

QuickChecking Lattice Properties of Abstract Domains

Tom GOALARD *ENS*

Rennes, France Benoit MONTAGU *INRIA*

Rennes, France Pierre LERMUSIAUX *INRIA*

Rennes, France

Abstract—The OCaml static analyzer Salto is developed by the Épicure team (Inria Rennes, IRISA) using an abstract interpretation formalism. In this formalism, OCaml values are represented by abstract values of carefully selected and implemented abstract domains in order to check correctness properties and detect uncaught exceptions. The Epicure team is therefore looking for guarantees on the correctness and precision of these abstract domains. In this paper, we expose a methodology called “property-based testing” to test the properties of the abstract domains and transfer functions of Salto using the OCaml library QuickCheck. It allowed us to find bugs and imprecisions not detected by the unit tests. Those bugs are now fixed, consequently increasing the confidence in the soundness and precision of the Salto static analyzer.

Index Terms—QuickCheck, property-based testing, static analysis, lattices, OCaml

I. INTRODUCTION

THE Épicure team (Inria Rennes, IRISA) develops a static analyzer for OCaml programs (Salto), one of whose aims is to detect uncaught exceptions [3]. This analyzer uses a new abstract domain to represent sets of OCaml values and defines transfer functions that over-approximate operations on these sets of values. Ensuring the correctness of these two elements (abstract domain, transfer functions) is essential to guarantee that the analyzer produces correct results. Consequently, our main goal is to test the properties of those abstract domains and transfer functions to increase confidence in the Salto static analyzer.

The technique used to test the properties of the abstract domains and transfer functions are called “property-based testing”. This is a technique for testing whether properties on certain pure functions are satisfied, by randomly generating inputs to the functions to be tested and confronting those values to mathematical properties that are expected to hold.

This work was done over a period of 9 weeks as part of an internship at the Épicure team. It was supervised by Benoit Montag and Pierre Lermusiaux.

II. BACKGROUND

A. Abstract Interpretation And Lattices

An abstract interpreter is a program that overapproximates the behavior of another program. This interpretation is symbolic and uses abstract values to represent concrete execution values. An abstract value denotes a set of concrete values (or a set of properties describing them) and is used to represent

the possible values of a variable at a given point in the program [4]. An abstract domain is a set of abstract values and the abstract operations that can be performed on them to approximate the concrete operations.

Definition II.1: Partial ordering [2]

A partial ordering is a relation $\sqsubseteq \in \mathcal{P}(L \times L)$ that is reflexive, transitive, and anti-symmetric.

Definition II.2: Partially ordered set [2]

A partially ordered set is a pair $(L, \sqsubseteq)$ where L is a set and $\sqsubseteq$ is a partial ordering on L .

Definition II.3: Complete lattice [2]

A complete lattice $(L, \sqsubseteq, \sqcup, \sqcap, \top, \perp)$ is a partially ordered set such that all subsets have a least upper bound as well as a greatest lower bound. Furthermore, $\perp = \sqcup \emptyset$ is the least element and $\top = \sqcap \emptyset$ is the greatest element.

In addition, we have the widening operator [3] ∇ that is used to ensure the termination of the analysis and make the analysis more efficient. The widening operator is an approximation of the join operator so we can use it instead of join. The interesting part of the widening operator is that if we take a sequence $(s_n)_{\mathbb{N}}$ defined by $s_0 = \perp$ and $s_{n+1} = s_n \nabla a_n$ where $a_1, \dots, a_n \in L$, the sequence is not strictly increasing. The thing is that the analysis of a while loop is a sequence like $(s_n)_{\mathbb{N}}$. So it comes in handy because we have a reliable way to ensure the termination of while loops analysis (or recursion).

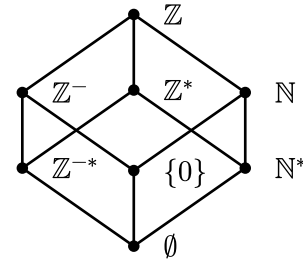


Figure 1: The lattice $(\{\emptyset, \mathbb{Z}, \mathbb{Z}^*, \mathbb{Z}^{-*}, \mathbb{Z}^-, \mathbb{N}, \mathbb{N}^*\}, \subseteq, \cup, \cap, \mathbb{Z}, \emptyset)$

B. Salto Abstract Domains

Most of the Salto abstract domains are classic domains used in static analysis. There is however an abstract domain that

is a new addition to the static analysis state of the art. In the following, we will call this domain the nabla domain [3]. This domain is used to represent an inductively defined set of values of the OCaml language. We will not go into the details of this domain as it is not the main topic here. However, to understand how it works, we can see an example of a nabla value in figure 2. This value represents unary numbers. The constructor S is used to represent the successor of a number. The constructor 0 is used to represent the number 0.

$$\mu\alpha.\{S(\alpha);0\}$$

Figure 2: A simplified nabla value to express unary numbers

The important part that we want to show in Figure 2 is the recursion. It is represented by the $\mu\alpha$. It means that in the following expression, we can replace α by the expression itself: $\mu\alpha.expr$ becomes $expr[\alpha \mapsto \mu\alpha.expr]$. This way allows to unroll the recursion as wished when needed. To represent the number 2 for instance, we start with the expression $\mu\alpha.S(\alpha)$. We unroll it a first time to get $S(\mu\alpha.S(\alpha))$ and then a second time to get $S(S(\mu\alpha.0))$. This simplifies then to $S(S(0))$ because the expression 0 does not contain any instance of α .

The OCaml implementation uses what is called De Bruijn indices [5] to represent the variables in the expression. This way, the variables are represented by a number typifying the number of μ that separates the variable from its definition. The figure 3 shows an example of a nabla value with De Bruijn indices. The arrows show the relation between the variables and their definition.

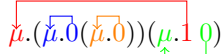


Figure 3: De Bruijn indices example

The De Bruijn indices are used to canonize how the variables are represented in the expression. Changing the name of a variable does not change the expression but changing an index does. If we take the expression in figure 3, $\mu\alpha.(\mu\beta.\beta(\mu\gamma.\gamma))(\mu\beta.\alpha\beta)$ is equivalent to the expression but so $\mu\zeta.(\mu\theta.\theta(\mu\alpha.\alpha))(\mu\theta.\zeta\theta)$ is. The De Bruijn indices are unique and it becomes easier to compare two expressions.

C. Property Based Testing

Formal verification can be used to ensure the soundness of the domain's properties and transfer functions. However, in our case, it is too complex and too time-consuming to be performed for all the domains and transfer functions. To test the lattice properties, we instead used an OCaml library called Quickcheck [1] to perform Property-Based testing with limited efforts. This methodology consists of generating tests rather than writing unit tests by hand. The main part of this methodology is to build a generator to create random values effectively to get interesting cases.

Once the generator is built, we have to define the mathematical property that should hold regardless of the given input. For instance, if we take the reflexivity of $\sqsubseteq$: We state that $\forall x.x \sqsubseteq x$

and for any generated input x , we check if the property holds. If it does not, we have a counterexample showing that the property does not hold, hence a bug in the code.

III. TESTING LATTICES CORRECTNESS

A. Algebraic Properties

To test the different properties of the lattices, we have to define them first. First of all, we must check that the provided $\sqsubseteq$ is indeed a partial order.

$$\forall a \in L.a \sqsubseteq a \quad (\text{Reflexivity})$$

$$\forall a, b, c \in L.a \sqsubseteq b \wedge b \sqsubseteq c \implies a \sqsubseteq c \quad (\text{Transitivity})$$

$$\forall a, b \in L.a \sqsubseteq b \wedge b \sqsubseteq a \implies a = b \quad (\text{Anti-symmetry})$$

If those properties are satisfied we can then check that the lattice is a bounded lattice. The three previous tests need to be done first as they are used in the following properties.

$$\forall a \in L.\perp \sqsubseteq a \quad (\perp \text{ lower bound})$$

$$\forall a \in L.a \sqsubseteq \top \quad (\top \text{ upper bound})$$

Definition III.1: Concretization function

The concretization function $\gamma : \mathbb{A} \mapsto \mathcal{P}(\mathbb{C})$ is a function that maps an abstract value of a domain $\mathbb{A}$ to a set of concrete values in the concrete domain $\mathbb{C}$.

For all lattices in general, we tested the soundness of $\sqsubseteq$, $\sqcup$, $\sqcap$ and ∇ operations.

$$\forall a, b \in L.a \sqsubseteq a \nabla b \wedge b \sqsubseteq a \nabla b \quad (\nabla \text{ upper bound})$$

$$\forall a, b \in L.a \sqsubseteq b \implies \gamma(a) \subseteq \gamma(b) \quad (1)$$

$$\forall a, b \in L.\gamma(a) \cup \gamma(b) \subseteq \gamma(a \sqcup b) \quad (2)$$

$$\forall a, b \in L.\gamma(a) \cap \gamma(b) \subseteq \gamma(a \sqcap b) \quad (3)$$

Equation 1 is used to check that the $\sqsubseteq$ operator is an over-approximation of the $\subseteq$ operator. Equation 2 is used to check that the $\sqcup$ operator is an over-approximation of the $\cup$ operator. Equation 3 is used to check that the $\sqcap$ operator is an over-approximation of the $\cap$ operator. We have to use the concretization functions here because it is a bridge between the abstract and the concrete world [4]. Otherwise, we can not compare the abstract values to the concrete values.

B. Transfer Functions

In the following, the notation $a^\#$ will be used to represent an abstract value.

The Salto analyser uses transfer functions to approximate the effect of concrete operations on the abstract values. The forward operators used to approximate concrete operators are specific to each lattice but the general idea is the same: if an abstract $r^\#$ is the result of an abstract function $f^\#$ applied to one or more abstract elements $a_n^\#$, then the result of the concrete function f applied to the $\gamma(a_n^\#)$ should be a subset of $\gamma(r^\#)$.

$$\forall a_1^\# \in L_1, \dots, a_n^\# \in L_n, r^\# \in L, f^\# : (L_1, \dots, L_n) \rightarrow L.$$

$$f^\#(a_1^\#, \dots, a_n^\#) = r^\# \implies f(\gamma(a_1^\#), \dots, \gamma(a_n^\#)) \subseteq \gamma(r^\#)$$

Another set of functions is the backward operators. They are used to get refined abstract values from an operation given its result and the operands. For those functions, there are two possible results: None if the result cannot be obtained from the inputs ($a * b = 0$ where a and b are not 0) or Some of the refined abstract value. If the result is None, we have to check that the result of the concrete operation is not in the concretization of the abstract result. If the result is Some, we have to check that for any set of concrete values in the operands such that the result of the concrete operation is in the concretization of the abstract result, those values are in the concretization of the refined values.

$$\begin{aligned} &\forall a_1^\# \in L_1, \dots, a_n^\# \in L_n, r^\# \in L, \\ &f^\# : (L_1, \dots, L_n, L) \rightarrow \{None, Some(L_1, \dots, L_n)\} \\ &\text{if } f^\#(a_1^\#, \dots, a_n^\#, r^\#) = Some(a_1'^\#, \dots, a_n'^\#) \text{ then} \\ &\quad \forall c_1 \in \gamma(a_1^\#), \dots, c_n \in \gamma(a_n^\#). \\ &\quad f(c_1, \dots, c_n) \in \gamma(r^\#) \\ &\quad \implies c_1 \in \gamma(a_1'^\#) \wedge \dots \wedge c_n \in \gamma(a_n'^\#) \\ &\text{else} \\ &\quad \forall c_1 \in \gamma(a_1^\#), \dots, c_n \in \gamma(a_n^\#). \\ &\quad f(c_1, \dots, c_n) \not\subseteq \gamma(r^\#) \\ &\text{end} \end{aligned}$$

IV. TESTING LATTICES PRECISION

Some properties are desirable but not necessarily always satisfied. In our case, we added some precision tests because the more precise the operations are, the more precise the analysis results will be. A majority of the previous properties hold if the tested operands always return $\top$. The thing is that $\top$ does not teach us any information so if possible we want to avoid getting it as a result. Another point about the analysis is that it should be semantics-based [2] meaning that we should test the transfer functions of Salto to ensure that the semantic of OCaml is respected. For instance, if we look at the abstract addition, it should approximate the OCaml addition.

A. Algebraic Properties

For the precision properties, we test whether the join is an upper bound and whether the meet is a lower bound. We also check for commutativity, associativity, and idempotence of the $\sqcup$ and $\sqcap$ operators.

$$\begin{aligned} \forall a, b \in L. a \sqsubseteq a \sqcup b & \quad (\sqcup \text{ upper bound}) \\ \forall a, b \in L. a \sqcap b \sqsubseteq a & \quad (\sqcap \text{ lower bound}) \\ \forall a \in L. \perp \nabla a = a & \quad (\perp \text{ neutral element for } \nabla) \\ \forall a, b \in L. a \sqsubseteq b \iff a \sqcup b = b & \quad (\sqcup - \sqsubseteq \text{ compatible}) \\ \forall a, b \in L. a \sqsubseteq b \iff a \sqcap b = a & \quad (\sqcap - \sqsubseteq \text{ compatible}) \end{aligned}$$

The following properties are the same for the $\sqcup$ and $\sqcap$ operators.

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

$$\begin{aligned} \forall a, b, c \in L. a \sqcup (b \sqcup c) &= (a \sqcup b) \sqcup c & (\sqcup \text{ associative}) \\ \forall a \in L. a \sqcup a &= a & (\sqcup \text{ idempotent}) \\ \forall a, b \in L. a \sqcup (a \sqcap b) &= a & (\sqcap \text{ absorption}) \\ \forall a, b \in L. a \sqcap (a \sqcup b) &= a & (\sqcup \text{ absorption}) \end{aligned}$$

Some properties are disabled on some lattices as they are not always true for them. For instance the recursive abstract domain implementation used in Salto [3] does not satisfy $\sqcup$ and $\sqcap$ associativity.

B. Transfer Functions

The backward operators can also be tested for precision: We can check that the refined abstract values are included in the non-refined abstract values if, of course, the result is not None. This is important because as stated before, the result $Some(\top, \dots, \top)$ is a valid result but not an interesting one as we want to have more refined results.

$$\begin{aligned} &\forall a_1^\# \in L_1, \dots, a_n^\# \in L_n, r^\# \in L, \\ &f^\# : (L, L_1, \dots, L_n) \rightarrow \{None, Some(L_1, \dots, L_n)\}. \\ &f^\#(r^\#, a_1^\#, \dots, a_n^\#) = Some(a_1'^\#, \dots, a_n'^\#) \\ &\implies a_1'^\# \sqsubseteq a_1^\# \wedge \dots \wedge a_n'^\# \sqsubseteq a_n^\# \quad (4) \end{aligned}$$

V. TESTING METHODOLOGY

A. General Approach

To test a piece of code, there are mainly two ways to do so: you can use either a white box or a black box approach. The white box approach consists of testing the code by knowing its internal structure. It provides more flexibility and allows us to test parts of the code that are unreachable from the public interface. The black box approach, on the other hand, consists of testing the code without knowing its internal structure. The test behaves like a user and can not know the internal structure of the code. This is the desirable approach as it allows the test to be independent of the code. In our case, the black box testing was a good potential solution. Still, when it comes to more complex data structures, the public interface of the OCaml modules is not expressive enough to generate conveniently random values or to test some parts of the code. Therefore, our approach is more like gray box testing: The code is tested as in a black box approach, but, in some cases, the internals of data structures and some utility functions are exposed to be used in the tests. To do so without modifying the main code too heavily, we used raw functor (Fig 4) that exposes the module internals.

The code in figure 4 shows an example of a module and its raw version. The module **Make** is the module used by the main codebase. The type t is private and the exposed functions are only the ones that are needed to manipulate the type t . The module **MakeRaw** is the raw version of the module. This is the version used by the tests. The type t is exposed and some utility functions are added to help the tests. This approach allows for more flexibility than a black-box approach while allowing to keep the tested code as clean as possible and to keep the tests as independent as possible from the tested code.

```

module Make : sig
  type t
  val foo : t -> t
  val bar : t -> bool
end

module MakeRaw : sig
  type t = someType
  val foo : t -> t
  val bar : t -> bool
  val internal : t -> int
end

```

Figure 4: Example of a module and its raw version

B. QCheck

QCheck is a QuickChecking library for OCaml. It allows to generate random test instances and test them against properties.

```

QCheck2.Test.make
  G.generator
  (fun x -> L.leq x x )

```

Figure 5: Reflexivity property test with QCheck

A test hence involves two main components: a generator and a property.

The generator is a function that generates the desired random values. In figure 5, the generator is the function `G.generator` where `G` is a module used to generate random values for the lattice. The lattice is represented by the module `L`. `L.leq` is the $\sqsubseteq$ operator for the lattice `L`.

The property is represented by a function of type `L.t -> ... -> bool` indicating whether the property is satisfied or not. `L.t` represent the type of abstract values of the lattice.

```

Random seed: 143896216
generated error fail pass/total time test name
[✓] 1000 0 0 1000/1000 0.0s reflexivity
=====
success (ran 4 tests)

```

Figure 6: Reflexivity property test result

If we run the tests with the default configuration, QCheck will run 1000 instances of tests (Fig 6). Assuming that the random generated values are interesting, that will be equivalent to 1000 unit tests while we only wrote one test and a generator that will be reused in other tests.

C. Generating Random Abstract Values

A large part of the testing process is to generate random values. This part is the most important part of the testing process as the quality of the test is directly related to the quality of the generated values. The difficulty of generating interesting values is more discussed in section VII but the main problem is to generate values that are interesting enough to test the properties. For instance, if we generate only trivial values, the tests will not be very useful. On the other hand, the size and complexity of the generated values should not be

too high. A too complex value make the tests too slow (section VI-C) or consume too much memory.

To generate such values, we used the QCheck generators that are monadic. A monad is used to model a computation. It is in our case represented by a function that maps an input to an output. The interest of the monad is that it can have side effects but the side effects happen in a controlled order. Another interesting thing is that the monads can pass to each other a state seamlessly. In our case, the side effect is updating the random generator state. It comes in very handy when you want to generate complex values based on the provided base generators. The provided operators allow us to chain the generators together to create complex values. To do so, there are three operators involved:

- `return` This operator allows to encapsulate of a value in the monad.
- `bind (>=>)` This operator allows to chain two monads together.
- `map (>|=)` This operator allows to apply a function to the value in the monad.

The problem with generating random values is that when you finally find a counter-example, you want to be able to understand it to fix the bug. Unfortunately, as the values are random, they are usually not readable or too big. To fix that, QCheck implements a way to reduce counter-examples to a smaller and more readable form. This reduction is handled by a function called a shrinker. And because the generators are monadic, we do not have to create our shrinker. QCheck will shrink the values for us by using the state given by the monad.

D. Generating Random Concrete Values

To test the soundness of the $\sqcup$ and $\sqcap$ operations, we need to have a way to generate concrete values to test them. We thus had to create two new OCaml functions:

- A concrete values generator, of type `L.t -> L.concrete`, used to generate values in the concretization of a given element
- A membership function, of type `L.t -> L.concrete -> bool`, to check if a concrete value is in the concretization of an abstract value

The reason we test random concrete values and not manipulate concrete sets of values is that sometimes the set might be too big to be tested or infinite. For example, we can look at the intervals domain. The concretization of $[0, +\infty]$ is an infinite set.

Generating concrete values is an addition to what is proposed in [1] as in the paper they did not check the correctness of properties that involve concrete values. In practice, we used QCheck to generate a random list of concrete values in the concretization of the abstract values we wanted to test and then iterate over them. This leads to more complex and time-consuming tests (VI-C) but this also makes our tests more precise: For each of the random values, we test between 100 and 500 concrete values on many of the tests.

VI. RESULTS

The test suite is now populated by 544 new tests allowing to check the properties of the abstract domains and transfer functions of Salto.

A. The Bugs Found

The goal of this work was to find bugs in Salto and we can say that this was a success. The test suite found about twenty bugs in the code. Those bugs were soundness issues as well as precision issues that were not detected by the unit tests. The bugs are now fixed and some counterexamples are now a part of the unit tests. This way, if the bug is reintroduced, the tests will fail and the bug will be detected. The randomness of the QCheck tests causes some bugs not to be found each time the tests are run. The unit tests are therefore still needed to ensure that the bugs found are fixed and not reintroduced.

A memory leak was also found in QCheck that was fixed in the latest version of the library. Although it was not the goal of this work, it is still a good thing to have found this bug as it allows us to improve the quality of QCheck while reducing the memory consumption of the tests. This ultimately benefits us and anyone using QCheck.

B. Code Coverage

An interesting metric to evaluate the quality of the tests is the code coverage. It is the percentage of lines of code that are executed by the tests. To perform this evaluation, we used the Bisect library which is a code coverage tool for OCaml. We have to keep in mind that a high code coverage does not mean that the code is well tested. It is still interesting to see which part of the code is not tested at all, as the random generators may not generate values that reach those parts of the code. Another point to keep in mind is that some functions are not used at all in the tests. For instance, the printer functions are not used. This reduces the code coverage as those functions are not covered. This is not a big deal as they are quite small most of the time and usually represent a small part of the code.

File	QCheck (%)	Unit (%)	Total (%)
nablaRec	81	87	87
arrays	78	30	82
constants	84	97	97
enumChar	73	58	96
flatString	76	55	88
heap	82	69	95
free_intervals	70	61	83
records	48	59	80
lazys	48	63	81

Table I: Code coverage of some abstract domains

In table I, we can see the code coverage of the QCheck tests, the unit tests, and the total code coverage if we run the

two types of tests consequently. The unit tests are the tests that were already in the codebase before this work. They mainly consist of expected output tests on sample code. There are also some unit tests to check the lattices on edge cases.

Note that the total is not entirely related to the coverage of the two methods separately as some parts are tested by the two methods and some are not tested at all. This table shows nonetheless that in general, the two methods complement each other as the code coverage of the two simultaneously is sometimes way higher than the two separately. If we take a look at the records line, we can see that the coverage of each method is not very high but the total coverage is still pretty good.

This allowed us to find some parts of the code that were not tested at all and to add tests for them. For instance, two transfer functions of the integer domains were not tested because the tests were not added to the test suite.

C. Performance

A black spot of our tests is the performance. It heavily depends on the generated values but overall, the tests are globally slow. Not too slow for a Continuous Integration but too slow for a developer to run them all at each modification, because of the number of tests performed each time. Another problem is caused by the nabla domain. The generator creates in some cases values that are quite slow to compute because one of the algorithms is quadratic in the size of the value.

To address this issue, a command line interface is implemented in the tests executable to run only tests for a specific domain or a specific group of tests for a given domain. This way, the developer can run only the tests he wants to run when needed.

Another problem found was memory consumption. This issue is addressed by grouping the tests in smaller groups run one after the other. This way, the memory is freed after each group of tests, and the memory consumption is reduced. This issue was also a problem because of a memory leak in the QCheck library that is now fixed in the latest version (See VI-A). This helped to reduce the memory consumption of the tests.

VII. DIFFICULTIES ENCOUNTERED

Randomly generating values that are interesting enough to test the properties and that respect some constraints is not an easy task for the more complex data structures. While generating random intervals or random sets of data is relatively easy, generating random values for the nabla domain or the constant domain is more complex.

The constant domain is a domain used to represent the constant values in OCaml. It means that the domain is a union of in our case, 31 bits integers, 63 bits integers, chars, and strings. The main concern with the constants is that we need to generate values with different types and manipulate random Generalized Algebraic Data Types (GADT) values. It is an advanced way to work with types in functional programming languages and allows fine grained control over the manipulated data. Defining a generator over GADT is challenging because

sometimes the type checker is not able to infer the type of the generated values or infers a type that is not generic. In the best-case scenario, we have to add type annotations to the generators but in the worst case, we have to rewrite the code to make it work. It was not the most time consuming part of the work. It was nevertheless quite annoying sometimes.

Another difficulty was to generate nabla values of the desired size, that fit some constraints like a small memory footprint. At first glance, it seems easy enough because the nabla domain is parametrized by the previous domains and we can use those generators to generate values. The only remaining thing to do is to generate a tree of values. It is helped by a generator included in QCheck that permits generating values recursively. When it comes to the recursive part of the domain however, the thing becomes harder.

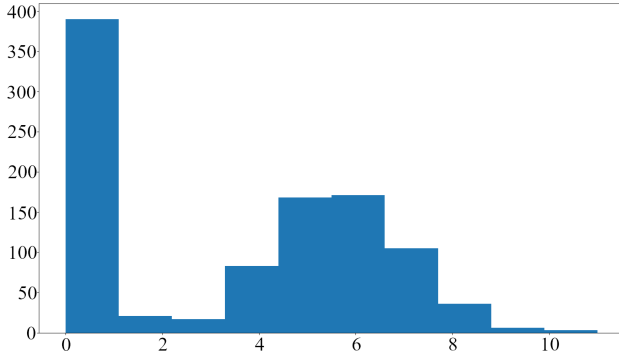


Figure 7: Buggy distribution of the height of the generated nabla for 1000 instances with no $\perp$

The first encountered issue was that the tree was too dense and hence used too much memory for no interesting purpose. Fixing that was just tweaking the generator probability. The second issue was the fact that the generated values were sometimes meaningless and thus simplified by the minimizing algorithm used in the domain. It happened because the values should respect properties of well-formedness [3]. For instance, $\mu\alpha.\alpha$, the value is infinite with no way to ever end the recursion. It is therefore minimized to $\perp$. We can see in Fig 7 that there are two wrong things in the distribution: the first one is the fact that there is more than a third of the value with height 0, e.g., $\perp$ values while in this configuration there should be none. The second one is that the maximum depth is 10 while it should be around 12 to 15.

To address this issue, the generator keeps track of the variables in the tree and adds nonvariable outcomes where needed to ensure that the tree is not infinite. This gives the distribution in Fig 8. In this case, the maximum height is in the expected range and there are no $\perp$ values anymore.

The next issue was to unroll randomly part of the tree. To do so, we have two methods: the first one is to unroll the μ like normal: $\mu\alpha.expr$ becomes $expr[\alpha \mapsto \mu\alpha.expr]$ and the second is to unroll only some part of it by doing $\mu\alpha.expr[\alpha \mapsto expr' \text{ or } \alpha]$ randomly where $expr'$ is $expr$ with updated De Bruijn Indices. Setting up those two methods was straightforward but choosing if the unrolling should be done or not was the hard part. As the unrolling can lead to an explosion of the tree size, we have to be careful when to do it. The

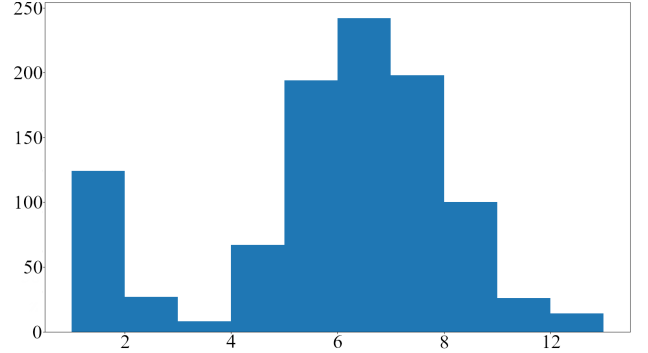


Figure 8: Distribution of the height of the generated nabla for 1000 instances with no $\perp$

method we used was to limit the unroll to μ that is not the root or a child of the root and by limiting the number of unrolls to 4 per tree. This prevents the tree from exploding while still allowing it to unroll some parts of it. There may be some better methods to achieve the same results but this method is not computationally expensive and simple to implement.

The final issue was the shrinking part of the values. The shrinking is done by a tree exploration of the random possibilities. This works fine most of the time but on nabla, the number of possibilities is so high that the shrinking can take a lot of time. To address this issue, the shrinking had to be changed and some parts of the nabla value are now prevented from being shrunk. This way, the shrinking is faster because the shrinking is done mainly on the nabla size and not on the huge number of possibilities added by some parts of the tree.

VIII. CONCLUSION

We have expanded the Salto test suite to include tests for the properties of the lattices and transfer functions of the Salto static analyzer. The property-based testing was shown to be effective for our work case as it allowed us to find bugs in the Salto static analyzer while being easier to perform than formal verification. The tests are mostly achieved in a modular and flexible way that allows to add new tests easily and to change them without too much effort. The lattices are now tested on thousands of cases, greatly increase the code coverage of the analyzer. This effectively increases the confidence in the soundness and precision of the analyzer.

REFERENCES

- [1] J. Midtgaard and A. Møller, “Quickchecking static analysis properties,” *Software Testing, Verification and Reliability*, vol. 27, no. 6, p. e1640, 2017, e1640 stvr.1640. [Online]. Available: <https://onlinelibrary.wiley.com/doi/abs/10.1002/stvr.1640>
- [2] F. Nielson, H. R. Nielson, and C. Hankin, *Principles of program analysis*. Berlin ; New York: Springer, 1999.
- [3] P. Lermusiaux and B. Montagu, “Detection of uncaught exceptions in functional programs by abstract interpretation,” in *Programming Languages and Systems*, S. Weirich, Ed. Cham: Springer Nature Switzerland, 2024, pp. 391–420.
- [4] P. Cousot and R. Cousot, “Static determination of dynamic properties of programs,” in *Proceedings of the Second International Symposium on Programming*. Dunod, Paris, France, 1976.
- [5] Wikipedia contributors, “De bruijn index — Wikipedia, the free encyclopedia,” https://en.wikipedia.org/w/index.php?title=De_Bruijn_index&oldid=1208839602, 2024, [Online; accessed 17-July-2024].