Automated Random Testing of Numerical Constrained Types

Ghiles Ziat ✉

ISAE-SUPAERO, Université de Toulouse, France

Matthieu Dien ✉

Université de Caen, France

Vincent Botbol ✉

Nomadic labs, Paris, France

Abstract

We propose an automated testing framework based on constraint programming techniques. Our framework allows the developer to attach a *numerical* constraint to a type that restricts its set of possible values. We use this constraint as a partial specification of the program, our goal being to derive *property-based* tests on such annotated programs. To achieve this, we rely on the user-provided constraints on the types of a program: for each function f present in the program, that returns a constrained type, we generate a test. The tests consists of generating uniformly pseudo-random inputs and checking whether f's output satisfies the constraint. We are able to automate this process by providing a set of generators for primitive types and generator combinators for composite types. To derive generators for constrained types, we present in this paper a technique that characterizes their inhabitants as the solution set of a numerical CSP. This is done by combining abstract interpretation and constraint solving techniques that allow us to efficiently and uniformly generate solutions of numerical CSP. We validated our approach by implementing it as a syntax extension for the OCaml language.

2012 ACM Subject Classification Software and its engineering → Dynamic analysis

Keywords and phrases Constraint Programming, Automated Random Testing, Abstract Domains, Constrained Types

Digital Object Identifier 10.4230/LIPIcs.CP.2021.59

Supplementary Material *Software (Source Code)*: `https://github.com/ghilesZ/Testify`
archived at `swh:1:dir:e80146ce697a659919067f8125461a6c5c9d553c`

Funding This work has been partially supported by the Defense Innovation Agency (AID) of the French Ministry of Defense under Grant No.: 2018.60.0072.00.470.75.01 and the RIN ALENOR project.

1 Introduction

In this article, we propose an automated testing framework that generates tests for a restricted class of dependent types, that is constrained types. Constrained types attach a membership predicate to a type and are used to restrict its set of possible values. For instance, to encode rationals as a pair of integer (n, d), one could add the constraint $d \neq 0$ to filter invalid representations. Moreover, providing the constraint that n and d are coprime with $d > 0$ defines a canonical representation for this type. This is desirable when a given term has several structurally different but semantically equivalent representations, e.g. $\frac{2}{4}$ would violate the constraint while $\frac{1}{2}$ would be valid. However, type systems with constrained types are generally undecidable making it hard to obtain strong static guarantees at compile-time [2]. Dynamic verification, on the other hand, while not preserving these strong formal guarantees, makes the approach both feasible and practical. One instance of this is property-based testing [9] that can potentially detect bugs in programs given a specification. Still, this

© Ghiles Ziat, Matthieu Dien, and Vincent Botbol;
licensed under Creative Commons License CC-BY 4.0

requires to manually provide tests that may be tedious and error-prone. For these reasons, we focus on the development of an automated test system for constrained types. The goal of our framework is to exhibit wrong behaviours in such programs by finding instances of a constrained type that violate their predicates. We achieve this by automatically generating tests for functions that manipulate constrained types. This requires from the developer to input a constraint from which we extract a partial specification of the program. The resulting generated tests consist in generating random inputs for the tested functions and checking their output against the given specification. Therefore, the main challenge we face is the automatic derivation of uniform value generators for constrained types. We address this by combining two approaches: Constraint Programming [26] and Abstract Interpretation [11]. Abstract interpretation provides modular, efficient and precise abstractions, in particular, for numerical values. Coupling it to standard constraint programming resolution scheme allows us to provide fast and uniform input generators for our automatically generated tests.

Our implementation targets the OCaml [21] programming language, and the examples we show are written in it. However, our approach is generic and could be ported to other languages, hence, this paper voluntarily disregards some specificities of the OCaml language.

1.1 Example: Putting Constraints in Programming

Consider the following example:

```
1 type nat = int [@satisfying (fun x -> x >= 0)]
```

This type declaration `type nat = int` is an alias of the primitive for machine-integers. Adding the annotation `[@satisfying (fun x -> x >= 0)]` specifies the constraint that values x of this type must respect the constraint $x \geq 0$, that is the positive integers. We then generate a test for each function whose return type is `nat`, as in the following example:

```
1 let add (x : nat) (y: nat) : nat = x * y
```

Compiling this program with our preprocessor will generate the following test:

```
1 let add_test () =
2   let x_rnd = range 0 max_int in
3   let y_rnd = range 0 max_int in
4   (multiply x y) >= 0
```

To achieve this, we have solved the constraint $x \geq 0 \wedge x \leq 2^{64}$ and determined the set of its solution. This allowed us to define the generator for the `nat` type. Here, `range` is a primitive of our framework that draws uniformly within an integer range.

Running this test will (usually) yield an error message stating that the return value of `add` violates the predicate `(fun x -> x >= 0)` for some input. Indeed, when a random input close to 2^{64} is passed to the function, an overflow may occur which would cause the return value to violate the constraint.

This example is designed to illustrate our testing framework process. Throughout the paper, we will describe how to derive generators from more complex constraints.

1.2 Contributions

This article focuses on type-driven automated test generation. Our contribution is threefold:

- Firstly, we propose an automated testing framework capable to derive a partial specification of a program using a set of constraints given by the developer. We then run tests against this specification.
- Secondly, we present an abstract domain based solving technique able to characterize constrained types in a way that enables the automatic derivation of uniform random generators, i.e. each instance has the same probability to be drawn.

- Finally, we give abstract domains (i.e. boxes, polyhedra) and their associated operators that permit generic random uniform generation.

1.3 Outline

This paper is organized as follows: Section 2 presents the related works. Section 3 introduces the derivation of generators and specification for constrained types. Section 4 recalls definitions of our abstract domains usage in constraint solving, and explains the relations between Numerical Constraint Satisfaction Problems (NCSP) and constrained types. Section 5 discusses the problem of uniform random sampling within heterogeneous abstract values, in particular for polyhedra and, related, the cardinality estimation problem with such domains. Section 6 presents our implementation, its current limitations and gives some experimental results. Finally, Section 7 summarizes our work and discusses its future continuations.

2 Related Works

Random testing has been thoroughly studied, for testing correctness [16, 28], exhaustiveness [33], complexity [5] etc. Several frameworks exist and relate more or less to our work. For instance, *American Fuzzy Lop* [37] inspects the execution paths of the program and apply mutations to each input to potentially discover new ones. This method increases the coverage of the test but requires the binaries to be instrumented. In the *OCaml* ecosystem, the *ppx-inline-tests* preprocessor makes it possible to inline tests in the source code. Also, several testing libraries for *OCaml* programs exist, such as monolith [31], or *QCheck* [13], inspired from Haskell's *QuickCheck* library [9]. These property-based testing frameworks are widely used [24, 23]. In particular, we re-use from *QCheck* some basic value generators for our automatic test generation. However, these works require the developer to manually write the tests, that is the generators and the properties to be checked. Our approach makes it possible to define constrained types that will act as a partial specification of the program and automatically extract the needed generators. Our framework automatically generates tests and is, thus, less error-prone and is easily maintainable: a constrained type annotation is sufficient to generate tests for all functions that return values of this type. This way, the test suite is updated automatically each time a new function is added. Such as *ppx_inline_tests*, our framework is implemented as a preprocessing mechanism that has no side-effects on the execution of the original program.

Constrained types have been widely studied, for example, Dependant ML [36] enriches the type system of ML with a restricted form of dependent types. Also, *Cayenne* [2] has dependent types and is able to encode predicate logic at the type level allowing types to be used as specifications of programs. However, this makes *Cayenne*'s type checking undecidable. In [27], the authors presented a framework for a Hindley/Milner kind of type systems with constraints as a set of formulas over a cylindric algebra. These approaches to constrained types are either undecidable in the general case or do not support the same constraint language as we do, that is numerical constraints. Closer to our work, the study of automatic generator derivation for data types has already been studied in previous works. For example, in [6], the authors adapt a Boltzmann model for random generation of algebraic data types, in particular for inductive types. Our approach is similar but we handle constrained types. In [15], the author explores the definition of generators for a class of dependent types. However, the proposed techniques are not automatic and do not aim to be uniform.

There exists several uniform sampling tools for *SAT* [14, 8]. We aim for the same goal but for NCSP instead. In [17], the authors focus on generating random uniform solutions for CSP, however they target discrete problems, with very small domain size, which are

not adapted to numerical variables with large cardinalities (e.g. machine-integers). The ese of CP techniques for test generation has already been explored in the CP literature. For instance, the *FocalTest* framework[7] uses a constraint-based approach to build inputs for property based testing. Another example is [18] in which the method proposed is used to generate white box tests. Our approach differs from these in several way: our resolution scheme is based on abstract domain instead of *clp(FD)*. Also our constraints are extracted from the types while theirs are derived from the statements of the program. Finally we aim at producing uniform generators which is not the case in these works.

We also explore the problem of generators definition using numerical abstract domains. In [34], the author proposes a way to quantify the precision of abstract values through a measure of volumes, including an approximated measuring of polyhedra. Also, [19] proposes an algorithm to solve boolean and linear constraints, and to randomly select values among the set of solutions using rejection sampling. This is close from what we do in Section 5 except that we have additional hypothesis that allow us to compute an exact volume and minimize the use of rejection sampling.

3 Testing Semantics

Random testing within a typed language requires the definition of pseudo-random generators for types. These generators are used to provide inputs for the functions whose output is checked against a given specification. A pseudo-random generator g for a data type τ is a function $g : \mathcal{S} \rightarrow \tau$, with $\mathcal{S}$ the type of random seeds, which is useful to make the tests reproducible. Previous work, such as [6], has shown that the derivation of efficient uniform random generators for algebraic data types can be made automatic, even in presence of recursive types. However, this is more difficult for constrained types: a constrained type can be seen as a pair $\langle \tau, p \rangle$, composed by an algebraic type τ and a predicate $p : \tau \rightarrow \texttt{bool}$. The set of its inhabitants is defined as $S = \{t \in \tau \mid p(t) = \texttt{true}\}$. As a result, a generator for a constrained type $\langle \tau, p \rangle$ needs to produce a value of type τ that satisfies p.

3.1 Type Language and Semantics

Our framework aims at generating tests that verify that some function does not violate the invariant attached to its return type. To do so, we provide to the developer the capability of constraining a type τ with an arbitrary predicate *i.e.* a function of type $\tau \rightarrow \texttt{bool}$. Therefore, our syntactic extension may be seen as a small but expressive annotation language. To be able to reason on the types of the program, we will suppose that all the values that we handle are explicitly typed. We consider a *ML*-like type language with constrained types. Its Bachus-Naur form (BNF) grammar is given in Figure 1.

A type declaration is composed of an identifier and a type expression. Type expressions can be: type identifiers, product types (tuples), sum types where each variant is differentiated by a unique constructor, and constrained type which we add to the language *via* the `[@satisfying]` annotation. Note that even if the syntax permits the definition of recursive types, these are not handled by our framework yet. The constraint language used for the definition of predicates over constrained types is relatively classic. Here, $\mathbb{I}$ and $\mathbb{F}$ are respectively the set of integers and floating point values, and $\mathbb{V}$ is the set of variable identifiers.

Annotating a type τ with a predicate p defines a partial specification for the program: all functions returning a value of type τ are expected to produce values that satisfy p. This property being in the general case out of the reach of a type checker [2], we propose to test it.

decl ::=	'type' ident '=' type	declaration
type ::=	ident	identifier
|	type {'*' type }	product
|	ident 'of' type { '|' ident 'of' type }	sum
|	type '[@satisfying' constraint ']'	constrained
constraint ::=	arith $\square$ arith	$\square \in \{\geq, \leq, =, \neq\}$
|	'not' constraint	negation
|	constraint '||' constraint	disjunction
|	constraint '&&' constraint	conjunction
arith ::=	i	$i \in \mathbb{I}$
|	f	$f \in \mathbb{F}$
|	v	$v \in \mathbb{V}$
|	arith $\diamond$ arith	$\diamond \in \{+, -, *, /, \%\}$
|	- arith	opposite

Figure 1 Grammar of the constrained type language.

3.2 Constraints semantics

We introduce inference rules that enriches a standard type algebra with constrained types: because types are composable, we need to define inference rules to propagate constraints from atomic types to composite types. For example, when a type τ is product or a sum of types that were constrained by a property, this property is lifted to τ accordingly.

▶ **Example 1.** Consider the following constrained types which define the type of positive `float` and 2D circles:

```
type positive = float [@satisfying (fun x -> x >= 0)]
type circle = (float * float) * positive}
```

Here, the type `circle` is not explicitly constrained, but as it depends on the type `positive`, an implicit constraint will be attached to it. More generally, whenever a type τ is defined using a constrained type τ', the constraint over τ is also inherited by τ'. Here, `circle` is implicitly constrained by the function: `(fun ((cx,cy),radius) -> radius >= 0)`.

In Figure 2, we give a formal definition of the semantics for constraints composition. We define it by induction over the syntax while considering a predicate environment ρ that stores, for each type identifier, the predicate that was attached to it. For clarity, we consider that an unconstrained type is a constrained type who is attached a tautology. Also, we suppose the initial environment ρ_0 already equipped with tautological constraints for primitives types: $\rho_0 = [\text{unit} \mapsto \lambda().\texttt{true}, \text{bool} \mapsto \lambda b.\texttt{true}, \text{int} \mapsto \lambda i.\texttt{true}, \text{float} \mapsto \lambda f.\texttt{true}]$. For each rule, the conclusion gives the derivation formula of a constraint for a given type, using the constraints derived in the premises. The constraint for product types is the conjunction of constraints attached to the type sub-components. For sum types, we determine for each variant the corresponding constraint and build a predicate based on pattern matching[1] to select a constructor using case by case reasoning. For type declarations, we use the notation $\rho[id \mapsto p_\tau]$ to denote the setting of the constraint associated to the type identifer id to its new value p_τ.

[1] `function patterns` is equivalent to `fun x -> match x with patterns`

$$\text{(declaration)}\frac{\rho(\tau) \to p_\tau}{(\mathit{type}\ \mathit{id} = \tau, \rho) \to (\rho[\mathit{id} \mapsto p_\tau])}$$

$$\text{(constrained)}\frac{\rho(\tau) \to p_\tau}{(\tau\,[\text{@}\mathit{satisfying}\ p], \rho) \to \lambda x.p(x) \wedge p_\tau(x)}$$

$$\text{(identifier)}\frac{p = \rho(\mathit{id})}{(\mathit{id}, \sigma) \to p}$$

$$\text{(product)}\frac{\rho(\tau_1) \to p_1 \quad \dots \quad \rho(\tau_n) \to p_n}{(\tau_1 * \dots * \tau_n, \rho) \to \lambda(x_1, \dots, x_n).p_1(x_1) \wedge \dots \wedge p_n(x_n)}$$

$$\text{(sum)}\frac{\rho(\tau_1) \to p_1 \quad \dots \quad \rho(\tau_n) \to p_n}{(c_1 \text{ of } \tau_1 \mid \dots \mid c_n \text{ of } \tau_n, \rho) \to \text{function } c_1(x_1) \mapsto p_1(x_1) \mid \dots \mid c_n(x_n) \mapsto p_n(x_n)}$$

Figure 2 Constraint semantics.

3.3 Generator semantics

The derivation of random generators for composite types given random generators for atomic types, is made following the same inductive principle as in the previous section. To keep track of which generator is associated to which type, we consider an environment σ which associates to each type identifier its generator. We suppose an initial environment σ_0 populated with uniform random generators for primitive types. Figure 3 presents the derivation of uniform random generators for constrained types.

$$\text{(declaration)}\frac{\sigma(\tau) \to g_\tau}{(\mathit{type}\ \mathit{id} = \tau, \sigma) \to (\sigma[\mathit{id} \mapsto g_\tau])}$$

$$\text{(constrained)}\frac{}{(\tau\,[\text{@}\mathit{satisfying}\ p], \sigma) \to \mathbf{solve}(\tau, p))}$$

$$\text{(identifier)}\frac{g = \sigma(\mathit{id})}{(\mathit{id}, \sigma) \to g}$$

$$\text{(product)}\frac{\sigma(\tau_1) \to g_1 \quad \dots \quad \sigma(\tau_n) \to g_n}{(\tau_1 * \dots * \tau_n, \sigma) \to \lambda i.(g_1(i), \dots, g_n(i))}$$

$$\text{(sum)}\frac{\sigma(\tau_1) \to \lambda i.c_1(g_1(i)) \quad \dots \quad \sigma(\tau_n) \to \lambda i.c_n(g_n(i))}{(c_1 \text{ of } \tau_1 \mid \dots \mid c_n \text{ of } \tau_n, \sigma) \to \mathbf{weighted}([(\mathbf{card}(\tau_1), g_n); \dots; (\mathbf{card}(\tau_n), g_n)])}$$

Figure 3 Generator semantics.

For product types (similar to Cartesian product of sets), generators are obtained by composing the generators obtained for their components. An important point of our work is to derive random generators with uniform distribution: each inhabitants of the type has the same probability to be drawn. Because uniform distributions do not compose so easily, especially in the case of sum types (union of sets), we have to take care of the cardinal of each type's population. Hence for sum types, we decompose the uniform sampling of a value in two steps: first choosing a constructor, and then drawing a value for this constructor. For this to be uniform, the first step takes into account the cardinality of the components: the more inhabitant one has, the more likely it has to be chosen. To achieve that, we introduce two procedures: **card** and **weighted**.

The procedure **card** gives the number of inhabitants of a given type. As we restrict ourselves to non-recursive types, keeping track of the cardinality of an algebraic data type is straightforward: cardinality for sum (resp. product) types is given by the sum (resp. product) of the cardinalities of their components. Computing cardinalities of constrained types is equivalent in our case to counting the solutions of a CSPs, which is a known hard problem [30]. Hence, we use approximations to compute the cardinal of constrained types which we present in the next section. The procedure **weighted** chooses a generator among a list of generators. To choose uniformly, each generator has a probability of being chosen equal to its associated weight.

Finally generators for constrained types are given by the procedure **solve** that, given a type declaration and a predicate, builds the generator corresponding to the constrained type. We give a definition of this procedure in the next section.

3.4 Test Generation

From the generators and the constraints we derived from the annotated program, we may now produce tests. We retrieve value declarations in the program for which the return type is a constrained one. If the value declaration is a constant c, we simply generate a test that consists of applying the predicate to c. If the value declaration is a function f, we will first retrieve, for each of its arguments, the associated generators to build the inputs. Equipped with input generators, we then apply f to the uniformly drawn inputs and validate the outputs using the constraint predicate.

4 Solving Constrained Types

We now study the derivation of generators and cardinality estimation for constrained types. One way to automatically do this is to extensively compute the set of values of a type and keep only those that satisfy a constraint. We may then randomly choose among those whenever a generator is called. This approach however is not practical and does not scale. Another possibility is to use a rejection sampling technique using the generator for τ and checks its values against p. This should yield a uniform generator but present an important flaw: when the cardinality of the constrained type $\langle \tau, p \rangle$ is small compared to the cardinality of the original type τ, this tends to be ineffective. Moreover, cardinality may only be an estimation, not an exact result. Instead, our approach is to solve constrained types by providing a measurable characterization of their inhabitants. We are then able to define uniform random generators over it. To do so, we see a constrained type as a constraint satisfaction problem, and its inhabitants as the solutions of this problem. We use for this task a hybrid approach, that mixes both techniques from Abstract Interpretation [11] and Constraint Programming [26], based on *abstract domains*. These are a key notion in abstract interpretation as they implement an abstract semantic for which they provide data-structures and define algorithmic aspects. They are designed to abstract program values and are thus particularly well-suited for our needs. Moreover, many constructive and systematic methods to design and compose such domains exist in the literature: numerical domains (intervals, congruences, polyhedra, octagons, etc.), domain composition operators (products, powersets, etc.).

4.1 CSP extraction from a constrained type

A constraint-satisfaction problem can be defined as a triplet $\langle \mathcal{V}, \mathcal{D}, \mathcal{C} \rangle$, where $\mathcal{V} = \{v_1, \ldots, v_n\}$ is a set of variables, $\mathcal{D} = \{d_1, \ldots, d_n\}$ a set of interval domains, each one being associated to a variable, and $\mathcal{C} = \{c_1, \ldots, c_m\}$ is a set of constraints over the variables. A solution of a CSP $(\mathcal{V}, \mathcal{D}, \mathcal{C})$ is an instance $i = \{v_1 \to x_1, ..., v_n \to x_n\}$ that satisfies all of the constraints by substitution of the variables with their value in i, that is:

$$\forall i, x_i \in d_i \wedge \forall c \in \mathcal{C}, c(\{v_1 \to x_1, ..., v_n \to x_n\})$$

In our case, we want to solve CSPs that are extracted from constrained types, to obtain an approximation of their sets of inhabitants. We can then tackle the problem of generating uniformly within this approximation. Consider the following declaration:

```
1 type itv = (int * int) [@satisfying (fun (inf,sup) -> inf <= sup)]
```

This type defines a bounded two-dimensional space, where dimensions correspond to the members of the tuple, named `inf` and `sup` in the predicate. These are constrained by the relation $\texttt{inf} \leq \texttt{sup}$. From a constraint solving point of view, the set of inhabitant of this type is the set of solutions of the CSP: $\langle \mathcal{V} = \{\texttt{inf}; \texttt{sup}\}, \mathcal{D} = \{[-2^{64}; 2^{64}]; [-2^{64}; 2^{64}], \mathcal{C} = \{\texttt{inf} \leq \texttt{sup}\}\} \rangle$

We have automated the process of CSP extraction for type definition by *inlining* their declaration: we give an identifier to each of the value of the type and then work within a simple numerical abstract element where each identifier corresponds to a variable. Doing so builds a CSP that abstracts some information about the *shape* of the type and hence, in parallel, we build a function that will reconstruct from the solutions of the CSP, a value with the correct shape.

4.2 CSP solving

Solving a numerical CSP usually means finding one or all the solutions of the problem. Because this is generally impossible when the domains of the variables are continuous (or just very large), solvers generally compute a set of boxes, that is a Cartesian product of intervals, that *covers* the solution space. The resolution of such problems works from above: given an initial coarse approximation, several heuristics are used until a sufficiently good cover is found. In order to build this cover, a constraint solver generally alternates two main steps:

- *Filtering:* which reduces the variables domains by removing values that do not satisfy the constraints.
- *Splitting:* which duplicates the problem to create two (or more) complementary sub-problems that are smaller, w.r.t. a certain measure, than the original one.

Repeating these two steps in turn does not necessarily terminate. Hence, this procedure generally goes on until the search space contains: no solution, only solutions, or is smaller than a given parameter. Producing such a cover can be sufficient for us if we are able to compute its exact number of solutions, and select one of these uniformly. However, using boxes poorly fits our need, in particular, when we are considering relational constraints, which appear fairly commonly in constrained types. Therefore, we use the solving method of [29], which is designed to be parameterized by any abstract domain. The algorithm introduced in [29] builds a set of abstract elements $\mathbb{S}$ that covers the solution space, i.e. for all instances i that satisfy all the constraints $\mathcal{C}$, we have $\exists e \in \mathbb{S}, i \in \gamma(e)$. Here γ is the usual concretization function for abstract values, that is the set of concrete instances abstracted by

an element. The algorithm starts from an initial abstract element e built from the domains of the variables. Then, e is filtered according to the set of constraints. If the filtered abstract element e' is not empty, three cases are possible:

- e' satisfies all the constraints, then it is added to the set of solutions.
- there is at least one constraint c that e' does not satisfy and its size is small enough with respect to a given threshold, it is also added to the set of solutions.
- otherwise, e' is divided into sub-elements using the split operator and the process is repeated with each of these sub-elements.

When all of the elements have been processed, the union of the element in $\mathbb{S}$ is a sound over-approximation of the solution space. We propose a slightly modified version of this algorithm, which, we believe, is better suited for our needs.

4.3 Solving Algorithm

Contrarily to the algorithm of [29], we distinguish inner elements from outer elements. Inner elements are the ones that are guaranteed to only contain solutions while outer elements may contain non-solutions. Our algorithm is defined in Figure 1.

Algorithm 1 Abstract solving for generator derivation.

1: **function** SOLVE($\mathcal{D}, \mathcal{C}, \epsilon, max$)
2: $\quad I \leftarrow \emptyset$
3: $\quad O \leftarrow \emptyset$
4: $\quad e =$**init**$(\mathcal{D})$
5: $\quad O \leftarrow$**insert**(e, O)
6: $\quad$**while** $\mu(I, O) > \epsilon \vee |I| < max$ **do**
7: $\quad\quad e \leftarrow$ **biggest**(O)
8: $\quad\quad e' \leftarrow \rho(e, \mathcal{C})$
9: $\quad\quad$**if** $e' \neq \bot$ **then**
10: $\quad\quad\quad$**if solution**(e', C) **then**
11: $\quad\quad\quad\quad$**push** e' in I
12: $\quad\quad\quad$**else**
13: $\quad\quad\quad\quad$**push split** (e) in O
14: **return** I, O

Our algorithm maintains two sets of elements: inner elements I, and outer elements O. Here, I under-approximates the solution set and O is such that $I \cup O$ over-approximates it. It first initializes an abstract element e and inserts it in the set of outer elements O. Then, the main loop proceeds repeating the steps: The biggest element of O is selected (which is more likely to contains more solutions), filtered using the propagator ρ, and pushed in I if it satisfies the constraints. Otherwise, it is split and the sub-elements are pushed back in O.

As we will see, the number of element in the cover is related to the size of the code generated for the random samplers. Also their rejection rate is closely related to the proportion of inner elements in the cover. Thus the tuning of the obtained generator may be controlled by the max and ϵ parameters: max is needed to avoid an exponential growth of the cover and ϵ fixes a rejection rate to reach. The next section gives insights about the code generation and defines precisely $\mu(I, O)$.

4.4 From covers to generators

Once we have computed a cover (I, O) for a constrained type $\langle \tau, p \rangle$, we have to compile it into a generator for $\langle \tau, p \rangle$. A cover is a set of inner elements and a set of outer elements. Thus, to compile a cover into a generator we start by compiling each element $e \in (I \cup O)$ into a generator. We then choose an element e of the cover, and generate an instance i using the generator associated with e. If e belongs to I, then i is kept, but if it belongs to O, then i must be checked against p to make sure that it is a valid member of the constrained type. In that case, we are forced to rely on rejection sampling.

Hence, the generator for the whole cover is actually a dispatcher to the generators of the elements: it randomly selects an element's generator with a probability proportional to the volume of the underlying set of solutions, then calls it. Note that this yields a uniform generator because the split operator produces elements that do not overlap. Otherwise, instances belonging to several elements would appear more often during the sampling.

To compute the cardinality of a constrained type $\langle \tau, p \rangle$ we must reason on the number of solutions of its associated cover. This exact number is given by:

$$\sum_{e \in I} |\gamma(e)| + \sum_{e \in O} |\, \{ i \in \gamma(e) \mid p(i) \} \,|$$

However, as the number of solutions of an outer element e depends on the predicate p, it is hard to compute exactly in the general case. Instead, we use an over-approximation of this number that is $|\{i \in \gamma(e)\}|$. Our cardinality estimation for a cover (I, O), denoted $card(I, O)$, is given by:

$$card(I, O) = \sum_{e \in I} |\gamma(e)| + \sum_{e \in O} |\gamma(e)|$$

Despite of this over-approximation, our sampling method is uniform, thanks to rejection sampling: rejecting erroneous solutions does not bias the uniform distribution. To ensure the sampling of the uniform distribution in case of rejection, the whole sampling process is restarted from the beginning *i.e.* an abstract element is drawn w.r.t. to its volume and so on.

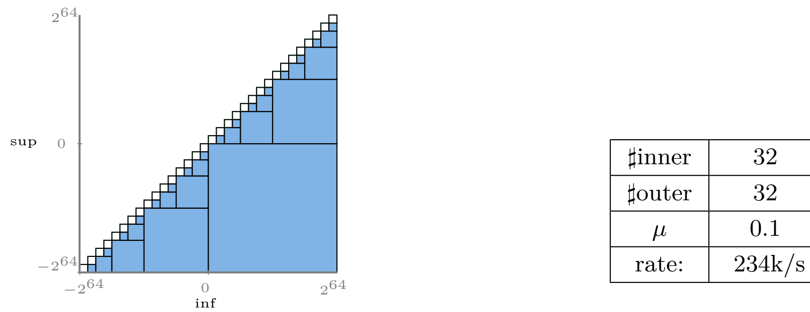
One way to mitigate the occurrences of rejections, is to minimize the volume of O. Hence the introduction of $\mu(I, 0)$ defined as

$$\mu = 1 - \frac{card(I, \emptyset)}{card(I, O)}$$

It measures the relative error between the over-approximation and the exact volume of an abstract element. At the start of the algorithm process $\mu(I, O) = 1$. Then it decreases at each iteration until reaching the desired precision. Note that $\mu = 0$ means that the abstract element corresponds exactly to the set of solutions of the CSP and the random sampling will be made without rejection.

5 Abstract Domains for Random Testing

The technique presented in the previous section relies on abstract domains to solve and derive generators for constrained types. To use abstract domains in the context of constraint resolution, the authors of [29] define several operators and requirements they must satisfy. Our use-case requires the same operators, plus an additional one for the random sampling of an element. The following definition gives these.



♯inner	32
♯outer	32
μ	0.1
rate:	234k/s

(a) Graphical resolution of the CSP. **(b)** Metrics over the resulting generator.

Figure 4 Solving the `itv` constrained type with boxes.

▶ **Definition 2.** *Abstract domains for constraint solving are given by*

- *a partial order* $\langle D, \sqsubseteq \rangle$ *and the usual abstract set operators and values* $\langle \top, \bot, \sqcap, \sqcup \rangle$
- *an abstraction* α *and a concretization function* γ
- *a size function* ***card****:* $D \rightarrow \mathbb{N}^+$
- *a split function* ***split****:* $D \rightarrow P(D)$*,*
- *a constraint filtering operator* $\rho : D \times \mathcal{C} \rightarrow D \cup \{\bot\}$*, which given an abstract value* e *and a constraint* c *computes the smallest abstract value (possibly empty) entailed by* c *and* e*.*
- *a generation function* ***uniform****:* $D \rightarrow \mathbb{R}^n$

For sake of concision, we will not detail the compilation of abstract elements into expressions. Instead we define random generation functions on abstract elements.

5.1 Boxes

A very studied abstract domain in continuous constraint solving is the abstract domain of boxes. Its main operators rely on interval arithmetic [1] and were already introduced in previous work [35]. In our case, the cardinality of a box $b = [a_1, b_1] \times \cdots \times [a_n, b_n]$ is its volume (defined as its Lebesgue measure) i.e. $card(b) = \prod_{i=1}^{n}(b_i - a_i)$. For the split operation, we use standard bisection of a variable with the so-called *largest-first* heuristic which chooses the variable with the biggest range as a variable selection strategy. Also, for the filtering operation we use the HC4 constraint propagation algorithm [3]. Finally, the uniform distribution over a n-dimensional box is sampled using n uniform (over $[0, 1]$) and independant random variables $(r_i)_{i \leq n}$ with the formula $(a_i + r_i * (b_i - a_i))_{i \leq n}$.

Defining these operators is sufficient to embed this domain within our testing framework. Using boxes, it is possible to derive efficient generators for constrained type that use non relational constraints, that are constraints that involve a single variable. However, producing fast generators in presence of relational constraints is harder *e.g.* the `itv` type of Example 4.1. Figure 4 shows graphically the approximation we obtain and the corresponding generated OCaml code is given in Appendix A.1. Filled elements correspond to inner elements, and empty elements correspond to outer elements. The Table 4b gives some metrics over the cover and the resulting generator: the row ♯inner (resp. ♯outer) gives the number of inner (resp. outer) elements, the third row gives the μ-score and the last line gives the generation rate of the obtained generator, which is measured experimentally and is given in number of calls per seconds.

We can see that when dealing with an affine constraint, the use of boxes misfits our needs: the precision needed to avoid rejections during the sampling leads to a high number of elements in the cover of the solution space and so a very large (in term of code size) and slow sampler. To solve this issue, we focus in the next subsection on a relational abstract domain.

5.2 Polyhedra

The polyhedra abstract domain [12] is a numerical relational abstract domain that approximates sets of points as convex closed polyhedra. Modern implementations [20] generally follow the "double description approach" and maintain two dual representations for each polyhedron: a set of linear constraints and a set of vertices. The constraint representation of the polyhedron is the intersection of the half-spaces defined by the linear constraints. The vertex representation is the convex hull of a set of points. These dual descriptions are very useful in practice as polyhedra operators [20] are generally easier to define on one representation rather than the other. We use both to define the filtering, splitting, measure and random sampling functions. Constraint filtering for polyhedra generally consists in building a sound linear approximation of a constraint (different approximations can be used e.g. quasi-linearization [25] or linearization for polynomial constraints [22]), and then adding it to the representation of a given polyhedron.

Volume computation and uniform random sampling within a polyhedron are notoriously hard tasks. The first problem is ♯P-hard (see [4] for example). For the sampling, the fastest algorithm (to our knowledge) has an expected (time) complexity in $\mathcal{O}^\star\left(n^3\right)$[2] (see Theorem 3.1.3 of [10]) whose result validity and running time are probabilistic.

We mitigate these problems by systematically considering a simpler case: simplices. A simplex is the most simple polyhedron with a non null volume, obtainable in an n-dimensional space as it is the convex hull of $n+1$ vertices. Instead of stopping the split and filtering procedure when an inner polyhedron is found, we continue to split elements until all are simplices. To do so, we use a split operator that favors the creation of simplices that can be summarized as follows. Suppose a polyhedron P lives in a n-dimensional space and is not already a simplex, i.e. it is defined using at least $n+2$ vertices:

- pick $n+1$ arbitrary vertices, $\{v_0, \ldots, v_n\}$
- compute Q the smallest polyhedron that encompasses $\{v_0, \ldots, v_n\}$
- return $Q \cup (P \ominus Q)$

Here, the difference operator ($\ominus$) used in the last step corresponds to the set difference of two polyhedra. This operator, illustrated by Figure 5, uses the constraint representation: the polyhedron Q is a space defined by the conjunction of a set of constraints C_Q, where each constraint is a linear inequality over the variables of the polyhedron. It is defined as:

$$P \ominus Q \triangleq \left\{\bigcup P \cap \neg c \mid c \in C_Q\right\}$$

Our simplex split allows us to decompose a polyhedron while performing the resolution, which allows us to define both the volume estimation and the uniform random sampling on simplices. The way to compute the volume of a simplex is well known and not discussed here.

In order to sample a point of a simplex, we first consider its mirror parallelotope: given a simplex polyhedron P defined by the set of vertices $V = \{v_1, \ldots, v_n\}$, we build its mirror parallelotope P' by adding one vertex t to V, obtained by translation of v_1 by

[2] The $\star$ in $\mathcal{O}^\star\left(n^3\right)$ hides logarithmic factor.

Figure 5 Difference operator: $P \ominus Q$.

(a) r is inside P, we keep it.

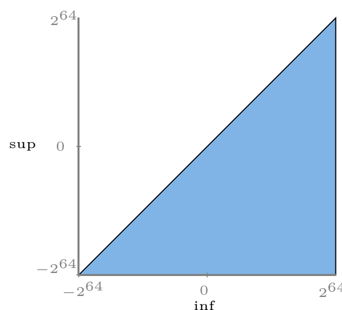
(b) r is not inside P, we take r' its reflection point according to b.

Figure 6 Simplex uniform generation procedure.

the vector $\delta = \overrightarrow{v_1v_2} + \overrightarrow{v_1v_3} + \ldots + \overrightarrow{v_1v_n}$. Then, a point r of P' is uniformly drawn by sampling $n-1$ independent and uniform random variables $(r_i)_{i \le n-1}$ over $[0,1]$, such that $r = r_1 \cdot \overrightarrow{v_1v_2} + r_2 \cdot \overrightarrow{v_1v_3} + \ldots + r_{n-1} \cdot \overrightarrow{v_1v_n}$.

- If r is inside P we keep it.
- Else, r is outside P, we keep r' its symmetrical according to b.

Where b is the barycenter of the opposite face to v_1, i.e. the $n-2$ dimensional face of corners $\{v_2, \ldots, v_n\}$. Figure 6 illustrates this procedure for a 2D-simplex. Note that every point in P has exactly two ways of being chosen, that is directly or by symmetry, which makes this procedure uniform. Equipped with this relational abstract domain, we can compare the result obtained for the CSP of the `itv` type, illustrated by Figure 7. The corresponding code, given in Appendix A.2, is approximately three times faster than the one obtained with boxes, thanks to the null rejection rate ($\mu = 0$).



♯ inner	1
♯ outer	0
μ	0
rate:	627k/s

Figure 7 Approximation of the `itv` type with polyhedra.

6 Current Implementation and Benchmarks

Our implementation is built as a syntax extension for *OCaml* based on a preprocessing mechanism. We use *OCaml*'s attributes to allow the user to annotate its types with constraints. Attributes are placeholders in the syntax tree which are ignored by the compiler but can be used by external tools such as ours. We have developed a prototype to demonstrate the interest of our technique. It is open-source and available at the url `https://github.com/ghilesZ/Testify`. It currently implements the work we have presented in Sections 3 and 4 plus some other features we describe briefly here. Our main focus is the constraint solving of constrained types to automatically derive generators. However for a more practical use, we provide the programmer two other ways of specifying a generator for a given type. The first one is using the **rejection** keyword, for example `type even = int [@satisfying rejection(fun x -> x mod 2 = 0)]`. Constraints tagged as rejected will not be handled by the constraint solver but will simply be treated *a posteriori* after the generation to keep or discard generated values. The second one is by manual annotation of functions: the developer can tag one of its own function of type $\mathcal{S} \to \tau$ as a custom generator for the type τ. This will overload the generator automatically derived by our framework. Also we have presented a type language with tuples and sum types but our actual implementation also handles polymorphic and record types. We have not detailed those here as they do not represent a specific challenge from a constraint solving perspective.

Also, as the numerical values we manipulate in our CSPs correspond to sets of machine integers and floating point numbers, the computation we perform are likely to produce round-off errors if made using standard precision. To bypass this issue, all of our computations are made using arbitrary precision integers and rationals. Moreover, our uniformity metrics are valid in $\mathbb{R}$ but not necessarily for floating point numbers. We believe that this is a reasonable approximation. Finally our current implementation suffers from some limitations, for example: we handle explicitly typed values only and do not enjoy the capabilities of *OCaml*'s type inference mechanism. Also, we do not have a generator derivation mechanism for recursive types and further work will be needed to lift these restrictions.

We have applied our testing framework on some open source *OCaml* libraries where we have identified and annotated some constrained types. These types use quite simple constraints that are mainly bound constraints, linear constraints of the form $x_i \leq x_j$, and some disjunctions of such constraints. This reflects the fact that when writing code, the developer keeps in mind a relatively simple representation of the set of possible values of its type. Table 1 presents some metrics over the tests we have generated using different configurations. The first two columns give some information about the constrained type from which we derive the CSP. The first column specifies the kind of constraints attached to the type: *bc* for bound constraints, *lin* for linear constraints, and *dis*, for disjunctions of linear constraints. The second column indicates the number of variables appearing in the corresponding CSP. The next columns give some quality metrics over the generators: the generation speed and uniformity. The column $\mathcal{B}_8$ (resp. $\mathcal{B}_{64}$) gives the measures for the boxes abstract domain with a cover size limited to 8, (resp. 64), the column $\mathcal{P}$ gives these values for polyhedra (with a cover size limited to 64). For comparison purposes, we add a supplementary column $\mathcal{RS}$ that gives the statistics we obtain using rejection sampling. The row μ shows the value of $\mu(I, O)$ at the end of Algorithm 1 and the last row gives the generation rate, i.e. the number of generated values (in thousands) per seconds. The results of Table 1 validates our intuition: constraint solving of constrained types helps producing efficient generators. All of our abstract domain based configurations outperform the rejection

Table 1 Generation rate and value of μ per configuration.

CSP		$\mathcal{B}_8$		$\mathcal{B}_{64}$		$\mathcal{P}$		$\mathcal{RS}$	
kind	♯var	rate	μ	rate	μ	rate	μ	rate	μ
bc	2	**10075**	0	10059	0	2604	0	816	0
bc	2	9860	0	**9996**	0	2586	0	842	0
lin	2	2551	0.6	**3312**	0.35	3262	0	1333	0
bc	1	26055	0	**26146**	0	7870	0	1922	0
lin	2	2602	0.6	3200	0.35	**3358**	0	1337	0
lin	2	2594	0.6	3166	0.35	**3366**	0	1080	0
lin	2	2622	0.6	**3312**	0.35	3234	0	428	0
bc	2	10246	0	**10436**	0	2768	0	1449	0
lin	2	2066	0.82	2812	0.44	**6394**	0	115	0
lin	3	615	1	716	1	**1362**	0	403	0
lin	2	2213	0.6	2750	0.35	**6146**	0	2278	0
dis	5	462	1	523	0.85	**1189**	0	452	0

sampling approach by one order of magnitude. In columns $\mathcal{B}_8$ and $\mathcal{B}_{64}$ the size limit for the cover varies and as expected, it decreases μ. Moreover, on the benchmarks it almost always increases the generation rate. This indicates that the execution time benefits more from reducing the rejection rate than from increasing the size of the cover. However, it is slower on most examples using bound constraints than the boxes due to the relative complexity of the simplex generation procedure compared to the one of boxes. For the examples with linear relational constraints, the samplers produced with polyhedrons are faster.

7 Conclusion

We have proposed in this paper an automated type-driven testing framework for programs that manipulate constrained types. We have defined an automated technique to derive efficient random uniform generators for numerical constrained types. To do so, we have proposed several abstract domains, both relational and non-relational, with the addition of random value generator. The strength of our method lies in its genericity: it allows us to shift the problem of uniform random sampling of CSP solution to the definition of a uniform generation operator for abstract domains. Further works include the study of such operators for some popular domains of abstract interpretation such as congruences, products, octagons, etc. Another idea would be to couple our methods with Boltzmann generation techniques (like in [6]) in order to build generators for recursive types with constraints.

Our techniques were implemented in a prototype that can be used as a preprocessor for *OCaml* programs. Even though we targeted *OCaml* and although we have taken advantage of its generic syntax extension mechanism, the process we have presented could be adapted to most programming languages. The tests we are able to generate are not made to be as pertinent as some hand written tests and our goal is not to replace those. However our approach being fully automatic and fast, it can be used *on the fly*, while programming, to find bugs quickly. We believe that random uniform generators for abstract domain open the way for hybrid techniques at the border between sound static analysis approaches and complete testing techniques. For example, one could imagine a backward analysis able to derive necessary preconditions (as in [32]) that exhibit bugs in a program and then uniformly generating tests within the corresponding abstract element to find actual bugs.

References

1 Ignacio Araya, Gilles Trombettoni, and Bertrand Neveu. Exploiting Monotonicity in Interval Constraint Propagation. In Association for the Advancement of Artificial Intelligence, editor, *Twenty-Fourth AAAI Conference on Artificial Intelligence*, pages 9–14, Atlanta, United States, July 2010. http://www.aaai.org/ocs/index.php/AAAI/AAAI10/paper/view/1699. URL: https://hal-enpc.archives-ouvertes.fr/hal-00654400.

2 Lennart Augustsson. Cayenne – a language with dependent types. In S. Doaitse Swierstra, José N. Oliveira, and Pedro R. Henriques, editors, *Advanced Functional Programming*, pages 240–267, Berlin, Heidelberg, 1999. Springer Berlin Heidelberg.

3 Frédéric Benhamou, Frédéric Goualard, Laurent Granvilliers, and Jean-Francois Puget. Revising hull and box consistency. In *Logic Programming: Proceedings of the 1999 International Conference on Logic Programming*, pages 230–244, January 1999.

4 G. Brightwell and P. Winkler. Counting linear extensions is ♯p-complete. In *STOC '91*, 1991.

5 Jacob Burnim, Sudeep Juvekar, and Koushik Sen. Wise: Automated test generation for worst-case complexity. In *Proceedings of the 31st International Conference on Software Engineering*, ICSE '09, page 463–473, New York, NY, USA, 2009. Association for Computing Machinery. doi:10.1109/ICSE.2009.5070545.

6 Benjamin Canou and Alexis Darrasse. Fast and sound random generation for automated testing and benchmarking in objective caml. In *Proceedings of the 2009 ACM SIGPLAN Workshop on ML*, ML '09, page 61–70, New York, NY, USA, 2009. Association for Computing Machinery. doi:10.1145/1596627.1596637.

7 Matthieu Carlier, Catherine Dubois, and Arnaud Gotlieb. Focaltest: A constraint programming approach for property-based testing. In José Cordeiro, Maria Virvou, and Boris Shishkov, editors, *Software and Data Technologies*, pages 140–155, Berlin, Heidelberg, 2013. Springer Berlin Heidelberg.

8 Supratik Chakraborty, Kuldeep S. Meel, and Moshe Y. Vardi. A scalable and nearly uniform generator of sat witnesses. In Natasha Sharygina and Helmut Veith, editors, *Computer Aided Verification*, pages 608–623, Berlin, Heidelberg, 2013. Springer Berlin Heidelberg.

9 Koen Claessen and John Hughes. Quickcheck: A lightweight tool for random testing of haskell programs. In *Proceedings of the Fifth ACM SIGPLAN International Conference on Functional Programming*, ICFP '00, page 268–279, New York, NY, USA, 2000. Association for Computing Machinery. doi:10.1145/351240.351266.

10 Benjamin Cousins. *Efficient high-dimensional sampling and integration.* PhD thesis, Georgia Institute of Technology, 2017.

11 Patrick Cousot and Radhia Cousot. Abstract interpretation: a unified lattice model for static analysis of programs by construction or approximation of fixpoints. In *Conference Record of the Fourth Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, pages 238–252, 1977.

12 Patrick Cousot and Nicolas Halbwachs. Automatic discovery of linear restraints among variables of a program. In *Proceedings of the 5th ACM SIGACT-SIGPLAN symposium on Principles of programming languages*, pages 84–96, 1978.

13 Simon Cruanes. *QuickCheck inspired property-based testing for OCaml.* URL: https://github.com/c-cube/qcheck.

14 Rafael Dutra, Kevin Laeufer, Jonathan Bachrach, and Koushik Sen. Efficient sampling of sat solutions for testing. In *2018 IEEE/ACM 40th International Conference on Software Engineering (ICSE)*, ICSE '18, page 549–559, New York, NY, USA, 2018. Association for Computing Machinery. doi:10.1145/3180155.3180248.

15 Peter Dybjer, Qiao Haiyan, and Makoto Takeyama. Random generators for dependent types. In Zhiming Liu and Keijiro Araki, editors, *Theoretical Aspects of Computing - ICTAC 2004*, pages 341–355, Berlin, Heidelberg, 2005. Springer Berlin Heidelberg.

16 Patrice Godefroid, Nils Klarlund, and Koushik Sen. Dart: Directed automated random testing. *SIGPLAN Not.*, 40(6):213–223, 2005. doi:10.1145/1064978.1065036.

17 Vibhav Gogate and Rina Dechter. A new algorithm for sampling csp solutions uniformly at random. In Frédéric Benhamou, editor, *Principles and Practice of Constraint Programming - CP 2006*, pages 711–715, Berlin, Heidelberg, 2006. Springer Berlin Heidelberg.

18 A. Gotlieb, B. Botella, and M. Watel. Inka: Ten years after the first ideas. In *19th International Conference on Software & Systems Engineering and their Applications (ICSSEA'06)*, Paris, France, December 2006.

19 Erwan Jahier and Pascal Raymond. Generating random values using Binary Decision Diagrams and Convex Polyhedra. In Frédéric Benhamou, Narendra Jussien, and Barry O'Sullivan, editors, *Trends in Constraint Programming*, page 416 pp. ISTE, 2007. URL: https://hal.archives-ouvertes.fr/hal-00389766.

20 Bertrand Jeannet and Antoine Miné. Apron: A library of numerical abstract domains for static analysis. In *Proceedings of the 21th International Conference Computer Aided Verification (CAV 2009)*, 2009.

21 Xavier Leroy, Damien Doligez, Alain Frisch, Jacques Garrigue, Didier Rémy, and Jérôme Vouillon. The ocaml system release 4.02. *Institut National de Recherche en Informatique et en Automatique*, 54, 2014.

22 Alexandre Maréchal, Alexis Fouilhé, Tim King, David Monniaux, and Michaël Périn. Polyhedral approximation of multivariate polynomials using handelman's theorem. In *17th International Conference on Verification, Model Checking, and Abstract Interpretation*, pages 166–184, 2016.

23 Jan Midtgaard. Quickchecking patricia trees. In Meng Wang and Scott Owens, editors, *Trends in Functional Programming*, pages 59–78, Cham, 2018. Springer International Publishing.

24 Jan Midtgaard, Mathias Justesen, Patrick Kasting, Flemming Nielson, and Hanne Nielson. Effect-driven quickchecking of compilers. *Proceedings of the ACM on Programming Languages*, 1:1–23, August 2017. doi:10.1145/3110259.

25 Antoine Miné. Symbolic methods to enhance the precision of numerical abstract domains. In *7th International Conference on Verification, Model Checking, and Abstract Interpretation*, 2006.

26 Ugo Montanari. Networks of constraints: Fundamental properties and applications to picture processing. *Information Science*, 7(2):95–132, 1974.

27 Martin Odersky, Martin Sulzmann, and Martin Wehr. Type inference with constrained types. *TAPOS*, 5:35–55, January 1999. doi:10.1002/(SICI)1096-9942(199901/03)5:1<35::AID-TAPO4>3.0.CO;2-4.

28 Michal Palka, Koen Claessen, Alejandro Russo, and John Hughes. Testing an optimising compiler by generating random lambda terms. *Proceedings - International Conference on Software Engineering*, January 2011. doi:10.1145/1982595.1982615.

29 Marie Pelleau, Antoine Miné, Charlotte Truchet, and Frédéric Benhamou. A constraint solver based on abstract domains. In *Proceedings of the 14th International Conference on Verification, Model Checking, and Abstract Interpretation (VMCAI 2013)*, 2013.

30 G. Pesant. Counting solutions of csps: A structural approach. In *IJCAI*, 2005.

31 F. Pottier. Strong automated testing of ocaml libraries. In *JFLA 2021 - 32es Journées Francophones des Langages Applicatifs*, 2020.

32 Xavier Rival. Understanding the origin of alarms in astrée. In Chris Hankin and Igor Siveroni, editors, *Static Analysis*, pages 303–319, Berlin, Heidelberg, 2005. Springer Berlin Heidelberg.

33 Colin Runciman, Matthew Naylor, and Fredrik Lindblad. Smallcheck and lazy smallcheck automatic exhaustive testing for small values. In *Proceedings of the First ACM SIGPLAN Symposium on Haskell*, volume 44, pages 37–48, January 2008.

34 Pascal Sotin. Quantifying the precision of numerical abstract domains, 2010. URL: https://hal.inria.fr/inria-00457324.

35 Charlotte Truchet, Marie Pelleau, and Frédéric Benhamou. Abstract domains for constraint programming, with the example of octagons. *International Symposium on Symbolic and Numeric Algorithms for Scientific Computing*, pages 72–79, 2010. doi:10.1109/SYNASC.2010.69.

36 H. Xi. Dependent ml an approach to practical programming with dependent types. *J. Funct. Program.*, 17:215–286, 2007.

37 M Zalewski. *American fuzzy lop*. URL: `https://lcamtuf.coredump.cx/afl/`.

A Generated code for generators

A.1 Generated code using the boxes abstract domain

The following code gives the generator for the **itv** type we derived using the boxes abstract domain. The `weighted` procedure chooses a generator w.r.t. their probability. To perform quickly on average the list is sorted in decreasing order of probabilities, first elements being bigger, and thus more likely to be chosen. The function is more thus likely to stop quickly in average without harming uniformity.

```
weighted
[(0.400000000001,
   ((fun x ->
       (fun i -> (((get_int "x") i), ((get_int "y") i)))
         ((fun rs ->
             [("y", ((mk_int_range 0 0x3fffffffffffffff) rs));
             ("x", ((mk_int_range 0x4000000000000000 (-1)) rs))]) x))));
(0.100000000001,
  (reject (fun (x, y) -> x <= y)
     (fun x ->
        (fun i -> (((get_int "x") i), ((get_int "y") i)))
          ((fun rs ->
              [("y",
                 ((mk_int_range 0x2000000000000000 0x3fffffffffffffff) rs));
              ("x", ((mk_int_range 0 0x1fffffffffffffff) rs))]) x))));
(0.100000000001,
  (reject (fun (x, y) -> x <= y)
     (fun x ->
        (fun i -> (((get_int "x") i), ((get_int "y") i)))
          ((fun rs ->
              [("y", ((mk_int_range 0 0x1fffffffffffffff) rs));
              ("x", ((mk_int_range 0 0x1fffffffffffffff) rs))]) x))));
(0.100000000001,
  (reject (fun (x, y) -> x <= y)
     (fun x ->
        (fun i -> (((get_int "x") i), ((get_int "y") i)))
          ((fun rs ->
              [("y", ((mk_int_range 0x6000000000000000 (-1)) rs));
              ("x",
                ((mk_int_range 0x4000000000000000 0x5fffffffffffffff) rs))])
             x))));
(0.100000000001,
  (reject (fun (x, y) -> x <= y)
     (fun x ->
        (fun i -> (((get_int "x") i), ((get_int "y") i)))
          ((fun rs ->
              [("y",
                 ((mk_int_range 0x4000000000000000 0x5fffffffffffffff) rs));
              ("x",
                ((mk_int_range 0x4000000000000000 0x5fffffffffffffff) rs))])
             x))));
(0.0500000000001,
  (reject (fun (x, y) -> x <= y)
     (fun x ->
        (fun i -> (((get_int "x") i), ((get_int "y") i)))
          ((fun rs ->
              [("y",
                 ((mk_int_range 0x2000000000000000 0x3fffffffffffffff) rs));
              ("x",
                ((mk_int_range 0x3000000000000000 0x3fffffffffffffff) rs))])
             x))));
(0.0500000000001,
  (reject (fun (x, y) -> x <= y)
     (fun x ->
        (fun i -> (((get_int "x") i), ((get_int "y") i)))
          ((fun rs ->
              [("y",
                 ((mk_int_range 0x2000000000000000 0x3fffffffffffffff) rs));
```

```
59                ("x",
60                  ((mk_int_range 0x2000000000000000 0x2fffffffffffffff) rs))])
61              x))));
62 (0.0500000000001,
63   (reject (fun (x, y) -> x <= y)
64      (fun x ->
65         (fun i -> (((get_int "x") i), ((get_int "y") i)))
66           ((fun rs ->
67                [("y", ((mk_int_range 0x6000000000000000 (-1)) rs));
68                ("x", ((mk_int_range 0x7000000000000000 (-1)) rs))]) x))));
69 (0.0500000000001,
70   (reject (fun (x, y) -> x <= y)
71      (fun x ->
72         (fun i -> (((get_int "x") i), ((get_int "y") i)))
73           ((fun rs ->
74                [("y", ((mk_int_range 0x6000000000000000 (-1)) rs));
75                ("x",
76                  ((mk_int_range 0x6000000000000000 0x6fffffffffffffff) rs))])
77              x))))]
78
```

A.2 Generated code using the polyhedra abstract domain

The following code was generated by our framework as a generator for the **itv** type using the polyhedra abstract domain. The `simplex` procedure corresponds to drawing method given in section 5.2.

```
1  fun x ->
2  (fun i -> (((get_int "x") i), ((get_int "y") i)))
3    ((simplex
4        [((mk_int 0x4000000000000000), "y");
5        ((mk_int 0x4000000000000000), "x")]
6        [[((mk_int 0x3ffffffffffffe00), "y");
7         ((mk_int 0x4000000000000000), "x")];
8        [((mk_int 0x3ffffffffffffe00), "y");
9        ((mk_int 0x3ffffffffffffe00), "x")]]
10       [((mk_int 0x3ffffffffffffe00), "y"); ((mk_int 0), "x")]) x)
```