
6. Binaryen: Compiler infrastructure and toolchain library for WebAssembly (Accessed: 2020-07-02), `https://github.com/WebAssembly/binaryen`
7. Cardelli, L.: Type systems. ACM Comput. Surv. **28**(1), 263–264 (Mar 1996)
8. Claessen, K., Hughes, J.: QuickCheck: A lightweight tool for random testing of Haskell programs. In: Proc. of ICFP'00. pp. 53–64 (2000)
9. Cruanes, S.: QCheck: QuickCheck inspired property-based testing for OCaml (Accessed: 2020-07-02), `https://github.com/c-cube/qcheck`
10. Donaldson, A.F., Evrard, H., Lascu, A., Thomson, P.: Automated testing of graphics shader compilers. PACMPL **1**(OOPSLA), 93:1–93:29 (2017)
11. GoogleChromeLabs: JSVU, JavaScript (engine) version updater (Accessed: 2020-07-04), `https://github.com/GoogleChromeLabs/jsvu`
12. Haas, A., Rossberg, A., Schuff, D.L., Titzer, B.L., Holman, M., Gohman, D., Wagner, L., Zakai, A., Bastien, J.: Bringing the web up to speed with WebAssembly. In: Proc. of PLDI'17. pp. 185–200 (2017)
13. Holler, C., Herzig, K., Zeller, A.: Fuzzing with code fragments. In: Proc. of the 21st USENIX Security Symposium (2012)
14. Hughes, J.: Experiences with QuickCheck: Testing the hard stuff and staying sane. In: A List of Successes That Can Change the World. LNCS, vol. 9600, pp. 169–186 (2016)
15. Koopman, P.W.M., Plasmeijer, R.: Testing with functional reference implementations. In: TFP'10, Revised Selected Papers. LNCS, vol. 6546, pp. 134–149 (2011)
16. Le, V., Afshari, M., Su, Z.: Compiler validation via equivalence modulo inputs. In: Proc. of PLDI'14. pp. 216–226 (2014)
17. McKeeman, W.M.: Differential testing for software. Digital Technical Journal **10**(1), 100–107 (1998)
18. Midtgaard, J.: Quickchecking Patricia trees. In: TFP'17, Revised Selected Papers. LNCS, vol. 10788, pp. 59–78 (2018)
19. Midtgaard, J., Justesen, M.N., Kasting, P., Nielson, F., Nielson, H.R.: Effect-driven quickchecking of compilers. PACMPL **1**(ICFP), 15:1–15:23 (2017)
20. Pałka, M.H., Claessen, K., Russo, A., Hughes, J.: Testing an optimising compiler by generating random lambda terms. In: Proc. of AST'11. pp. 91–97 (2011)
21. Purdom, P.: A sentence generator for testing parsers. BIT **12**(3), 366–375 (1972)
22. Reich, J.S., Naylor, M., Runciman, C.: Lazy generation of canonical test programs. In: IFL'11, Revised Selected Papers. LNCS, vol. 7257, pp. 69–84 (2012)
23. Rossberg, A.: WebAssembly reference interpreter (Accessed: 2020-07-02), `https://github.com/WebAssembly/spec/tree/master/interpreter`
24. Runciman, C., Naylor, M., Lindblad, F.: SmallCheck and Lazy SmallCheck: automatic exhaustive testing for small values. In: Proc. of Haskell'08. pp. 37–48 (2008)

25. Watt, C.: Mechanising and verifying the WebAssembly specification. In: Proc. of CPP'18. pp. 53–65 (2018)
26. WebAssembly: Official website (Accessed: 2020-07-02), https://webassembly.org/
27. Yang, X., Chen, Y., Eide, E., Regehr, J.: Finding and understanding bugs in C compilers. In: Proc. of PLDI'11. pp. 283–294 (2011)