QuickChecking Static Analysis Properties

Jan Midtgaard
Department of Computer Science*
Aarhus University
Email: mail@janmidtgaard.dk

Anders Møller
Department of Computer Science
Aarhus University
Email: amoeller@cs.au.dk

***Abstract*—A static analysis can check programs for potential errors. A natural question that arises is therefore: who checks the checker? Researchers have given this question varying attention, ranging from basic testing techniques, informal monotonicity arguments, thorough pen-and-paper soundness proofs, to verified fixed point checking. In this paper we demonstrate how quickchecking can be useful for testing a range of static analysis properties with limited effort. We show how to check a range of algebraic lattice properties, to help ensure that an implementation follows the formal specification of a lattice. Moreover, we offer a number of generic, type-safe combinators to check transfer functions and operators on lattices, to help ensure that these are, e.g., monotone, strict, or invariant. We substantiate our claims by quickchecking a type analysis for the Lua programming language.**

I. Introduction

Fundamentally, most static analyses boil down to monotone functions operating over lattices [19]. To gain confidence in a static analysis implementation, one would thus hope that the code implements, at least, (1) the formal specification of being a lattice, and (2) functions that are in fact monotone. For example, consider a simple two-element lattice expressed as an OCaml module:

```
module L = struct
  let name = "example␣lattice"
  type elem = Top | Bot
  let leq a b = match a,b with
    | Bot, _ -> true
    | _, Top -> true
    | _, _   -> false
  let join e e' = if e = Bot then e' else Top
  let meet e e' = if e = Bot then Bot else e'
  (* ... *)
  let to_string e = if e = Bot then "Bot" else "Top"
end
```

How can we be sure that the above module implements a lattice? Provided we extend the lattice module with a generator of arbitrary elements, as in

```
let arb_elem = Arbitrary.among [Bot; Top]
```

we present a framework that offers a range of property tests to boost confidence in the implementation (here shown with the user's input in bold font):

```
# let module LTests = GenericTests(L) in run_tests
LTests.suite;;
  check 19 properties...
testing property leq reflexive in example lattice...
  [✓] passed 1000 tests (0 preconditions failed)
testing property leq transitive in example lattice...
  [✓] passed 1000 tests (0 preconditions failed)
testing property leq anti symmetric in example lattice...
  [✓] passed 1000 tests (0 preconditions failed)
... (32 additional lines cut)...
tests run in 0.02s
[✓] Success! (passed 19 tests)
```

Furthermore, to ensure that a static analysis implementation is guaranteed to reach a fixed point, the involved lattice operations should be monotone. For example, consider the following non-monotone operation:

```
let flip e = if e = L.Bot then L.Top else L.Bot
```

In this paper we provide a type-safe, embedded domain-specific language (EDSL) to check and catch such operator errors, by expressing property signatures in a syntax that resembles the standard mathematical syntax, `flip` $: L \xrightarrow{\sqsubseteq} L$:

```
# let flip_desc = ("flip",flip) in
  run (testsig (module L) -<-> (module L) =: flip_desc);;
  testing property 'flip monotone in argument 1'...
  [✗] 270 failures over 1000 (print at most 1):
  (Bot, Top)
```

Pure coverage testing would leave such property-based errors uncaught. In fact one needs only two inputs (top and bottom) to achieve full coverage of `flip`'s implementation. Undoubtedly, this is a simplistic example, but the need to test implementations still stands. Today's static analyses can establish interesting properties about programs in higher-order, dynamically typed languages, for example JavaScript. The intricate semantics of such languages induces complexity in the underlying lattices and in the operations over these. In this paper we demonstrate how *quickchecking* [6] can be used as an effective lightweight methodology to test a range of algebraic properties in static analyses. In situations where lattices and transfer functions are subject to change, for example, in the design phase or in the revision of an analysis, the approach can become a valuable tool. Our approach can also act as a supplement to pen-and-paper proofs or mechanized reasoning within a proof assistant. Towards this goal, this paper makes the following contributions:

- We explore how the ideas in quickchecking (briefly summarized in Section II) can be applied to static analysis.
- We demonstrate how to lift generators of simple lattices to generators of composite lattices and how to use the gener-

*Current affiliation: DTU Compute, Technical University of Denmark

ators to check a number of fundamental lattice properties expressed as a reusable lattice test suite (Section III).
- We formulate a type-safe EDSL of property signatures for testing operations over lattices for a number of desirable properties (Section IV).
- We present a case study of quickchecking a nontrivial static type analysis for Lua, where the tests supplement a basic test suite of hand-written programs to collectively achieve nearly full coverage (Section V).

II. Background

This section provides relevant background information on QuickCheck, Lua, and static type analysis.

A. A QuickCheck summary

Quickchecking [6] is a popular methodology within the functional programming community for performing *property-based testing*. It involves two components: (1) a *generator* for producing arbitrary input, and (2) *properties* that should be tested on the arbitrary input. Two domain-specific languages (DSLs) are used for this purpose, one for each component. The original QuickCheck technique was based on DSLs embedded into Haskell using a Haskell library [6]. Since then, the approach has been ported to numerous other programming languages, both statically and dynamically typed. For the remainder of this paper, we use the qcheck implementation of QuickCheck in OCaml. However, we stress that the approach is not specific to OCaml.

Suppose we wish to test the (incorrect) property from the introduction, that `flip` is a monotone function, hence satisfying the property $\forall$a,b. a $\sqsubseteq$ b $\Rightarrow$ flip a $\sqsubseteq$ flip b. By translating the universally quantified variables a and b into function parameters, this property can be expressed as an OCaml function with Boolean result type:

```
fun (a,b) ->
   Prop.assume (L.leq a b); L.leq (flip a) (flip b)
```

The implication of our specification is modeled using the operator `Prop.assume` of the property DSL, which will test its precondition and (i) continue if it is **true**, or (ii) accept the test and bail early if it is **false** (while keeping track of the number of failed preconditions). This faithfully models logic implication: **false** implies anything. In general, properties concerning a value of type `'a` become predicates of type `'a -> bool`.

To test the above property on arbitrary pairs, we need to generate some input. Recall `Arbitrary.among : 'a list -> 'a Arbitrary.t` from the introduction: it is an example of a combinator from the generator DSL that will supply arbitrary elements selected from its argument list. We can lift this generator to a generator of pairs, using another built-in combinator, `Arbitrary.pair` (from here on we will sometimes abbreviate `qcheck`'s `Arbitrary` module to `Arb`):

```
let arb_pair = Arbitrary.pair L.arb_elem L.arb_elem
```

We are now in position to write a test with `mk_test` and subsequently run it, which exposes the error:

```
# let mon_test =
    mk_test arb_pair
     (fun (a,b) ->
       Prop.assume (L.leq a b); L.leq (flip a) (flip b));;
  val mon_test : QCheck.test = <abstr>
# run mon_test;;
testing property <anon prop>...
  [✗] 27 failures over 100
```

The error message neither names the failed property nor provides a counterexample. To do so, `mk_test` accepts a range of optional, named arguments (prefixed with tilde in OCaml). An example:

```
mk_test ~n:1000 ~pp:pp_pair ~name:"flip␣monotone"
  arb_pair
  (fun (a,b) ->
     Prop.assume (L.leq a b); L.leq (flip a) (flip b))
```

This will instead run the test on 1000 arbitrary pairs, it will identify the particular failed property by the supplied string `"flip␣monotone"`, and it will pretty-print up to ten counterexamples, using a supplied pretty-printer `pp_pair` (defined as a combination of its component's pretty-printers, `let pp_pair = PP.pair L.to_string L.to_string`).

B. The Lua programming language

Lua is a dynamically typed programming language in the ALGOL family of lexically scoped languages. In addition to the usual built-in data types, such as numbers and strings, it features both first-class hash tables and first-class functions. Lua is widely adopted within the computer game industry [15]. Appendix A provides a simplified BNF of the language, leaving out a number of details that are inessential for this presentation. As an example, consider the following, higher-order Lua program:

```
function mktable(f)
  return { x = f("x"), y = f("y") }
end

mktable(function (z) return z.."␣component" end)
```

The program calls a function `mktable`, passing a function as parameter. The function `mktable` will then allocate a table with two entries, `x` and `y`, initialize the entries with the result of invoking the function parameter, and return the resulting table.

C. A static analysis for Lua

To statically predict dynamic type properties of Lua programs, we build a forward, interprocedural static analysis along the lines of TAJS, Jensen et al.'s type analysis of JavaScript [16]. Fundamentally, the analysis consists of a composite lattice to model the state of Lua programs, as well as operations (e.g., transfer functions) over the involved lattices, and a tree-walker to model program execution.

The analysis is centered around an *allocation site abstraction* [5] in which tables and function values are identified by unique labels denoting their origin. Hence, we assume that table literals and functions are uniquely labeled.

Diagrammatically, we illustrate the lattice structure of the type analysis in Fig. 1. Starting from the bottom, we compute

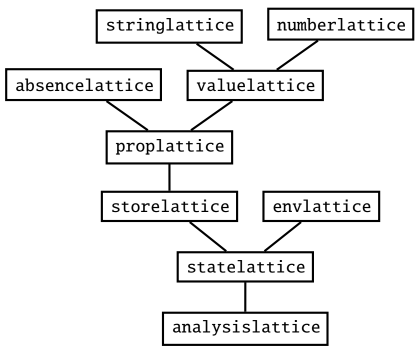

Fig. 1: Lattice structure of type analysis.

invariants over `analysislattice`, which holds an abstract state (`statelattice`) for each program point (before and after each statement) of an input program. An abstract state consists of an abstract store (`storelattice`) and an abstract environment (`envlattice`) that represents scope chains. An abstract store associates to each label ℓ an element from `proplattice`, representing the properties (keys and values) of tables originating from label ℓ. Unlike JavaScript, keys in Lua tables can be any value (even tables), except the special `nil` value. The lattice `proplattice` therefore uses an additional lattice `valuelattice` to over-approximate these. The latter is a Cartesian product of `stringlattice`, `numberlattice` and a few set-based lattices to keep track of allocation sites (of tables and functions) and other value tags (e.g., Booleans and `nil`).

We express each of the above lattices as OCaml modules with a signature satisfying Fig. 2. Each of the components corresponds to an entry in the formal definition of a lattice: $\langle L; \sqsubseteq, \bot, \top, \sqcup, \sqcap \rangle$. We leave an explicit top element optional, as it is not needed in practice for all lattices, for example, `valuelattice`. For lattices with an explicit top element we provide an extended lattice signature. We also include a `to_string` coercion operation in the signature for pretty printing.

When applied to the example program of Section II-B, our analysis will infer that the resulting store after line 5 contains a table allocated in line 2. With the help of `absencelattice`, the lattice `proplattice` reveals that the allocated table definitely contains `x` and `y` entries, and `valuelattice` reveals that both entries can be any string. Because the analysis is monomorphic [19],

```
 module type LATTICE_TOPLESS =
 sig
   type elem
   val leq       : elem -> elem -> bool
   val bot       : elem
(* val top       : elem *)
   val join      : elem -> elem -> elem
   val meet      : elem -> elem -> elem
   val to_string : elem -> string
 end
```

Fig. 2: Lattice signature without explicit top element.

passing two different string arguments to `f` forces the result to top in `stringlattice`.

A static analysis such as the above is typically tested on a range of example programs, to ensure that the analysis soundly accounts for all corner cases of the language. However, a number of underlying properties are seldom given similar attention. In the following sections we will develop the infrastructure for quickchecking such properties.

III. Testing lattices

Formally, a lattice $\langle L; \sqsubseteq, \bot, \top, \sqcup, \sqcap \rangle$ satisfies a number of properties, which should be reflected in an implementation. First, $\langle L; \sqsubseteq \rangle$ is a partial order, meaning the ordering is reflexive ($\forall a \in L.\ a \sqsubseteq a$), transitive ($\forall a,b,c \in L.\ a \sqsubseteq b \ \wedge\ b \sqsubseteq c \Rightarrow a \sqsubseteq c$), and anti-symmetric ($\forall a,b \in L.\ a \sqsubseteq b \ \wedge\ b \sqsubseteq a \Rightarrow a = b$). Second, a lattice has a range of algebraic properties:

$$\forall a \in L.\ \bot \sqsubseteq a \ \wedge\ a \sqsubseteq \top \qquad (\bot/\top \text{ is lower/upper bound})$$

$$\forall a,b \in L.\ a \sqcup b = b \sqcup a \ \wedge\ a \sqcap b = b \sqcap a \qquad (\sqcup,\sqcap \text{ commutative})$$

$$\forall a,b,c \in L.\ (a \sqcup b) \sqcup c = a \sqcup (b \sqcup c) \ \wedge\ (a \sqcap b) \sqcap c = a \sqcap (b \sqcap c) \qquad (\sqcup,\sqcap \text{ associative})$$

$$\forall a \in L.\ a \sqcup a = a \ \wedge\ a \sqcap a = a \qquad (\sqcup,\sqcap \text{ idempotent})$$

$$\forall a,b \in L.\ a \sqcup (a \sqcap b) = a \ \wedge\ a \sqcap (a \sqcup b) = a \qquad (\sqcup\text{-}\sqcap,\sqcap\text{-}\sqcup \text{ absorption})$$

$$\forall a,b \in L.\ a \sqsubseteq b \Leftrightarrow a \sqcup b = b \Leftrightarrow a \sqcap b = a \qquad (\sqsubseteq\text{-}\sqcup\text{-}\sqcap \text{ compatible})$$

In addition, $\bot$ should be the *greatest* lower bound and $\top$ should be the *least* upper bound (in lattices with an explicit $\top$ element).

In order to test such properties, we need (1) a way to compare elements for equality (e.g., to test commutativity), and (2) a scheme for generating arbitrary lattice elements. To this end, we extend the lattice signature of Fig. 2 with two additional operations for testing equality and for generating arbitrary elements:

```
val eq       : elem -> elem -> bool
val arb_elem : elem Arbitrary.t
```

To avoid clutter and to separate our partial analysis specification from the implementation, we keep the quickchecking source code (including generators such as the above) in separate modules, on which the analysis proper does not depend. We can then achieve an "object-oriented sub-classing effect" by means of OCaml's module system: quickchecking code defines extended lattice modules that include the original lattice modules and extend their signature with additional operations, such as, `arb_elem`. In the following sections, we investigate how to formulate this operation.

A. Basic generators

We first consider the simple two-element lattice `absencelattice` that is used to signal whether a table entry is definitely present, in which case a table lookup is bound to succeed. Since there are only two choices of elements, an arbitrary element will necessarily be one of the two. Hence `absencelattice`'s definition of `arb_elem` coincides with that of `L` from the introduction.

For infinitely wide lattices, such as, the flat constant propagation of strings, the situation is more interesting. What do we mean by an arbitrary element? Potentially this could mean several things, for example: (1) a "uniform choice" where each element is equally likely to be chosen, (2) an "algebraic choice" where each datatype constructor (e.g., `Bot | Const of string | Top` in the string lattice) represents a choice reflected in the code and hence should give rise to equally likely choices, or (3) a "concretization choice" where each element is chosen based on weights reflecting how many concrete elements it represents.

For the flat constant propagation lattice, which is infinitely wide, a uniform choice would mean that top and bottom are unlikely to be drawn, even if we restrict the constants to, e.g., all 32-bit integers. Similarly, based on a concretization choice, top is most likely to be chosen as all concrete sets of size 2 or more abstract to it, whereas bottom is unlikely to be chosen. Hence, from a testing point of view, a distribution based on algebraic choice is preferable to uniformly cover all cases in lattice-relevant dispatches. Furthermore, this choice echoes our decision from the simpler two-element lattice.

We express the resulting generator as a choice between three simpler generators: a constant bottom generator built with `Arb.return`, the builtin string generator lifted into the `elem` type, and a constant top generator.

```
let arb_elem = Arb.(choose [return bot;
                            lift const string;
                            return top])
```

For set-based lattices, e.g., sets of allocation site labels, we need to build up a set of arbitrarily chosen elements. For this purpose, we use a fixed point combinator for generating recursive values: `fix : (base:'a Arb.t) -> ('a Arb.t -> 'a Arb.t) -> 'a Arb.t`. As a base case, we provide `LabelSet.empty`, the constructor of an empty set, suitably cast as a constant generator. As the inductive case, we provide an arbitrary-set transformer, by lifting `LabelSet.add` (OCaml's builtin set addition operation) into the generators, using `arb_label` (a generator of arbitrary labels, represented as integers) and `lift2 : ('a -> 'b -> 'c) -> 'a Arb.t -> 'b Arb.t -> 'c Arb.t`:

```
Arb.(fix ~base:(return LabelSet.empty)
         (lift2 LabelSet.add arb_label))
```

Effectively, this generator will generate an empty `LabelSet`, then iterate an arbitrary number of times (the default maximum is 15), using `LabelSet.add` to add an arbitrary label from `arb_label` in each iteration. Next, we lift these basic lattice generators to generators for composite lattices.

B. *Composite generators*

We can easily form generators for product lattices by composing the generators of the sub-lattices. For example, if `A` and `B` are lattice modules extended with generators, we can form a generator for the pair lattice of elements `A.elem * B.elem`:

```
let arb_elem = Arbitrary.pair A.arb_elem B.arb_elem
```

Concretely, we use this approach to form a generator for `statelattice` (represented as two-element records) of the generators for `storelattice` and `envlattice`.

To build arbitrary elements of function lattices, such as `storelattice` and `analysislattice`, we first formulate a helper for building maps. The helper takes three arguments: `mt` for building the empty map, `add` for adding arguments, and finally an association list `kvs` of (key, value) pairs. It proceeds by recursion over the input list and adds each list entry to a constant map generator. Note how the nonempty case utilizes `(>>=) : 'a Arb.t -> ('a -> 'b Arb.t) -> 'b`, the monadic bind of generators to temporarily name the recursively built map:

```
let build_map mt add kvs =
  let rec build ls = match ls with
    | [] ->
      Arb.return mt
    | (k,v)::ls ->
      Arb.(build ls >>= fun tbl ->
                         return (add k v tbl)) in
  build kvs
```

Maps are built based on the input list's element order. If `build_map` is applied to the same association list twice, albeit with the elements permuted, the resulting map (and hence OCaml's underlying balanced tree) will likely result in a differently structured tree, thereby avoiding the pitfall of skewing the generator into generating only a particular subset of map shapes [14]. We can now generate arbitrary maps. For `storelattice`, e.g., we form a generator of arbitrary (`label`, `proplattice`) association lists, and feed the outcome to `build_storemap` (`build_map` parameterized to build `StoreMaps`):

```
let arb_entries =
  Arb.(list ~len:(int 20) (pair arb_label pl_arb_elem))
let build_storemap =
  build_map StoreMap.empty StoreMap.add
let arb_elem = Arb.(arb_entries >>= build_storemap)
```

To summarize, we have seen how to build (and combine) generators for a simple two-element lattice, for a constant propagation lattice, for a set-based lattice, for product lattices, and for function lattices. Collectively, these lattices can be combined to a non-trivial static type analysis such as TAJS [16].

C. *Testing lattice properties*

With element generators in place for all lattices, we are now in position to check the lattice properties we set out to. For example, we can formulate a generic join commutativity test for any lattice `L` that satisfies the `LATTICE_TOPLESS` signature:

```
let join_comm = (* forall a,b. a \/ b = b \/ a *)
  mk_test ~n:1000 ~pp:pp_pair
    ~name:("join␣commutative␣in␣" ^ L.name)
    arb_pair (fun (a,b) -> L.(eq (join a b) (join b a)))
```

where `arb_pair` and `pp_pair` are defined as in Section II-A. We can subsequently test all nine of our example lattices for commutativity of their join operations.

Consider now a conditional property, such as transitivity of the lattice ordering. Again this translates directly to a test:

```
(* forall a,b,c. a <= b /\ b <= c  =>  a <= c *)
let leq_trans =
  mk_test ~n:1000 ~pp:pp_triple
    ~name:("leq␣transitive␣in␣" ^ L.name)
    arb_triple (fun (a,b,c) -> Prop.assume (L.leq a b);
                               Prop.assume (L.leq b c);
                               L.leq a c)
```

Here, arb_triple and pp_triple are generic helper functions for generating and pretty-printing arbitrary triples, analogous to arb_pair and pp_pair. This approach is insufficient for more complex lattices, however, as the probability of generating arbitrary triples that are ordered (and thus satisfying the precondition) decreases with the number of lattice elements. For example, if we run the above test on 1000 arbitrary generated input triples of valuelattice elements we see a problem:

```
testing property leq transitive in value lattice...
  [✓] passed 1000 tests (1000 preconditions failed)
```

Not a single generated triple satisfies the precondition.

Rather than cranking up the number of generated triples to increase the chance of generating a few ordered ones we instead equip lattices with a generator to help generate ordered tuples. To this end, we further extend the lattice signature of Fig. 2 with an operation for generating arbitrary elements less or equal to a given argument:

```
val arb_elem_le : elem -> elem Arbitrary.t
```

The two-element absencelattice is straightforward to extend:

```
let arb_elem_le e =
  if e = Top then arb_elem else Arb.return Bot
```

The extension to the flat stringlattice is not considerably more complex:

```
let arb_elem_le e = match e with
  | Bot     -> Arb.return Bot
  | Const s -> Arb.among [Bot; Const s]
  | Top     -> arb_elem
```

To build an arbitrary subset of a given set, which we need for generating ordered tuples involving set-based lattices, we first formulate a helper function build_set akin to build_map:

```
let rec build_set mt sglton union ls = match ls with
  | []  -> Arb.return mt
  | [l] -> Arb.return (sglton l)
  | _   ->
    Arb.(int (1 + List.length ls) >>= fun i ->
         let ls,rs = split i ls in
         lift2 union (build_set mt sglton union ls)
                     (build_set mt sglton union rs))
```

Similarly to build_map, build_set is parameterized with a builder for the empty set, a builder for singletons, a set union operation, and a list of elements. For input lists of length two or more, build_set will split the input list at some arbitrary point, recurse on both halves, and union their results. We thereby again avoid the pitfall of generating skewed data structures.

Next, we formulate a helper function le_gen to aid with subset selection. The function is parameterized with a list and a builder. It will first permute its input list, split the resulting list in two sublists, and finally pass one of these to the builder:

```
let le_gen es build =
  let es_gen = permute es in
  Arb.(es_gen >>= fun es ->
       int (1 + List.length es) >>= fun i ->
       let smaller_es,_ = split i es in
       build smaller_es)
```

Within valuelattice we use this approach repeatedly to generate subsets of its set-based lattices. For example, given an argument e from valuelattice, we serialize its tags into a list using TagSet.elements, and pass the result to le_gen for subset selection and subsequent building of a set structure:

```
let build_tagset =
  build_set TagSet.empty TagSet.singleton TagSet.union
let le_tag_gen =
  le_gen (TagSet.elements e.tags) build_tagset
```

Assuming the outputs of permute and Arb.int are arbitrary, this approach provides equal chance of each set size. Alternatively, one could consider an approach with equal chance of each subset, by flipping a coin to decide whether each element is included in the subset.

As to the composite lattices, formulating arb_elem_le for product lattices is a straightforward lifting that generates and combines less-or-equal elements for each sub-lattice. For function lattices under pointwise ordering, we first serialize its bindings into an association list. We then reuse the le_gen function from above to choose a subset of bindings. This has the effect of choosing fairly between each subset size of bindings. (Alternatively we could have used a coin toss per binding, similar to above.) We then iterate over the resulting association list using le_entries below, which invokes its argument arb_elem_le on each entry in order to obtain a result that may be pointwise less than its argument. Finally we pass the resulting association list to build_map.

```
let le_entries arb_elem_le kvs =
  let rec build es = match es with
    | [] ->
      Arbitrary.return []
    | (k,v)::es ->
      Arbitrary.(build es >>= fun es' ->
                 arb_elem_le v >>= fun v' ->
                 return ((k,v')::es')) in
  build kvs
```

With arb_elem_le in hand, we can now generate arbitrary ordered pairs and triples for any lattice L to test, e.g., transitivity. For example, we can define ord_pair as follows:

```
let ord_pair =
  Arb.(L.arb_elem >>= fun e ->
       pair (L.arb_elem_le e) (return e))
```

IV. Testing lattice operations

We now turn to operations on lattices. Besides the monotonicity property mentioned in the introduction, what properties are desirable of an analysis operator? Strictness, $f(\bot) = \bot$, is an obvious candidate. Depending on the lattices, this can mean “no output values produced when given no input values” or “output state is unreachable if input state is unreachable”. Since we are

```
(* forall s. bot = s ^ bot *)
let concat_strict_snd =
  mk_test ~n:1000 ~pp:Str.to_string
    ~name:("concat␣strict␣in␣arg.2")
    Str.arb_elem (fun s -> Str.(eq bot (concat s bot)))

(* forall s,s',s''. s' <= s''  =>  (s ^ s') <= (s ^ s'') *)
let concat_monotone_snd =
  mk_test ~n:1000 ~pp:(PP.pair Str.to_string pp_pair)
     ~name:("concat␣monotone␣in␣arg.2")
    (Arbitrary.pair Str.arb_elem ord_pair)
    (fun (s,(s',s'')) -> Prop.assume (Str.leq s' s'');
                         Str.(leq (concat s s') (concat s s'')))

(* forall s,s',s''. s' ~ s''  =>  (s ^ s') ~ (s ^ s'') *)
let concat_invariant_snd =
  mk_test ~n:1000 ~pp:(PP.pair Str.to_string pp_pair)
    ~name:("concat␣invariant␣in␣arg.2")
    (Arbitrary.pair Str.arb_elem Str.equiv_pair)
    (fun (s,(s',s'')) -> Prop.assume (Str.eq s' s'');
                         Str.(eq (concat s s') (concat s s'')))
```

Fig. 3: Examples of specific `stringlattice` operation tests.

only interested in sound analyses, operator strictness is not a formal requirement: returning any over-approximation of $\perp$ is safe, yet an analysis should be as precise as possible.

As a third operator property, we include invariance,[1] $\forall x,x'.\ x = x' \Rightarrow f(x) = f(x')$: operators should yield equal results when applied to equal arguments. Mathematically speaking, this should be obvious, yet it is less so in an implementation, where, e.g., identical strings may be located at different places in memory, or data structures containing the same elements may be differently shaped depending on the insertion order. In the broader context of quickchecking, Holdermans [14] has further argued for supplementing "axiom driven tests" with invariance tests to help catch an otherwise undiscovered class of errors (this issue is discussed more in Section VI). In order to test for invariance, we extend the lattice signature with a generator of equivalent element pairs:

```
val equiv_pair : (elem * elem) Arbitrary.t
```

One must then again go through the lattices to extend them with such a generator. For example, for `stringlattice`, our generator uses OCaml's builtin `String.copy` to create pairs of equivalent, yet differently located strings. For set lattices, `build_set` is already geared to create potentially differently shaped trees for each invocation. For function lattices, extra care must be taken to avoid that duplicate key entries in the initial association list will result in a different function lattice element under a different addition order. We do so by first building one map, then serialize its bindings into an association list, which we can subsequently permute and use to build a second, equivalent map.

Returning to the topic of lattice operations, consider the tests in Fig. 3, which express properties specific to the `stringlattice` (`Str`) operation `concat`: strictness, monotonicity, and invariance. By studying Fig. 3, one can observe a pattern in the code for the three tests, to the point that it is needlessly repetitive. To avoid writing such repetitive lines, we seek to distill a basis of primitives (an EDSL) from which all such tests can be expressed concisely.

[1]This property is also known as 'congruence'.

```
    mname ::= (module NAME)
 baseprop ::= op_monotone
            | op_strict
            | op_invariant
rightprop ::= baseprop
            | pw_right mname (rightprop)
 leftprop ::= rightprop
            | pw_left mname (leftprop)
     prop ::= finalize (leftprop mname mname)
```

(a) Combinator-based notation.

```
   mname ::= (module NAME)
baseprop ::= mname -<-> mname
           | mname -$-> mname
           | mname -~-> mname
    prop ::= (testsig [mname --->]*
                 baseprop [---> mname]* ) for_op
```

(b) Infix notation.

Fig. 4: EBNF grammar of our EDSL.

The EDSL consists of three primitives for expressing properties of unary operations: `op_strict`, `op_monotone`, and `op_invariant`. In addition, we add generic operations, `pw_left` and `pw_right` for adding arguments to the left, resp. right, of the parameter in question. Finally, we add a generic operator for building a test out of the pieces, effectively sealing off the signature (we omit a few optional parameters to `mk_test`):

```
let finalize opsig (opname,op) =
  opsig
    (fun (pp,gen,prop,pname,leftargs) ->
       mk_test ~n:1000 ~pp:pp (* ... *)
         ~name:(Printf.sprintf "'%s␣%s␣in␣argument␣%i'"
            opname pname leftargs)
         gen (prop op))
```

This definition hints to the implementation of our framework: as the combinators traverse the signature description, they will structurally build up a pretty-printer `pp`, a generator `gen`, the property `prop`, the property name `pname`, and a left argument counter `leftargs`. By passing the operator name `opname` as a string, we can thus obtain nice error messages (as illustrated in Section I). Because of the embedding into OCaml, the EDSL will furthermore statically ensure that each module satisfies the lattice signature and that the described signature and the operator's signature agree. In Fig. 4(a) we summarize the syntax of the combinator-based EDSL. For example, we can build a monotonicity test of `Str.concat` in its second argument, based solely on a description of the signature (`=::` is shorthand, infix syntax for `finalize`):

```
# let sc = ("Str.concat",Str.concat) in
  pw_left (module Str) op_monotone (module Str) (module Str) =:: sc;;
- : QCheck.test = <abstr>
```

Such a description may still not be quite satisfactory, as the connection to the corresponding mathematical notation $Str \longrightarrow Str \xrightarrow{\sqsubseteq} Str$ is unclear. To remedy this mismatch we provide convenient infix syntax in the form of arrows with and without annotation: `-$->` for strictness, `-<->` for monotonicity, `-~->` for invariance, and `--->` for a plain function arrow. We can now write `Str.concat`'s signature as follows, where `testsig` and `for_op` act as delimiters of the signature:

```
# (testsig (module Str) ---> (module Str) -<-> (module Str)) for_op;;
- : string * (Str.elem -> Str.elem -> Str.elem) -> QCheck.test
  = <fun>
```

Note how OCaml infers that the EDSL expression further expects a string (describing the operator, e.g., by name) and a `Str.elem -> Str.elem -> Str.elem` operator to yield a QuickCheck test. We thereby statically prevent type errors in our tests, e.g., an attempt to quickcheck a lattice operator over an incorrect signature.

Technically, the type-safe embedding is achieved by left-associativity of function application and of all infix operators in OCaml beginning with the minus '-' character. As a result, OCaml itself will create an underlying AST in the form of Fig. 5. By suitable function definitions for the `testsig`, `for_op`, and the infix arrow operators, one can obtain a depth-first traversal compatible with an AST such as Fig. 5's. The challenge is to combine the combinators in such a traversal: we do not know the arguments to, e.g., `op_monotone` until we meet the `-<->` node (the rightmost `Str` in its left subtree and the right child of the root). Furthermore, the role of additional arguments changes above and below a `-<->` node: below they should be added as left arguments with the combinator `pw_left`, and above they should be added as right arguments with the combinator `pw_right`. We solve these issues by implementing the traversal as a state machine. One register of the state machine keeps track of the latest encountered module argument (initialized with `testsig`'s argument). Two other state machine registers accumulate both a left- and a right argument-adding transformer simultaneously. Upon meeting an `--->` arrow, we register the latest module argument and add the previous to both argument-adding transformers. Upon meeting a 'property arrow', such as the `-<->` node, all previously visited arguments were to the left, so we move the left transformer to the right transformer for any remaining right arguments to be added. Upon completion, a full argument-adding transformer is therefore available to `for_op`.

Statically typing something of this form may seem hard, as the type of a signature varies with its shape. The situation is similar to the static typing of C's `printf`, which will vary with its format string. Inspired by Danvy's solution to the `printf` problem [11], we utilize the polymorphism of result types in continuation-passing style (CPS). In this context, `for_op` will instantiate the polymorphic result type. The EDSL's infix syntax is summarized in Fig. 4(b). With `=:` as an additional infix

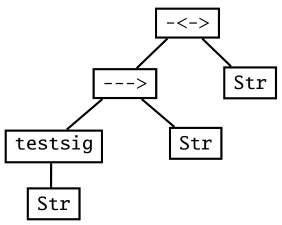

Fig. 5: OCaml's underlying AST.

```
module type ARB_ARG =
sig
  type elem
  val arb_elem  : elem Arbitrary.t
  val to_string : elem -> string
end
```

Fig. 6: Relaxed argument signature.

synonym for `for_op`, we can thus write the equivalent tests of Fig. 3 as compactly as follows:

```
let concat_tests =
 let sc = ("Str.concat",Str.concat) in [
 testsig (module Str) ---> (module Str) -$-> (module Str) =: sc;
 testsig (module Str) ---> (module Str) -<-> (module Str) =: sc;
 testsig (module Str) ---> (module Str) -~-> (module Str) =: sc;
 (* ... and similar for the first argument ... *) ]
```

A. *Checking predicates*

A number of lattice operations naturally involve only lattice arguments. However, many operations have a different signature. One such class is predicate functions with a Boolean result type. For example, `VL.may_be_proc : VL.elem -> bool` takes a `valuelattice` (`VL`) element and checks if the set-lattice that over-approximates (allocation sites of) procedure values is nonempty. In fact, this is still an operator on lattices, as the Booleans form a lattice, whose elements are ordered by implication, and `VL.may_be_proc` is monotone over this lattice. For this purpose, we have implemented a short Boolean lattice module, tested the lattice implementation (using quickchecking, of course), and finally quickchecked `VL.may_be_proc` and a number of similar queries using the Boolean lattice. For other queries, e.g., `VL.is_bot`, which checks whether a given `valuelattice` element is bottom, we use the dual lattice that is ordered by reverse implication.

We cannot expect the arguments of all operations to have been designed as lattices. Some arguments and results nevertheless turn out to be so in retrospect. Rather than force developers to revise their analysis implementations, we also provide OCaml functors to easily build, e.g., list and pair lattices for such quickchecking.

B. *Beyond pure lattice operations*

Our EDSL of infix signature syntax between modules satisfying `LATTICE_TOPLESS` can handle a majority of the operations in our Lua analysis implementation. However, the requirement that arguments must satisfy the `LATTICE_TOPLESS` signature is sometimes too restrictive: some operations simply accept an allocation site label or a `string` (representing a variable name or a property), yet can still be strict, monotone, and/or invariant in the other arguments.

To handle such cases, we relax the parameter requirements to the `pw_left` and `pw_right` combinators, to match the simpler `ARB_ARG` signature in Fig. 6. Based on this relaxation we can then write the last tests in the (still type-safe, but less readable) combinator syntax. We have not found a type-safe approach to allow these in the infix syntax at this point.

An avenue for a different, non-signature based class of tests is that of soundness. To a limited degree this is feasible. For example, we include the following string concatenation test over the `stringlattice` in our test suite.

```
(* forall s,s'. abs(s ^ s') = abs(s) ^ abs(s') *)
let concat_sound =
  mk_test ~n:1000 ~pp:pp_pair ~name:("Str.concat␣sound")
    Arbitrary.(pair string string)
    (fun (s,s') -> Str.(eq (const (s ^ s'))
                           (concat (const s) (const s'))))
```

This, however, will quickly require a reference interpreter in some form as well as the ability to generate arbitrary syntax trees. We leave such an exploration for future work.

V. Experiments and discussion

Our hypothesis is that quickchecking a static analysis increases confidence in its implementation. Indeed, we have found errors in our initial Lua analysis using the lattice property tests and using the lattice operation tests. For later extensions of the analysis, quickchecking became a natural part of the development cycle. As an example, we found an early copy-paste error in the implementation of `meet` of `absencelattice`. This error was caught early because `meet` failed the algebraic tests. We have found several lattice operations that were not strict but could be improved to be so. As such, quickchecking helped make our analysis more precise. More importantly, we found two (unrelated) operations that were not monotone even though they were supposed to be. At closer inspection, the issue turned out to be a lack of relational coordination across lattices: operations would iterate over labels found in the `valuelattice`, yet some labels would not exist as entries in the `storelattice` and hence looking them up would fail, ultimately leading to non-monotonicity. Even though such issues represent corner cases and perhaps never occur in a proper analysis execution, having the code act meaningfully is preferable.

We have generally found that quickchecking is a good reason to (re)consider what properties a lattice operation should have. For example, not all operators should necessarily be strict. One such operation is `PL.add`: since the `proplattice` bottom element means "empty table", we expect an addition of an entry to the empty table to return a nonempty table. In general, the observation of using quickchecking to revisit program specifications agrees with that of the original QuickCheck authors [6].

To further test our hypothesis we devised an experiment to measure coverage of the quickchecked analysis code. To measure coverage we instrumented the source code using the 'Bisect' tool.[2] It works as a preprocessor to OCaml code, by statically annotating program points with labels and dynamically tracking the visited program points. Table I reports the percentages of visited program points. We omit coverage of lattice pretty-printing routines as these are irrelevant to the properties being tested. Column 3 shows that quickchecking achieves reasonable coverage (77–100%). `storelattice` stands a bit out, mainly because of the intricate semantics of Lua's 'metatables': these require other tables (and functions) to be installed at special strings entries of a table, and the corresponding code is thus not easily exercised by a naive generator. By itself, our original test suite consisting of 155 hand-written programs obtains slightly worse coverage (51–91%), as shown in column 2. However, if we combine the two approaches, we achieve full coverage in 5 out of 9 lattices and close to full coverage in general (92–100%). Our interpretation of these numbers is that quickchecking is useful to exercise the esoteric paths in lattice code. The black-box nature of the approach for testing these paths is obviously no silver bullet. Yet, when combined with a standard test suite of programs, the two complement each other well.

Lattice module	Test suite coverage (in %)	QuickCheck coverage (in %)	Combined coverage (in %)
`absencelattice`	72	100	100
`numberlattice`	88	100	100
`stringlattice`	72	100	100
`valuelattice`	78	98	100
`envlattice`	71	92	92
`proplattice`	66	97	98
`storelattice`	91	77	98
`statelattice`	51	93	99
`analysislattice`	81	95	100

TABLE I: Coverage of analysis code (excluding tool front-end). The second column lists coverage of our original test suite. The third column lists coverage of the QuickCheck-based tests.

Our implementation of the testing framework consists of a compact 378 line OCaml module, `LCheck`. This contains a functor with 19 lattice property tests as well as the EDSL code and a number of lattices (the Boolean lattice and its dual, a list lattice, etc.). Since this module can be reused across many static analyses it is separately available for download.[3] Applying the EDSL to the Lua type analysis takes an additional 1100 lines of code. This code then checks 810 properties, distributed between 271 lattice properties and 539 properties of their associated operations. If we include the reusable EDSL module code, this makes for approximately two lines per checked property.[4] We have not attempted to quickcheck the analysis's tree-walker over arbitrary syntax trees at this point.

We would like to extend the infix syntax to handle multiple properties in *one* test signature. Overall this should lower the lines-of-code/property ratio. Technically, this would require changing the underlying CPS building into one returning a list answer type. We would also like to handle more signature properties, e.g., for extensive functions. In addition, one could consider to revise the current infix syntax using the camlp4 preprocessor, to avoid having to write **`module`** repeatedly and to allow arrow syntax that coincides with OCaml's builtin type signature syntax. For now, we have chosen to keep within the bounds of pure OCaml, to limit the number of dependencies.

We stress that although the current development has taken place within OCaml, it could just as well have been formulated, e.g., with Haskell's type classes. The type-safe embedding of

[2]http://bisect.x9c.fr/

[3]https://github.com/jmid/lcheck

[4]The coverage reports, the source code of the analysis, and the tests are available at https://github.com/jmid/luata-quickcheck.

the EDSL utilizes the Hindley-Milner based type system of OCaml, thereby statically preventing type errors in the tests, e.g., after lattice or transfer function revisions. A less type-safe embedding could be implemented, e.g., in Java. One popular application of quickchecking is to test programs written in the dynamically typed programming language Erlang. Indeed, nothing of the present framework mandates a statically typed programming language: the extension of lattice interfaces and an EDSL of signatures could just as well be written in JavaScript, Lua, or Scheme.

VI. Related work

Previous work on increasing confidence in static analyses, range from basic testing to rigorous pen-and-paper proofs such as Astrée's [8]. Our approach can be beneficial to analyses from both ends of the spectrum, to help ensure that an implementation captures the intended meaning — be it in a programmer's mind or in a rigorous pen-and-paper formalization. The growing interest in proof assistants, such as, Coq, led Pichardie et al. [4], [22] to formalize abstract interpretation in constructive logic. Combined with Coq's ability to extract, e.g., OCaml code from its constructive proofs, this minimizes the 'trusted computing base' to Coq itself. In a recent endeavor, Blazy et al. [2] investigated the formalization of a value analysis for C integrated into the CompCert framework. To keep things manageable, this approach relies on a common abstract domain interface akin to a bare bones version of Fig. 2, and it formalizes (and extracts) a *fixed-point verifier* in Coq rather than the fixed-point computing value analysis itself. A rather different approach was taken by Murawski and Yi [17], by developing a static monotonicity analysis formulated as a type-and-effects system. Their analysis conservatively accepts (or rejects) λ-definable functions over finite lattices fed to a static analysis generator. We believe that quickchecking as demonstrated in this paper offers a lightweight alternative to the above approaches. It is type safe, it is reusable, and it supplements basic testing well.

The OCaml implementation of the Lua type analysis benefits from lack of side effects, e.g., assignments. This makes it easier to check and gain confidence in the individual lattice operations as their output is determined solely by the input parameters. The static analysis community has previously benefited from analysis implementations in functional languages: The Astrée static analyzer [8] is implemented in OCaml, the CIL infrastructure for analysis and transformation of C program [18] is implemented in OCaml, Frama-C [10], an industrial-strength static analysis framework for the C programming language, is implemented in OCaml, MathWorks's (formerly, PolySpace Technologies) PolySpace verifier [12] is written in Standard ML, Simon's value analysis of C [23] is implemented in Haskell, and SLAM (subsequently, Static Driver Verifier) was originally implemented in OCaml [1]. There are therefore plenty of existing analysis implementations that might benefit from our methodology. In addition, both Cousot et al. [8] and Jensen et al. [16] report to have arrived at their lattice and transfer function designs after multiple revisions: a potentially fruitful scene for our suggested methodology.

In a follow-up paper to the original, Claessen and Hughes [7] develop a QuickCheck framework for testing monadic code. One of their key insights is to characterize observational equivalence of such code ("in all contexts, one piece of imperative code is indistinguishable from another"), in terms of a little language of traces (for arbitrarily generated context traces, perform an equality test) which one can easily test for. As our analysis computes over approximate states, it has an imperative flavor similar to the examples of Claessen and Hughes [7]. For example, we would like to investigate methods for generating relational lattice values (e.g., `valuelattice` elements whose sets of labels all belong to the generated `storelattice` value). Furthermore, the tree-walker of the Lua type analysis is written monadically, and hence should be a likely target for their techniques. Potentially this could utilize some of the techniques of Pałka et al [21] for generating random abstract syntax trees.

In the context of testing abstract datatypes, Holdermans [14] recalls how naively lifting axioms from an algebraic specification to quickcheck properties leaves programmers with a false sense of security: a buggy implementation can still pass a seemingly complete property-based test suite. In the pitfall he investigates, the randomly generated tests are insufficient to cover all concrete representations of an abstract data type. As a remedy he extends the test suite with invariance tests. We have chosen to follow Holdermans's recommendation in our work.

In the broader programming language community, a number of tools utilize randomized testing. In the field of compiler testing, we briefly touched upon the work of Pałka et al [21] used to test the Glasgow Haskell Compiler's strictness analyser. Another prominent representative is Csmith by Yang et al. [24]: a tool that uses randomized differential testing to generate random C programs free of C's notorious undefined behavior, yet capable of finding numerous errors in production compilers. Cuoq et al. [9] use Csmith specifically for testing static analyzers by modifying the Frama-C static analysis tool, effectively making it an interpreter without abstraction that can be tested against an ordinary C compiler. PLT Redex [13] is a general tool for semantics engineering that can also quickcheck properties on randomly generated input. In contrast to OCaml's statically typed programs and properties, PLT Redex's internal language is dynamically typed. In the presence of (almost) full coverage, one can argue that static typing plays a less dominant role. Static types however provide pedagogical guidelines for composing composite lattice generators out of simpler ones.

Numerous techniques have been suggested for automated testing in related settings. For example, Randoop [20] performs feedback-directed random testing for object-oriented programs, and Korat [3] generates test inputs based on formal specifications of pre- and post-conditions. Such techniques can in principle also be applied to test static analyses, however, unlike our approach, they do not exploit the underlying structured domains and the algebraic properties common in static analysis.

The Lua type analysis is heavily inspired by TAJS [16], a type analysis for JavaScript. In contrast to the present functional OCaml implementation, TAJS is implemented in Java. Despite extensive testing, the manually developed test suite for TAJS does not achieve full coverage of, for example, its value lattice domain, which plays a central role. This is partially due to algorithmic interference: TAJS's worklist algorithm decides when states should be joined in the lattice, which makes it hard to craft an input program that will force the abstract interpreter down a certain lattice code path. Although unit tests have also been made specifically for that part of the code, not enough resources have been invested in making them sufficiently comprehensive. However, quickchecking is well suited to test such paths extensively and with much less effort.

VII. Conclusion

We have presented a lightweight methodology for quickchecking static analyses to check a range of properties, and as a result raise confidence in their implementation. We can do so in a non-intrusive and scalable manner: lattice properties of both basic lattices and complex compositional lattices share the same property tests and can quickly be checked on thousands of generated inputs. To quickcheck lattice operations we have developed a type-safe EDSL for expressing common properties. With our EDSL, much of the infrastructure becomes reusable across analyses, and testing a lattice operation property involves little more than writing a type signature.

Based on this positive experience, we encourage static analysis developers to quickcheck their next analysis tool for many of the generic properties that may otherwise be cumbersome to test. Our OCaml-based EDSL may serve as a useful foundation, or as a source for inspiration if using other languages.

Acknowledgements: This work benefited from Magnus Madsen's comments, from Simon Cruanes's `qcheck` quickchecking library, from Xavier Clerc's Bisect coverage analysis tool, and from support by the Danish Research Council for Technology and Production.

References

[1] T. Ball and S. K. Rajamani. The SLAM project: Debugging system software via static analysis. In *Proc. of the 29th Annual ACM Symposium on Principles of Programming Languages*, pages 1–3, 2002.

[2] S. Blazy, V. Laporte, A. Maroneze, and D. Pichardie. Formal verification of a C value analysis based on abstract interpretation. In *Static Analysis, 19th International Symposium, SAS 2013*, volume 7935 of *LNCS*, pages 324–344, 2013.

[3] C. Boyapati, S. Khurshid, and D. Marinov. Korat: automated testing based on Java predicates. In *Proc. International Symposium on Software Testing and Analysis (ISSTA)*, pages 123–133, 2002.

[4] D. Cachera, T. Jensen, D. Pichardie, and V. Rusu. Extracting a data flow analyser in constructive logic. *Theoretical Computer Science*, 342(1):56–78, 2005.

[5] D. R. Chase, M. Wegman, and K. F. Zadeck. Analysis of pointers and structures. In *Proc. of the ACM SIGPLAN 1990 Conference on Programming Languages Design and Implementation*, pages 296–310, 1990.

[6] K. Claessen and J. Hughes. QuickCheck: A lightweight tool for random testing of Haskell programs. In *Proc. of the Fifth ACM SIGPLAN International Conference on Functional Programming (ICFP'00)*, pages 53–64, 2000.

[7] K. Claessen and J. Hughes. Testing monadic code with QuickCheck. *SIGPLAN Notices*, 37(12):47–59, 2002.

[8] P. Cousot, R. Cousot, J. Feret, L. Mauborgne, A. Miné, D. Monniaux, and X. Rival. The ASTRÉE analyser. In *Programming Languages and Systems, 14th European Symposium on Programming, ESOP 2005*, volume 2618 of *LNCS*, pages 21–30, 2005.

[9] P. Cuoq, B. Monate, A. Pacalet, V. Prevosto, J. Regehr, B. Yakobowski, and X. Yang. Testing static analyzers with randomly generated programs. In *NASA Formal Methods - 4th International Symposium, NFM 2012. Proc.*, volume 7226 of *LNCS*, pages 120–125, 2012.

[10] P. Cuoq, J. Signoles, P. Baudin, R. Bonichon, G. Canet, L. Correnson, B. Monate, V. Prevosto, and A. Puccetti. Experience report: OCaml for an industrial-strength static analysis framework. In *Proc. of the 14th ACM SIGPLAN International Conference on Functional Programming (ICFP'09)*, pages 281–286, 2009.

[11] O. Danvy. Functional unparsing. *Journal of Functional Programming*, 8(6):621–625, 1998.

[12] A. Deutsch. Static verification of dynamic properties, 2003. PolySpace Technologies.

[13] M. Felleisen, R. B. Findler, and M. Flatt. *Semantics Engineering with PLT Redex*. The MIT Press, 2009.

[14] S. Holdermans. Random testing of purely functional abstract datatypes. In *PPDP'13: Proc. of the 15th International Symposium on Principles and Practice of Declarative Programming*, pages 275–284, 2013.

[15] R. Ierusalimschy, L. H. de Figueiredo, and W. Celes. The evolution of Lua. In *Proc. of the Third ACM SIGPLAN Conference on History of Programming Languages*, HOPL III, pages 2–1–2–26, 2007.

[16] S. H. Jensen, A. Møller, and P. Thiemann. Type analysis for JavaScript. In *Static Analysis, 16th International Symposium, SAS 2009*, volume 5673 of *LNCS*, pages 238–255, 2009.

[17] A. S. Murawski and K. Yi. Static monotonicity analysis for λ-definable functions over lattices. In *VMCAI*, volume 2294 of *LNCS*, pages 139–153, 2002.

[18] G. C. Necula, S. McPeak, S. P. Rahul, and W. Weimer. CIL: Intermediate language and tools for analysis and transformation of C programs. In *CC'02: Proc. of the 11th International Conference on Compiler Construction*, volume 2304 of *LNCS*, pages 213–228, 2002.

[19] F. Nielson, H. R. Nielson, and C. Hankin. *Principles of Program Analysis*. Springer, 1999.

[20] C. Pacheco, S. K. Lahiri, M. D. Ernst, and T. Ball. Feedback-directed random test generation. In *29th International Conference on Software Engineering (ICSE)*, pages 75–84, 2007.

[21] M. H. Pałka, K. Claessen, A. Russo, and J. Hughes. Testing an optimising compiler by generating random lambda terms. In *Proc. of the 6th International Workshop on Automation of Software Test, AST 2011*, pages 91–97, 2011.

[22] D. Pichardie. *Interprétation abstraite en logique intuitioniste: extraction d'analyseurs Java certifiés*. PhD thesis, Université de Rennes 1, 2005.

[23] A. Simon. *Value-Range Analysis of C Programs: Towards Proving the Absence of Buffer Overflow Vulnerabilities*. Springer, 2008.

[24] X. Yang, Y. Chen, E. Eide, and J. Regehr. Finding and understanding bugs in C compilers. In *Proc. of the ACM SIGPLAN 2011 Conference on Programming Languages Design and Implementation*, pages 283–294, 2011.

Appendix

$$
\begin{array}{rl}
\mathit{lvalue} ::= & \mathit{id} \\
 & \mid \mathit{exp}\,\texttt{.}\,\mathit{id} \\
 & \mid \mathit{exp}\,\texttt{[}\,\mathit{exp}\,\texttt{]} \\
\mathit{exp} ::= & \mathit{lit} \\
 & \mid \mathit{lvalue} \\
 & \mid \mathit{unop}\ \mathit{exp} \\
 & \mid \mathit{exp}\ \mathit{binop}\ \mathit{exp} \\
 & \mid \mathit{exp}\ \texttt{and}\ \mathit{exp} \\
 & \mid \mathit{exp}\ \texttt{or}\ \mathit{exp} \\
 & \mid \mathit{exp}\,\texttt{(}\mathit{exp}^*\texttt{)} \\
 & \mid \mathit{exp}\,\texttt{.}\,\mathit{id}\,\texttt{(}\mathit{exp}^*\texttt{)} \\
 & \mid \texttt{(}\,\mathit{exp}\,\texttt{)} \\
\mathit{block} ::= & \mathit{stmt}^*
\end{array}
\qquad
\begin{array}{rl}
\mathit{lit} ::= & \texttt{nil} \\
 & \mid \mathit{bool} \\
 & \mid \mathit{string} \\
 & \mid \mathit{number} \\
 & \mid \texttt{\{}\,\mathit{exp}^*\,\texttt{;}\ (\mathit{id}\,\texttt{=}\,\mathit{exp})^*\,\texttt{\}} \\
 & \mid \texttt{function}\,\texttt{(}\mathit{id}^*\texttt{)}\ \mathit{block}\ \texttt{end} \\
\mathit{stmt} ::= & \texttt{break} \\
 & \mid \texttt{if}\ \mathit{exp}\ \texttt{then}\ \mathit{block}\ \texttt{else}\ \mathit{block}\ \texttt{end} \\
 & \mid \texttt{while}\ \mathit{exp}\ \texttt{do}\ \mathit{block}\ \texttt{end} \\
 & \mid \texttt{do}\ \mathit{block}\ \texttt{end} \\
 & \mid \mathit{lvalue}^+\ \texttt{=}\ \mathit{exp}^+ \\
 & \mid \texttt{local}\ \mathit{exp}^+\ (\texttt{=}\ \mathit{exp}^+)? \\
 & \mid \mathit{exp}\,\texttt{(}\mathit{exp}^*\texttt{)} \\
 & \mid \mathit{exp}\ \texttt{.}\ \mathit{id}\,\texttt{(}\mathit{exp}^*\texttt{)} \\
 & \mid \texttt{return}\ \mathit{exp}^*
\end{array}
$$

A simplified BNF grammar of Lua. ?, $*$, and $+$ denote optional elements, empty and nonempty Kleene sequences, respectively.