Witness Generation for JSON Schema

Lyes Attouche
Université Paris-Dauphine – PSL
lyes.attouche@dauphine.fr

Mohamed-Amine Baazizi
Sorbonne Université, LIP6 UMR 7606
baazizi@ia.lip6.fr

Dario Colazzo
Université Paris-Dauphine – PSL
dario.colazzo@dauphine.fr

Giorgio Ghelli
Dip. Informatica, Università di Pisa
ghelli@di.unipi.it

Carlo Sartiani
DIMIE, Università della Basilicata
carlo.sartiani@unibas.it

Stefanie Scherzinger
Universität Passau
stefanie.scherzinger@uni-passau.de

ABSTRACT

JSON Schema is an important, evolving standard schema language for families of JSON documents. It is based on a complex combination of structural operators, Boolean operators, including full negation, and mutually recursive variables. The static analysis of JSON Schema documents comprises practically relevant problems, including schema satisfiability, inclusion, and equivalence. These three can be reduced to witness generation: given a schema, generate an element of the schema — if it exists — otherwise report failure. Schema satisfiability, inclusion, and equivalence have been shown to be decidable, by reduction to reachability in alternating tree automata. However, no witness generation algorithm has yet been formally described. We contribute a first, direct algorithm for JSON Schema witness generation. We study its effectiveness and efficiency, in experiments over several schema collections, including thousands of real-world schemas. Our focus is on the completeness of the language (where we only exclude the `"uniqueItems"` operator) and on the ability of the algorithm to run in reasonable time on a large set of real-world examples, despite the exponential complexity of the problem.

KEYWORDS

JSON Schema, witness generation, inclusion, equivalence

1 INTRODUCTION

This paper is about witness generation for JSON Schema [33], the de-facto standard schema language for JSON [10, 11, 18, 35].

JSON Schema is a schema language based on a set of *assertions* that describe features of the JSON values described and on logical and structural combinators for these assertions.

The semantics of this language can be subtle. For instance, the two schemas below differ in their syntax, but are in fact equivalent. Schema a) explicitly states that any instance must be an object, and that a property named "foo" is not allowed. Schema b) implicitly requires the same: the `required` keyword has implicative semantics, stating that *if* the instance is an object, it must contain a property named "foo". Via negation, it is enforced that the instance must be an object, where a property named "foo" is not allowed. While this specific example is artificial, it exemplifies the most common usage of `not` in JSON Schema [13].

```
(a) { "type": "object",
      "properties": { "foo": false } }
(b) { "not": { "required": ["foo"] } }
```

Validation of a JSON value J with respect to a JSON Schema schema S, denoted $J \vDash S$, is a well-understood problem that can be solved in time $O(|J|^2|S|)$ [35]. The JSON Schema Test Suite [34], a collection of validation tests, lists over 50 validator tools, at the time of writing. Yet there are static analysis problems, equally relevant, where we still lack well-principled tools. We next outline these problems, and then point out that they can be ultimately reduced to JSON Schema witness generation, the focus of this work.

Inclusion $S \subseteq S'$: does, for each value J, $J \vDash S \Rightarrow J \vDash S'$? Checking schemas for inclusion (or containment) is of great practical importance: if the output format of a tool is specified by a schema S, and the input format of a different tool by a schema S', the problem of format compatibility is equivalent to schema inclusion $S \subseteq S'$; given the high expressive power of JSON Schema, this "format" may actually include detailed information about the range of specific parameters. For example, the IBM ML framework LALE [16] adopts an incomplete inclusion checking algorithm for JSON Schema, to improve safety of ML pipelines [28].

Schema inclusion also plays a central role in schema evolution, with questions of the kind: will a value that respects the new schema still be accepted by tools designed for legacy versions? If not, what is an example of a problematic value?

Equivalence $S \equiv S'$: does, for each value J, $J \vDash S \Leftrightarrow J \vDash S'$? Checking equivalence builds upon inclusion, and is relevant in designing workbenches for schema analysis and simplification [24].

Satisfiability of S: does a value J exist such that $J \vDash S$?

Note that the above problems are strictly interrelated. Indeed, as JSON Schema includes the Boolean algebra, schema inclusion and satisfiability are equivalent: $S \subseteq S'$ if and only if $S \wedge \neg S'$ is not satisfiable, and S is satisfiable if and only if $S \not\subseteq$ `false`, where `false` is the schema that no JSON document can match.

Witness generation for S, a constructive generalization of satisfiability: given S, generate a value J such that $J \vDash S$, or return "unsatisfiable" if no such value exists. In the first case, we call J a *witness*. Schema inclusion $S \subseteq S'$ can be immediately reduced to witness generation for $S \wedge \neg S'$, but with a crucial advantage: if a witness J for $S \wedge \neg S'$ is generated, we can provide users with an explanation: S is not included in S' *because* of values such as J. We can similarly solve a "witnessed" version of equivalence: given S and S', either prove that one is equivalent to the other, or provide an explicit witness J that belongs to one, but not to the other.

A witness generation algorithm, besides its use for the solution of witnessed inclusion, is the first step in the design of *complete enumeration* and *example generation* algorithms. Here, *complete enumeration* is any algorithm, in general non-terminating, that, for a given S, enumerates every J that satisfies S. With *example generation*, we indicate any enumeration algorithm that is not necessarily complete, but pursues some "practical" criterion in the choice of the generated witnesses, such as the "realism" of the base values, or some form of coverage of the different cases allowed by the schema. Example generation is extremely useful in the context of test-case generation, and also as a tool to understand complex schemas through realistic examples.

Open challenges. Witness generation for JSON Schema is difficult. Existing tools are incomplete and struggle with this task (as we will show in our experiments). First of all, JSON Schema includes conjunction, disjunction, negation, modal (or *structural*) operators, recursive second-order variables, and recursion under negation. Secondly, for each JSON type, the different structural operators have complex interactions, as in the following example, where "required" and the negated "patternProperties" force the presence of fields whose names match "^a" and "^abz$" (this is explained in the paper), "maxProperties" : 1 forces these two fields to be one, and, finally, "patternProperties" forces the value of that field to satisfy *var2*, since "abz" also matches "z$".

```
{"required":["abz"],
 "not":{"patternProperties":{"^a":{"$ref":"#/$defs/var1"}}},
 "maxProperties":1,
 "patternProperties":{"z$":{"$ref":"#/$defs/var2"}},
 "$defs" : ...
}
```

Each aspect would make the problem computationally intractable by itself. Their combination exacerbates the difficulty of the design of a *complete* algorithm that is *practical*, that is, of an algorithm that is correct and complete by design, but is also able to run in a reasonable time over the vast majority of real-world schemas.

Contributions. The main contribution of this paper is an original sound and complete algorithm for checking the satisfiability of an input schema S, generating a witness J when the schema is satisfiable. Our algorithm supports the whole language without uniqueItems. While the existence of an algorithm for this specific problem follows from the results in [18], where the problem is proved to be EXPTIME-complete, we are the first to explicitly describe an algorithm, and specifically one that has the potential to work in reasonable time over schemas of realistic size. Our algorithm is based on a set of formal manipulations of the schema, some of which, such as *preparation*, are unique to JSON Schema, and have not been proposed before in this form. Particularly relevant in this context is the notion of *lazy and-completion*, which we will describe later. In this paper, we detail each algorithm phase, show that each is in $O(2^{poly(N)})$, and focus on preparation and generation of objects and arrays, the phases completely original to this work.

The practical applicability of our algorithm is proved by our experimentation, which is another contribution of this work. Our experiments are based on four real-world datasets, on a synthetic dataset, and on a handwritten dataset. Real-world datasets comprise 6,427 unique schemas extracted, through an extensive data cleaning process, from a large corpus of schemas crawled from GitHub [14] and curated by us for errors and redundancies; the other datasets, already used in [28], are related to specific application domains and originated from Snowplow[5], The Washington Post [36], and Kubernetes [30]. The synthetic dataset is synthesized from the standard schemas provided by *JSON Schema Org* [34], from which we derive schemas that are known to be satisfiable or unsatisfiable by design [7]. The handwritten dataset is specifically engineered to test the most complex aspects of the JSON Schema language. The experiments show that our algorithm is complete, and that, despite its exponential complexity, it behaves quite well even on schemas with tens of thousands of nodes. Overall, we can show that our contributions advance the state-of-the-art.

Our implementation of the witness generation algorithm is available as open source. The code is part of a fully automated reproduction package [4], which contains all input data, as well as the data generated in our experiments. For convenience, our implementation is also accessible as an interactive web-based tool [3].

Paper outline

The rest of the paper is organized as follows. In Section 2 we analyze related work. In Section 3 we briefly describe JSON and JSON Schema. In Sections 4 and 5 we introduce our algebraic framework. In Sections 6, 7, and 8, we describe the structure of the algorithm, the initial phases, and the last phases. In Section 9 we present an extensive experimental evaluation of our approach. In Section 10, we draw our conclusions.

2 RELATED WORK

Overviews over schema languages for JSON can be found in [10, 11, 18, 35]. Pezoa et al. [35] introduced the first formalization of JSON Schema and showed that it cannot be captured by MSO or tree automata because of the uniqueItems constraints. While they focused on validation and proved that it can be decided in $O(|J|^2|S|)$ time, they also showed that JSON Schema can simulate tree automata. Hence, schema satisfiability is EXPTIME-hard.

In [18] Bourhis et al. refined the analysis of Pezoa et al. They mapped JSON Schema onto an equivalent modal logic, called recursive JSL, and proved that satisfiability is PSPACE-complete for schemas without recursion and uniqueItems, it is in EXPSPACE for non recursive schemas with uniqueItems, it is EXPTIME-complete for recursive schemas without uniqueItems, and it is in 2EXPTIME for recursive schemas with uniqueItems. Their work is extremely important in establishing complexity bounds. Since they map JSON Schema onto recursive JSL logic, and provide a specific kind of alternating tree automata for this logic, they already provide an indirect indication of an algorithm for witness generation. However, classical reachability algorithms for alternating automata are designed to prove complexity upper bounds, not as practical tools. They are typically based on the exploration of all subsets of the state set of the automaton [20], hence on a sequence of complex operations on a set of sets whose dimension may be in the realm of $2^{10,000}$. While exponentiality cannot be avoided in the worst case, it is clear that we need a different approach when designing a practical algorithm.

To the best of our knowledge, the only tool that is currently available to check the satisfiability of a schema is the containment checker described by Habib et al. [28]. While it has been designed for schema containment checking, e.g., $S_1 \subseteq S_2$, it can also be exploited for schema satisfiability since S is satisfiable if and only if $S \nsubseteq S'$, where S' is an empty schema. The approach of Habib et al. bears some resemblances to ours, e.g., schema canonicalization has been first presented there, but its ability to cope with negation is very limited as well as its support for recursion.

Several tools (see [17] and [1]) for example generation exist. They generate JSON data starting from a schema. These tools, however, are based on a trial-and-error approach and cannot detect unsatisfiable schemas. We compare our tool with [17] in our experiments. There are also grammar-based approaches for generating JSON values. The tool by Gopinath et al. allows for data generation under Boolean constraints [27], which have to be specified *manually.*

In [22], Benac Earle et al. present a systematic approach to testing behavioral aspects of Web Services that communicate using JSON data. In particular, this approach builds a finite state machine capturing the schema describing the exchanged data, but this machine is only used for generating data and is restricted to atomic values, objects and to some form of boolean expressions.

Own prior work. In our technical report [15], we discuss negation-completeness for JSON Schema, that is, we show how pairs of JSON Schema operators such as "patternProperties"-"required" and "items"-"contains" are *almost* dual under negation, as $\wedge$-$\vee$ or $\forall$-$\exists$ are, but not exactly. In the process, we define an algorithm for not-elimination, that we actually developed for its use in the witness generation algorithm that we describe here. In Section 7.2 we will rapidly recap this algorithm.

An earlier prototype implementation has been presented in tool demos [8, 9, 24]. Meanwhile, we have optimized our algorithm, and formalized the proofs, as presented in this paper.

A preliminary version of the algorithm described in the current paper has been presented in [12] (informal proceedings).

3 PRELIMINARIES

3.1 JSON data model

Each JSON value belongs to one of the six JSON Schema types: nulls, Booleans, decimal numbers Num (hereafter, we just use *numbers* to refer to decimal numbers), strings Str, objects, arrays. Objects represent sets of members, each member being a name-value pair, where no name can be present twice, and arrays represent ordered sequences of values.

J ::=	$B \mid O \mid A$		**JSON expressions**
B ::=	null $\mid$ true $\mid$ false $\mid q \mid s$		
		$q \in \mathsf{Num}, s \in \mathsf{Str}$	**Basic values**
O ::=	$\{l_1 : J_1, \ldots, l_n : J_n\}$		
		$n \geq 0, \quad i \neq j \Rightarrow l_i \neq l_j$	**Objects**
A ::=	$[J_1, \ldots, J_n]$	$n \geq 0$	**Arrays**

Definition 1 (Value equality and sets of values). We interpret a JSON object $\{l_1 : J_1, \ldots, l_n : J_n\}$ as a *set* of pairs (*members*) $\{(l_1, J_1), \ldots, (l_n, J_n)\}$, where $i \neq j \Rightarrow l_i \neq l_j$, and an array $[J_1, \ldots, J_n]$ as an ordered list; JSON value equality is defined accordingly, that is, by ignoring member order when comparing objects.

Sets of JSON values are defined as collections with no repetition with respect to this notion of equality.

3.2 JSON Schema

JSON Schema is a language for defining the structure of JSON documents. Many versions have been defined for this language, notably Draft-03 of November 2010, Draft-04 of February 2013 [25], Draft-06 of April 2017 [41], Draft 2019-09 of September 2019 [39], and Draft 2020-12 of December 2020 [40]. Draft 2019-09 introduced a major semantic shift, since it made assertion validation dependent on annotations, and has not been amply adopted up to now, hence we decided to base our work on Draft-06. However, we decided also to include the operators "minContains" and "maxContains" introduced with Draft 2019-09 since they are very interesting in the context of witness generation and they do not present the problematic dependency on annotations of the other novel operators.

JSON Schema uses JSON syntax. A schema is a JSON object that collects *assertions* that are members, i.e., name-value pairs, where the name indicates the assertion and the value collects its parameters, as in "minLength" : 3, where the value is a number, or in "items" : {"type" : ["boolean"]}, where the value for "items" is an object that is itself a schema, and the value for "type" is an array of strings.

A JSON Schema document (or *schema*) denotes a set of JSON documents (or *values*) that satisfy it. The language offers the following abilities.

- Base type specification: it is possible to define complex properties of collections of base type values, such as all strings that satisfy a given regular expressions ("pattern"), all numbers that are multiple of a given numbers ("multipleOf") and included in a given interval ("minimum", "maximum",...).
- Array specification: it is possible to specify the types of the elements for both uniform arrays and non-uniform arrays ("items"), to restrict the minimum and maximum size of the array, to bound the number of elements that satisfy a given property ("contains", "minContains", ...), and also to enforce uniqueness of the items ("uniqueItems").
- Object specification: it is possible to require for certain names to be present or to be absent, to specify the schemas of both optional or mandatory members, all of this by denoting classes of names using regular expressions (via "properties", "patternProperties" and "required"). It it possible to specify that some assertions depend on the presence of some members ("dependencies"), and it is possible to limit the number of members that are present.
- Boolean combination: one can express union, intersection, and complement of schemas ("anyOf", "allOf", "not"), and also a generalized form of mutual exclusion ("oneOf").
- Mutual recursion: mutually recursive schema variables can be defined ("definitions", "$ref").

In the next section we describe JSON Schema by giving its translation into a simpler algebra.

4 THE ALGEBRA

4.1 The *core* and the *positive* algebras

In JSON Schema, the meaning of some assertions is modified by the surrounding assertions, making formal manipulation much more difficult. Moreover, the language is rich in redundant operators, such as "if" – "then" – "else" and "dependencies", which can both be easily translated in terms of "not" and "anyOf".

$m \in \mathsf{Num}^{-\infty}, M \in \mathsf{Num}^{\infty}, l \in \mathbb{N}_{>0}, i \in \mathbb{N}, j \in \mathbb{N}^{\infty}, q \in \mathsf{Num}, k \in \mathsf{Str}$

$$
\begin{array}{lcl}
T & ::= & \mathsf{Arr} \mid \mathsf{Obj} \mid \mathsf{Null} \mid \mathsf{Bool} \mid \mathsf{Str} \mid \mathsf{Num} \\
r & ::= & \text{Any regular expression} \mid \overline{r} \mid r_1 \sqcap r_2 \\
b & ::= & \mathtt{true} \mid \mathtt{false} \\
S & ::= & \mathsf{ifBoolThen}(b) \mid \mathsf{pattern}(r) \mid \mathsf{betw}_m^M \mid \mathsf{xBetw}_m^M \\
 & & \mid \mathsf{mulOf}(q) \mid \mathsf{props}(r : S) \mid \mathsf{req}(k) \mid \mathsf{pro}_i^j \\
 & & \mid \mathsf{item}(l : S) \mid \mathsf{items}(i^+ : S) \mid \mathsf{cont}_i^j(S) \\
 & & \mid \mathsf{type}(T) \mid x \mid S_1 \wedge S_2 \mid S_1 \vee S_2 \\
\text{core:} & & \mid \neg S \\
\text{positive:} & & \mid \mathsf{notMulOf}(q) \mid \mathsf{pattReq}(r : S) \mid \mathsf{contAfter}(i^+ : S) \\
E & ::= & x_1 : S_1, \ldots, x_n : S_n \\
D & ::= & S \text{ defs } (E)
\end{array}
$$

Figure 1: Syntax of the *core* and *positive* algebras.

For these reasons, in our implementation, we translate JSON Schema onto a *core algebra*, that is an algebraic version of JSON Schema with less redundant operators.

This algebra is very similar (apart the syntax) to the recursive JSL logic defined in [18], but has a different aim. While JSL is an elegant and minimal logic upon which JSON Schema is translated, and an excellent tool for theoretical research, our algebra is an implementation tool with two aims:

(1) simplify the implementation by its algebraic nature and its reduced size;
(2) simplify the formal discussion of the implementation.

Both aims are facilitated by the algebraic nature and the reduced size of the algebra, but we also value a certain degree of adherence to JSON Schema.

The first step of our approach is the translation of an input schema into an algebraic representation, and the second step is not-elimination (Section 7.2). For the first step we use a *core algebra* that is defined by a subset of JSON Schema operators. For not-elimination, we use a *positive algebra* where we remove negation but we add three new operators: $\mathsf{notMulOf}(n)$, $\mathsf{pattReq}(r : S)$, and $\mathsf{contAfter}(i^+ : S)$. Our algebras extend JSON Schema regular expressions with external intersection $\sqcap$ and complement $\overline{r}$ operators; this extension is discussed in Section 4.4. The syntax of the two algebras, *core* and *positive*, which are expressive enough to capture all JSON Schema assertions of Draft-06, plus the extra operators `"minContains"` and `"maxContains"` of Draft 2019-09, is presented in Figure 1.

In $\mathsf{mulOf}(q)$, q is a number. In betw_m^M and in xBetw_m^M, m is either a number or $-\infty$, M is either a number or ∞. In pro_i^j, in $\mathsf{items}(i^+ : S)$, in $\mathsf{cont}_i^j(S)$, and in $\mathsf{contAfter}(i^+ : S)$, i is an integer with $i \geq 0$, and j is either an integer with $j \geq 0$, or ∞, while in $\mathsf{item}(l : S)$, l is an integer with $l \geq 1$, and k in $\mathsf{req}(k)$ is a string.

We distinguish Boolean operators ($\wedge$, $\vee$ and $\neg$), variables (x), and *Typed Operators* (TO — all the others). All TOs different from $\mathsf{type}(T)$ have an implicative semantics: "if the instance belongs to the type T then ...", so that they are trivially satisfied by every instance not belonging to type T. We say that they are *implicative typed operators* (ITOs).

The operators of the core algebra strictly correspond to those of JSON Schema, and in particular to their implicative semantics. The exact relationship between core algebra and JSON Schema is discussed in Section 5.

Informally, an instance J of the core or positive algebra satisfies an assertion S if:

- $\mathsf{ifBoolThen}(b)$: *if* the instance J is a boolean, *then* $J = b$.
- $\mathsf{pattern}(r)$: *if* J is a string, *then* J matches r.
- betw_m^M: *if* J is a number, *then* $m \leq J \leq M$. xBetw_m^M is the same with extreme excluded.
- $\mathsf{mulOf}(q)$: *if* J is a number, *then* $J = q \times i$ for some integer i. q is any number, i.e., any *decimal* number (Section 3.1).
- $\mathsf{props}(r : S)$ *if* J is an object and *if* (k, J') is a member of J where k matches the pattern r, *then* J' satisfies S. Hence, it is satisfied by any instance that is not an object and also by any object where no member name matches r.
- $\mathsf{req}(k)$: *if* J is an object, *then* it contains at least one member whose name is k.
- pro_i^j: *if* J is an object, *then* it has between i and j members.
- $\mathsf{item}(l : S)$: *if* J is an array $[J_1, \ldots, J_n]$ $(n \geq 0)$ and *if* $l \leq n$, *then* J_l satisfies S. Hence, it is satisfied by any J that is not an array and also by any array that is strictly shorter than l, such as the empty array: it does not force the position l to be actually used.
- $\mathsf{items}(i^+ : S)$: *if* J is an array $[J_1, \ldots, J_n]$, *then* J_l satisfies S for every $l > i$. Hence, it is satisfied by any J that is not an array and by any array shorter than i.
- $\mathsf{cont}_i^j(S)$: *if* J is an array, *then* the total number of elements that satisfy S is included between i and j.
- $\mathsf{type}(T)$ is satisfied by any instance belonging to the predefined JSON type T (Str, Num, Bool, Obj, Arr, and Null).
- x is equivalent to its definition in the environment E associated with the expression.
- $S_1 \wedge S_2$: both S_1 and S_2 are satisfied.
- $S_1 \vee S_2$: either S_1, or S_2, or both, are satisfied.
- $\neg S$: S is not satisfied.
- $\mathsf{notMulOf}(n)$: *if* J is a number, *then* is not a multiple of n.
- $\mathsf{pattReq}(r : S)$: *if* J is an object, *then* it contains at least one member (k, J) where k matches r and J satisfies S
- $\mathsf{contAfter}(i^+ : S)$: *if* J is an array $[J_1, \ldots, J_n]$, *then* it contains at least one element J_j with $j > i$ that satisfies S.
- An environment $E = x_1 : S_1, \ldots, x_n : S_n$ defines n mutually recursive variables, so that x_i can be used as an alias for S_i inside any of $S_1, \ldots, S_n$.
- $D = S$ defs $(x_1 : S_1, \ldots, x_n : S_n)$: J satisfies S when every x_i is interpreted as an alias for the corresponding S_i.

Variables in $E = x_1 : S_1, \ldots, x_n : S_n$ are mutually recursive, but we require recursion to be *guarded*. Let us say that x_i *directly depends* on x_j if some occurrence of x_j appears in the definition of x_i without being in the scope of an ITO. For example, in "x : $(\mathsf{props}(r : y) \wedge z)$", x directly depends on z, but not on y. Recursion is not guarded if the transitive closure of the relation "directly depends on" contains a reflexive pair (x, x). Informally, recursion is guarded iff every cyclic chain of dependencies traverses an ITO.

Hereafter we will often use the derived operators $\mathbf{t}$ and $\mathbf{f}$. $\mathbf{t}$ stands for "always satisfied" and can be expressed, for example, as pro_0^{∞}, which is satisfied by any instance. **f** stands for "never satisfied" and can be expressed, for example, as $\neg\mathbf{t}$.

4.2 Semantics of the core algebra

The semantics of a schema S with respect to an environment E is the set of JSON instances $[\![S]\!]_E$ that *satisfy* that schema, as specified in Figure 2. Hereafter, $E(x)$ indicates the schema that E associates to x. $L(r)$ denotes the regular language generated by r. For T in Null, Bool, Str, Num, Obj, Arr, $\mathcal{J}Val(T)$ is the set of JSON values of that type, and $\mathcal{J}Val(*)$ is the set of all JSON values. $\mathbb{Z}$ is the set of all integers. Universal quantification on an empty set is true, and the set $\{1..0\}$ is empty.

The definition can be read as follows (ignoring the index p for a moment): the semantics of $\mathsf{props}(r : S)$ specifies that $J \in [\![\mathsf{props}(r : S)]\!]_E \Leftrightarrow$ *if* J is an object, *if* $(k_i : J_i)$ is a member where k_i matches r, *then* $J_i \in [\![S]\!]_E$, as informally specified in the previous section.

$$
\begin{array}{lcl}
[\![\mathsf{ifBoolThen}(b)]\!]^p_E & = & \{\!|\, J \mid J \in \mathcal{J}Val(\mathsf{Bool}) \Rightarrow J = b \,|\!\} \\
[\![\mathsf{pattern}(r)]\!]^p_E & = & \{\!|\, J \mid J \in \mathcal{J}Val(\mathsf{Str}) \Rightarrow J \in L(r) \,|\!\} \\
[\![\mathsf{betw}^M_m]\!]^p_E & = & \{\!|\, J \mid J \in \mathcal{J}Val(\mathsf{Num}) \Rightarrow m \leq J \leq M \,|\!\} \\
[\![\mathsf{xBetw}^M_m]\!]^p_E & = & \{\!|\, J \mid J \in \mathcal{J}Val(\mathsf{Num}) \Rightarrow m < J < M \,|\!\} \\
[\![\mathsf{mulOf}(q)]\!]^p_E & = & \{\!|\, J \mid J \in \mathcal{J}Val(\mathsf{Num}) \Rightarrow \\
 & & \exists i \in \mathbb{Z}.\ J = i \cdot q \,|\!\} \\
[\![\mathsf{props}(r : S)]\!]^p_E & = & \{\!|\, J \mid J = \{(k_1 : J_1), \ldots, (k_n : J_n)\} \Rightarrow \\
 & & \forall i \in \{1..n\}.\ k_i \in L(r) \ \Rightarrow J_i \in [\![S]\!]^p_E \,|\!\} \\
[\![\mathsf{req}(k)]\!]^p_E & = & \{\!|\, J \mid J = \{(k_1 : J_1), \ldots, (k_n : J_n)\} \Rightarrow \\
 & & \exists i \in \{1..n\}.\ k_i = k \,|\!\} \\
[\![\mathsf{pro}^j_i]\!]^p_E & = & \{\!|\, J \mid J = \{(k_1 : J_1), \ldots, (k_n : J_n)\} \Rightarrow \\
 & & i \leq n \leq j \,|\!\} \\
[\![\mathsf{item}(l : S)]\!]^p_E & = & \{\!|\, J \mid J = [J_1, \ldots, J_n] \Rightarrow \\
 & & n \geq l \Rightarrow J_l \in [\![S]\!]^p_E \,|\!\} \\
[\![\mathsf{items}(i^+ : S)]\!]^p_E & = & \{\!|\, J \mid J = [J_1, \ldots, J_n] \Rightarrow \\
 & & \forall j \in \{1..n\}.\ j > i \Rightarrow J_j \in [\![S]\!]^p_E \,|\!\} \\
[\![\mathsf{cont}^j_i(S)]\!]^p_E & = & \{\!|\, J \mid J = [J_1, \ldots, J_n] \Rightarrow \\
 & & i \leq\ |\{\!|\, l \mid J_l \in [\![S]\!]^p_E \,|\!\}|\ \leq j \,|\!\} \\
[\![\mathsf{type}(T)]\!]^p_E & = & \mathcal{J}Val(T) \\
[\![S_1 \wedge S_2]\!]^p_E & = & [\![S_1]\!]^p_E \cap [\![S_2]\!]^p_E \\
[\![S_1 \vee S_2]\!]^p_E & = & [\![S_1]\!]^p_E \cup [\![S_2]\!]^p_E \\
[\![\neg S]\!]^p_E & = & \mathcal{J}Val(*) \setminus [\![S]\!]^p_E \\
[\![x]\!]^0_E & = & \emptyset \\
[\![x]\!]^{p+1}_E & = & [\![E(x)]\!]^p_E \\
[\![S]\!]_E & = & \bigcup_{i \in \mathbb{N}} \bigcap_{p \geq i} [\![S]\!]^p_E \\
[\![S \ \mathsf{defs}\ (E)]\!] & = & [\![S]\!]_E
\end{array}
$$

Figure 2: Semantics of the algebra with explicit negation.

The index p is used since otherwise the definition $[\![x]\!]_E = [\![E(x)]\!]_E$ would not be inductive: $E(x)$ is in general bigger than x, while the use of the index makes the entire definition inductive on the lexicographic pair $(p, |S|)$. However, we need to define an appropriate notion of limit for the sequence $[\![S]\!]^p_E$. We cannot just set $[\![S]\!]_E = \bigcup_{p \in \mathbb{N}} [\![S]\!]^p_E$, since, because of negation, this sequence of interpretations is not necessarily monotonic in p. For example, if we have a definition $y : \neg(x)$, then $[\![y]\!]^0_E$ contains the entire $\mathcal{J}Val(*)$. However, since the interpretation converges when p grows, we can extract an exists-forall limit from it, by stipulating that an instance J belongs to the limit $[\![S]\!]_E$ if an i exists such that J belongs to every interpretation that comes after i:

$$[\![S]\!]_E = \bigcup_{i \in \mathbb{N}} \bigcap_{j \geq i} [\![S]\!]^j_E$$

Now, it is easy to prove that this interpretation satisfies JSON Schema specifications, since, for guarded schemas, it enjoys the properties expressed in Theorem 4, stated below.

Definition 2. An environment $E = x_1 : S_1, \ldots, x_n : S_n$ is *guarded* if recursion is guarded in E. An environment $E = x_1 : S_1, \ldots, x_n : S_n$ is *closing* for S if all variables in $S_1, \ldots, S_n$ and in S are included in $x_1, \ldots, x_n$.

Lemma 3 (Convergence). *There exists a function I that maps every triple J, S, E, where E is guarded and closing for S, to an integer $i = I(J, S, E)$ such that:*

$$(\forall j \geq i.\ J \in [\![S]\!]^j_E) \vee (\forall j \geq i.\ J \notin [\![S]\!]^j_E)$$

Proof. For any guarded E, we can define a function d_E from assertions to natural numbers such that, when x directly depends on y, then $d_E(x) > d_E(y)$. Specifically, we define the degree $d_E(S)$ of a schema S in E as follows. If S is a variable x, then $d_E(x) = d_E(E(x)) + 1$. If S is not a variable, then $d_E(S)$ is the maximum degree of all unguarded variables in S and, if it contains no unguarded variable, then $d_E(S) = 0$. This definition is well-founded thanks to the guardedness condition. We now define a function $I(J, S, E)$ with the desired property by induction on $(J, d_E(S), S)$, in this order of significance.

(i) Let $S = x$. We prove that $I(J, x, E) = I(J, E(x), E) + 1$ has the desired property. We want to prove that

$(\forall j \geq I(J, E(x), E) + 1.\ J \in [\![x]\!]^j_E) \vee (\forall j \geq I(J, E(x), E) + 1.\ J \notin [\![x]\!]^j_E)$

We rewrite $[\![x]\!]^j_E$ as $[\![E(x)]\!]^{j-1}_E$:

$(\forall j \geq I(J, E(x), E)+1.\ J \in [\![E(x)]\!]^{j-1}_E) \vee (\forall j \geq I(J, E(x), E)+1.\ J \notin [\![E(x)]\!]^{j-1}_E)$

i.e., $(\forall j \geq I(J, E(x), E).\ J \in [\![E(x)]\!]^j_E) \vee (\forall j \geq I(J, E(x), E).\ J \notin [\![E(x)]\!]^j_E)$

This last statement holds by induction, since $d_E(x) = d_E(E(x)) + 1$, hence the term J is the same but the degree of $E(x)$ is strictly smaller than that of x.

(ii) Let $S = \neg S'$. We prove that $I(J, \neg S', E)$ defined as $I(J, S', E)$ has the desired property. We want to prove that, for any J:

$(\forall j \geq I(J, S', E).\ J \in [\![\neg S']\!]^j_E) \vee (\forall j \geq I(J, S', E).\ J \notin [\![\neg S']\!]^j_E)$

By definition of $[\![\neg S']\!]^j_E$, we need to prove that for any J:

$(\forall j \geq I(J, S', E).\ J \notin [\![S']\!]^j_E) \vee (\forall j \geq I(J, S', E).\ J \in [\![S']\!]^j_E)$

which holds by induction on S, since the term J is the same and the degree is equal.

(iii) Let $S = S' \wedge S''$. In this case, we let
$I(J, S' \wedge S'', E) = max(I(J, S', E), I(J, S'', E))$. We want to prove that:

$(\forall j \geq max(I(J, S', E), I(J, S'', E)).\ J \in [[S' \wedge S'']]^j_E)$
$\vee(\forall j \geq max(I(J, S', E), I(J, S'', E)).\ J \notin [[S' \wedge S'']]^j_E)$

This follows immediately from the following two properties, that hold by induction on $(J, d_E(S), S)$, since both S_1 and S_2 have a degree less or equal to S, and are strict subterms of S:

$(\forall j \geq I(J, S', E).\ J \in [[S']]^j_E) \vee (\forall j \geq I(J, S', E).\ J \notin [[S']]^j_E)$

$(\forall j \geq I(J, S'', E).\ J \in [[S'']]^j_E) \vee (\forall j \geq I(J, S'', E).\ J \notin [[S'']]^j_E)$

The same proof holds for the case $S = S' \vee S''$.

(iv) Let $S = \text{items}(n^+ : S')$. If J is not an array, then we can take $I(J, S, E) = 0$, since J satisfies S for any index. If $J = [J_1, \ldots, J_m]$, then we fix

$$I([J_1, \ldots, J_m], S, E) = max_{i \in \{1..m\}} I(J_i, S', E) \qquad (*)$$

which is well defined by induction, since every J_i is a strict subterm of J. Observe that the fact that each J_j is *strictly* smaller than J, and not just less-or-equal, is essential since, in general, the degree of S' may be bigger than the degree of S, since S' is in a guarded position inside S. Consider the semantics of $\text{items}(n^+ : S')$:

$\{\!| J \mid J = [J_1, \ldots, J_m] \Rightarrow \forall l \in \{1..m\}.\ l > n \Rightarrow J_l \in [[S']]^p_E |\!\}$.

Now, because of $(*)$, $\forall j \geq I(J, S, E)$, either $J_l \in [[S']]^j_E$ or $J_l \notin [[S']]^j_E$, hence $(\forall j \geq I(J, S, E).\ J \in [[\text{items}(n^+ : S')]]^j_E) \vee (\forall j \geq I(J, S, E).\ J \notin [[\text{items}(n^+ : S')]]^j_E)$

Informally, for any l and for any $j \geq max_{i \in \{1..m\}} I(J_i, S', E)$, the question "does J' belong to $J_l \in [[S']]^j_E$" has a fixed answer, hence the question "does J belong to $\text{items}(n^+ : S')$" has a fixed answer as well.

All other TOs can be treated in the same way.

□

Theorem 4. *For any E guarded, the following equality holds:*

$$[[E(x)]]_E = [[x]]_E$$

Moreover, for each equivalence in Figure 2, the equivalence still holds if we substitute every occurrence of $[[S]]^p_E$ with $[[S]]_E$, obtaining for example:

$$[[\text{item}(l : S)]]_E \ = \ \{\!| J \mid J = [J_1, \ldots, J_n] \Rightarrow n \geq l \Rightarrow J_l \in [[S]]_E |\!\}$$

from

$$[[\text{item}(l : S)]]^p_E \ = \ \{\!| J \mid J = [J_1, \ldots, J_n] \Rightarrow \ n \geq l \Rightarrow J_l \in [[S]]^p_E |\!\}$$

Proof. This is an immediate consequence of convergence. Consider any equation such as:

$$[[\text{item}(l : S)]]^p_E \ = \ \{\!| J \mid J = [J_1, \ldots, J_n] \Rightarrow n \geq l \Rightarrow J_l \in [[S]]^p_E |\!\}$$

That is:

$$J \in [[\text{item}(l : S)]]^p_E \ \Leftrightarrow \ (J = [J_1, \ldots, J_n] \Rightarrow n \geq l \Rightarrow J_l \in [[S]]^p_E)$$

If we consider any integer I that is bigger than $I(J, \text{item}(l : S), E)$ and of every $I(J_l, S, E)$, then, if the equation holds for one index $p \geq I$, then it holds for every such index, hence it holds for the limit. This is the general idea, and we now present a more formal proof.

We first prove that:

$$\bigcup_{i \in \mathbb{N}} \bigcap_{j \geq i} [[x]]^j_E = \bigcup_{i \in \mathbb{N}} \bigcap_{j \geq i} [[E(x)]]^j_E$$

Assume that $J \in \bigcup_{i \in \mathbb{N}} \bigcap_{j \geq i} [[x]]^j_E$. Then,
$\exists i. \forall j \geq i. J \in [[x]]^j_E$. Let I be one i with that property. We have that
$\forall j \geq I. J \in [[x]]^j_E$, i.e.,
$\forall j \geq I. J \in [[E(x)]]^{j-1}_E$, which implies that
$\forall j \geq I. J \in [[E(x)]]^j_E$, hence
$\exists i. \forall j \geq i. J \in [[E(x)]]^j_E$.
In the other direction, assume $J \in \bigcup_{i \in \mathbb{N}} \bigcap_{j \geq i} [[E(x)]]^j_E$. Hence,
$\exists i. \forall j \geq i. J \in [[E(x)]]^j_E$. Let I be one i with that property. We have that
$\forall j \geq I. J \in [[E(x)]]^j_E$, i.e.,
$\forall j \geq I. J \in [[x]]^{j+1}_E$, i.e.,
$\forall j \geq (I+1). J \in [[x]]^j_E$, i.e.,
$\exists i. \forall j \geq i. J \in [[x]]^j_E$.

For the second property, the crucial case is that for $J \in [[\neg S]]_E$, where we want to prove:

$$J \in [[\neg S]]_E \ \Leftrightarrow J \notin [[S]]_E$$

.

$J \in [[\neg S]]_E \ \Leftrightarrow$

$\exists i. \forall j \geq i.\ J \in [[\neg S]]^j_E \ \Leftrightarrow$

$\exists i. \forall j \geq i.\ J \notin [[S]]^j_E \ \Leftrightarrow (***)$

$\forall i. \exists j \geq i.\ J \notin [[S]]^j_E \ \Leftrightarrow$

$\neg(\exists i. \forall j \geq i.\ J \in [[S]]^j_E) \ \Leftrightarrow J \notin [[S]]_E$

For the crucial $\Leftrightarrow$ $(***)$ step, the direction $\Rightarrow$ is immediate. For the direction $\Leftarrow$ we use the convergence Lemma 3: if we assume that $\forall i. \exists j \geq i.\ J \notin [[S]]^j_E$, then, by considering the case $i = I(J, S, E)$, we have that $\exists j \geq I(J, S, E).\ J \notin [[S]]^j_E$, hence, by Lemma 3, $\forall j \geq I(J, S, E).\ J \notin [[S]]^j_E$, hence $\exists i. \forall j \geq i.\ J \notin [[S]]^j_E$.

All other cases follow easily from convergence. Consider for example the case where $J \in [[\text{cont}^M_m(S')]]_E$. We want to prove:

$$J \in [[\text{cont}^M_m(S')]]_E$$
$$\Leftrightarrow (J = [J_1, \ldots, J_n] \Rightarrow m \leq \ |\{\!| l \mid J_l \in [[S']]_E |\!\}| \ \leq M)$$

If J is not an array, the double implication holds trivially. Consider now the case $J = [J_1, \ldots, J_n]$:

$J \in [[\text{cont}^M_m(S')]]_E \ \Leftrightarrow$

$\exists i. \forall j \geq i.\ J \in [[\text{cont}^M_m(S')]]^j_E \ \Leftrightarrow$

$\exists i. \forall j \geq i.\ m \leq \ |\{\!| l \mid J_l \in [[S']]^j_E |\!\}| \ \leq M \Leftrightarrow$

Here, we choose an I that is greater than $I(J, \text{cont}^M_m(S'), E)$ and is greater than $I(J_l, S', E)$ for every J_l (from the proof of Lemma 3 we know that $I(J, \text{cont}^M_m(S'), E)$ as defined in that proof would do the work):

$$\exists i. \forall j \geq i.\ m \leq\ |\{\!|\ l\ |\ J_l \in [[S']]^j_E\ |\!\}|\ \leq M \Leftrightarrow$$
$$\forall j \geq I.\ m \leq\ |\{\!|\ l\ |\ J_l \in [[S']]^j_E\ |\!\}|\ \leq M \Leftrightarrow$$
$$m \leq\ |\{\!|\ l\ |\ \forall j \geq I.\ J_l \in [[S']]^j_E\ |\!\}|\ \leq M \Leftrightarrow$$
$$m \leq\ |\{\!|\ l\ |\ J_l \in [[S']]_E\ |\!\}|\ \leq M$$

□

The official JSON Schema semantics specifies that x is the same as $E(x)$ for all schemas where such interpretation never creates a loop (i.e., for all guarded schemas) and describes, verbally, the equations that we wrote in the form without the index. Hence, Theorem 4 proves that our semantics exactly captures the official JSON Schema semantics (provided that we wrote the correct equations).

4.3 Semantics of the three extra operators of the positive algebra

The three operators added in the positive algebra are redundant in presence of negation. They do not correspond to JSON Schema operators, but can still be expressed in JSON Schema, through the negation of `"multipleOf"`, `"patternProperties"`, and `"additionalItems"`. The semantics of these operators can be easily expressed in the core algebra with negation, as shown in Figure 3; hereafter, we use $S_1 \Rightarrow S_2$ as an abbreviation for $\neg S_1 \vee S_2$:

$$\begin{array}{lcl} \mathsf{notMulOf}(n) & = & \mathsf{type}(\mathsf{Num}) \Rightarrow \neg\mathsf{mulOf}(n) \\ \mathsf{pattReq}(r : S) & = & \mathsf{type}(\mathsf{Obj}) \Rightarrow \neg\mathsf{props}(r : \neg S) \\ \mathsf{contAfter}(i^+ : S) & = & \mathsf{type}(\mathsf{Arr}) \Rightarrow \neg\mathsf{items}(i^+ : \neg S) \end{array}$$

Figure 3: Semantics of additional operators.

Observe that the semantics of the additional operators is implicative, as for all the others ITOs.

The definition of $\mathsf{pattReq}(r : S)$ deserves an explanation. The implication $\mathsf{type}(\mathsf{Obj}) \Rightarrow \ldots$ just describes its implicative nature – it is satisfied by any instance that is not an object. Since $r : \neg S$ means that, if a name matching r is present, then its value satisfies $\neg S$, any instance that does not satisfy $r : \neg S$ must possess a member name that matches r and whose value does not satisfy $\neg S$, that is, satisfies S. Hence, we exploit here the fact that the negation of an implication forces the hypothesis to hold.

4.4 About regular expressions

4.4.1 Undecidability of JSON Schema regular expressions. JSON Schema regular expressions (REs) are ECMA regular expressions. Universality of these REs is undecidable [23], hence the witness generation problem for any sublanguage of JSON Schema that includes $\neg\mathsf{pattern}(r)$ is undecidable. In our implementation we sidestep this problem by mapping every JSON Schema RE unto a standard RE, as supported by the brics library [32], using a simple incomplete algorithm.[1] When the algorithm fails, we raise a failure. This approach allows us to manage the vast majority of our corpus.[2]

We limit our complexity analysis to the schemas where our RE translation succeeds, hence, we will hereafter assume that every *JSON Schema regexp* that appears in the source schema, can be translated to a standard RE with a linear expansion, similarly to the approach adopted in [18], where the analysis is restricted to standard REs.

4.4.2 Extending REs with external complement and intersection. In our algebra, we use a form of *externally extended REs* (EEREs), where the two extra operators are not first class RE operators, so that one cannot write $(\overline{r})*$, but they can be used at the outer level:

$$r ::= \textit{Any regular expression} \mid \overline{r} \mid r_1 \sqcap r_2$$

This extension does not affect the expressive power of regular expressions , since the set of regular languages is closed under intersection and complement, but affects their succinctness, hence the complexity of problems such as emptiness checking. We are going to exploit this expressive power in four different ways:

1. in order to translate `"additionalProperties"` : S as $\mathsf{props}(\overline{(r_1|\ldots|r_m)} : \langle S\rangle)$, where $\overline{r}$ is applied to a standard RE (Section 5);
2. in order to translate `"propertyNames"` : S, where a complex boolean combination of `pattern` assertions inside S produces a corresponding complex boolean combination of patterns in the translation (Section 5);
3. during not-elimination (Section 7.2), where $\mathsf{pattern}(\overline{r})$ is used to rewrite $\neg\mathsf{pattern}(r)$;
4. during object preparation (Section 8.3.3), where we must express the intersection and the difference of patterns that appear in $\mathsf{props}(r : S)$ and $\mathsf{pattReq}(r : S)$ operators.

During the final phases of our algorithm (Section 8.3), we need to solve the following *i-enumeration* problem (which generalizes emptiness) for our EEREs: for a given EERE r and for a given i, either return i words that belong to $L(r)$, or return "impossible" if $|L(r)| < i$. It is well-known that emptiness of REs extended (internally) with negation and intersection is non-elementary [37]. However, for our external-only extension i-enumeration and emptiness can be solved in time $O(i^2 \times 2^n)$.

Property 1. *If r is an EERE, its language can be recognized by a DFA with $O(2^{|r|})$ states, which can be built in time $O(2^{|r|})$.*

Proof. Let us define a *circuit* of REs to be a term rr generated by the following grammar, where the graph of dependencies induced by $x_1 : r_1, \ldots, x_n : r_n$ is acyclic:

$$\begin{array}{lcl} r & ::= & \textit{Any regular expression} \mid \overline{r} \mid r_1 \sqcap r_2 \mid x \\ rr & ::= & r\ \mathsf{defs}\ (x_1 : r_1, \ldots, x_n : r_n) \end{array}$$

The semantics of such a circuit is defined by recursively substituting every x with its definition, which is guaranteed to terminate because the dependencies are acyclic. Circuits of *RE*s generalize our EEREs; we prove the desired property for any circuit since this result will be useful in Section 5.3. We prove that any circuit rr of REs can be simulated by an automaton with $O(2^{|rr|})$ states. We

[1] The rewriting algorithm was suggested to us by Dominik Freydenberger in personal communication.

[2] We are currently able to translate more than 97% of the unique patterns in our corpus. The other ones mostly contain look-ahead and look-behind.

first transform each basic RE r_i that appears in the circuit into a *DFA* A_i of size $O(2^{|r_i|})$, in time $O(2^{|r_i|})$, using standard techniques [26]. We build the product automaton $A_\Pi = A_1 \times \ldots \times A_n$, whose states are tuple of states of $A_1 \times \ldots \times A_n$ in the standard fashion [29]; the states of this automaton grow as $O(2^{|r_1|}) \times \ldots \times 2^{|r_n|})$, i.e. $O(2^{|r_1|+\ldots+|r_n|})$, i.e., $O(2^{|rr|})$. We associate to each subexpression r in the circuit a set $F(r, rr)$ of states of A_Π that are "accepting" for r in the natural way: for each basic r_i, we define $F(r_i, rr)$ to be the states of A_Π whose i-projection is accepting for A_i. We set $F(r \sqcap r', rr) = F(r, rr) \cap F(r', rr)$, $F(\overline{r}, rr) = Q \setminus F(\overline{r}, rr)$, where Q are the states of A_Π, and we set $F(x, r \text{ defs } (E)) = F(E(x), r \text{ defs } (E))$, which is terminating since variables form a DAG. To each subexpression r of rr we associate the automaton A_r whose states and transitions are the same as A_Π, and whose final states are $F(r, rr)$. We define $d_E(r)$ as in the proof of Lemma 3, and we prove by induction on $(d_E(r), r)$ that A_r recognizes the language of r defs $(x_1 : r_1, \ldots, x_n : r_n)$. When $r = x$, this is true by induction, since $A_x = A_{E(x)}$ and $d_E(x) < d_E(E(x))$. When $r = \overline{r}$ or $r = r_1 \sqcap r_2$, the result follows by induction on r. □

Property 2. *For any extended RE r generated by our grammar starting from standard REs, the i-enumeration problem can be solved in time* $O(i^2 \times 2^{|r|})$.

Proof sketch. By Property 1, a DFA $A(r)$ for r with less than $2^{|r|}$ states can be built in time $O(2^{|r|})$.

Finally, given an automaton of size $2^{|r|}$, it is easy to see that the enumeration of i words can be performed in $O(i^2 \times 2^{|r|})$. □

5 FROM JSON SCHEMA TO THE ALGEBRA

5.1 Structure of the chapter

A JSON Schema schema is a JSON object whose fields are assertions. Essentially, the translation $\langle S\rangle$ of a schema S applies some simple rules to the single assertions, and combines them by conjunction, as follows:

$$\begin{array}{lcl} \langle\{\texttt{"a1"} : S1, \ldots, \texttt{"an"} : Sn\}\rangle & = & \langle\texttt{"a1"} : S1\rangle \wedge \ldots \wedge \langle\texttt{"an"} : Sn\rangle \\ \langle\texttt{"multipleOf"} : q\rangle & = & \mathsf{mulOf}(q) \\ \ldots & & \end{array}$$

However, there are some exceptions, that we describe in this chapter. We first describe how we map the complex referencing mechanism of JSON Schema into our simpler S defs (E) construct. We then describe the translation of the redundant operators `propertyNames`, `const`, `enum`, and `oneOf` into the core algebra. Finally, we describe the non-algebraic JSON Schema operators, where a group of related operators must be translated together, and we finish with the easy cases.

5.2 Representing definitions and references

JSON Schema defines a `$ref` : *path* operator that allows any subschema of the current schema to be referenced, as well as any subschema of a different schema that is reachable through a URI, hence implementing a powerful form of mutual recursion. The path *path* may navigate through the nodes of a schema document by traversing its structure, or may retrieve a subdocument on the basis of a special `id`, `$id`, or `$anchor` member (`$anchor` has been added in Draft 2019-09), which can be used to associate a name to the surrounding schema object. However, according to our collection of JSON schemas, the subschemas that are referred are typically just those that are collected inside the value of a top-level `definitions` member. Hence, we defined a referencing mechanism that is powerful enough to translate every collection of JSON schemas, but that privileges a direct translation of the most commonly used mechanism.

When all references in a JSON Schema document refer to a name defined in the `definitions` section, we just use the natural translation:

$$\begin{array}{l} \langle\{a_1 : S_1, \ldots, a_n : S_n, \texttt{definitions} : \{x_1 : S'_1, \ldots, x_m : S'_m\}\}\rangle \\ = \langle\{a_1 : S_1, \ldots, a_n : S_n\}\rangle \text{ defs } (x_1 : \langle S'_1\rangle, \ldots, x_m : \langle S'_m\rangle) \end{array}$$

In the general case, we collect all paths that are used in any reference assertion `$ref` : *path* and that are different from `definitions/k`, we retrieve the referred subschema and copy it inside the `definitions` member where we give it a name *name*, and we substitute all occurrences of `$ref` : *path* with `$ref` : `definitions/name`, until we reach the shape (1) above. In principle, this may cause a quadratic increase in the size of the schema, in case we have paths that refer inside the object that is referenced by another path. It would be easy to define a more complex mechanism with a linear worst-case size increase, but this basic approach does not create any size problem on the schemas we collected.[3]

Example 1. *We consider the following JSON Schema document*

```
{ "properties": {
    "Country": { "type": "string" },
    "City":    { "$ref": "#/properties/Country" } }
}
```

Definition normalization produces the following, equivalent schema:

```
{"properties": {
   "Country": {"type": "string" },
   "City":    {"$ref": "#/definitions/properties_Country"}},
 "definitions": {"properties_Country": {"type": "string" }}
}
```

Which is translated as:

$$\begin{array}{l} \mathsf{props}(\mathit{Country} : \mathsf{type}(\mathsf{Str})) \wedge \mathsf{props}(\mathit{City} : \mathit{properties\_Country}) \\ \mathsf{defs}(\mathit{properties\_Country} : \mathsf{type}(\mathsf{Str})) \end{array}$$

5.3 `"propertyNames"` : S **encoded as** $\mathsf{props}(\overline{r_S} : \mathbf{f})$

The JSON Schema assertion `"propertyNames"` : S requires that, if the instance is an object, then every member name satisfies S. Our translation to the algebra proceeds in two steps. We first translate to a new, redundant, algebraic operator $\mathsf{pNames}(S)$ that has the semantics that we just described:

$$\begin{array}{l} [\![\mathsf{pNames}(S)]\!]_E \\ = \{\!|\, J \mid J = \{k_1 : J_1, \ldots, k_m : J_m\} \Rightarrow \forall l \in \{1..m\}.\ k_l \in [\![S]\!]_E \,|\!\} \end{array}$$

Hence, $J \in [\![\mathsf{pNames}(S)]\!]_E$ means that no member name violates S. Hence, if we translate S into a pattern $r = \mathit{PattOfS}(S, E)$ that exactly describes the strings that satisfy S (whose variables are interpreted by E), we can translate $\mathsf{pNames}(S)$ into $\mathsf{props}(\mathit{PattOfS}(\neg S, E)$:

[3] When we have a collection of documents with mutual references, we first merge the documents together and then apply the same mechanism, but this functionality has not yet been integrated into our published code.

$\mathbf{f}$), which means: if the instance is an object, it cannot contain any member whose name does not match $\mathit{PattOfS}(S, E)$.

For all the ITOs S whose type is not Str, such as $\mathsf{mulOf}(q)$, we define $\mathit{PattOfS}(S, E) = .*$, since they are satisfied by any string:

$$\mathit{PattOfS}(\mathsf{mulOf}(a), E) \ = \ \mathit{PattOfS}(\mathsf{cont}_i^j(S), E) \ = \ \ldots \ = \ .*$$

For the other operators, $\mathit{PattOfS}(S, E)$ is defined as follows.

$$\begin{array}{lcl}
\mathit{PattOfS}(\mathsf{type}(T), E) & = & \overline{.*} \quad \text{if } T \neq \mathsf{Str} \\
\mathit{PattOfS}(\mathsf{type}(Str), E) & = & .* \\
\mathit{PattOfS}(\mathsf{pattern}(r), E) & = & r \\
\mathit{PattOfS}(S_1 \wedge S_2, E) & = & \mathit{PattOfS}(S_1, E) \sqcap \mathit{PattOfS}(S_2, E) \\
\mathit{PattOfS}(S_1 \vee S_2, E) & = & \overline{\overline{\mathit{PattOfS}(S_1, E)} \sqcap \overline{\mathit{PattOfS}(S_2, E)}} \\
\mathit{PattOfS}(\neg S, E) & = & \overline{\mathit{PattOfS}(S, E)} \\
\mathit{PattOfS}(x, E) & = & \mathit{PattOfS}(E(x), E)
\end{array}$$

Above, while $\mathit{PattOfS}(\mathsf{mulOf}(q), E) = .*$ since $\mathsf{mulOf}(q)$ is an Implicative Typed Operator, $\mathit{PattOfS}(\mathsf{type}(\mathsf{Num}), E) = \overline{.*}$, since $\mathsf{type}(\mathsf{Num})$ is not implicative, and is not satisfied by any string.

Since $\mathit{PattOfS}(S, E)$ does not depend on the schemas that are guarded by an ITO, the above definition is well-founded when recursion is guarded: after a variable x has been expanded, x is guarded in the result of any further expansion, hence we will not need to expand it again.

It is easy to prove the following equivalences, which allow us to translate pNames, hence `propertyNames`, into the core algebra.

Property 3. *For any assertion S and for any environment E guarded and closing for S, the following equivalences hold.*

$$\begin{array}{lcl}
[[\mathsf{type}(\mathsf{Str}) \wedge S]]_E & = & [[\mathsf{type}(\mathsf{Str}) \wedge \mathsf{pattern}(\mathit{PattOfS}(S, E))]]_E \\
[[\mathsf{pNames}(S)]]_E & = & [[\mathsf{props}(\mathit{PattOfS}(\neg S, E) : \mathbf{f})]]_E
\end{array}$$

This translation expands each variable with its definition, hence there exist schemas where $\mathit{PattOfS}(\neg S, E)$ is exponential in the size of (S, E). In practice, this is not a problem: in all schemas that we collected, `"propertyNames" : S` (which is quite rare) is invariably used with a very simple S, whose expansion is always small.

To ensure linear-size translation, we should extend regular expressions with a variable mechanism, for example in the following way, where we would impose a non-cyclic dependencies constraint to variable environments, so that an expression rr is actually a *Boolean circuit* of regular expressions.

$$\begin{array}{lcl}
r & ::= & \textit{Any regular expression} \mid \overline{r} \mid r_1 \sqcap r_2 \mid x \\
rr & ::= & r \ \mathsf{defs}\ (x_1 : r_1, \ldots, x_n : r_n)
\end{array}$$

Lifting $\overline{r}$ and $r \sqcap r'$ from EEREs to circuits is very easy. We can prove that the complexity of i-generation (Section 4.4) for circuits has the same bound as for EEREs, hence this extension would not create complexity problems. We can now translate an environment

$$E = \ldots x_i : S_i \ldots$$

with a pattern environment

$$\mathit{patt\_E} = \ldots \mathit{patt\_x}_i : \mathit{PattOfS}(S_i, E) \ldots$$

and we can then define

$$\mathit{PattOfS}(x, E) = \mathit{patt\_x} \ \mathsf{defs}\ (\mathit{patt\_E}).$$

Then, size expansion would be polynomial and not exponential.

Since the problem has, at the moment, no practical relevance, we decided to avoid this complication, hence we limit our complexity analysis to those schemas that are `propertyNames`-small, according to the following definition. If we encounter families of schemas that violate this property, we just need to extend our implementation, and our analysis, by supporting Boolean circuits of REs.

Definition 5 (`propertyNames`-small). A schema S defs (E) of the core algebra extended with $\mathsf{pNames}(S)$ is `propertyNames`-*small* if

$$|\mathit{PNExpand}(S)| \leq 2 \times |S \ \mathsf{defs}\ (E)|$$

where *PNExpand* is the function that translates all instances of $\mathsf{pNames}(S')$ with $\mathsf{props}(\mathit{PattOfS}(\neg S', E) : \mathbf{f})$.

Hence, by definition, the translation of `propertyNames` only causes a linear increase in `propertyNames`-small schemas.

5.4 Translation of `const` and `enum`

The assertions `"const"` : J and `"enum"` : $[J_1, \ldots, J_n]$, used to restrict a schema to a finite set of values, can be translated by first rewriting them into their algebraic counterparts $\mathsf{enum}(J_1, \ldots, J_n)$ and $\mathsf{const}(J)$, and then by applying the rules in Figure 4, similar to those presented in [28]. Hereafter, we use $\underline{k}$ to denote a pattern that only matches k;[4] when k is a string, so that `"const"` : k can be translated as $\mathsf{type}(\mathsf{Str}) \wedge \mathsf{pattern}(\underline{k})$.

5.5 Translation of `oneOf`

The assertion `"oneOf"` : $[S_1, \ldots, S_n]$ requires that J satisfies one of $S_1, \ldots, S_n$ and violates all the others. It can be expressed as follows, where the x_i's are fresh variables, and the defs part must actually be added to the outermost level:

$$\bigvee_{i \in \{1..n\}} (\neg x_1 \wedge \ldots \wedge \neg x_{i-1} \wedge x_i \wedge \neg x_{i+1} \wedge \ldots \wedge \neg x_n)$$
$$\mathsf{defs}\ (x_1 : \langle S_1 \rangle, \ldots, x_n : \langle S_n \rangle)$$

The definition of the fresh variables is fundamental in order to avoid that a single subschema is copied many times, which may cause an exponential size increase. The outermost $\bigvee$ has size $O(n^2)$, hence this encoding may still cause a quadratic size increase; this increase can be avoided using a more sophisticated linear encoding that we present in [15].[5]

5.6 The remaining assertions

While most JSON Schema assertions can be translated one by one, as described in Section 5.1, we have four groups of exceptions, that is, four families of assertions whose semantics depends on the occurrence of other assertions of the same family as members of the same schema. These families are:

(1) `if`, `then`, `else`;
(2) `additionalProperties`, `properties`, `patternProperties`;
(3) `additionalItems`, `items`;
(4) in Draft 2019-09: `minContains`, `maxContains`, `contains`.

[4] Using standard notation, $\underline{k}$ would generally coincide with k, unless k contains special characters, such as ".", "|", or "*", that need to be escaped.

[5] In our implementation we adopted the basic algorithm, having verified that, in our schema corpus, `oneOf` has on average 2.3 arguments, and, moreover, the quadratic encoding behaves better than the linear one when submitted to DNF expansion.

$\mathsf{enum}(J_1,\ldots,J_n)$	$=$	$\mathsf{const}(J_1) \vee \ldots \vee \mathsf{const}(J_n)$	
$\mathsf{const}(\mathtt{null})$	$=$	$\mathsf{type}(\mathsf{Null})$	
$\mathsf{const}(b)$	$=$	$\mathsf{type}(\mathsf{Bool}) \wedge \mathsf{ifBoolThen}(b)$	$b \in \mathsf{type}(\mathsf{Bool})$
$\mathsf{const}(n)$	$=$	$\mathsf{type}(\mathsf{Num}) \wedge \mathsf{betw}^{n}_{n}$	$n \in \mathsf{Num}$
$\mathsf{const}(s)$	$=$	$\mathsf{type}(\mathsf{Str}) \wedge \mathsf{pattern}(\underline{s})$	$s \in \mathsf{Str}$
$\mathsf{const}([J_1,\ldots,J_n])$	$=$	$\mathsf{type}(\mathsf{Arr}) \wedge \mathsf{cont}^{n}_{n}(\mathbf{t}) \wedge \mathsf{item}(1:\mathsf{const}(J_1)) \wedge \ldots \wedge \mathsf{item}(n:\mathsf{const}(J_n))$	
$\mathsf{const}(\{k_1:J_1,\ldots,k_n:J_n\})$	$=$	$\mathsf{type}(\mathsf{Obj}) \wedge \mathsf{req}(k_1,\ldots,k_n) \wedge \mathsf{pro}^{n}_{0} \wedge \mathsf{props}(\underline{k_1}:\mathsf{const}(J_1);\mathbf{t}) \wedge \ldots \wedge \mathsf{props}(\underline{k_n}:\mathsf{const}(J_n);\mathbf{t})$	

Figure 4: Elimination of enum and const.

When translating a schema object, we first partition it into families, we complete each family by adding the predefined default value for missing operators (for example, a missing `else` becomes `"else" : true`), and we then translate each family as we specify below. All other assertions are just translated one by one.

The assertion group `"if"` $: S_1,$ `"then"` $: S_2,$ `"else"` $: S_3$ is translated as follows, where $x : \langle S_1\rangle$ is inserted in order to avoid duplication of $\langle S_1\rangle$, and is actually lifted at the outermost level, as we do with `oneOf`:

$$((x \wedge \langle S_2\rangle) \vee (\neg x \wedge \langle S_3\rangle))\ \mathsf{defs}\ (x : \langle S_1\rangle)$$

The `properties` family is translated as follows, where we use pattern complement $\overline{r}$ to translate `additionalProperties`, which associates a schema to any name that does not match either `properties` or `patternProperties` arguments:

$$\begin{aligned}&\langle \texttt{"properties"} : \{k_1 : S_1,\ldots,k_n : S_n\},\\ &\ \texttt{"patternProperties"} : \{r_1 : PS_1,\ldots,r_m : PS_m\},\\ &\ \texttt{"additionalProperties"} : S\rangle\\ &= \mathsf{props}(\underline{k_1} : \langle S_1\rangle) \wedge \ldots \wedge \mathsf{props}(\underline{k_n} : \langle S_n\rangle)\\ &\quad \wedge \mathsf{props}(r_1 : \langle PS_1\rangle) \wedge \ldots \wedge \mathsf{props}(r_m) : \langle PS_m\rangle\\ &\quad \wedge \mathsf{props}(\overline{(\underline{k_1}|\ldots|\underline{k_n}|r_1|\ldots|r_m)} : S)\end{aligned}$$

`items` may have either a schema S or an array $[S_1,\ldots,S_n]$ as argument; in the first case, it is equivalent to $\mathsf{items}(0^+ : S)$, and a co-occurring `additionalItems` is ignored, while in the second case it is equivalent to $(\mathsf{item}(1 : S_1) \wedge \ldots \wedge \mathsf{item}(n : S_n))$, and `"additionalItems"` $: S'$ means $\mathsf{items}(n^+ : \langle S'\rangle)$. The family is hence translated as follows.

$$\begin{aligned}\langle \texttt{"additionalItems"} : S'\rangle &= \mathsf{items}(0^+ : \langle S'\rangle)\\ \langle \texttt{"items"} : S\rangle &= \mathsf{items}(0^+ : \langle S\rangle)\\ \langle \texttt{"items"} : S, \texttt{"additionalItems"} : S'\rangle &= \mathsf{items}(0^+ : \langle S\rangle)\\ \langle \texttt{"items"} : [S_1,\ldots,S_n]\rangle &= (\mathsf{item}(1 : \langle S_1\rangle) \wedge \ldots \wedge \mathsf{item}(n : \langle S_n\rangle))\end{aligned}$$

$$\begin{aligned}&\langle \texttt{"items"} : [S_1,\ldots,S_n], \texttt{"additionalItems"} : S'\rangle\\ &\quad = (\mathsf{item}(1 : \langle S_1\rangle) \wedge \ldots \wedge \mathsf{item}(n : \langle S_n\rangle)) \wedge \mathsf{items}(n^+ : \langle S'\rangle)\end{aligned}$$

The `contains` family is translated as follows - a missing lower bound defaults to 1 (rather than the usual 0), and a missing upper bound defaults to ∞:

$$\langle \texttt{"contains"} : S, \texttt{"minContains"} : m, \texttt{"maxContains"} : M\rangle = \mathsf{cont}^{M}_{m}(\langle S\rangle)$$

Then, we have the `dependencies` assertion:

$$\begin{aligned}&\texttt{"dependencies"} : \{k_1 : [k^1_1\ldots,k^1_{m_1}],\ldots,k_n : [k^n_1\ldots,k^n_{m_n}]\}\\ &\texttt{"dependencies"} : \{k_1 : S_1,\ldots,k_n : S_n\}\end{aligned}$$

The first form specifies that, for each $i \in \{1..n\}$, if the instance is an object and if it contains a member with name k_i, then it must contain all of the member names $k^i_1\ldots,k^i_{m_i}$. The second form specifies that, under the same conditions, the instance must satisfy S_i. Both forms are translated using req and $\Rightarrow$:

$$\begin{aligned}&\langle \texttt{"dependencies"} : \{k_1 : [r^1_1\ldots,r^1_{m_1}],\ldots,k_n : [r^n_1\ldots,r^n_{m_n}]\}\rangle\\ &\quad = ((\mathsf{type}(\mathsf{Obj}) \wedge \mathsf{req}(k_1)) \Rightarrow \mathsf{req}(r^1_1\ldots,r^1_{m_1}))\\ &\qquad \wedge \ldots \wedge ((\mathsf{type}(\mathsf{Obj}) \wedge \mathsf{req}(k_n)) \Rightarrow \mathsf{req}(r^n_1\ldots,r^n_{m_n}))\\ &\langle \texttt{"dependencies"} : \{k_1 : S_1,\ldots,k_n : S_n\}\rangle\\ &\quad = ((\mathsf{type}(\mathsf{Obj}) \wedge \mathsf{req}(k_1)) \Rightarrow \langle S_1\rangle)\\ &\qquad \wedge \ldots \wedge ((\mathsf{type}(\mathsf{Obj}) \wedge \mathsf{req}(k_n)) \Rightarrow \langle S_n\rangle)\end{aligned}$$

Finally, all the other JSON Schema assertions are translated one by one in the natural way, as reported in Table 1, where we omit the symmetric cases (e.g. `"maximum"`: M, `"exclusiveMaximum"` : M, etc) that can be easily guessed.

⟨`"minimum"`: m⟩	$=$	$\mathsf{betw}^{\infty}_{m}$
⟨`"exclusiveMinimum"`: m⟩	$=$	$\mathsf{xBetw}^{\infty}_{m}$
⟨`"multipleOf"`: n⟩	$=$	$\mathsf{mulOf}(n)$
⟨`"minLength"`: m⟩	$=$	$\mathsf{pattern}(\hat{}.\{m,\}\$)$
⟨`"pattern"`: r⟩	$=$	$\mathsf{pattern}(r)$
⟨`"minItems"`: m⟩	$=$	$\mathsf{cont}^{\infty}_{m}(\mathbf{t})$

Table 1: Translation rules for JSON Schema.

5.7 How we evaluate complexity

We have seen that JSON Schema can be translated to the algebra with a polynomial (actually, linear) size increase, and in the rest of the paper we show that our algorithm runs in $O(2^{poly(N)})$ with respect to the size of the input algebra, but with one important caveat: hereafter, we assume that all i and j constants different from ∞ that appear in $\mathsf{item}(i : S)$, $\mathsf{items}(i^+ : S)$, $\mathsf{contAfter}(i^+ : S)$, $\mathsf{cont}^{j}_{i}(S)$, and pro^{j}_{i}, are smaller than the input size, and we call this assumption the *linear constant assumption*. This is a reasonable assumption, since in practical cases these numbers tend to be extremely small when compared with the input size. Hereafter, whenever a result depends on this assumption, we will say that explicitly.

6 WITNESS GENERATION

6.1 The structure of the algorithm

In a recursive algorithm for witness generation, in order to generate a witness for an ITO such as $\mathsf{pattReq}(r : S)$, one can generate a witness J for S and use it to build an object with a member whose name matches r and whose value is J. The same approach can be followed for the other ITOs. For the Boolean operator $S_1 \vee S_2$, one recursively generates witnesses of S_1 and S_2.

Negation and conjunction are much less direct: there is no way to generate a witness for $\neg S$ starting from a witness for S. Also, given a witness for S_1, if it is not a witness for $S_1 \wedge S_2$, we may need to try infinitely many others before finding one that satisfies S_2 as well.[6] We solve this problem as follows. We first eliminate $\neg$ using not-elimination, then we bring all definitions of variables into DNF so that conjunctions are limited to sets of ITOs that regard the same type (Section 7). We then perform a form of *and-elimination* over these homogeneous conjunctions (*preparation*), and we finally use these "prepared" homogeneous conjunctions to generate the witnesses, through a bottom-up iterative process (Section 8).

Preparation is the crucial step: here we make all the interactions between the conjuncted ITOs explicit, which may require the generation of new variables. This phase is delicate because it is exponentially hard in the general case, and we must organize it in order to run fast enough in typical case. Moreover, it may generate infinitely many new variables, which we avoid with a technique based on ROBDDs, that we define in Section 7.1.

7 TRANSFORMATION IN POSITIVE, STRATIFIED, GROUND, CANONICAL DNF

We will illustrate the preliminary phases of our algorithm by exploiting the running example of Figure 5.

7.1 Premise: ROBDD reduction

Two expressions built with variables and Boolean operators are *Boolean-equivalent* when they can be proved equivalent using the laws of the Boolean algebra. An ROBDD (Reduced Ordered Boolean Decision Diagram) is a data structure that provides the same representation for two such expressions if, and only if, they are Boolean-equivalent [19]. Hence, whenever we define a variable x whose body S_x is a Boolean combination of variables, in any phase of the algorithm, we perform the *ROBDD reduction*: we compute the ROBDD representation of S_x, $robdd(S_x)$, and we store a pair $x : robdd(S_x)$ in the ROBDDTab table, unless a pair $y : robdd(S_y)$ with $robdd(S_x) = robdd(S_y)$ is already present. In this case, we substitute every occurrence of x with y. This technique makes the entire algorithm more efficient and, crucially, it ensures termination of the preparation phase (Section 8.3.3).

7.2 Not-elimination

Not-elimination, described in detail in our technical report [15], proceeds in two phases.

[6] One may actually solve the problem by ordered generation of witnesses for S_1 and S_2 and a merge-sort implementation of intersection, but the algorithms that we explored with this approach seem far more expensive than ours.

(a)

$$
\begin{aligned}
&r : \mathsf{pattReq}(b : x) \vee \mathsf{props}(a : y) \vee \mathsf{props}(a.* : \neg r \vee x),\\
&x : \mathsf{type}(\mathsf{Arr}), \qquad y : \mathsf{type}(\mathsf{Num})
\end{aligned}
$$

(b)

$$
\begin{aligned}
&r : \mathsf{pattReq}(b : x) \vee \mathsf{props}(a : y) \vee \mathsf{props}(a.* : co(r) \vee x),\\
&x : \mathsf{type}(\mathsf{Arr}), \qquad y : \mathsf{type}(\mathsf{Num}),\\
&co(r) : \mathsf{type}(\mathsf{Obj}) \wedge \mathsf{props}(b : co(x)) \wedge \mathsf{pattReq}(a : co(y))\\
&\qquad\qquad \wedge \mathsf{pattReq}(a.* : r \wedge co(x)),\\
&co(x) : \mathsf{type}(\mathsf{Null}) \vee \mathsf{type}(\mathsf{Bool}) \vee \mathsf{type}(\mathsf{Num}) \vee \mathsf{type}(\mathsf{Str}) \vee \mathsf{type}(\mathsf{Obj}),\\
&co(y) : \mathsf{type}(\mathsf{Null}) \vee \mathsf{type}(\mathsf{Bool}) \vee \mathsf{type}(\mathsf{Str}) \vee \mathsf{type}(\mathsf{Obj}) \vee \mathsf{type}(\mathsf{Arr})
\end{aligned}
$$

(c)

$$
\begin{aligned}
&r : \mathsf{pattReq}(b : x) \vee \mathsf{props}(a : y) \vee \mathsf{props}(a.* : crx),\\
&co(r) : \mathsf{type}(\mathsf{Obj}) \wedge \mathsf{props}(b : co(x)) \wedge \mathsf{pattReq}(a : co(y))\\
&\qquad\qquad \wedge \mathsf{pattReq}(a.* : rcx),\\
&crx : co(r) \vee x, \qquad rcx : r \wedge co(x)
\end{aligned}
$$

(d)

$$
\begin{aligned}
&crx : \{\mathsf{type}(\mathsf{Obj}), \mathsf{props}(b : co(x)), \mathsf{pattReq}(a : co(y)),\\
&\qquad \mathsf{pattReq}(a.* : rcx)\} \quad \vee \quad \{\mathsf{type}(\mathsf{Arr})\},\\
&rcx : \{(\mathsf{pattReq}(b : x), \mathsf{type}(\mathsf{Null})\} \vee \{(\mathsf{pattReq}(b : x), \mathsf{type}(\mathsf{Bool})\}\\
&\quad \vee \{(\mathsf{pattReq}(b : x), \mathsf{type}(\mathsf{Num})\} \vee \{(\mathsf{pattReq}(b : x), \mathsf{type}(\mathsf{Str})\}\\
&\quad \vee \{(\mathsf{pattReq}(b : x), \mathsf{type}(\mathsf{Obj})\}\\
&\quad \vee \{\mathsf{props}(a : y), \mathsf{type}(\mathsf{Null})\} \vee \{\mathsf{props}(a : y), \mathsf{type}(\mathsf{Bool})\}\\
&\quad \vee \{\mathsf{props}(a : y), \mathsf{type}(\mathsf{Num})\} \vee \{\mathsf{props}(a : y), \mathsf{type}(\mathsf{Str})\}\\
&\quad \vee \{\mathsf{props}(a : y), \mathsf{type}(\mathsf{Obj})\}\\
&\quad \vee \{\mathsf{props}(a.* : \mathrm{crx}), \mathsf{type}(\mathsf{Null})\} \vee \ldots\\
&\quad \vee \{\mathsf{props}(a.* : \mathrm{crx}), \mathsf{type}(\mathsf{Obj})\}
\end{aligned}
$$

(e)

$$
\begin{aligned}
&r : \{\mathsf{type}(\mathsf{Obj}), \mathsf{pattReq}(b : x)\} \vee \{\mathsf{type}(\mathsf{Obj}), \mathsf{props}(a : y)\}\\
&\quad \vee \{\mathsf{type}(\mathsf{Obj}), \mathsf{props}(a.* : crx\} \vee \{\mathsf{type}(\mathsf{Null})\}\\
&\quad \vee \{\mathsf{type}(\mathsf{Bool})\} \vee \{\mathsf{type}(\mathsf{Num})\} \vee \{\mathsf{type}(\mathsf{Str})\} \vee \{\mathsf{type}(\mathsf{Arr})\},\\
&rcx : \{\mathsf{type}(\mathsf{Obj}), \mathsf{pattReq}(b : x)\} \vee \{\mathsf{type}(\mathsf{Obj}), \mathsf{props}(a : y)\}\\
&\quad \vee \{\mathsf{type}(\mathsf{Obj}), \mathsf{props}(a.* : crx)\} \vee \{\mathsf{type}(\mathsf{Null})\}\\
&\quad \vee \{\mathsf{type}(\mathsf{Bool})\} \vee \{\mathsf{type}(\mathsf{Num})\} \vee \{\mathsf{type}(\mathsf{Str})\} \vee \{\mathsf{type}(\mathsf{Arr})\}
\end{aligned}
$$

Figure 5: (a) Original term. (b) After not-elimination. (c) After stratification, omitting unaffected variables. (d) After transformation to GDNF. (e) After canonicalization.

(1) Not-completion of variables: for every variable $x_n : S_n$ we define a corresponding $not\_x_n : \neg S_n$.[7]
(2) Not-rewriting: we rewrite every expression $\neg S$ into an expression where the negation has been pushed inside.

Not-completion of variables. Not-completion of variables is the operation that adds a variable not_x for every variable x as follows:

$$
\begin{aligned}
&\text{not-completion}(x_0 : S_0, \ldots, x_n : S_n) =\\
&\quad x_0 : S_0, , \ldots, x_n : S_n,\\
&\quad not\_x_0 : \neg S_0, \ldots, not\_x_n : \neg S_n
\end{aligned}
$$

After not-completion, every variable has a complement variable $co(x_i) = not\_x_i$ and $co(not\_x_i) = x_i$. The complement $co(x)$ is used for not-elimination (and also in the preparation phase).

Not-rewriting. We rewrite $\mathsf{req}(k)$ as $\mathsf{pattReq}(\underline{k} : \mathbf{t})$, and then we inductively apply the rules in Figure 6. It is easy to prove that not-elimination can be performed in linear time and increases the

[7] We do this, unless a variable whose body is Boolean-equivalent to $\neg S_n$ already exists, in which case that variable is used through ROBDD reduction

$\neg(\mathsf{ifBoolThen}(\mathtt{true}))$	=	$\mathsf{type}(\mathsf{Bool}) \wedge \mathsf{ifBoolThen}(\mathtt{false})$
$\neg(\mathsf{ifBoolThen}(\mathtt{false}))$	=	$\mathsf{type}(\mathsf{Bool}) \wedge \mathsf{ifBoolThen}(\mathtt{true})$
$\neg(\mathsf{pattern}(r))$	=	$\mathsf{type}(\mathsf{Str}) \wedge \mathsf{pattern}(\overline{r})$
$\neg(\mathsf{betw}_m^M)$	=	$\mathsf{type}(\mathsf{Num}) \wedge (\mathsf{xBetw}_{-\infty}^m \vee \mathsf{xBetw}_M^\infty)$
$\neg(\mathsf{xBetw}_m^M)$	=	$\mathsf{type}(\mathsf{Num}) \wedge (\mathsf{betw}_{-\infty}^m \vee \mathsf{betw}_M^\infty)$
$\neg(\mathsf{mulOf}(q))$	=	$\mathsf{type}(\mathsf{Num}) \wedge \mathsf{notMulOf}(q)$
$\neg(\mathsf{notMulOf}(q))$	=	$\mathsf{type}(\mathsf{Num}) \wedge \mathsf{mulOf}(q)$
$\neg(\mathsf{props}(r : S))$	=	$\mathsf{type}(\mathsf{Obj}) \wedge \mathsf{pattReq}(r : \neg S)$
$\neg(\mathsf{pattReq}(r : S))$	=	$\mathsf{type}(\mathsf{Obj}) \wedge \mathsf{props}(r : \neg S)$
$\neg(\mathsf{pro}_i^j)$	=	$\mathsf{type}(\mathsf{Obj}) \wedge (\mathsf{pro}_0^{i-1} \vee \mathsf{pro}_{j+1}^\infty)$
$\neg(\mathsf{item}(l : S))$	=	$\mathsf{type}(\mathsf{Arr}) \wedge \mathsf{item}(l : \neg S_i) \wedge \mathsf{cont}_l^\infty(\mathbf{t})$
$\neg(\mathsf{items}(i^+ : S))$	=	$\mathsf{type}(\mathsf{Arr}) \wedge \mathsf{contAfter}(i^+ : \neg S)$
$\neg(\mathsf{contAfter}(i^+ : S))$	=	$\mathsf{type}(\mathsf{Arr}) \wedge \mathsf{items}(i^+ : \neg S)$
$\neg(\mathsf{cont}_i^j(S))$	=	$\mathsf{type}(\mathsf{Arr}) \wedge (\mathsf{cont}_0^{i-1}(S) \vee \mathsf{cont}_{j+1}^\infty(S))$
$\neg(\mathsf{type}(T))$	=	$\bigvee(\mathsf{type}(T') \mid T' \neq T)$
$\neg(x)$	=	$co(x)$
$\neg(S_1 \wedge S_2)$	=	$(\neg S_1) \vee (\neg S_2)$
$\neg(S_1 \vee S_2)$	=	$(\neg S_1) \wedge (\neg S_2)$
$\neg(\neg S)$	=	S

Figure 6: Not-pushing rules — unsatisfiable disjuncts, such as pro_0^{-1} or $\mathsf{pro}_\infty^\infty$, are generated as f.

schema size of a linear factor. We report here the following result from [15].

Property 4. *For any system where recursion is guarded, not elimination preserves the semantics of every variable.*

From now on, every other phase of the algorithm will only produce schemas that belong to the positive algebra.

7.3 Stratification

We say that a schema is *stratified* when every schema argument of every ITO is a variable, so that $\mathsf{pattReq}(a : x \wedge y)$ is not stratified while $\mathsf{pattReq}(a : w)$ is stratified.

Stratification makes it easy to build a witness for a typed group such as

$$\{\mathsf{Obj}, \mathsf{pattReq}(\text{\textasciicircum a\$} : x), \mathsf{pattReq}(\text{\textasciicircum b\$} : y)\}$$

after a witness for each involved variable has been built.

In this phase, for every ITO that has a subschema S in its syntax, such as $\mathsf{cont}_i^j(S)$, when S is not a variable, we create a new variable $x : S$, and we substitute S with x. For every variable $x : S$ that we define, we must also define its complement $not\_x : \neg S$, and perform not-elimination and stratification on $\neg S$ — see Figure 5(c). As specified in Section 7.1, we apply ROBDD reduction to $x : S$ and $not\_x : \neg S$.

Property 5. *Stratification transforms a schema S* defs (E) *into a schema S'* defs (E') *such that $[[S]]_E = [[S']]_{E'}$.*

Property 6. *Stratification transforms a schema S* defs (E) *into a schema S'* defs (E') *such that $|S'$ defs $(E')|$ is in $O(N)$, where $N = |S$ defs $(E)|$.*

Proof. Assume that stratification is performed bottom up, so that $\mathsf{cont}_i^j(\mathsf{cont}_l^k(S))$ is first transformed into $\mathsf{cont}_i^j(\mathsf{cont}_l^k(x))$ with $x : S$ and $not\_x : \neg S$, and then in $\mathsf{cont}_i^j(y)$ with $y : \mathsf{cont}_l^k(x)$ and $not\_y : \neg\mathsf{cont}_l^k(x)$. In this way, every S that is moved to the environment is only copied twice (once below negation), and each such operation generates two instances of x and one of $\neg x$. Hence, each node in the original tree corresponds to a constant number of nodes in the stratified tree - in the worst case, it generates three variables, one negation, and two copies of the original node. At this point we apply not-elimination, and this step is linear as well. □

7.4 Transformation in Canonical GDNF

Guarded DNF. A schema is in Guarded Disjunctive Normal Form (GDNF) if it has the shape $\vee(\wedge(S_{1,1}, \ldots, S_{1,n_1}), \ldots, \wedge(S_{l,1}, \ldots, S_{l,n_l}))$ and every $S_{i,j}$ is a TO. Every conjunction may be trivial ($n_i = 1$), and so may be the disjunction ($l = 1$).

To produce a new environment E^G in GDNF starting from a positive and stratified environment E, we first define an ordered enumeration $\{\!|x_1, \ldots, x_o|\!\}$ of the variables in $Vars(E)$ such that when x_i directly depends of x_j (as defined in Section 4.1) then $j < i$. We know that such enumeration exists because recursion is guarded. We now compute $E^G(x_i)$ starting from x_1 and going onward, so that, when we compute $E^G(x_i)$, $E^G(x_j)$ has already been computed for each $j < i$.

Let $\mathcal{T}$ denote the set of all TOs that appear in E as subterms of $E(y)$ for any y, so that, if

$$E \;=\; x : (\mathsf{type}(\mathsf{Num}) \wedge \mathsf{pattReq}(\text{\textasciicircum}a\$: x)) \vee \mathsf{mulOf}(3)$$

then $\mathcal{T} = \{\!|\mathsf{type}(\mathsf{Num}),\ \mathsf{pattReq}(\text{\textasciicircum}a\$: x),\ \mathsf{mulOf}(3)|\!\}$. As we will show, reduction in GDNF does not create any new typed expression, hence every term in GDNF corresponds to a set DC (*Disjunction of Conjunctions*) of subsets of $\mathcal{T}$ as follows.

$$E^G(x) = \bigvee_{C \in DC_x} \bigwedge_{S \in C} S \quad \text{where} \quad DC_x \in \mathcal{P}(\mathcal{P}(\mathcal{T}))$$

To compute this set-of-sets representation $g(E(x))$ of the GDNF of the body $E(x)$ of every x defined in E, we apply the following rules:

$$\begin{array}{lcll} g(S) & = & \{\!|\{\!|S|\!\}|\!\} & \text{if } S \text{ is a TO} \\ g(y) & = & E^G(y) & \\ g(S_1 \vee S_2) & = & g(S_1) \cup g(S_2) & \\ g(S_1 \wedge S_2) & = & \bigcup_{(C_1,C_2) \in g(S_1) \times g(S_2)} (C_1 \cup C_2) & \end{array}$$

When S is a typed expression, it is translated into a trivial GDNF. Each variable y inside $E(x)$ had its body already transformed. The rule for $\vee$ is trivial, while the rule for $\wedge$ is Boolean algebra distributivity: for each conjunction $\bigwedge_{S \in C_1} S$ of S_1 and for each conjunction $\bigwedge_{S \in C_2}$ of S_2, the conjunction $\bigwedge_{S \in C_1} S \wedge \bigwedge_{S \in C_1} S = \bigwedge_{S \in C_1 \cup C_2} S$ is inserted in the result.

Reduction to GDNF can lead to an exponential explosion, and it is actually the most expensive phase of our algorithm, according to our measures (Section 9).

Property 7. *For a given schema x* defs (E)*, such that $n = |x$* defs $(E)|$*, the size of x* defs (E^G) *is in $O(2^n)$, and it can be build in time $O(2^n)$.*

Proof. The schema x defs (E^G) has $O(n)$ variables. The body of each variable can be represented as a set DC belonging to $\mathcal{P}(\mathcal{P}(\mathcal{T}))$ The set $\mathcal{P}(\mathcal{T})$ has size $O(2^n)$, hence every set of sets DC contain at most $O(2^n)$ sets, and each of these sets can be represented using n bits. This yields a total upper bound of $O(n) \times O(n) \times O(2^n)$ for x defs (E^G). As for the construction time, the most expensive part is the computation of $\bigcup_{(C_1,C_2)\in g(S_1)\times g(S_2)} (C_1 \cup C_2)$, that may take place once for each variable. The size of $g(S_1) \times g(S_2)$ is in $O(2^n)$, the size of C_1 and C_2 is in $O(n)$, hence this computation is in $O(2^n)$. ◻

Canonicalization. Canonicalization is a process defined along the lines of [28]. We say that a conjunction that contains exactly one assertion type(T) and a set of ITOs of that same type T is a *typed group* of type T; canonicalization splits every conjunct of the GDNF into a set of *typed groups* (Figure 5(e), where we also applied elementary equivalences, such as idempotence of $\vee$).

In order to transform a conjunction C of a GDNF DC into a typed group, we first repeatedly apply the following rewriting rules, which preserve the meaning of the conjunction. In the third rule, $ITO(T')$ are the ITOs associated to type T', which are trivially satisfied when in conjunction with a type(T) with $T \neq T'$:

$$
\begin{array}{llll}
\mathsf{type}(T), \mathsf{type}(T) & \rightarrow & \mathsf{type}(T) & \\
\mathsf{type}(T), \mathsf{type}(T') & \rightarrow & \mathbf{f} & T \neq T' \\
\mathbf{f}, S & \rightarrow & \mathbf{f} & \\
\mathsf{type}(T), S & \rightarrow & \mathsf{type}(T) & S \in ITO(T'),\ T' \neq T
\end{array}
$$

The first three rules ensure that the result is either $\mathbf{f}$, which is then deleted from the disjunction, or has exactly one type(T) assertion, or has none. If it has exactly one type(T) assertion, then the fourth rule ensures that all the ITOs refer to type T. If it has no type(T) assertion, we transform it in the following equivalent disjunction, where $filter(\{\mathsf{S}_1, \ldots, \mathsf{S}_n\}, T)$ is the conjunction of those ITOs in $\{\mathsf{S}_1, \ldots, \mathsf{S}_n\}$ whose type is T:

$$
\begin{array}{ll}
(\mathsf{type}(\mathsf{Null})) & \vee (\mathsf{type}(\mathsf{Bool}) \wedge filter(C, \mathsf{Bool})) \\
& \vee (\mathsf{type}(\mathsf{Str}) \wedge filter(C, \mathsf{Str})) \ldots
\end{array}
$$

so that every $C \in DC$ denotes a set of values of the same type.

By construction, every phase described in this section transforms a JSON Schema document into an equivalent one.

Property 8 (Equivalence). *The phases of not-elimination, stratification, transformation into Canonical GDNF, transform a JSON Schema document into an equivalent one.*

8 PREPARATION AND WITNESS GENERATION

8.1 Assignments and bottom-up semantics

Let us define an assignment A for an environment E as a function mapping each variable of E to a set of JSON values. An assignment is sound when it maps each variable to a subset of its semantics. We order assignments by variable-wise inclusion.

Definition 6 (Assignments, Soundness, Order). An assignment A for an environment E is a function mapping each variable of E to a set of JSON values. An assignment A for E is sound iff for all $y \in Vars(E)$: $A(y) \subseteq [\![y]\!]_E$. We say that $A \leq A'$ iff $\forall y.\ A(y) \subseteq A'(y)$.

Given a schema S defs (E), an assignment A for E defines an *assignment-evaluation* for S by applying the rules in Figure 7, which are the same rules that define environment-based semantics $[\![S]\!]_E$, with the only difference that a variable x is not interpreted by interpreting the schema $E(x)$, but directly as the set of values $A(x)$ (we always assume that every schema S defs (E) is closed and guarded).

For all schemas not containing subschemas, such as ifBoolThen(b), we just define $\langle\!\langle \mathsf{ifBoolThen}(b) \rangle\!\rangle_A = [\![\mathsf{ifBoolThen}(b)]\!]_E$, and neither A nor E play any role in the definition

$$
\begin{array}{lcl}
\langle\!\langle x \rangle\!\rangle_A & = & A(x) \\
\langle\!\langle \mathsf{ifBoolThen}(b) \rangle\!\rangle_A & = & \{\!| J \mid J \in \mathcal{J}Val(\mathsf{Bool}) \Rightarrow J = b |\!\} \\
\langle\!\langle \mathsf{props}(r : S) \rangle\!\rangle_A & = & \{\!| J \mid J = \{(k_1 : J_1), \ldots, (k_n : J_n)\} \Rightarrow \\
& & \forall i \in \{1..n\}.\ k_i \in L(r) \ \Rightarrow J_i \in \langle\!\langle S \rangle\!\rangle_A |\!\} \\
\langle\!\langle \mathsf{item}(l : S) \rangle\!\rangle_A & = & \{\!| J \mid J = [J_1, \ldots, J_n] \Rightarrow \\
& & n \geq l \Rightarrow J_l \in \langle\!\langle S \rangle\!\rangle_A |\!\} \\
\langle\!\langle \mathsf{cont}_i^j(S) \rangle\!\rangle_A & = & \{\!| J \mid J = [J_1, \ldots, J_n] \Rightarrow \\
& & i \leq \ |\{\!| l \mid J_l \in \langle\!\langle S \rangle\!\rangle_A |\!\}| \ \leq j |\!\} \\
\langle\!\langle S_1 \wedge S_2 \rangle\!\rangle_A & = & \langle\!\langle S_1 \rangle\!\rangle_A \cap \langle\!\langle S_2 \rangle\!\rangle_A \\
\langle\!\langle S_1 \vee S_2 \rangle\!\rangle_A & = & \langle\!\langle S_1 \rangle\!\rangle_A \cup \langle\!\langle S_2 \rangle\!\rangle_A \\
\ldots & &
\end{array}
$$

Figure 7: Rules for assignment-evaluation.

For schemas in the positive algebra, iterated assignment-evaluation yields an alternative notion of semantics, as follows.

Definition 7. For a given positive environment E, the corresponding assignment transformation $T_E(\_)$ is the function from assignments to assignments defined as follows:

$$\forall y \in Vars(E).\ T_E(A)(y) = \langle\!\langle E(y) \rangle\!\rangle_A$$

Intuitively, if A collects witnesses for the variables in E, then $T_E(A)$ uses E in order to build new witnesses starting from those in A. For example, if E contains $y : \{\mathsf{type}(\mathsf{Arr}), \mathsf{items}(0^+ : x), \mathsf{cont}_1^3(\mathbf{t})\}$, if $A(x) = \{\!|J|\!\}$, then $T_E(A)(y) = \{\!| [J], [J, J], [J, J, J] |\!\}$.

For any positive environment E, the corresponding assignment transformation is monotone in A, by positivity of E, hence T_E has a minimal fix-point, that is the limit $\mathcal{A}_E^\infty$ of the sequence $\mathcal{A}_E^i$ defined accordingly to Tarski theorem, starting from the empty assignment and then reapplying T_E.

Definition 8 ($\mathcal{A}_E^i$, $\mathcal{A}_E^\infty$). For a given positive environment E, the sequence of assignments $\mathcal{A}_E^i$ is defined as follows:

$$
\begin{array}{l}
\forall y \in Vars(E).\ \mathcal{A}_E^0(y) = \emptyset \\
\mathcal{A}_E^{i+1} = T_E(\mathcal{A}_E^i)
\end{array}
$$

The assignment $\mathcal{A}_E^\infty$ is defined as $\bigcup_{i\in\mathbb{N}} \mathcal{A}_E^i$.

Property 9. *For any positive E, the assignment $\mathcal{A}_E^\infty$ is the minimal fix-point of the assignment transformation T_E.*

In Section 4.2, we adopted the official top-down semantics for JSON schema in order to follow the standard and because it also applies to negative operators. However, on positive schemas, the top-down semantics and the bottom-up fix-point coincide.

Property 10. *For any positive schema S* defs (E), *the following equality holds:*

$$[\![S]\!]_E = \langle\!\langle S\rangle\!\rangle_{\mathcal{A}_E^\infty}$$

Proof sketch. We prove, by induction on i and, when i is equal, on S, that for all i, and for any positive assertion S that is closed wrt E, the following holds:

$$[\![S]\!]_E^i = \langle\!\langle S\rangle\!\rangle_{\mathcal{A}_E^i}$$

For the inductive step $i+1$, if S is an operator that contains no schema subterm, the equality

$$[\![S]\!]_E^{i+1} = \langle\!\langle S\rangle\!\rangle_{\mathcal{A}_E^{i+1}}$$

is immediate. If S is a variable, we have, by definition, $[\![y]\!]_E^{i+1} = [\![E(y)]\!]_E^i$ and $\langle\!\langle y\rangle\!\rangle_{\mathcal{A}_E^{i+1}} = (\mathcal{A}_E^{i+1})(y) = \langle\!\langle E(y)\rangle\!\rangle_{\mathcal{A}_E^i}$; we can conclude since $[\![E(y)]\!]_E^i = \langle\!\langle E(y)\rangle\!\rangle_{\mathcal{A}_E^i}$ holds by induction on i. For $S = S_1 \wedge S_2$ we reason by induction on S as follows:

$$\begin{aligned}[\![S_1 \wedge S_2]\!]_E^{i+1} &= [\![S_1]\!]_E^{i+1} \cap [\![S_2]\!]_E^{i+1}\\ &= \langle\!\langle S_1\rangle\!\rangle_{\mathcal{A}_E^{i+1}} \cap \langle\!\langle S_2\rangle\!\rangle_{\mathcal{A}_E^{i+1}} = \langle\!\langle S_1 \wedge S_2\rangle\!\rangle_{\mathcal{A}_E^{i+1}}\end{aligned}$$

For all other operators we reason in the same way.

Finally, the base case $i = 0$. When $S = x$, then both $[\![x]\!]_E^0$ and $\langle\!\langle x\rangle\!\rangle_{\mathcal{A}_E^0}$ are the empty set. In all other cases, we reason as in case $i > 0$.

Now, since $[\![S]\!]_E^p$ coincides with $\langle\!\langle S\rangle\!\rangle_{\mathcal{A}_E^p}$ for any p, then $[\![S]\!]_E^p$ is a succession of sets that grows with p, hence $\bigcap_{p\geq i}[\![S]\!]_E^p = \langle\!\langle S\rangle\!\rangle_{\mathcal{A}_E^i}$, hence $\bigcup_{i\in N}\bigcap_{p\geq i}[\![S]\!]_E^p = \bigcup_{i\in\mathbb{N}}\langle\!\langle S\rangle\!\rangle_{\mathcal{A}_E^i} = \langle\!\langle S\rangle\!\rangle_{\mathcal{A}_E^\infty}$. □

Any JSON value J has a *depth* $\delta(J)$, that is the number of levels of its tree representation, formally defined as follows.

Definition 9 (Depth $\delta(J)$, $\mathcal{J}^d$). The depth of a JSON value J, $\delta(J)$, is defined as follows, where $\max(\{\!|\ |\!\})$ is defined to be 0:

J belongs to a base type :	$\delta(J) = 1$
$J = [J_1, \ldots, J_n]$:	$\delta(J) = 1 + \max(\{\!\|\delta(J_1), \ldots, \delta(J_n)\|\!\})$
$J = \{\ a_1 : J_1, \ldots, a_n : J_n\ \}$:	$\delta(J) = 1 + \max(\{\!\|\delta(J_1), \ldots, \delta(J_n)\|\!\})$

$\mathcal{J}^d$ is the set of all JSON values J with $\delta(J) \leq d$.

The assignment $\mathcal{A}_E^i$ includes all witnesses of depth i: for any depth i, it can be proved that $([\![y]\!]_E \cap \mathcal{J}^i) \subseteq \mathcal{A}_E^i(y)$.

Bottom-up semantics is the basis of bottom-up witness generation: we will compute a witness for S defs (E) by approximating the sequence $\mathcal{A}_E^i$.

8.2 Bottom-up iterative witness generation

Since S defs (E) is equivalent to x defs $(x : S, E)$, we will discuss here, for simplicity, generation for the x defs (E) case.

Our algorithm for bottom-up iterative witness generation for a schema x defs (E) produces a sequence of finite assignments A^i, each approximating the assignment $\mathcal{A}_E^i$, until we reach either a witness for x or an "unsatisfiability fix-point", which is a notion that we will introduce shortly.

A^i is built as follows: $A^0 = \mathcal{A}_E^0$; then, at step i, for each $y \in Vars(E)$, we compute a set of new values for y based on the current assignment A^i using a generation algorithm $\mathsf{Gen}(E(y), A^i)$ that computes a subset of $\langle\!\langle E(y)\rangle\!\rangle_{A^i}$; formally, $A^{i+1}(y) = \mathsf{Gen}(E(y), A^i)$. Our specific Gen algorithm is defined in the next section, but we show now that any generic algorithm g can be used to approximate $\langle\!\langle E(y)\rangle\!\rangle_{A^i}$, provided that g is *sound* and *generative*.

We first introduce a notion of *i-witnessed assignment A*: if a variable y has a witness J with $\delta(J) \leq i$, then y has a witness in an *i-witnessed assignment A*.

Definition 10 (i-witnessed). For a given environment E, and an assignment A for E, we say that A is i-witnessed if:

$$\forall y \in Vars(E).\ ([\![y]\!]_E \cap \mathcal{J}^i) \neq \emptyset \implies A(y) \neq \emptyset$$

Generativity of g means that, if A is i-witnessed, then the assignment computed using g is (i+1)-witnessed, so that, by repeated application of g starting from A^0, every non-empty variable will be eventually "witnessed" (Property 11).

Hereafter, we say that a triple (S, E, A) is coherent if E is guarded and closing for S, and if $Vars(E) = Vars(A)$.

Definition 11 (Soundness of g). A function $g(\_,\_)$ mapping each pair assertion-assignment to a set of JSON values is *sound* iff, for every coherent (S, E, A), if A is sound for E, then $g(S, A) \subseteq [\![S]\!]_E$.

Definition 12 (Generativity of g). A function $g(\_,\_)$ mapping each pair assertion-assignment to a set of JSON values is *generative* for an assertion S iff for any E and A such that (S, E, A) is coherent:

(1) if $([\![S]\!]_E \cap \mathcal{J}^1) \neq \emptyset$, then $g(S, A) \neq \emptyset$;
(2) for any $i \geq 1$, if A is i-witnessed, and if $([\![S]\!]_E \cap \mathcal{J}^{i+1}) \neq \emptyset$, then $g(S, A) \neq \emptyset$.

g is *generative for E* if it is generative for $E(y)$ for each variable $y \in Vars(E)$.

Soundness of Gen inductively implies that every assignment in every A^i is sound. Generativity implies that each A_i computed by the i-th pass of the algorithm is i-witnessed, so that, if a variable has a witness J of depth d, then $A^i \neq \emptyset$ for every $i \geq d$.

We can now define our bottom-up algorithm (Algorithm 1) as follows.

Algorithm 1: Bottom-up witness generation

```
BottomUpGenerate(x,E)
    Prepare (E);
    ∀y. A[y] := nextA[y] := ∅ ;
    while A[x] == ∅ do
        for y in vars(E) where A[y] == ∅ do
            nextA[y] := Gen(E(y),A)
        if (∀y. nextA[y] == A[y]) then return (unsatisfiable);
        else
            ∀y. A[y] := nextA[y];
    return (A[x]);
```

Prepare(E) rewrites E and prepares all the extra variables needed for generation, as explained later. Then, we initialize A^0 as the empty assignment $\lambda y.\ \emptyset$. We repeatedly execute a pass that sets $A^i(y) = \mathsf{Gen}(E(y), A^{i-1})$ for any y such that $A^{i-1}(y) = \emptyset$ — we call it "pass i". We say that a pass i is *useful* if there exists y such that $A^i(y) \neq \emptyset$ while $A^{i-1}(y) = \emptyset$, and we say that pass i was *useless* otherwise. Before each pass i, if $\langle\!\langle x\rangle\!\rangle_{A^{i-1}} \neq \emptyset$, then the algorithm stops with success. After pass i, if the pass was useless, the algorithm stops with "unsatisfiable".

We can now prove that this algorithm is correct and complete, as follows.

Property 11 (Correctness and completeness). *If* Gen *is sound and is generative for E after preparation, then Algorithm 1 enjoys the following properties.*

1. *If the algorithm terminates with success after step i, then $A^i(x)$ is not empty and is a subset of $[[x]]_E$.*
2. *If the algorithm terminates with "unsat.", then $[[x]]_E = \emptyset$.*
3. *The algorithm terminates after at most $|Vars(E)| + 1$ passes.*

Proof. Property (1) is immediate: by induction and by soundness of Gen, we have that A^i is sound for any i, that is, $\langle\langle S\rangle\rangle_{A^i} \subseteq [[S]]_E$.

For (2), we first prove the following property: if the algorithm terminates with "unsatisfiable" after step j, then, for every variable y:

$$A^j(y) = \emptyset \implies [[y]]_E = \emptyset.$$

Assume, towards a contradiction, that there is a non empty set of variables Y such that

$$y \in Y \implies (A^j(y) = \emptyset \ \wedge\ [[y]]_E \neq \emptyset).$$

Let d be the minimum depth of $\bigcup_{y\in Y}[[y]]_E$, and let w be a variable in Y and such that d is the minimum depth of the values in $[[w]]_E$. Minimality of d implies that every variable z with a value in $[[z]]_E$ whose depth is less than $d-1$ has a witness in A^j, hence, since the step j was useless, every such z has a witness in A^{j-1}, hence A^{j-1} is $(d-1)$-witnessed, hence, by generativity, w should have a witness generated during step j, which contradicts the hypothesis.

If the algorithm terminates with "unsatisfiable", this means that $\langle\langle x\rangle\rangle_{A^{j-1}} = \emptyset$, hence $\langle\langle x\rangle\rangle_{A^j} = \emptyset$ since the step j was useless, hence $[[x]]_E = \emptyset$, since we proved that

$$A^j(y) = \emptyset \Rightarrow [[y]]_E = \emptyset.$$

Property (3) is immediate: at every useful pass the number of variables such that $A^i(y) \neq \emptyset$ diminishes by at least 1, hence we can have at most $|Vars(E)|$ useful passes plus one useless pass. □

We can finally describe the phases of preparation and generation for all typed groups.

Preparation is a crucial phase, where we make explicit the interactions between different object or array operators found in a same typed group, and we create new variables to manage these interactions.

8.3 Object group preparation and generation

8.3.1 Constraints and requirements. We say that an assertion $S = \mathsf{props}(r : x)$ or $S = \mathsf{pro}_0^M$ is a *constraint*. A *constraint* has the following features: (a) $\{\ \} \in [[S]]_E$ and (b) $\{k_1 : J_1, \ldots, k_n : J_n, k_{n+1} : J_{n+1}\} \in [[S]]_E \Rightarrow \{k_1 : J_1, \ldots, k_n : J_n\} \in [[S]]_E$ — constraints can prevent the addition of members, but they never require the presence of a member, similarly to a *for all fields* quantifier.

We say that an assertion $S = \mathsf{pattReq}(r : x)$ or $S = \mathsf{pro}_m^\infty$ with $m > 0$ is a *requirement*. A *requirement S* has the following features: (a) $\{\ \} \notin [[S]]_E$ and (b) $\{k_1 : J_1, \ldots, k_n : J_n\} \in [[S]]_E \Rightarrow \{k_1 : J_1, \ldots, k_n : J_n, k_{n+1} : J_{n+1}\} \in [[S]]_E$ — requirements can require the addition of a member, but they never prevent adding a member, similarly to an *exists field* quantifier.

As a consequence, a possible algorithm to build an object is: start from the empty object, add one member at a time until all requirements are satisfied, but, whenever you add a member to satisfy *some* requirements, verify that it satisfies *all* constraints too.

8.3.2 Preparation and generation. For a typical object group, where every pattern is trivial and where each type in each pattReq is just $x_{\mathbf{t}}$, object generation is very easy. Consider the following group:

$$\{\ \mathsf{type}(\mathsf{Obj}), \mathsf{props}(\text{"}a\text{"} : x), \mathsf{pattReq}(\text{"}a\text{"} : x_{\mathbf{t}}), \mathsf{pattReq}(\text{"}c\text{"} : x_{\mathbf{t}})\ \}$$

In order to generate a witness, we just need to generate a member $k : J$ for each required key, respecting the corresponding props constraint if present. Hence, here we generate a member "a" : J where $J \in A^i(x)$, and a member "c" : J', where J' is arbitrary.

Unfortunately, in the general case where we have non-trivial patterns and where the pattReq operator specifies a non-trivial schema for the required member, the situation is much more complex, and we must keep into account the following issues:

1. need to compute the intersections between patterns of different assertions;
2. need to generate new variables when patterns intersect;
3. possibility for one member to satisfy many requirements.

To exemplify the first two problems, consider the following object group: $\{\ \mathsf{type}(\mathsf{Obj}), \mathsf{props}(p : x), \mathsf{pattReq}(r : y), \mathsf{pro}_1^1\ \}$.

There are two distinct ways of producing a witness $\{\ k : J\ \}$ for the object above: either we generate a k that matches $r \sqcap \overline{p}$, and a witness J for y, or we generate a k that matches $r \sqcap p$, and a witness J for $x \wedge y$. This exemplifies the first two issues above:

1. patterns: we need to compute which of the combinations $r \sqcap \overline{p}$ and $r \sqcap p$ have a non-empty language, in order to know which approaches are viable w.r.t. to pattern combination;
2. new variables: we need a new variable whose body is $x \wedge y$, in order to generate a witness for this conjunctive schema.

Let us say that a member $k : J$ has shape $r : S$ when $k \in L(r)$ and J is a witness for S. Then, we can rephrase the example above by saying that an object $\{\ k : J\ \}$ satisfies that object group iff $k : J$ either has shape $(r \sqcap \overline{p} : y)$ or $(r \sqcap p : x \wedge y)$.

To exemplify the last problem — one member possibly satisfying many requirements — consider the following object group:

$$\{\ \mathsf{type}(\mathsf{Obj}), \mathsf{pattReq}(r_1 : y_1), \mathsf{pattReq}(r_2 : y_2), \mathsf{pro}_{min}^{Max}\}$$

In order to satisfy both requirements, we have two possibilities:

1. producing just one member with shape $r_1 \sqcap r_2 : y_1 \wedge y_2$;
2. producing two members, with shapes $r_1 : y_1$ and $r_2 : y_2$.

In order to explore all possible ways of generating a witness, we need to consider both possibilities. But, in order to consider the first possibility, we need a new variable whose body is equivalent to $y_1 \wedge y_2$.

We solve all these issues by transforming, during the preparation phase, every object into a form where all possible interactions between assertions are made explicit, and we create a fresh new variable for every conjunction of variables that is relevant for witness generation. The generative witness-generation function that is used during bottom-up evaluation, and that will be described in the Section 8.3.4, will be applied to this prepared form.

8.3.3 Object group preparation. Consider a generic object group

$$\{\ \text{type(Obj)},\quad \text{props}(p_1 : x_1), \ldots, \text{props}(p_m : x_m), \text{pattReq}(r_1 : y_1), \ldots, \text{pattReq}(r_n : y_n), \text{pro}^{Max}_{min}\ \}$$

We use CP (*constraining part*) to denote the set of props assertions $\{\!|\text{props}(p_i : x_i) \mid i \in 1..m|\!\}$ and RP (*requiring part*) to denote the set of pattReq assertions. Any witness for this object group is a collection of fields (k, J) where every field satisfies every constraint $\text{props}(p_i : x_i)$ such that $k \in L(p_i)$, and such that every requirement $\text{pattReq}(r_j : y_j)$ is satisfied by a matching field. Hence, every field is associated to a set $CP' \subseteq CP$ of constraints and to a set $RP' \subseteq RP$ of requirements. Only some pairs of sets (CP', RP') make sense, because of pattern compatibility. Object preparation generates all, and only, the pairs (actually, the *triples*, as we will see) that will be useful to the task of exploring all ways of generating a witness.

Formally, to every pair (CP', RP'), where $CP' \subseteq CP$ and $RP' \subseteq RP$, we associate a *characteristic pattern* $cp(CP', RP')$ that describes all strings (maybe none) that match every pattern in (CP', RP') and no pattern in $(CP \setminus CP', RP \setminus RP')$, as follows.

Definition 13 (Characteristic pattern). Given an object group $\{\text{type(Obj)}, CP, RP, \text{pro}^{Max}_{min}\}$ and two subsets $CP' \subseteq CP$ and $RP' \subseteq RP$, the characteristic pattern $cp(CP', RP')$ is defined as follows:

$$\begin{aligned} & cp(CP', RP') \\ & \quad = \ (\textstyle\bigsqcap_{\text{props}(p:\_) \in CP'} p\,) \sqcap (\textstyle\bigsqcap_{\text{props}(p:\_) \in (CP \setminus CP')} \overline{p}\,) \\ & \qquad \sqcap (\textstyle\bigsqcap_{(\text{pattReq}(r:\_) \in RP'} r\,) \sqcap (\textstyle\bigsqcap_{(\text{pattReq}(r:\_) \in (RP \setminus RP')} \overline{r}\,) \end{aligned}$$

Consider for example the following object group, corresponding, modulo variable names, to a fragment of our running example (Figure 5(d)):

$$\{\text{type(Obj)}, \text{props}(\text{"}b\text{"} : x), \text{pattReq}(\text{"}a\text{"} : y1), \text{pattReq}(\text{"}a.*\text{"} : y2)\}$$

For space reason, we adopt the following abbreviations for the assertions that belong to CP and RP:

$$pb = \text{props}(\text{"}b\text{"} : x), \qquad ra = \text{pattReq}(\text{"}a\text{"} : y1),$$
$$ras = \text{pattReq}(\text{"}a.*\text{"} : y2)$$

Here we have 2^3 pairs (CP', RP') that are elementwise included in (CP, RP), each pair defining its own characteristic pattern; for each pattern we indicate an equivalent extended regular expression (".+" stands for any non-empty string) or $\emptyset$ when the pattern has an empty language:

$$\begin{array}{lcll}
cp(\{\!|\,|\!\}, \{\!|\,|\!\}) & = & \overline{b} \sqcap \overline{a} \sqcap \overline{a.*} & \equiv \overline{b} \sqcap \overline{a.*} \\
cp(\{\!|\,|\!\}, \{\!|ra|\!\}) & = & \overline{b} \sqcap a \sqcap \overline{a.*} & \equiv \emptyset \\
cp(\{\!|\,|\!\}, \{\!|ras|\!\}) & = & \overline{b} \sqcap \overline{a} \sqcap a.* & \equiv a.+ \\
cp(\{\!|\,|\!\}, \{\!|ra, ras|\!\}) & = & \overline{b} \sqcap a \sqcap a.* & \equiv a \\
cp(\{\!|pb|\!\}, \{\!|\,|\!\}) & = & b \sqcap \overline{a} \sqcap \overline{a.*} & \equiv b \\
cp(\{\!|pb|\!\}, \{\!|ra|\!\}) & = & b \sqcap a \sqcap \overline{a.*} & \equiv \emptyset \\
cp(\{\!|pb|\!\}, \{\!|ras|\!\}) & = & b \sqcap \overline{a} \sqcap a.* & \equiv \emptyset \\
cp(\{\!|pb|\!\}, \{\!|ra, ras|\!\}) & = & b \sqcap a \sqcap a.* & \equiv \emptyset
\end{array}$$

All different pairs (CP', RP') define languages that are mutually disjoint by construction, but many of these are empty, as in this example. The non-empty languages cover all strings, by construction, hence they always define a partition of the set of all strings.

Consider now a member $k : J$ which we may use to build a witness of the object group. The key k matches exactly one non-empty characteristic pattern $cp(CP', RP')$, hence J must be a witness for all variables x_i such that $\text{props}(p_i : x_i) \in CP'$, since each relevant constraint must be satisfied, but, as far as the assertions $\text{pattReq}(r_j : y_j) \in RP'$ are concerned, there is much more choice. If J is a witness for every such y_j, then this member satisfies all requirements in RP'. But it may be the case that some of these y_j's are mutually exclusive, hence we must choose which ones will be satisfied by J. Or, maybe, none of the y_j is satisfied by J, but we may still use $k : J$ in order to satisfy a pro^{∞}_{m} requirement with $m \neq 0$. Hence, in order to explore all different ways of generating a member $(k : J)$ for a witness of the object group, we must choose a pattern $cp(CP', RP')$, and a subset RP'' of RP' that we require J to satisfy. Hence, we define a *choice* to be a triple (CP', RP', RP''), with $RP'' \subseteq RP'$. The $(CP', RP', \_)$ part specifies the pattern that is satisfied by k, while the $(CP', \_, RP'')$ part, with $RP'' \subseteq RP'$, specifies the variables that J must satisfy.

We also distinguish *R-choices*, where RP'' is not empty, hence they are useful in order to satisfy some requirements in RP, and *non-R-choices*, where RP'' is empty, hence they can only be used to satisfy a pro^{∞}_{m} requirement. The only choices that may describe a member are those where the set of strings $L(cp(CP', RP'))$ is not empty; we call them *non-cp-empty choices.*

Definition 14 (Choice, R-Choice, cp-empty choice). Given an object group $\{\ \text{type(Obj)}, CP, RP, \text{pro}^{M}_{m}\ \}$ with constraining part $CP = \{\!|\text{props}(p_i : x_i) \mid i \in 1..m|\!\}$ and $RP = \{\!|\text{pattReq}(r_j : y_j) \mid j \in 1..n|\!\}$, a choice is a triple (CP', RP', RP'') such that $CP' \subseteq CP$, $RP'' \subseteq RP' \subseteq RP$. The *characteristic pattern* $cp(CP', RP', RP'')$ of the choice is defined by its first two components, as follows:

$$cp(CP', RP', RP'') = cp(CP', RP')$$

The *schema* of the choice $s(CP', RP', RP'')$ is defined by the first and the third component, as follows:

$$s(CP', RP', RP'') \ = \bigwedge_{\text{props}(p:x) \in CP'} x \ \wedge \bigwedge_{\text{pattReq}(r:y) \in RP''} y$$

A choice is *cp-empty* if $L(cp(CP', RP', RP''))$ is empty, is *non-cp-empty* otherwise.

A choice is an *R-choice* if $RP'' \neq \{\!|\,|\!\}$, is a *non-R-choice* otherwise.

In the object group of our previous example we have 4 non-cp-empty pairs, $(\{\!|\,|\!\}, \{\!|\,|\!\})$, $(\{\!|pb|\!\}, \{\!|\,|\!\})$, $(\{\!|\,|\!\}, \{\!|ras|\!\})$, $(\{\!|\,|\!\}, \{\!|ra, ras|\!\})$, which correspond to the following 8 non-cp-empty choices – for each, we indicate the corresponding schema.

$$\begin{array}{lcll}
s(\{\!|\,|\!\}, \{\!|\,|\!\}, \{\!|\,|\!\}) & = & x_{\mathbf{t}} & \text{non-R-choice} \\
s(\{\!|pb|\!\}, \{\!|\,|\!\}, \{\!|\,|\!\}) & = & x & \text{non-R-choice} \\
s(\{\!|\,|\!\}, \{\!|ras|\!\}, \{\!|\,|\!\}) & = & x_{\mathbf{t}} & \text{non-R-choice} \\
s(\{\!|\,|\!\}, \{\!|ras|\!\}, \{\!|ras|\!\}) & = & y2 & \text{R-choice} \\
s(\{\!|\,|\!\}, \{\!|ra, ras|\!\}, \{\!|\,|\!\}) & = & x_{\mathbf{t}} & \text{non-R-choice} \\
s(\{\!|\,|\!\}, \{\!|ra, ras|\!\}, \{\!|ra|\!\}) & = & y1 & \text{R-choice} \\
s(\{\!|\,|\!\}, \{\!|ra, ras|\!\}, \{\!|ras|\!\}) & = & y2 & \text{R-choice} \\
s(\{\!|\,|\!\}, \{\!|ra, ras|\!\}, \{\!|ra, ras|\!\}) & = & y1 \wedge y2 & \text{R-choice}
\end{array}$$

The schema of a choice is always a conjunction of variables, say $x_1 \wedge \ldots \wedge x_n$. During bottom-up generation, we need to know which non-cp-empty choices have a witness in the current assignment A^i, hence we need to associate every non-cp-empty choice with just one variable, not with a conjunction. Hence, we need to create a new variable y for each conjunction $x_1 \wedge \ldots \wedge x_n$ that we have never seen before, then we execute GDNF normalization over $x_1 \wedge \ldots \wedge x_n$, transforming it into a guarded disjunction of typed groups S, then we add $y : S$ to the current environment and we apply *preparation* again to this new variable; we call this process *and-completion.* In the example above, this may be the case for $y1 \wedge y2$, unless $y1 \wedge y2$ is Boolean-equivalent to some variable that already exists.

Preparation can be regarded as a sophisticated form of and-elimination. Here, *and-completion* plays the same role that not-completion plays for not-elimination: it creates the new variables that we need in order to push conjunction through the object group operators. But, crucially, and-completion is *lazy*: we do not pre-compute every possible conjunction, but only those that are really needed by some specific non-cp-empty choice. This laziness is crucial for the practical feasibility of the algorithm: when different constraints, or requirements, are associated to disjoint patterns, we have very few non-cp-empty choices, and in most cases they do not need any fresh variable, as in the example. Despite laziness, this prepare-generate-normalize-prepare loop can still generate a huge number of variables. We keep their number under control using the ROBDDTab data structure that we introduced in Section 7.1, which allows us to create a new variable only when none of the existing variables is boolean-equivalent to its body; this crucial optimization also ensures that this phase can never generate an infinite loop.

Hence, object preparation proceeds as follows:

(1) determine the set of non-cp-empty pairs (CP', RP'), that is the pairs such that $cp(CP', RP')$ is not empty;
(2) for each non-cp-empty pair (CP', RP') compute the corresponding choices (CP', RP', RP'') and, if the variable intersection $s(CP', RP', RP'')$ has no equivalent variable in the environment, add a new variable $x : s(CP', RP', RP'')$ to the environment, apply GDNF reduction to $s(CP', RP', RP'')$, apply preparation to the GDNF-reduced conjunction.

When we describe object generation, we will show how the set of all prepared choices can be used in order to enumerate all possible ways of generating a witness for an object group.

Step (1) has, in the worst case, an exponential cost, but in practice it is much cheaper: in the common case where every pattern matches a single string, a set of n properties and requirements generates at most $n+1$ non-empty pairs (one for each string plus one for the complement of the string set), n R-choices, and $n+1$ non-R-choices. Since before preparation we have at most $O(N)$ distinct variables (where N is the input size), step (2) may generate at most $O(2^N)$ new variables, each of which has a body which can be prepared in time $O(2^{poly(N)})$. Hence, the global cost of this phase is still $O(2^{poly(N)})$.

Our experiments show that this cost is, for most real-world schemas, tolerable.

Property 12. *Object preparation can be performed in $O(2^{poly(N)})$ time.*

Remark 1. *In our implementation, the generation of all non-cp-empty pairs is not performed by brute force enumeration, but using an algorithm based on the following schema: it matches every pair of patterns $r1$ and $r2$ coming from either CP and RP and, in case the two are neither equal nor disjoint, splits them into three patterns $r1 \sqcap \overline{r2}$, $r1 \sqcap r2$ and $\overline{r1} \sqcap r2$. This algorithm has a cost that is quadratic in the number of non-empty pairs that are generated. Hence, it is $O(2^n)$ in the worst case but is just quadratic in the typical case, the one where the number of non-empty pairs is linear in the size of the object group.*

8.3.4 *Witness generation from a prepared object group.*

After the object group has been prepared once for all, at each pass of bottom-up witness generation we use the following sound and generative algorithm, listed as Algorithm 2, to compute a witness for the prepared object group starting from the current assignment A^i.

In a nutshell, we (1) pick a list of choices that contains enough R-choices to satisfy all requirements — each choice will correspond to one field in the generated object, and vice versa; (2) we verify that the list is *pattern-viable*, i.e., that it does not require two fields with the same name; (3) to satisfy any unfulfilled pro_m^∞ requirement, we add some non-R-choices, still keeping the choice list *pattern-viable*, as defined above. In order to keep the search space in $O(2^{poly(N)})$, we limit ourselves to the subset of the *disjoint* solutions, and we prove that it is big enough to have a complete algorithm.

In greater detail, consider a generic object group with the form $\{\ \mathrm{type(Obj)}, CP, RP, \mathrm{pro}_m^M\ \}$ and assume that the corresponding non-cp-empty choices have been prepared.

To generate an object, we first choose a list of choices that satisfies all of RP. To reduce the search space, we first observe that a single object can be described by many different choice lists. For example, assume that '1' belongs to both $[[x]]_E$ and $[[y]]_E$ and assume that:

$$rx = \mathrm{pattReq}("a|b" : x)$$
$$ry = \mathrm{pattReq}("a|b" : y)$$
$$RP = \{\ rx, ry\ \}$$

then $\{\ "a"\ :\ 1, "b"\ :\ 1\ \}$ is described by each the following four choice lists (and by others), where every choice could be used to generate/describe each of the two members:

$$\begin{array}{lll}
CL_1 = [\ (\{\!|\,|\!\}, \{\!|rx, ry|\!\}, \{\!|rx|\!\}), & (\{\!|\,|\!\}, \{\!|rx, ry|\!\}, \{\!|ry|\!\}) &] \\
CL_2 = [\ (\{\!|\,|\!\}, \{\!|rx, ry|\!\}, \{\!|rx, ry|\!\}), & (\{\!|\,|\!\}, \{\!|rx, ry|\!\}, \{\!|\,|\!\}) &] \\
CL_3 = [\ (\{\!|\,|\!\}, \{\!|rx, ry|\!\}, \{\!|rx, ry|\!\}), & (\{\!|\,|\!\}, \{\!|rx, ry|\!\}, \{\!|rx, ry|\!\}) &] \\
CL_4 = [\ (\{\!|\,|\!\}, \{\!|rx, ry|\!\}, \{\!|rx, ry|\!\}), & (\{\!|\,|\!\}, \{\!|rx, ry|\!\}, \{\!|rx|\!\}) &]
\end{array}$$

This example shows that we do not need to explore any possible choice list, but just *enough* choice lists to generate *all* witnesses. To this aim, we focus on *disjoint solutions*, defined as follows, whose completeness will be proved in Theorem 18.

Definition 15 (Disjoint solution, Minimal disjoint solution). Fixed a set of requirements RP, a size limit M, and a set of choices $\mathbb{C}$, a multiset $\mathbb{C}' = \{\!|(C_l, R'_l, R''_l) \mid l \in L|\!\}$ with elements in $\mathbb{C}$ is a *solution* (for the fixed RP and M) iff:

$$\bigcup_{l \in L} R''_l = RP \ \text{ and } \ |\mathbb{C}'| \leq M$$

The solution is *disjoint* if: $i \neq j \Rightarrow R''_i \cap R''_j = \emptyset$.

The solution is *minimal* if every choice in $\mathbb{C}'$ is an R-choice.

In the previous example, only CL_1 and CL_2 are disjoint, and only CL_1 is disjoint and minimal.

Every object described by a solution for an object group is a witness for the that group.

Definition 16 (describes-in-A). A choice $C = (CP', RP', RP'')$ for a prepared object group *describes in an assignment A* a field $k : J$, iff $k \in L(cp(C))$ and $J \in A(var(C))$. A choice list $\mathbb{C}$ *describes* in A an object J if there is a bijection mapping each field $k : J'$ in J to a choice C in $\mathbb{C}$ such that C describes $k : J'$.

Property 13. *For any prepared object group*

$$S = \{\mathsf{type}(\mathsf{Obj}), \mathsf{CP}, \mathsf{RP}, \mathsf{pro}_{\mathsf{m}}^{\mathsf{M}}\}$$

with the corresponding environment E and choices $\mathbb{C}$, if $\mathbb{C}'$ is a choice list over $\mathbb{C}$ with $m \leq |\mathbb{C}'| \leq M$ that is a solution for RP, if A is sound for E, and if J is described in A by $\mathbb{C}$, then $J \in [[S]]_E$.

Object generation depends on the current assignment A^i. We say that a variable x is *Populated* (in A^i) when $A^i(x) \neq \emptyset$, and is *Open* otherwise. We say that a choice is *Populated*, or *Open*, when its schema variable is *Populated*, or is *Open*. In order to generate a witness, we first generate a *disjoint minimal solution* for *RP* with bound M, only using R-choices that are *Populated*. Then, in order to deal with the constraint that all names in an object are distinct, we check that the solution is *pattern-viable*. Informally, pattern-viability ensures that, if we have n choices in the solution with the same characteristic pattern cp, then the language of cp has at least n different strings, which can be used to build n different members corresponding to those n choices. We will exemplify the issue after the definition.

Definition 17 (Pattern-viable). A set of choices $\mathbb{C}$ is pattern-viable iff for every pair (CP', RP'), the number of choices in $\mathbb{C}$ with shape $(CP', RP', \_)$ is smaller than the number of words in $L(cp(CP', RP'))$:

$$\forall CP', RP'.$$
$$|\, \{\!|(CP', RP', RP'') \mid (CP', RP', RP'') \in \mathbb{C}|\!\}\, | \;\leq\; |\, L(cp(CP', RP'))\, |$$

For example, the following choice list $\mathbb{C}$ is not viable since it describes an object with two members that share the same characteristic pattern "a" that only contains one string:

$$rx = \mathsf{pattReq}(\text{"}a\text{"} : x),\ \ ry = \mathsf{pattReq}(\text{"}a\text{"} : y)$$
$$\mathbb{C} = [\quad (\{\!|\,|\!\}, \{\!|rx, ry|\!\}, \{\!|rx|\!\}), \quad (\{\!|\,|\!\}, \{\!|rx, ry|\!\}, \{\!|ry|\!\}) \quad]$$

But it would be viable if the pattern "a" were substituted by "$a|b$".

Finally, for each viable disjoint solution, we check whether it also satisfies the pro_m^{∞} requirement (line 6 of Algorithm 2). If it does not, we try and extend the solution by adding some *Populated* non-R-choices (line 7). Observe that the disjoint solution contains each R-choice (CP', RP', RP'') at most once, because of disjointness; however, we can add the same non-R-choice as many times as we need in order to reach m members. A non-R-choice C can only be added if the result remains viable; hence, a minimal disjoint solution $\mathbb{C}$ may have a viable extension $\mathbb{C}'$ of length m, obtained by adding a multiset of non-R-choices (lines 6-13), or it may not have such a viable extension, and then we need to start from a different minimal solution. If no viable disjoint solution admits a viable extension of length at least m, then the algorithm returns "Open" (according to the current assignment). Otherwise, we use the extended solution $\mathbb{C}'$ to build a witness: for each choice $C \in \mathbb{C}'$, we generate a name k satisfying $cp(C)$, we pick a value J from $A^i(var(C))$, and the set of members $k : J$ that we obtain is a witness for the object group. When n different choices inside $\mathbb{C}'$ have the same characteristic pattern, we generate n different names, which is always possible since the solution is viable — this is the n-enumeration problem for EEREs that we introduced in Section 4.4.

Algorithm 2: Object witness generation

```
1  Gen(RPart, WitRChoices,WitNonRChoices, min, Max,)
2      for Solution in minDisjointSols (WitRChoices,RPart,Max) do
3          if (viable(Solution)) then
4              missing := min − size(Solution);
5              nonViableChoices := ∅;
6              while (missing > 0 and nonViableChoices != WitNonRChoices)
                 do
7                  choose NRC from (WitNonRChoices-nonViableChoices);
8                  if (viable([NRC]++Solution)) then
9                      Solution := [NRC]++Solution;
10                     missing := missing-1;
11                 else nonViableChoices := [NRC]++nonViableChoices;
12             if (missing == 0) then
13                 return ("Populated", WitnessFrom(Solution));
14     return ("Open");
```

Theorem 18 (Soundness and generativity). *Algorithm* Gen *is sound and generative.*

Proof. Our algorithm is sound by construction. For generativity, assume that the object group

$$S = \{\ \mathsf{type}(\mathsf{Obj}), CP, RP, \mathsf{pro}_{min}^{Max}\ \}$$

has a witness of depth $d+1$ in $[[S]]_E$. Assume that A is d-witnessed for E. We want to prove that Gen, applied to S and A, will generate at least one witness. Let

$$J = \{a_1 : J_1, \ldots, a_l : J_l\}$$

be a witness for S in E with depth $d+1$. We can now extract from

$$\{a_1 : J_1, \ldots, a_l : J_l\}$$

a set of choices (C_i', R_i', R_i'') with $i \in \{1..l\}$, as follows. C_i' and R_i' are defined by the only pair (C_i', R_i') whose language includes a_i. In order to define R_i'', we observe that, since J satisfies RP, then, we can associate to each S in RP one member i such that $a_i : J_i$ satisfies S — if many such members exist, we just choose one. The inverse of this relation associates to each member i a subset R_i'' of R_i'. The collection of choices $\mathbb{C} = \{\!|\ (C_i', R_i', R_i'') \mid i \in \{1..l\}\ |\!\}$ that we have defined is actually a multiset, since a non-R-choice may appear more than once, and is a disjoint solution since, by construction, $\bigcup_{i \in \{1..l\}} R_i'' = RP$, $l \leq Max$, and $1 \leq i < j \leq l \Rightarrow R_i'' \cap R_j'' = \emptyset$, since every requirement is mapped to exactly one member. We now prove that all these choices are *Populated* in A. To this aim, consider a choice $C = (C_i', R_i', R_i'')$ in $\mathbb{C}$ and the field $a_i : J_i$ that we used to define it. By construction, the schema $s(C)$ is the conjunction of the variables of all constraints C_i' that must be satisfied by and J that is associated to a_i in any witness of S, plus the variables a set of requirements R_i'' whose variables are satisfied by J_i, hence $J_i \in [[s(C)]]_E$, hence, by definition of $var(C)$, $J_i \in [[var(C)]]_E$. Since J has depth $d+1$, then $\delta(J_i) \leq d$, hence $A(var(C)) \neq \emptyset$ since A is d-witnessed, hence every choice in $\mathbb{C}$ is *Populated* in A.

Now we prove that our algorithm would generate at least one subsequence of $\mathbb{C}$ that is a solution, unless it stops since it is able to generate a different solution; in both cases, our algorithm generates a solution for the group.

To prove this, we remove every non-R-choice from $\mathbb{C}$, and so we get a collection $\mathbb{C}'$ that is a minimal disjoint solution. If $min > |\mathbb{C}'|$, then we choose $min - |\mathbb{C}'|$ non-R-choices out of $\mathbb{C}$ and add them to $\mathbb{C}'$. Being a subset of $\mathbb{C}$, the result is viable and, by construction, is an extension of a minimal disjoint solution $\mathbb{C}'$ with a multiset of non-R-choices. Our algorithm scans every such extension of every minimal disjoint solution, hence, if it is not stopped because it finds a different solution, it finds this one, and it generates a corresponding witness. □

Property 14 (Complexity). *Given a schema of size N, each run of the* Gen *algorithm has a complexity in $O(2^{poly(N)})$.*

Proof. Let N we the size of the original schema. Let us first focus on a single, arbitrary, group. For any object group, RP has at most N elements, and any choice has a size that is $O(N)$. Let M be an upper bound for the number of non-empty choices for an arbitrary object group. Since every minimal disjoint solution contains at most $|RP| \leq N$ choices, we can generate all minimal disjoint solutions by scanning the list of all N-tuples of choices, which can be done in time $O(M^N)$. We then need to scan the list of all non-R-choices for at most min times, which adds another $O(M^N)$ factor, since $min \leq N$ by the linear constants assumption, hence we arrive at $O(M^{2N})$ solutions. For every solution that contains i choices, we need to solve at most i times the i-enumeration problem, with $i \leq N$, in order to verify viability and to generate the witness when a witness exists. The pattern expression $cp(C)$ of each choice C of the solution has a size that is in $O(poly(N))$, hence running i times the i-enumeration problem has a cost that is $O(2^{poly(N)})$, hence we can examine $O(M^{2N})$ solutions in time $O(M^{2N} \cdot poly(N) \cdot 2^{poly(N)})$. Since M is in $O(2^{poly(N)})$, each pass of object generation is in $O(2^{poly(N)})$ for each prepared object group. Since we have less then $O(2^{poly(N)})$ groups, each pass of object generation is in $O(2^{poly(N)})$. □

8.4 Array group preparation and generation

8.4.1 Constraints and requirements. As with objects, we say that an assertion $S = \mathsf{contAfter}(i^+ : x)$ or $S = \mathsf{cont}_i^\infty(x)$ with $i > 0$ is a *requirement*, since it is not satisfied by [ ] and, if J^+ extends J, then $J \in [\![S]\!]_E \Rightarrow J^+ \in [\![S]\!]_E$.

We say that an assertion $S = \mathsf{item}(l : x)$, $S = \mathsf{items}(i^+ : x)$, or $S = \mathsf{cont}_0^j(x)$ is a *constraint*, since it is satisfied by [ ] and if J^+ extends J, then $J^+ \in [\![S]\!]_E \Rightarrow J \in [\![S]\!]_E$.

An assertion $S = \mathsf{cont}_i^j(x)$ with $i \neq 0$ and $j \neq \infty$ combines a requirement and a constraint.

8.4.2 Array group preparation. An array group is a set of assertions with the following shape:

$$\{\ \mathsf{type}(\mathsf{Arr}), IP, AP, KP\ \}$$

Here, IP is a set of *item* constraints $\mathsf{item}(l : x)$ and $\mathsf{items}(i^+ : x)$, AP is a set of *contains-after* requirements with shape $\mathsf{contAfter}(l^+ : x)$, KP is a set of *counting* assertions $\mathsf{cont}_i^j(x)$, where every assertions combines a requirement $\mathsf{cont}_i^\infty(x)$ and a constraint $\mathsf{cont}_0^j(x)$.[8]

In theory, arrays and objects are almost identical, since they are both finite mappings from labels to values, but arrays have some extra issues:

(1) Arrays have a domain downward closure constraint, that specifies that, when a value is associated to a label $n+1$, then a value is associated to n as well, for every $n \geq 1$; objects do not have anything similar.
(2) The $\mathsf{cont}_i^j(x)$ operator specifies an upper bound, and requires counting, while $\mathsf{pattReq}(a : x)$ only specifies the existence of at least one member matching a with schema x, with no upper bound and no counting ability.

Consider for example the following array group.

$$\{\ \mathsf{type}(\mathsf{Arr}), \mathsf{item}(2 : x), \mathsf{contAfter}(0^+ : y), \mathsf{cont}_1^1(z), \mathsf{cont}_2^2(x_\mathbf{t})\}$$

It describes an array of exactly two elements. The one at position 2 must satisfy x. At least one of the two elements must satisfy y. One, but only one, of the two elements must satisfy z.

Let us say that an array has shape $[S_1, \ldots, S_k]$ if it contains exactly k items $[J_1, \ldots, J_k]$, and if each item J_i satisfies S_i. Then, the group above is satisfied by arrays with one of the following four shapes:

$$\begin{array}{llll} [\ y \wedge z, & x \wedge co(z)\ ], & [\ y \wedge co(z), & x \wedge z\ ], \\ [\ z, & x \wedge y \wedge co(z)\ ], & [\ co(z), & x \wedge y \wedge z\ ] \end{array}$$

We recognize the two problems that we have seen with objects: interaction between constraints and requirements, resulting in conjunctions of x with other variables in position 2, and the possibility of one element to satisfy two requirements, resulting in $y \wedge z$ conjunctions, but we have the extra problem of the upper bound, that results in the presence of the dual variable $co(z)$ in some positions.

Hence, our algorithm to prepare arrays and to generate the corresponding witnesses is somehow different from that of objects, although similar in spirit. It obviously differs in the presence of dual variables like $co(z)$, motivated by upper bounds, but also differs in the strategy that we use to explore the space of witnesses. Instead of starting the exploration from the requirements, hence from the "first choices", here we are guided by the domain closure constraint, hence we start the exploration from the first position of the array.

We need to define some terminology. We first define a notion of head-length for an array group S (Definition 19): intuitively, when the head-length of S is h, then, for any witness J of S, if the elements of J from position $h+1$ onwards — which constitute the *tail* of J — are permuted, then J is still a witness; the elements in positions 1 to h constitute the *head*, and their position may matter. For example, an array group $\{\mathsf{type}(\mathsf{Arr}), \mathsf{item}(3 : x)\}$ has head-length 3. The head-length n may be 0, and actually this is the most common head-length that we encounter in practice. The interval of an assertion $In(S)$ is the interval of positions of the array that the assertion describes, which may belong to the head of the group, to the tail, or may cross both.

Definition 19 ($[i, j]$, $HL(S)$, $In(S)$). $[i, j]$, with $i \in \mathbb{N}$, $j \in \mathbb{N}^\infty$, denotes the interval between i and j, which is infinite when $j = \infty$,

[8] For the sake of simplicity, in our formal treatment we do not distinguish $\mathsf{cont}_i^j(x_\mathbf{t})$ from the other counting assertions, where $x_\mathbf{t}$ here indicates the variable whose body is $\mathbf{t}$, although in the implementation we actually exploit its special properties for efficiency reasons.

and is empty when $i > j$. The head-length $HL(S)$ and the interval $In(S)$ of an array ITO S, and of an array group, are defined as follows:

$$
\begin{array}{lcl}
[i, j] & = & \{\!| \; l \mid l \in \mathbb{N}, \; i \leq l \leq j \; |\!\} \\
HL(\mathsf{item}(l : S)) & = & l \\
HL(\mathsf{items}(i^+ : S)) & = & i \\
HL(\mathsf{contAfter}(l^+ : S)) & = & l \\
HL(\mathsf{cont}_i^j(S)) & = & 0 \\
HL((\{\mathsf{type}(\mathsf{Arr}), IP, AP, KP\}) & = & \max_{S \in IP \cup AP}(HL(S)) \\
In(\mathsf{item}(l : S)) & = & [l, l] \\
In(\mathsf{items}(i^+ : S)) & = & [i + 1, \infty] \\
In(\mathsf{contAfter}(l^+ : S)) & = & [l + 1, \infty] \\
In(\mathsf{cont}_i^j(S)) & = & [1, \infty]
\end{array}
$$

Property 15 (Irrelevance of tail position). *If S is an array typed group, $J = [J_1, \ldots, J_n] \in [\![S]\!]_E$, for all i, j with $HL(S) < i \leq j \leq n$, if J' is obtained from J by exchanging J_i with J_j, then $J' \in [\![S]\!]_E$.*

In order to define a choice we need a last definition: for a set of assertions $\mathcal{S}$, we define its restriction to $[i, j]$, denoted by $\mathcal{S} \cap [i, j]$, as the subset of $\mathcal{S}$ containing the assertions whose interval intersects $[i, j]$.

Definition 20 ($\mathcal{S} \cap [i, j]$).

$$\mathcal{S} \cap [i, j] = \{\!| \; S \mid S \in \mathcal{S}, \; ([i, j] \cap In(S)) \neq \emptyset \; |\!\}$$

Now, we define a choice for an array group IP, AP, KP with $h = HL(IP \cup AP)$, as a quintuple $([i, j], IP', AP', KP^+, KP^-)$ where:

(1) either $i = j \leq h$ or $i = h + 1$ and $j = \infty$, hence a choice describes either a single element $[i, i]$ in the head of the array group, or an element in the tail interval $[h + 1, \infty]$;
(2) IP' is equal to $IP \cap [i, j]$;
(3) AP' is a subset of $AP \cap [i, j]$;
(4) KP^+ is a subset of KP;
(5) KP^- is a subset of $KP \setminus KP^+$.

Hence, for each interval $[i, j]$, the element IP' is fixed, but we may still have many choices for AP', KP^+ and KP^-. Intuitively, a choice $([i, j], IP', AP', KP^+, KP^-)$ describes an element in a position that belongs to $[i, j]$, that satisfies all the constraints in $IP \cap [i, j]$, that satisfies the assertions in AP' and in KP^+, and does not satisfy any assertion in KP^-. With respect to object choices, here the label is not represented by a pair of sets of assertions (CP', RP'), but just by an interval $[i, j]$, while the schema is a bit more complex since it has three positive components IP', AP' and KP^+, playing the roles of CP' and RP'', but also a negative component KP^-. Observe that, while IP' and AP' are restricted to the assertions that apply to $[i, j]$, we do not have this restriction for KP, since every counting assertion analyzes all positions of the array. Hence, the schema of a choice is defined as follows.

Definition 21 ($s([i, j], IP', AP', KP^+, KP^-)$).

$$
\begin{array}{l}
s([i, j], IP', AP', KP^+, KP^-) \\
\quad = \; (\bigwedge_{(\mathsf{item}(l:x)) \in IP'} x) \wedge (\bigwedge_{(\mathsf{items}(i^+:x)) \in IP'} x) \\
\quad\quad \wedge \, (\bigwedge_{(\mathsf{contAfter}(l^+:x)) \in AP'} x) \\
\quad\quad \wedge \, (\bigwedge_{(\mathsf{cont}_i^j(x)) \in KP^+} x) \wedge (\bigwedge_{(\mathsf{cont}_i^j(x)) \in KP^-} co(x))
\end{array}
$$

As with object groups, a generative exploration of the space of all possible solutions does not require the generation of all possible choices, and different strategies are possible. In our implementation, we limit ourselves to the choices where $KP^- = KP \setminus KP^+$, which we call here the co-maximal choices. We prove later that this strategy ensures the generativity property that we need. More optimized strategies would be possible, but we believe that they are not worth the effort, since in practice the array types that we have to deal with are usually quite simple.

Hence, array preparation consists of the following steps.

(1) compute $h = HL(IP, AP)$;
(2) for each interval $[i, i]$ corresponding to an $i \in [1, h]$, and for each subset AP' of AP and KP' of KP produce the corresponding co-maximal choice:

$$([i, i], IP \cap [i, i], AP', KP', KP \setminus KP')$$

and check whether the variable intersection that corresponds to the schema of that choice is equivalent to some existing variable, and, if not, create a new variable that will become the schema of that choice, and apply preparation to the body of this new variable, as in the case of object preparation;
(3) do the same for the interval $[h + 1, \infty]$, and for each subset AP' of AP and KP' of KP.

As happens with object preparation, also array preparation has an exponential cost that is quite low in practice, since in the vast majority of cases the head-length of array groups is zero or one, and the set $AP \cup KP$ is either empty or a singleton. For this reason, we did not put any special effort into the optimization of this phase.

Property 16. *Array preparation can be performed in time $O(2^N)$, where N is the size of the input schema.*

8.4.3 Witness generation from a prepared array group.

Array preparation applied to an array group $\{ \; \mathsf{type}(\mathsf{Arr}), IP, AP, KP \; \}$ with head-length h produces a set of co-maximal choices, each characterized by an interval $[i, j]$ with shape $[i, i]$ when $i \leq h$, or $[h + 1, \infty]$ otherwise, and by two subsets AP', KP' of AP, KP. We indicate with $C(i, AP', KP')$ the co-maximal choice that is characterized by these three parameters, and with $s(i, AP', KP')$ and $s(i, AP', KP')$ its schema and the associated variable, as follows:

$$
\begin{array}{l}
C(i, AP', KP') \quad \text{with } 1 \leq i \leq h \\
\quad = \; ([i, i], IP \cap [i, i], AP', KP', KP \setminus KP') \\
C(h + 1, AP', KP') \\
\quad = \; ([h + 1, \infty], IP \cap [h + 1, \infty], AP', KP', KP \setminus KP') \\
s(i, AP', KP') \; = \; s(C(i, AP', KP')) \\
var(i, AP', KP') \; = \; var(C(i, AP', KP'))
\end{array}
$$

A choice $C(i, AP', KP')$ is a *head choice* when $i \leq h$, and is a *tail choice* when $i = h + 1$. At any pass of the generation algorithm, a choice is *Populated* or *Open*, depending on its schema variable.

Given a list of choices $\mathbb{C}$ and a set of contains-after and counting assertions $\{\!|AP, KP|\!\}$ (where $\{\!|AP', KP'|\!\}$ abbreviates $AP' \cup KP'$), we define the *incidence* of $\mathbb{C}$ over $\{\!|AP, KP|\!\}$ as a function that maps each $S \in \{\!|AP, KP|\!\}$ to the number of elements of $\mathbb{C}$ that are guaranteed to satisfy S, as follows:

$$\begin{array}{ll} \text{if } S \notin (AP' \cup KP'): & I_{C(i,AP',KP')}(S) = 0 \\ \text{if } S \in (AP' \cup KP'): & I_{C(i,AP',KP')}(S) = 1 \\ I_{[C_1,\ldots,C_n]}(S) = \sum_{i\in\{1..n\}} I_{C_i}(S) & \end{array}$$

We say that a list of choices $\mathbb{C}$ is a solution for $\{\!|AP, KP|\!\}$ when the incidence of the list satisfies all requirements and does not violate any constraint, as follows.

Definition 22 (Well formed list, Solution). A list of choices for an array group is well-formed for head-length h iff

(1) every choice in the list has either an interval $[i, i]$ with $i \leq h$ or the interval $[h + 1, \infty]$;

(2) if two consecutive choices in the list have intervals $[i, \_]$ and $[j, \_]$, then either $j = i + 1$ or $j = i = h + 1$.

For example, $[\ ([3, 3], \ldots), [4, 4], \ldots), ([5, \infty], \ldots), ([5, \infty], \ldots)\ ]$, $[\ ([5, \infty], \ldots)\ ]$, and $[\ ]$ are well formed for head-length 4.

Definition 23 (Solution). Fixed an array group { type(Arr), IP, AP, KP } with head-length h, a choice list $\mathbb{C}$ is a solution for the array group iff all the following hold:

(1) it is well formed for h;

(2) either $\mathbb{C}$ is empty or the first choice has interval $[1, \_]$;

(3) for every assertion $\mathsf{cont}_m^M(x) \in KP$ we have $I_{\mathbb{C}}(S) \leq M$;

(4) for every assertion $\mathsf{cont}_m^M(x) \in KP$ we have $I_{\mathbb{C}}(S) \geq m$;

(5) for every requirement $S \in AP$ we have $I_{\mathbb{C}}(S) > 0$.

Observe that an incidence $I_{\mathbb{C}}(S) = n$ guarantees that an array described by $\mathbb{C}$ has exactly n elements that satisfy S if $S \in KP$, and *at least* n elements that satisfy S if $S \in AP$. This happens by design, and is sufficient to guarantee the essential property that every array described by a solution is a witness for the corresponding group.

Definition 24 (describes-in-A). A choice $C = ([i, j], \ldots)$ for a prepared array group *describes in an assignment* A an element J_l of an array $[J_1, \ldots, J_n]$, iff $l \in [i, j]$ and $J \in A(var(C))$. A choice list $[C_1, \ldots, C_n]$ *describes* in A an array $J = [J_1, \ldots, J_n]$ if every C_l describes in A the element J_l.

Property 17. *For any prepared array group*

$$S = \{\mathsf{type(Arr), IP, AP, KP}\}$$

with the corresponding environment E and choices $\mathbb{C}$*, if A is sound for E, if the choice list* $\mathbb{C}'$ *over* $\mathbb{C}$ *is a solution for S, and if J is described in A by* $\mathbb{C}$*, then* $J \in [\![S]\!]_E$*.*

Proof. Consider a prepared group $S = \{\mathsf{type(Arr), IP, AP, KP}\}$ and the corresponding choices $\mathbb{C}$ and environment E. Let A be sound for E and assume that $\mathbb{C}' = [C_1, \ldots, C_n]$ describes $J = [J_1, \ldots, J_n]$.

By definition of $I_{\mathbb{C}'}(S)$, for any $S = \mathsf{contAfter}(i^+ : x) \in AP$, if $I_{\mathbb{C}'}(S) = k$, then there are exactly k choices C in $\mathbb{C}'$ such that $C = C(l, AP', KP')$, and $S \in AP'$. By definition of $s(C)$ and $var(C)$, for all of these choices we have that $s(C)$ is a conjunction of x with other variables, hence $[\![var(C)]\!]_E \subseteq [\![x]\!]_E$. For all of these choices, the corresponding J_l belongs to $A(var(C))$, since $\mathbb{C}'$ describes in A J. Since A is sound for E, we conclude that, for these choices, we have that $J_l \in [\![x]\!]_E$. Hence, if $I_{\mathbb{C}'}(S) > 0$ with $S = \mathsf{contAfter}(i^+ : x)$, we have at least one element of J which satisfies x. We must now prove that the position of that elements is greater than i. By definition of choice, every choice that includes $\mathsf{contAfter}(i^+ : x)$ has an interval that intersects $[i + 1, \infty]$. Since the head-length of the object group is at least i, every choice whose interval intersects $[i + 1, \infty]$ is either a head-choice with interval $[j, j]$ and $j > i$ or a tail choice with interval $[h + 1, \infty]$ and $h \geq i$. In both cases, every position described by that choice is strictly greater than i.

By definition of $I_{\mathbb{C}'}(S)$, for any $S = \mathsf{cont}_m^M(x) \in KP$, if $I_{\mathbb{C}'}(S) = k$, this implies that there are exactly k choices C in $\mathbb{C}'$ such that $C = C(l, AP', KP')$, and $S \in KP'$, and, as in the previous case, for all of these choices we have that $[\![var(C)]\!]_E \subseteq [\![x]\!]_E$. Since we only consider co-maximal choices, for all the other $n - k$ choices we have that $S \in KP^-$, hence for the other choices we have that $s(C)$ is a conjunction of $co(x)$ with other variables, hence $[\![var(C)]\!]_E \cap [\![x]\!]_E = \emptyset$. Since A is sound for E, and $\mathbb{C}'$ describes in A J, we conclude that *exactly* k elements of J belong to $[\![x]\!]_E$. Since $m \leq I_{\mathbb{C}'}(S) \leq M$, we conclude that J satisfies $\mathsf{cont}_m^M(x)$.

Consider any $S = \mathsf{item}(l : x) \in IP$ and any choice C whose interval intersects $[l, l]$. By construction, $[\![var(C)]\!]_E \subseteq [\![x]\!]_E$, hence, by soundness of A, the element described by C satisfies S.

Consider any $S = \mathsf{items}(i^+ : x) \in IP$ and any choice C whose interval intersects $[i + 1, \infty]$. By construction, $[\![var(C)]\!]_E \subseteq [\![x]\!]_E$, hence, by soundness of A, the element described by C satisfies S.

Hence, every assertion in {type(Arr), IP, AP, KP} is satisfied by J. ◻

We finally need a notion of *useful choices*, which is similar in spirit to the *R-choices* that we defined for the object case, and which will be crucial to ensure the termination of the algorithm: a choice C is *useful* for a set $\{\!|AP, KP|\!\}$ iff some assertion in $\{\!|AP, KP|\!\}$ is affected by C.

Definition 25 (useful choice). A choice $C(i, AP', KP')$ is *useful* for a set of assertions $\{\!|AP'', KP''|\!\}$ iff

$$(\{\!|AP', KP'|\!\} \cap \{\!|AP'', KP''|\!\}) \neq \emptyset.$$

We can now describe our algorithm.

Our algorithm *cList(hLen, aList, fLen, fInc, pChoices)* recursively solves the following generalized problem: assume you have a list of assertions *aList* and you already have a choice list *firstC* of length *fLen*, whose incidence on *aList* is *fInc*; find the rest of the list — that is, find a well formed choice list $\mathbb{C}$ such that the concatenation of *firstC* with $\mathbb{C}$ is a solution for *aList*.

If *aList* is already satisfied by *fInc*, then *cList* returns the empty choice list (line 2). Otherwise, for each C in *pChoices* that can describe position *fLen+1*, we try to solve the subproblem *cList(hLen, aList, fLen+1, fInc', pChoices')*, where *fInc'* is the incidence updated after C, and, when the position *fLen+1* belongs to the tail, *pChoice'* only contains the elements of *pChoice* that are still useful to solve *aList* after a *CLFirst* with incidence *fInc* — this reduction of *pChoice* will be commented later on. If such a C exists, and $\mathbb{C}$ is a solution for *cList(hLen, aList, fLen+1, fInc', pChoices')*, then we return $[C]$++$\mathbb{C}$ (lines 9-11). If *pChoices* contains no choice C such that *cList(hLen,* *aList, fLen+1, fInc', pChoices')* has a solution, then we return "unsatisfiable".

Hence, at each pass, we start from an assignment A, we collect all choices that are *Populated* wrt A in a list *pChoices*, and we invoke the algorithm *cList(head-length,o,⦃AP,KP⦄,allZeroes,pChoices)*. Termination is ensured by the fact that, once we arrive to the tail, we only keep the useful choices, hence every choice that is selected either (a) increments to one the incidence over an assertion $\mathsf{contAfter}(i^+ : x)$ whose incidence was zero, or (b) increments by one the incidence over an assertion $\mathsf{cont}_m^M(x)$ whose incidence was still below m, hence the algorithm stops after not more than *MaxSteps* steps:

$$MaxSteps \;=\; h + |AP| + \Sigma_{\mathsf{cont}_m^M(x) \in KP}\; m$$

Here, h is the head-length, $|AP|$ is an upper bound for the (a) steps, and $\Sigma_{\ldots}\; m$ is an upper bound for the steps of type (b). If the algorithm returns a solution, we use it to generate a witness by substituting each choice with a witness from the corresponding *Populated* schema.

Algorithm 3: Pseudo-code for array solution generation

```
cList(hLen, aList, fLen, fInc, pChoices)
    if emptyListSatisfies(aList, fInc) then return [ ];
    if fLen >= hLen then
        pChoices ← tailUsefulChoices(pChoices, aList, fInc, hLen);
    for C in pChoices where inInterval(hLen+1,C) do
        newFInc ← updateIncAfterChoice(aList, fInc, C);
        if maxViolated(aList, newFInc) then continue;
        else
            restSolution = cList(hLen, aList, fLen+1, newFInc, pChoices);
            if restSolution is not null then return ([C] ++ restSolution);
            else continue;
    return null;
tailUsefulChoices(choices, aList, fInc, hLen)
    result = [ ];
    for C in choices where start(C)==hLen+1 do
        if exists ContAftInC in APPrimeOf(C)
        where fInc(ContAftInC)=0 then
            add C to result;
        if exists MinMaxInC in KPPrimeOf(C)
        where min(MinMaxInC) > fInc(MinMax) then
            add C to result;
    return results;
```

This algorithm is sound and generative.

Property 18 (Soundness and generativity). *The algorithm* cList *is sound and generative.*

Proof. Assume that an array group $S = \{\, \mathsf{type}(\mathsf{Arr}), IP, AP, KP \,\}$ with head-length h has a witness with depth $d+1$, and consider such a witness $J = [J_1, \ldots, J_o]$. For every i of $\{1..o\}$, we define

$$A(i) \;=\; \{\!| \, S \;|\; S = \mathsf{contAfter}(l^+ : x),\; S \in AP,\; i > l,\; J_i \in [[x]]_E \,|\!\}$$

$$K(i) \;=\; \{\!| \, S \;|\; S = \mathsf{cont}_m^M(x),\; S \in KP,\; J_i \in [[x]]_E \,|\!\}$$

Now we build a choice list $\mathbb{C}$ that is derived from J, as follows.

We define an index i, initialized to 1, and a *cumulative incidence* function *in*, that maps every assertion to 0. If the function *in* satisfies already both AP and KP, then $\mathbb{C} = []$. Otherwise, we consider the choice $C(i, A(i), K(i))$. We say that a choice is useful for $\{\!|AP, KP|\!\}$ "after a list of choices described by *in*", if the choice contains some requirements from $\{\!|AP, KP|\!\}$ that are not yet satisfied by an array that is described by a list of choices whose incidence is *in*, which can be verified as described by function *tailUsefulChoices* in the algorithm. If $i \geq h+1$ and $C(i, A(i), K(i))$ is not a useful choice for $\{\!|AP, KP|\!\}$ after a list of choices described by *in*, then we can remove J_i from the array and what we obtain is still a witness: all requirements are already satisfied by the part of the array with incidence *in*, and the fact that all elements after J_i decrease their position by 1 is irrelevant since we are in the tail. If we are not in the tail, or we are in the tail and $C(i, A(i), K(i))$ is a useful choice, then we leave J_i in the array witness, we put $C(min(h+1, i), A(i), K(i))$ in $\mathbb{C}$, we update the cumulative incidence function *in*, we increment i, and we continue.

At the end of this process, we have a new witness J', obtained by deleting some elements from the tail of J, and a choice list $\mathbb{C}$ that describes J'. By the definition of $A(i)$ and $K(i)$, every J'_i in J' belongs to $[[x]]_E$ for all variables x that appear positively in $s(C(i, AP', KP'))$ and does not belong to $[[x]]_E$ for all variables x that appear complemented in $s(C(i, AP', KP'))$, hence it belongs to $[[\,co(x)]]_E$ for all these variables. Since J' is a witness for S, then J'_i also satisfies all applicable constraints in IP, hence it belongs to $[[s(C(i, AP', KP'))]]_E$, hence it belongs to $[[var(C(i, AP', KP'))]]_E$. If we assume that J has depth $d+1$, then every J'_i has a depth smaller than d, hence, for any A that is d-witnessed, every variable $var(C(i, AP', KP'))$ in the list $\mathbb{C}$ is populated. Hence, the choice list $\mathbb{C}$ is a list of choices that are populated, such that every tail choice C is useful after the choices that have been chosen before C, hence the choice list $\mathbb{C}$ would be generated by our algorithm unless a different solution were generated, hence our algorithm is generative.

Soundness of the algorithm is immediate. □

Property 19 (Complexity). *For any array group whose size is in $O(N)$, each pass of algorithm* cList *has a complexity in $O(2^{poly(N)})$.*

Proof. The *cList* algorithm explores at most $O(2^{poly(N)})$ choices at each step, and the total number of steps is at most:

$$MaxSteps \;=\; h + |AP| + \Sigma_{\mathsf{cont}_m^M(x) \in KP}\; m$$

By the linear constants assumption, *MaxSteps* is in $O(N^2)$, hence the algorithm explores at most $O((2^{poly(N)})^{N^2}) = O(2^{poly(N)*N^2})$ tuples, and the operation that must be executed for each tuple can be performed in time $O(2^{poly(N)})$. □

8.5 Witness Generation from Base Typed Groups

Witness generation for groups with a base type needs no preparation, is fully accomplished during the first pass, and is not difficult, as detailed below.

8.5.1 *Witness generation from a canonical schema of type* Null *or* Bool.

A canonical group of type Null has the shape $\{\mathsf{type}(\mathsf{Null})\}$ and generates `null`.

A group of type Bool that does not contain any $\mathsf{ifBoolThen}(b)$ operator will generate either `true` or `false`. If it contains a collection of $\mathsf{ifBoolThen}(\mathtt{true})$ operators, it will only generate `true`, and similarly for $\mathsf{ifBoolThen}(\mathtt{false})$. If it contains both, it is not satisfiable, and will return "unsatisfiable".

8.5.2 Witness generation from a canonical schema of type Str. A canonical group of type Str is just the conjunction of zero or more extended regular expressions, which we reduce to one by computing their intersection, whose size is linear in the size of the input regular expressions. At this point, we generate a witness for this regular expression, which can be done in time $O(2^{poly(N)})$ (Section 4.4).

8.5.3 Witness generation from a canonical schema of type Num. For a canonical schema of type Num, we can first merge all intervals into one and all $\mathsf{mulOf}(m)$ operators into one, let us call it $\mathsf{mulOf}(M)$; if the group contains an assertion $\mathsf{notMulOf}(n)$ with $M = n \times i$ for any integer i, then the group returns "unsatisfiable". Otherwise, we obtain one interval (if none is present, we add $\mathsf{betw}_{-\infty}^{\infty}$), a set of zero or many $\mathsf{notMulOf}(n)$ constraints, and one optional $\mathsf{mulOf}(m)$ with $m \neq n \times i$ for every $i \in \mathbb{Z}$ and for every $\mathsf{notMulOf}(n)$. At this point, to simplify some operations, we substitute any negative argument n of $\mathsf{mulOf}(n)$ or $\mathsf{notMulOf}(n)$ with its opposite. The interval may be open at both extremes, closed at both, or mixed. We distinguish five cases. In the last three cases we describe an open interval $\mathsf{xBetw}_{min}^{Max}$, but the reasoning when one extreme, or both, are included, is essentially the same.

(1) Empty interval: we return "unsatisfiable".
(2) One-point interval betw_{m}^{m}: if m satisfies all notMulOf and mulOf assertions we return m, otherwise we return "unsatisfiable".
(3) No $\mathsf{mulOf}(m)$, i.e., many-points interval $\mathsf{xBetw}_{min}^{Max}$ with no $\mathsf{mulOf}(m)$ constraint and l $\mathsf{notMulOf}(n_j)$ constraints: choose ϵ such that

$$0 < \epsilon \leq \frac{min((Max - min), n_1, \ldots, n_l)}{l + 2}$$

If we consider the set $B = \{\!\!|\ min + i \times \epsilon \mid i \in \{1..(l + 1)\}\ |\!\!\}$, then every value in B satisfies $\mathsf{xBetw}_{min}^{Max}$, and no assertion $\mathsf{notMulOf}(n_j)$ can be violated by two distinct values in B, hence at least one value in B is a witness.
(4) Finite $Max - min$ and mulOf, i.e., interval $\mathsf{xBetw}_{min}^{Max}$ with a $\mathsf{mulOf}(m)$ constraint and finite values for both min and Max: we list all multiples of m starting from min (excluded in case of xBetw) until we find one that satisfies all notMulOf assertions, or until we go over Max (excluded or included depending on the interval), in which case we return "unsatisfiable".
(5) Infinite $Max{-}min$ and mulOf, i.e., interval $\mathsf{xBetw}_{min}^{Max}$ where either min or Max is not finite, and with a $\mathsf{mulOf}(m)$ constraint: bring all arguments of $\mathsf{mulOf}(m)$ and $\mathsf{notMulOf}(n)$ into a fractional form where they share the same denominator d, as in $\mathsf{mulOf}(M/d)$, $\mathsf{notMulOf}(n_j/d)$. Select any prime number p that is strictly bigger than every n_j and such that either $p \times M/d$ or its opposite belongs to the interval. Such a number clearly exists, and it is easy to prove that primality of p and the fact that $(M/d) \neq (n_j/d) \times i$ for every $i \in \mathbb{Z}$ and for every $\mathsf{notMulOf}(n_j/d)$, imply that $p \times M/d$ satisfies all notMulOf assertions.

Property 20. *If a group of type* Num *has a witness, then the above algorithm will return a witness.*

Proof. The only difficult case is case (5). Assume, towards a contradiction, that exists n_j/d and an integer i with $p \times M/d = i \times (n_j/d)$, that is $p \times M = i \times n_j$. Since p is prime and is bigger than n_j, then p is prime wrt n_j. Since p is a factor of $i \times n_j$ and is prime wrt n_j, then p is a factor of i, hence there exists an integer i' such that $i = i' \times p$, that is, $p \times M = i' \times p \times n_j$, that is, $M = i' \times n_j$, which is impossible. □

Property 21. *If a group of type* Num *has a witness, one can be generated in time $O(2^{poly(N)})$, where N is the size of the input schema. If a group of type* Num *has a witness, this fact can be proved in time $O(2^{poly(N)})$.*

Proof. Here we do not need the linear constant assumption over any of the involved parameters. Let N be the size of the input schema. In case (3), we try $O(N)$ witnesses. In case (4), we must try at most $(Max - min)/m$ possible witnesses, which is in $O(2^N)$, because of binary notation. In case (5), we exploit the fact that the numbers are decimal, hence the number of digits of d is linear in N, hence the size of every n_j is still limited by N. We must also assure that either $p \times M/d$ or its opposite belongs to the interval. For example, when min is finite, p must satisfy $(p \times M)/d > min$ hence $p > min \times d/M$, and again all the constants have a bitmap representation linear in N. A prime number greater than k can be generated in time that is polynomial in k, hence we are still in $O(2^{poly(N)})$. □

9 EXPERIMENTAL ANALYSIS

9.1 Implementation and experimental setup

We implemented our witness generation algorithm for JSON Schema Draft-06 in Java 11, using the Brics library [32] to generate witnesses from patterns, and the *jdd* library [38] for ROBDDs. Our experiments were run on a Precision 7550 laptop with a 12-core Intel i7 2.70GHz CPU, 32 GB of RAM , running Ubuntu 21.10. We set the JVM heap size to 10 GB. Witnesses were validated by an external tool [2] (version 1.0.65), and additionally by hand, since the external tool reported false negatives in a few cases. Each schema is processed by a single thread, and all reported times are measured for a single run. Our reproduction package [4] can be used to confirm our results.

9.2 Tools for comparative experiments

Due to the lack of equivalent tools, we compare our tool against a Data Generator and a Containment Checker.

Data generator (DG). We use an open source test data generator for JSON Schema [17] (version 0.4.6). This Java implementation pursues a try-and-fail approach: an example is first generated, then validated against the schema, and potentially refined if validation fails, exploiting the error message. This tool lends itself to a comparison although it is not able to detect schema emptiness: given an unsatisfiable schema, it will always return an (invalid) instance.

Containment checker (CC). We compare our tool against the containment checker by Habib et al. [21] (version 0.0.5), described in [28], and designed to check interoperability of data transformation operators [16]. Typically, these schemas do not contain negation or recursion. The "CC tool" only supports Draft-04 schemas, a limitation that we consider when comparing against this tool.

9.3 Schema collections

We conduct experiments with six different schema collections: four real-world and two synthetic. Table 2 states their origin, the number of schemas, broken down into satisfiable and unsatisfiable schemas, and the average and maximal size of schemas.

Real-world schemas. The largest of the real-world schemas collection was obtained from GitHub. We retrieved virtually every accessible, open source-licensed JSON file from GitHub that presents the features of a schema, based on a BigQuery search on the GitHub public dataset; Google hosts a snapshot of all open source-licensed on GitHub, refreshed on a regular basis. The schemas were downloaded in July 2020, and are shared online [14]. We obtained over 80K schemas. As can be expected, we encountered a multitude of problems in processing these non-curated, raw files: files with syntactic errors, files which do not comply to any JSON Schema draft, and files with references that we are unable to resolve. Notably, there is a large share of duplicate schemas, with small variations in syntax and semantics. We rigorously removed such files, eliminating schemas with the same occurrences of keywords, condensing the corpus down to 7,046. We further excluded 619 schemas which are either ill-formed, or use specialized types (audio, video) that we do not support, or use an old draft with a different syntax, or employ patterns not supported by the third-party automaton library, or use unguarded recursion. More precisely, we excluded 17 ill-formed schemas, 105 schemas with specialized types, 355 schemas expressed in Draft-3, 61 schemas whose patterns contain negative lookahead, 68 schemas using unreachable references or references to fragments expressed inside specific keywords (like *properties*) that our tool does not yet correctly handle, and 13 schemas using unguarded recursion. Of the remaining 6,427 schemas, 40 are well-formed but unsatisfiable. We identified these schemas using our tool, and then performed a manual verification on all of them.

The three remaining real-world collections correspond to specifications of standards for deploying applications (Kubernetes [30]), ruling interactions within a specific system (Snowplow [5]), and describing data produced by content management systems (Washington Post [36]). To increase the number of processable schemas, we inlined references to external schemas. An earlier version of these collections where already used in [28] to check inclusion. Almost all schemas are satisfiable, except 5 from Kubernetes.

Hand-written schemas. Real-world schemas reflect real usage and can be quite big, but they focus on the commonest operators and combination of operators. Hence, for stress-testing, we inserted in our reproduction packages 233 handwritten schemas that are small but have been crafted to exemplify complex interactions between the language operators. To illustrate such an interaction, consider the following schema.

$$\{\, r : \mathsf{props}(a : x) \wedge \mathsf{props}(a.{*} : y) \wedge \mathsf{req}(a),\\ x : \mathsf{type}(\mathsf{Str}) \wedge \mathsf{pattern}(a(c|e)),\\ y : \mathsf{type}(\mathsf{Str}) \wedge \mathsf{pattern}(a(b|c)) \,\}$$

Here we have an interaction between two props and a req with overlapping patterns, and associated with two different variables x and y whose schema present non-trivial overlapping.

Array operators also present interactions, as in the following example.

$$\{\, r : \mathsf{item}(1 : x) \wedge \mathsf{cont}_1^1(y),\\ x : \mathsf{type}(\mathsf{Arr}) \wedge \mathsf{cont}_2^\infty(t),\\ y : \mathsf{cont}_1^\infty(\mathsf{type}(\mathsf{Num}) \wedge \mathsf{mulOf}(3))\}$$

This example describes an array with schema r that contains another array with schema $x \wedge y$, this one having at least two elements (because of $\mathsf{cont}_2^\infty(t)$), one of which is multiple of 3.

The collection has been built by systematically considering operators for objects, arrays, strings and numbers, following software-engineering principles for testing complex programs. Ultimately, this collection has proved particularly helpful in debugging.

More precisely, we considered the following combinations of typed operators by involving boolean operators with the goal of testing virtually all non-trivial interactions.

- for objects, we test interactions among props (as in the previous example) and between props and pro_i^j by setting one bound at a time than both the lower and the upper bounds,
- for arrays, we test the interactions among $\mathsf{item}(l : S)$ and $\mathsf{items}(i^+ : S)$, but also between these operators and $\mathsf{cont}_i^j(S)$,
- for strings, we basically test the interaction between patterns (pattern) and the lower/upper-bound for the length of string, which, in our algebra is captured in the pattern itself,
- for numbers, we test the interaction among betw_m^M and xBetw_m^M, $\mathsf{mulOf}(q)$, than any combination thereof.

Synthesized schemas. We include schemas that are neither real-world nor hand-written, but they are *synthesized*, that is, they are generated from the reference test suite for JSON Schema validation [34], designed to cover all language operators. The derivation is described in [6, 7], and yields triples (S_1, S_2, b) where the Boolean b specifies whether $S_1 \subseteq S_2$ holds for schemas S_1, S_2. Here, we restrict ourselves to schemas in Draft-04, since the CC-tool is restricted to this version. We excluded selected schemas that contain features that we do not yet support, such as the `format` keyword (a mere technicality) or references to external files.

We check a containment $S_1 \subseteq S_2$ by trying to generate a witness for the schema $S_1 \wedge \neg S_2$, which is unsatisfiable if, and only if, $S_1 \subseteq S_2$ holds; we thus obtain both satisfiable and unsatisfiable schemas. The CC tool accepts two schemas as input and does not need this encoding. We also test the DG tool, where comparison is only meaningful for pairs where $S_1 \wedge \neg S_2$ is satisfiable, since the DG tool cannot recognize unsatisfiable schemas.

9.4 Research hypotheses

We test the following hypotheses: (H1) *correctness* of our implementation, that we test with the help of an external tool that verifies the generated witnesses; (H2) *completeness* of our implementation, that we test by using an ample and diverse test-set; (H3) it can be used to fulfill some specific tasks better than existing tools; (H4) it can be implemented to run in *acceptable time* on sizable real-world schemas, despite its asymptotic complexity. We test the latest hypothesis by applying our tool to a vast set of real-world schemas.

9.5 Experimental results

9.5.1 Correctness and completeness. In each run of each tool, we distinguish four outcomes:

- *success*, when a result is returned and it is correct;
- *failure*: when the code raises a run-time error or a timeout, that we set at 3,600 secs (1 hour);
- *logical error on satisfiable schema*, when the input schema S is satisfiable but the code returns either "unsatisfiable" or a witness that does not actually satisfy S;
- *logical error on unsatisfiable schema*, when the input schema is unsatisfiable but a witness is nevertheless returned.

We consider two kinds of experiments. The first uses both the GitHub schemas and the hand-written schemas, comparing against the test data generator DG. The second uses the containment test suite and compares our tool with both the data generator (DG) and the containment checker (CC). We summarize the results in Table 2, together with the average and median runtimes.

Our tool. Our tool produces no logical error in any of our schema collections. With the GitHub schemas, it fails with "timeout" for 0.56% of schemas (35 schemas), and with "out of memory", when calling the automata library, for 0.36% of schemas (23 schemas). (We refer to Section 9.6.1 for a breakdown of problematic schemas.) No failures arise in the other two schema collections, supporting hypothesis H1.

The data generator. The DG tool successfully handles 93.45% of the GitHub schemas, and has similar correctness ratio for the other real-world schemas but it performs poorly regarding correctness on handwritten schemas, and cannot be really used for inclusion checking, since it does not detect unsatisfiability. It is difficult to compare run-times between tools. Essentially, on most schemas the two tools have comparable times, evident when looking at the median times, but there is a small percentage of files where our tool takes a very long time, and this is reflected on our disproportionately high average time.

The containment checker. The synthesized schemas show that our tool supports a much wider range of language features (hypothesis H2), which is natural since the CC tool targets a language subset, while completeness is core to our work.

We can conclude that our tool advances the state-of-the-art for containment checking and witness generation, especially for schemas that present aspects of complexity (hypothesis H3).

9.5.2 Runtime on real-world schemas. We next test hypothesis H4, assessing runtime on real-world schemas. In the three biggest collections, 95% of the files are elaborated in less than 2.1 secs, with median ≤40 msecs, and average ≤2.5 secs. The smaller Washington Post collection presents higher times, which will be discussed in Section 9.6. These results are coherent with hypothesis H4

9.6 Qualitative Insights

Several interesting insights can be extracted from an analysis of the space-time relationship for the GitHub collection, represented by the scatterplot in Figure 8b. The histograms at the top and at the right hand side indicate that schema size and run-time are distributed along 6 orders of magnitude, with a strong concentration on the low part of both axes, which forced us to use a log-log scale. In the log-log plot, we observe a cloud with a slope of about 1, suggesting a linear correlation, but we also observe that every file-size exhibits many outliers, and that long-running schemas can be found everywhere along the file-size axis. This clearly indicates that the runtime is affected more by the presence of specific combinations of operators, which may take little space but cause exponential runtime, than by schema size.

Indeed, our complexity analysis shows that exponential complexity is triggered by some specific operations, among which (1) object preparation, when different patterns overlap, requiring the generation of an exponential number of *choices* and of new variables; (2) reduction to DNF; and (3) pattern manipulation.

We tried to complement this theoretical knowledge with observations on the data. We applied data-mining techniques to correlate features of the schemas with the run-time. The feature that correlates more clearly with very long run-time is the presence of a `"maxLength"`: n statement with $n > 65000$, which induces the creation of a large automaton. Other features with a strong correlation with high run-time are the presence of `"enum"` with extremely long lists of arguments, that may then cause the generation of very big terms during DNF reduction, and of `"oneOf"` with long lists of argument, which again can generate big terms during DNF, since `"oneOf"` generates a conjunction during its translation.

We also resorted to visual inspection of problematic schemas, which indicated that nested objects with overlapping patterns may also require a lot of time, as indicated by the theoretical analysis.

The Washington Post collection required a specific analysis to explain its high 95% percentile time and average time. It is a smallish collection (125 schemas), where approximately 20% of the files require around 20 secs for their elaboration. All these files are very similar, with more than 2K nodes in their syntax trees and complex combinations of operators. By selectively deleting specific subtrees, we could conclude that the high time is typically due to pattern overlapping between an instance of `"patternProperties"` and a corresponding instance of `"properties"`, confirming our theoretical knowledge of the strong influence of pattern overlapping over the complexity of object preparation. The small number of files in this collection and their high homogeneity explains the anomaly of the result.

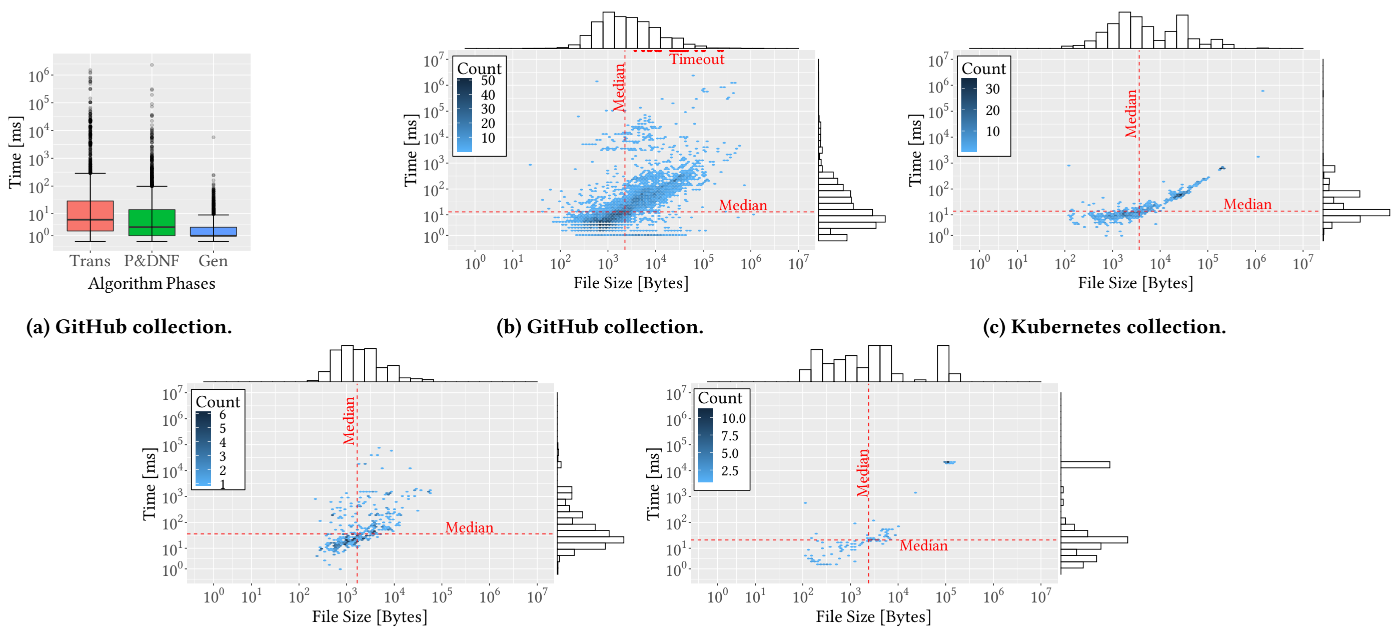
Hence, the overall indication is that our algorithm fulfills its aim of proving that this exponential problem can be successfully tackled on sizable real-world schema with a reasonable execution time, and that a careful analysis of the results of experiments over our vast and diverse dataset may guide further optimization efforts.

Runtime for the other collections is comparable to that of GitHub with fewer timeouts for two Snowplow schemas, which contain a `maxLength` assertion whose argument is 10^6. Another interesting observation is a schema from Kubernetes whose root consists in a `oneOf` with a list of 600 arguments, most of which are non-trivial, and which is elaborated in 5 mn. This confirms that the use of `oneOf` may increase the running time but is not sufficient to create a blowup

9.6.1 Problematic schemas. Our data suggests that a very long run-time does not really depend of the size of the schema but on the presence of specific arrangements of operators.

Table 2: Schema collections, correctness and completeness results, median/95th percentile/average runtime (in seconds).

Collection	#Total	#Sat/ #Unsat	Size (KB) Avg/Max	Tool	Success	Failure	Errors sat.	Errors unsat.	Med. Time	95% -tile	Avg. Time
GitHub [14]	6,427	6,387/40	8.7/1,145	Ours	99.08%	0.92%	0%	0%	0.013 s	0.600 s	2.711 s
				DG	93.45%	4.89%	1.21%	0.45%	0.054 s	0.103 s	0.089 s
Kubernetes [30]	1,092	1,087/5	24.0/1,310.7	Ours	100%	0%	0%	0%	0.014 s	0.606 s	0.605 s
				DG	99.54%	0%	0%	0.46%	0.078 s	0.144 s	0.088 s
Snowplow [5]	420	420/0	3.8/54.8	Ours	99.52%	0.48%	0%	no unsat	0.036 s	1.483 s	0.892 s
				DG	94.76%	0%	5.24%	no unsat	0.053 s	0.112 s	0.062 s
WashingtonPost [36]	125	125/0	21.1/141.7	Ours	100%	0%	0%	no unsat	0.021 s	20.773 s	3.622 s
				DG	96.8%	0%	3.2%	no unsat	0.090 s	0.181 s	0.107 s
Handwritten [4]	233	195/38	0.7/2.3	Ours	100%	0%	0%	0%	0.043 s	5.960 s	2.454 s
				DG	7.57%	36.87%	48.99%	6.57%	0.072 s	0.280 s	0.091 s
Containment-draft4 [7]	1,331	450/881	0.5/2.9	Ours	100%	0%	0%	0%	0.002 s	0.018 s	0.005 s
				DG	29.83%	28.85%	0.30%	41.02%	0.051 s	0.119 s	0.060 s
				CC	35.91%	62.96%	0.15%	0.98%	0.003 s	0.096 s	0.036 s



(a) GitHub collection. **(b) GitHub collection.** **(c) Kubernetes collection.**

(d) Snowplow collection. **(e) Washington Post collection.**

Figure 8: (a) Boxplots of processing times (in milliseconds, log scale) for the 3-phase witness generation algorithm, applied to GitHub schemas. Boxes range from the lower to the upper quartile, horizontal line indicating the median. Whiskers end at the 5th/95th percentile. Outliers above the whiskers are shown as individual dots, darker dots indicate overlapping values. (b-e) Scatterplot showing size of the schema vs. time for generating a witness for the different schema collections. Along top and right edge, a stylized histogram shows the distribution. Top right, the sizes of the files causing timeouts are shown in (b).

Our tool fails, with a timeout, only on 40 files, during the phase which interleaves between preparation and DNF. In order to better understand which operator usages create problems to our algorithm, with a focus on those cases where the runtime is definitely too high, we inspected these schemas, and verified that they all feature at least one of the following characteristics:

- object specification with a very long list of properties (reaching 142 for some schema), leading to the object preparation examining a very high number of combinations ;
- string assertions with an argument of `maxLength` exceeding 10^6 (Snowplow) or complex pattern expression combined with a relatively high argument for `maxLength` (reaching 5,000): both situations lead to manipulating very large automata increasing the total cost of the entire analysis;
- the use if recursive definitions involving the root and a negation of a complex object definition, this entails a problem during object preparation and DNF construction

While these schemas present a tiny portion of the GitHub-crawled corpora, they turn out to be very useful for stress-testing our tool and for indicating optimization opportunities.

9.7 Lessons learned

The experiment was not only useful to verify our hypotheses, but lead us also to other relevant insights, which we summarize here.

9.7.1 Patterns are important. Patterns appear in the `pattern` and `patternProperties` operators, and can be used to encode operators such as `minLength`, `maxLength`, and `additionalProperties`. Since these operators are not extremely common in real-world schemas (see the empirical study in [13]), it is easy to overlook the practical relevance of patterns in JSON Schema, but we discovered that the high complexity of regular expression operations has noticeable impact on the performance of the algorithm. We now believe that, while it is a good idea to rely on a high-quality external library to deal with the general case, a robust tool for witness generation must also dedicate extra effort to the special cases that arise in this specific application.

9.7.2 Easy schemas are very common. Manual inspection reveals that most GitHub schemas are very simple, using a subset of the operators in a repetitive way, and especially the largest schemas tend to be simplistic, often having been automatically generated (as also observed in [31]). This suggests that the average speed of any tool would greatly benefit from optimization targeted at this specific class of schemas.

9.7.3 Polynomial phases can be relevant. The boxplot shows that the polynomial phases of the algorithm take, on average, more time than the exponential phases. Although we did hope that the exponential phase were manageable, this inversion was for us a surprise, and also a lesson: do not underestimate the phases that appear inexpensive.

9.7.4 `oneOf` usually means `anyOf`. By a manual inspection of the schemas, we discovered that many schema designers define the different branches of a `oneOf` to be disjoint, as in

`"oneOf" : [{"type" : "null"}, {"type" : "string"}].`

Hence, the designer is using `oneOf` to tell the reader of the schema that the branches are disjoint, but if we substitute that `oneOf` with `anyOf`, the semantics of the schema remains exactly the same. This is extremely relevant, since `oneOf` is a very common operator, and `oneOf` is much more complex than `anyOf`, since it requires to compute the conjunction of each branch with the negation of all other branches. We acted upon this observation, and implemented a very simple optimization, where we first rewrite any `oneOf` to `anyOf`, generate a witness for this simplified schema, check the witness against the original schema, and fall back on the complete algorithm only in the extremely rare case when the generated witness was not valid. This simple optimization proved extremely effective.

10 CONCLUSIONS

JSON Schema is widely used in data-centric applications. The decidability and complexity of satisfiability and containment were known, but no explicit algorithm had been defined, and it was not obvious whether the high asymptotic complexity of the problem was compatible with a practical algorithm. In this paper we have addressed this open problem. We have described an algorithm for witness generation, satisfiability, and containment, that is based on a specific combination of known and original techniques, to take into account the specific features of JSON Schema object and array operators, and the need to run in a reasonable time.

Our extensive experiments prove the practical viability of the approach, and provide insight into the actual behavior of the algorithm on real-world schemas. These experiments are a necessary step for any redesign or re-factoring of the algorithm.

We have left the implementation of the `uniqueItems` operator out of the scope of the current paper in order to keep the size and complexity of this work under control, but the fundamental techniques that we have designed, for object and array preparation and generation, still apply, with some important generalizations that we believe deserve a dedicated analysis.

Acknowledgments. The research has been partially supported by the MIUR project PRIN 2017FTXR7S "IT-MaTTerS" (Methods and Tools for Trustworthy Smart Systems) and by the *Deutsche Forschungsgemeinschaft* (DFG, German Research Foundation) – 385808805.

We thank Dominik Freydenberger for proposing an algorithm to translate between ECMAScript and Brics REs. We thank Avraham Shinnar for feedback on an earlier version of this paper. We thank Stefan Klessinger for creating the charts and the reproduction package. We thank the students who contributed to our implementation effort: Francesco Falleni, Cristiano Landi, Luca Escher, Lukas Ellinger, Christoph Köhnen, and Thomas Pilz.

REFERENCES

[1] 2022. JSON Schema Faker. Available on GitHub at https://github.com/json-schema-faker/json-schema-faker and as an interactive tool at https://json-schema-faker.js.org.
[2] 2022. JSON schema validator. https://github.com/networknt/json-schema-validator
[3] 2022. JSONSchemaTool. Available at https://jsonschematool.ew.r.appspot.com.
[4] 2022. Reproduction Package on GitHub. Temporarily available at GitHub from https://github.com/sdbs-uni-p/JSONSchemaWitnessGeneration, will be moved to Zenodo, for long-term availability.
[5] Snowplow Analytics. 2022. Iglu Central. https://github.com/snowplow/iglu-central, commit hash 726168e.
[6] Lyes Attouche, Mohamed Amine Baazizi, Dario Colazzo, Yunchen Ding, Michael Fruth, Giorgio Ghelli, Carlo Sartiani, and Stefanie Scherzinger. 2021. Reproduction package: A Test Suite for JSON Schema Containment. Available on Zenodo at https://zenodo.org/record/5336931#.YshD0XZBxD8 and maintained on GitHub at https://github.com/sdbs-uni-p/json-schema-containment-testsuite.
[7] Lyes Attouche, Mohamed Amine Baazizi, Dario Colazzo, Yunchen Ding, Michael Fruth, Giorgio Ghelli, Carlo Sartiani, and Stefanie Scherzinger. 2021. A Test Suite for JSON Schema Containment. In *Proc. ER 2021*. 19–24. http://ceur-ws.org/Vol-2958/paper4.pdf
[8] Lyes Attouche, Mohamed Amine Baazizi, Dario Colazzo, Francesco Falleni, Giorgio Ghelli, Cristiano Landi, Carlo Sartiani, and Stefanie Scherzinger. 2021. A Tool for JSON Schema Witness Generation. In *Proc. EDBT 2021*. 694–697. https://doi.org/10.5441/002/edbt.2021.86 Tool Demo.
[9] Lyes Attouche, Mohamed-Amine Baazizi, Dario Colazzo, Francesco Falleni, Giorgio Ghelli, Cristiano Landi, Carlo Sartiani, and Stefanie Scherzinger. 2021. Un Outil de Génération de Témoins pour les schémas JSON A Tool for JSON Schema Witness Generation. In *Proc. Actes de la conférence BDA*. Informal proceedings.
[10] Mohamed Amine Baazizi, Dario Colazzo, Giorgio Ghelli, and Carlo Sartiani. 2019. Schemas And Types For JSON Data. In *Proc. EDBT*. 437–439.
[11] Mohamed Amine Baazizi, Dario Colazzo, Giorgio Ghelli, and Carlo Sartiani. 2019. Schemas and Types for JSON Data: From Theory to Practice. In *Proc. SIGMOD Conference*. 2060–2063.
[12] Mohamed-Amine Baazizi, Dario Colazzo, Giorgio Ghelli, Carlo Sartiani, and Stefanie Scherzinger. 2020. Not Elimination and Witness Generation for JSON Schema. In *Proc. Actes de la conférence BDA*. Informal proceesings, article available online at https://hal.archives-ouvertes.fr/hal-03190106/document.
[13] Mohamed Amine Baazizi, Dario Colazzo, Giorgio Ghelli, Carlo Sartiani, and Stefanie Scherzinger. 2021. An Empirical Study on the "Usage of Not" in Real-World JSON Schema Documents. In *Proc. ER*. 102–112. https://doi.org/10.1007/978-3-030-89022-3_9
[14] Mohamed Amine Baazizi, Dario Colazzo, Giorgio Ghelli, Carlo Sartiani, and Stefanie Scherzinger. 2021. A JSON Schema Corpus. A corpus of over 80thousand JSON Schema documents, collected from open source GitHub repositories, using Google BigQuery, in July 2020. Available on Zenodo (10.5281/zenodo.5141199) and maintained on GitHub (https://github.com/sdbs-uni-p/json-schema-corpus).
[15] Mohamed-Amine Baazizi, Dario Colazzo, Giorgio Ghelli, Carlo Sartiani, and Stefanie Scherzinger. 2022. Negation-Closure for JSON Schema. arXiv:2202.13434 [cs.DB] Accompanying technical report, available online https://arxiv.org/abs/2202.13434.
[16] Guillaume Baudart, Martin Hirzel, Kiran Kate, Parikshit Ram, and Avraham Shinnar. 2020. LALE: Consistent Automated Machine Learning, In Proc. KDD Workshop on Automation in Machine Learning (AutoML@KDD). *Computing Research Repository* abs/2007.01977. https://arxiv.org/abs/2007.01977
[17] Jim Blackler. 2022. JSON Generator. Available at https://github.com/jimblackler/jsongenerator.
[18] Pierre Bourhis, Juan L. Reutter, Fernando Suárez, and Domagoj Vrgoc. 2017. JSON: Data model, Query languages and Schema specification. In *Proc. PODS*. 123–135. https://doi.org/10.1145/3034786.3056120
[19] Randal E. Bryant. 1986. Graph-Based Algorithms for Boolean Function Manipulation. *IEEE Trans. Computers* 35, 8 (1986), 677–691. https://doi.org/10.1109/TC.1986.1676819
[20] Hubert Comon, Max Dauchet, Rémi Gilleron, Florent Jacquemard, Denis Lugiez, Christof Löding, Sophie Tison, and Marc Tommasi. 2008. *Tree Automata Techniques and Applications*. 262 pages. Available online at https://hal.inria.fr/hal-03367725/file/tata.pdf.
[21] IBM Corp. 2021. jsonsubschema. https://github.com/IBM/jsonsubschema
[22] Clara Benac Earle, Lars-Ake Fredlund, Ángel Herranz-Nieva, and Julio Mariño. 2014. Jsongen: a quickcheck based library for testing JSON web services. In *Proceedings of the Thirteenth ACM SIGPLAN workshop on Erlang, Gothenburg, Sweden, September 5, 2014*, Laura M. Castro and Hans Svensson (Eds.). ACM, 33–41. https://doi.org/10.1145/2633448.2633454
[23] Dominik D. Freydenberger. 2013. Extended Regular Expressions: Succinctness and Decidability. *Theory Comput. Syst.* 53, 2 (2013), 159–193. https://doi.org/10.1007/s00224-012-9389-0
[24] Michael Fruth, Kai Dauberschmidt, and Stefanie Scherzinger. 2021. New Workflows in NoSQL Schema Management. In *Proc. SEA-Data@VLDB (CEUR Workshop Proceedings, Vol. 2929)*. CEUR-WS.org, 38–39.
[25] Francis Galiegue and Kris Zyp. 2013. *JSON Schema: interactive and non interactive validation - draft-fge-json-schema-validation-00*. Technical Report. Internet Engineering Task Force. https://tools.ietf.org/html/draft-fge-json-schema-validation-00
[26] Wouter Gelade and Frank Neven. 2012. Succinctness of the Complement and Intersection of Regular Expressions. *ACM Trans. Comput. Log.* 13, 1 (2012), 4:1–4:19. https://doi.org/10.1145/2071368.2071372
[27] Rahul Gopinath, Hamed Nemati, and Andreas Zeller. 2021. Input Algebras. In *Proc. ICSE*. 699–710.
[28] Andrew Habib, Avraham Shinnar, Martin Hirzel, and Michael Pradel. 2021. Finding Data Compatibility Bugs with JSON Subschema Checking. In *Proc. ISSTA*. 620–632. https://doi.org/10.1145/3460319.3464796
[29] John E. Hopcroft, Rajeev Motwani, and Jeffrey D. Ullman. 2007. *Introduction to automata theory, languages, and computation, 3rd Edition*. Addison-Wesley.
[30] Kubernetes. 2022. Kubernetes JSON Schemas. https://github.com/instrumenta/kubernetes-json-schema, commit hash 133f848.
[31] Benjamin Maiwald, Benjamin Riedle, and Stefanie Scherzinger. 2019. What Are Real JSON Schemas Like? - An Empirical Analysis of Structural Properties. In *Proc. EmpER@ER*, Vol. 11787. Springer, 95–105. https://doi.org/10.1007/978-3-030-34146-6_9
[32] Anders Møller. 2021. dk.brics.automaton – Finite-State Automata and Regular Expressions for Java. Available at https://www.brics.dk/automaton/.
[33] JSON Schema Org. 2022. JSON Schema. Available at https://json-schema.org.
[34] JSON Schema Org. 2022. JSON Schema Test Suite. https://github.com/json-schema-org/JSON-Schema-Test-Suite.
[35] Felipe Pezoa, Juan L. Reutter, Fernando Suárez, Martín Ugarte, and Domagoj Vrgoc. 2016. Foundations of JSON Schema. In *Proc. WWW*. 263–273. https://doi.org/10.1145/2872427.2883029
[36] The Washington Post. 2022. ans-schema. https://github.com/washingtonpost/ans-schema, commit hash abdd6c211.
[37] Larry J. Stockmeyer. 1974. *The Complexity of Decision Problems in Automata Theory and Logic*. Ph. D. Dissertation. Massachusetts Institute of Technology.
[38] Arash Vahidi. 2020. JDD. https://bitbucket.org/vahidi/jdd/src/master/
[39] A. Wright, H. Andrews, and B. Hutton. 2019. *JSON Schema Validation: A Vocabulary for Structural Validation of JSON - draft-handrews-json-schema-validation-02*. Technical Report. Internet Engineering Task Force. https://tools.ietf.org/html/draft-handrews-json-schema-validation-02
[40] A. Wright, H. Andrews, and B. Hutton. 2020. *JSON Schema Validation: A Vocabulary for Structural Validation of JSON - draft-bhutton-json-schema-validation-00*. Technical Report. Internet Engineering Task Force. https://tools.ietf.org/html/draft-bhutton-json-schema-validation-00
[41] A. Wright, G. Luff, and H. Andrews. 2017. *JSON Schema Validation: A Vocabulary for Structural Validation of JSON - draft-wright-json-schema-validation-01*. Technical Report. Internet Engineering Task Force. https://tools.ietf.org/html/draft-wright-json-schema-validation-01