Negation-Closure for JSON Schema

Mohamed-Amine Baazizi[a], Dario Colazzo[b], Giorgio Ghelli[c], Carlo Sartiani[d,*], Stefanie Scherzinger[e]

[a] *Sorbonne Université, LIP6 UMR 7606, 4 place Jussieu, 75252, Paris, France*
[b] *Université Paris-Dauphine, PSL Research University, Place du Maréchal de Lattre de Tassigny, Paris, 75775, France*
[c] *Dipartimento di Informatica, Università di Pisa, Largo Bruno Pontecorvo, 3, Pisa, 56127, Italy*
[d] *DIMIE, Università della Basilicata, Via dell'Ateneo Lucano, 10, Potenza, 85100, Italy*
[e] *Universität Passau, Innstr. 43, Passau, 94032, Germany*

Abstract

JSON Schema is an evolving standard for describing families of JSON documents. It is a logical language, based on a set of *assertions* that describe features of the JSON value under analysis and on logical or structural combinators for these assertions, including a negation operator. Most logical languages with negation enjoy *negation closure*: for every operator, they have a negation-dual that allows negation to be pushed through the operator. We show that this is not the case for JSON Schema, study how that changed with the latest versions of the Draft, and discuss how the language may be enriched accordingly. To this aim, we exploit an algebraic reformulation of JSON Schema, which is helpful for the formal manipulation of the language.

Keywords: JSON Schema, Negation Closure, Schema Languages

1. Introduction

JSON is a simple data language whose terms represent trees constituted by nested records and arrays, with atomic values at the leaves. JSON is now widely used for data exchange on the Web. JSON Schema [1] is an ever-evolving specification for describing families of JSON terms, and heavily used for specifying web applications and REST services [2], operator compatibility in data science [3] and cloud computing pipelines [4, 5], or as schemas in NoSQL stores [6, 7].

Specifically, JSON Schema is a logical language based on a set of *assertions* that describe features of JSON values and on a set of boolean and structural combinators for these assertions, including negation and recursion. The expressive power of this complex language and the complexity of validation and satisfiability have recently been studied: Pezoa et al. [8] relied on tree automata and MSO to study the expressive power, while Bourhis et al. [9] mapped JSON Schema onto an equivalent modal logic, called JSL, to investigate the complexity of validation and satisfiability. Those papers proved that satisfiability is in 2EXPTIME in the general case.

[*]Corresponding author

Email addresses: `baazizi@ia.lip6.fr` (Mohamed-Amine Baazizi), `dario.colazzo@dauphine.fr` (Dario Colazzo), `ghelli@di.unipi.it` (Giorgio Ghelli), `carlo.sartiani@unibas.it` (Carlo Sartiani), `stefanie.scherzinger@uni-passau.de` (Stefanie Scherzinger)

We study here the problem of *negation-closure* for JSON Schema, that we define as the property of a logical formalism that allows every negated assertion to be rewritten into a negation-free one. Most logical languages with negation enjoy *negation-closure* because, for every operator, they have a “dual” that allows negation to be pushed through the operator, as happens for the pairs *and-or* and *forall-exists* in first order logics, and for the modal-logic pair $\Diamond$-$\Box$ used in JSL to encode JSON Schema [9]. Negation-closure is an important design principle for a logical system, since it ensures that, for every algebraic property that involves an operator, a “symmetric” algebraic property holds for its dual. This facilitates both reasoning and automatic manipulation.

We prove that JSON Schema, despite having the same expressive power as JSL, does not enjoy negation-closure, by showing that both objects and arrays are described by pairs of operators that “almost” describe the negation of the other, but not “exactly”. We show that, in the most common use-cases, negation can indeed be pushed through JSON Schema operators, and we exactly characterize the cases when this is not possible. We focus here on Draft-06 [10] of the JSON Schema language, but we also discuss what happens with the more recent versions, which introduce important novel issues.

Based on these results, we present a negation-closed extension of JSON Schema, with the same expressive power as the original language, but where all operators have a negation dual, together with a simple and complete not-elimination algorithm for the extended language.

Our extension has already found practical applications: (1) we have built a tool to verify equivalence and inclusion between JSON Schema documents through the generation of witnesses for satisfiable schemas [11, 12]. The not-elimination algorithm here described is the first step of the satisfiability verification algorithm employed in that tool. Thanks to this algorithm, our tool is capable of dealing with negation without any limitation, while other tools impose strong restrictions on the use of negation [2] or do not support negation at all [13]. (2) Our extension was also implemented by a third party within a tool for debugging and repairing machine learning pipelines [14].

We believe that the present study sheds light on aspects of JSON Schema that have not been studied in previous works [8, 9], and that may be useful for the development of further tools for JSON Schema manipulation.

Main Contributions.

(i) We study the problem of negation-closure of JSON Schema, that is, which operators are endowed with a negation dual. We show that JSON Schema structural operators are *not* negation-closed, and we exactly characterize the schemas where negation cannot be rephrased. To this aim, we use a reformulation of JSON Schema that is *algebraic*, i.e., where no operator depends on an adjacent one, and where a subschema can be freely substituted by an equivalent one. This property is essential for defining a denotational semantics for JSON Schema, which in turn is essential for using induction when proving properties. 1.6

(ii) We then extend our algebraic presentation of JSON Schema so that it becomes negation-closed, and we prove negation closure by defining a not-elimination algorithm for this closed language. The algorithm is based on a technique to deal with negated recursive variables that is simple but, to our knowledge, original.

(iii) We extend our study to some new operators introduced in Draft 2019-09, that have a significant impact on negation-closure. Existing works on formalizing JSON Schema take into account Draft-04 only, and therefore do not consider these novel operators.

```
{ "properties": { "VAL": { "type": "number" },
                  "NEXTOBJ":  { "type": "object"  }},
  "patternProperties": { "^NEXT":  { "$ref": "#" },
                         "STRING$": { "type": "string" }},
  "additionalProperties": { "anyOf": [{"$ref": "#/definitions/OBJROOT" },
                                      {"type": "boolean"},
                                      {"type": "string"}]},
  "required": ["VAL", "NEEDED"],
  "minimum": 1,
  "definitions": { "OBJROOT": { "allOf": [{ "$ref": "#" },
                                          { "type": "object" }]}}
}
```

Figure 1: A JSON Schema document.

Paper Outline. In Section 2 we provide an overview of JSON Schema. In Section 3, we define the formal semantics of JSON Schema, based on our algebraic presentation. Sections 4 and 5 1.1
study *negation closure*. Section 6 analyzes the `minContains` and `maxContains` operators introduced in Draft 2019-09. Section 7 presents our experiments. We conclude after a review of related work.

2. JSON and JSON Schema

JSON Schema is a logical language whose terms are expressed as JSON terms, and describe sets of JSON terms. Therefore, we first briefly review the JSON data model, and then introduce the main features of JSON Schema through examples. We will discuss its semantics in Section 3.

2.1. The data model

JSON values are either basic values, objects, or arrays. Basic values B include the null value, booleans, numbers m, and strings s. Objects O represent sets of *members*, each member (or *field*) being a name-value pair (k, J) with no name appearing in two distinct members, and arrays A represent sequences of values with positional access. Objects and arrays may be empty. Two JSON values are equal if, and only if, they only differ in the order of members in objects.

In JSON syntax, a name is itself a string, and hence surrounded by quotes. Below, we specify the data model syntax, where n is a natural number with $n \geq 0$, and $k_i \in \mathsf{Str}$ for $i = 1, \ldots, n$.

$$
\begin{array}{rlrl}
J ::= & B \mid O \mid A & & \text{JSON expressions} \\
B ::= & \texttt{null} \mid \texttt{true} \mid \texttt{false} \mid m \mid s & m \in \mathsf{Num}, s \in \mathsf{Str} & \text{Basic values} \\
O ::= & \{k_1 : J_1, \ldots, k_n : J_n\} & n \geq 0,\ i \neq j \Rightarrow k_i \neq k_j & \text{Objects} \\
A ::= & [J_1, \ldots, J_n] & n \geq 0 & \text{Arrays}
\end{array}
$$

2.2. JSON Schema by example

Consider the schema document of Figure 1. We use here *schema* to indicate a subterm of a JSON Schema document that is either an object or a boolean (`true`/`false`) and that is used to validate instances.

Assertions. Any JSON Schema document is a schema, which usually contains many nested schemas. The members of a schema whose name is chosen among the *assertion keywords* are called *assertions*, and here we have five of them at the outermost level: `properties`, `patternProperties`, `additionalProperties`, `required`, and `minimum`. A set of assertions collected inside a schema denotes a conjunction: an instance must satisfy all those assertions to satisfy the schema.

`definitions` is not an assertion keyword. It is just a placeholder, which does not assert anything about the JSON instance, but whose value contains a set of name-schema pairs: here, it is used to associate the name *OBJROOT* with the schema "{ "`allOf`" : [...] }". These associations are used by the $`ref` operator, described later.

JSON values belong to one of the six JSON types: number, null, boolean, string, object, array. All assertions that analyze the values of a specific type T are *conditional* on T, that is, their definition is "*if* the instance belongs to T *then*...", so that every instance that does *not* belong to T satisfies that assertion.

For example, the `properties` assertion in line 1 specifies that, if the instance is an object, then if the instance has a *VAL* member, then its value is a number, and, if it has a *NEXTOBJ* member, then its value is an object. This assertion is conditional on the type object, and hence it is satisfied by anything that is not an object, and is conditional on the presence of the members *VAL* and *NEXTOBJ*. Hence, it is satisfied by any object without those members.

The `required` assertion is itself conditional on the instance type being object, but in that case, if forces the presence of the listed member names. The names listed in the `properties` assertions and those that are listed by `required` are, in principle, not related: in the example, we restrict the value of *NEXTOBJ* member without forcing its presence, and we require the *NEEDED* member without explicitly restricting its value.

`patternProperties` specifies that *if* the instance is an object and *if* a member name matches some pattern, then the member value must satisfy the corresponding schema. JSON Schema patterns are matched against any substring of the target string unless they are anchored to the beginning of the target (using "^" as in "^NEXT") or to the end (using "$" as in "STRING$"). Hence, any member whose name begins with *NEXT* must satisfy the assertion "{ "$`ref`": "#" }" (to be discussed later) and any member whose name ends in *STRING* must satisfy the assertion "{ "`type`": "string"}". Hence, a member named *NEXTxxxSTRING* must satisfy both, and a member named *NEXTOBJ* must satisfy both the schema "{ "`type`": "object"}" associated with *NEXTOBJ* and the schema "{ "$`ref`": "#" }".

`additionalProperties` is context-dependent: *if* the instance is an object, if a member name does not match any `properties` or `patternProperties` that co-occur at the top level of the schema where `additionalProperties` occurs, then the member value must satisfy the schema of `additionalProperties`. Since *NEEDED* has not been explicitly constrained in `properties` or `patternProperties`, its values must conform to the schema specified in `additionalProperties`. Hence, "`additionalProperties`" : S is equivalent to "`patternProperties`" : { cp : S } where cp is a pattern that matches any string that is not matched by any `properties` or `patternProperties` assertions in the same schema object.

"`minimum`" : 1 is a conditional assertion about numbers, meaning: if the instance is a number, then its minimum value is 1. The combination of this assertion with the previous assertions, that are all conditional on the type object, means that: if the instance is a number, it must satisfy "`minimum`" : 1. If it is an object, it must satisfy the other constraints described. If it has any other type (null, boolean, string, array), then it satisfies this schema.

Reference assertions. A reference assertion "`$ref`": "`path`" is a recursive reference, denoting a subschema (possibly also of a remote resource) reached by a path. Path "`#`" denotes the root of the current document, while "`#/definitions/OBJROOT`" is the schema that is the value of the *OBJROOT* member of the *definitions* member of the root. This allows any subschema to refer to any other subschema.

Boolean combinators. Besides the conditional assertions, JSON Schema features boolean combinators such as `allOf` for boolean conjunction, `anyOf` for boolean disjunction, `not` for boolean negation, and others. For example (see lines 10 and 11), "`allOf`": [{ "`$ref`": "`#`" } , { "`type`": "object"}] in the example requires the instance to satisfy the root schema and to be an object, hence excluding all the other types.

JSON instances. In Figure 2 we show three JSON fragments satisfying the schema of Figure 1. The string "`notAnObject`" satisfies that schema since all assertions are conditional, and therefore satisfied by anything that is neither an object nor a number. All instances that are objects must contain both *VAL* and *NEEDED*, but *NEEDED* values may have any type. The *NEXTOBJ* value must be an object because of the `properties` assertion, and must satisfy the entire specification since this name matches "`^NEXT`".

```
"notAnObject"
```

```
{ "VAL" : 16,
  "NEEDED" : "Number of performance cores" }
```

```
{ "VAL" : 8,
  "NEEDED" : false,
  "NEXTOBJ" :  { "VAL" : 4,
                 "NEEDED"  : true }
}
```

Figure 2: Three JSON fragments satisfying the schema of Figure 1.

2.3. A negation example

Reasoning about negation can be tricky, due to some elusive aspects of JSON Schema semantics [15]. Notably, in its security guidelines for designing JSON Schema documents, NSA explicitly advises against the use of negation [16]. In Figure 3, we provide a supplementary example that contains the `not` operator. Lines 1 through 3 state which values are allowed for the fields *color* and *size* if the instance is an object and if it contains those members. Line 4 actually declares that the instance *must* be an object and *must not* contain a field *size*. This can be surprising, since the assertion might be mistakenly read for a field not being required, and therefore, *optional*. In fact, in real-world schemas, the most frequent argument of `not` is `required` [15].

Figure 4 shows an equivalent schema, where negation has been *pushed down*, to the point of elimination. It is now explicit that property `size` is not allowed (line 3). Moreover, line 4 states that the schema allows only instances of type object, an assertion that was only implicit in the original schema. The full equivalence of these schemas is not obvious.

```
{ "properties": {
    "color": {"enum": ["white", "black"]},
    "size":  {"enum": ["S", "M", "L"]} },
  "not": {"required": ["size"]}
}
```

Figure 3: T-shirt schema with negation.

```
{ "properties": {
    "color": {"enum": ["white", "black"]},
    "size": false },
  "type": "object"
}
```

Figure 4: T-shirt schema after not-elimination.

As a more intricate (although artificial) example, consider the schema of Figure 5.

```
{"properties": {"a": {"not": {"$ref": "#"}}}}
```

Figure 5: Recursive schema.

This schema may be puzzling, but it can be rewritten to be more readable: it describes an instance that, in case it is an object with the field "a", then the value of this field must be an object, which must contain an "a" field, whose value must recursively satisfy the same specification.

We will describe later (Example 6) how our not-elimination algorithm can be used to rewrite the compact specification of Figure 5 into the readable `oneOf` Figure 6.

```
{ "properties" :
      { "a": { "type" : "object" ,
               "required" : ["a"],
               "properties" : { "a" : { "$ref": "#" } }
             }
      }
}
```

Figure 6: Recursive schema after not-elimination.

2.4. JSON Schema Drafts

At the time of writing, there are five major versioned drafts of JSON Schema: Draft-03 of November 2010 [17], Draft-04 of February 2013 [18], Draft-06 of April 2017 [10], Draft 2019-09 of September 2019 [19], and Draft 2020-12 [20] (Draft-05 was essentially a cleanup of Draft-04 and used the same meta-schema, while versions before Draft-03 have been absorbed by that one).

To analyze the real-world usage of JSON Schema, 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 Google GitHub public dataset. Over 80K schemas were downloaded in July 2020, and are shared online [21].

In this GitHub corpus, the vast majority of schemas respect Draft-04, some use the new features of Draft-06, and we found almost no examples of Draft 2019-09.

JSON Schema validation produces a validity result and it also annotates the validated instance. Annotations are an important feature of JSON Schema but, up to Draft-06, one can define a semantics of validation that does not depend on them, hence keeping the two issues separated. Draft 2019-09 introduces a major semantic change in the language: the satisfaction of assertions `unevaluatedProperties` and `unevaluatedItems` does not only depend on the satisfaction of the sub-assertions that they contain, but also on the specific *annotations* that the sub-assertions produce. Once this dependence of validation from annotations is added, not only the semantics becomes more complicated but, unfortunately, De Morgan rules lose their validity.

Since the overwhelming majority of the schemas we found do not use the post-Draft-06 operators, we decided to focus our analysis on Draft-06, and to defer a detailed analysis of negation closure for the annotation-dependent Drafts for future work. However, we decided to include in our study the operators `minContains` and `maxContains`, added in Draft 2019-09, since they do not depend on annotations and they are strictly related to the themes of our study.

3. Algebraic syntax and denotational semantics for JSON Schema

We say that a language is *algebraic* when it enjoys *substitutability*, meaning that the substitution of every subterm with a semantically equivalent subterm preserves the semantics of the entire term, independently of the surrounding context. The following example shows that JSON Schema does not enjoy this property, because of the dependency of some operators on adjacent operators. 1.1

Example 1. The assertion

```
"properties": { "size": true }
```

specifies that, if the instance is an object, then if the instance has a *size* member, then its member must satisfy `true`. In other terms, it is a trivial assertion, verified by every JSON instance, which has the same trivial semantics as

```
"properties": { "color": true }
```

Consider now the following JSON Schema document:

```
1 { "type": "object",
2   "additionalProperties": false,
3   "properties": { "size": true }
4 }
```

This schema is satisfied by the empty object, and by any JSON object having a unique property named `"size"`. If we substitute the trivial assertion in line 3 with the equivalent trivial expression `"properties": { "color": true }`, the meaning of the entire schema changes, and now only a `"color"` member is accepted. This happens because the "`properties`" assertion, trivial when considered alone, influences the meaning of an "`additionalProperties`" assertion that co-occurs at the top level of the same schema. Hence, the substitution of an assertion with one satisfied by the same set of instances may modify the meaning of the surrounding schema: the dependency of the "`additionalProperties`" operator on adjacent operators prevents substitutability.

Substitutability allows one to define a denotational semantics, that is, to define the semantics of an operator as a function applied to the semantics of its operands, and, more generally, substitutability facilitates algebraic manipulations and proofs by induction. For this reason, we define here an alternative syntax for JSON Schema, a syntax which enjoys substitutability since the dependent operators such as `"additionalProperties"` are enriched with explicit contextual information — the property names used by the adjacent operators — that makes them independent of the other operators. 1.1,1.5

Another cause of non substitutability is the navigational nature of JSON Schema references. Consider the following schema, where the `"$ref"` operator refers, through navigation, to the `{"not":{"multipleOf":3}}` subschema.

```
{ "$ref": "#/definitions/MofThree/not",
  "definitions":
    { "MofThree":  { "not":  { "not":  { "multipleOf": 3 } } } }
}
```

If we substitute the subschema `{"not":{"not":{"multipleOf":3}}}` with the equivalent schema `{"multipleOf":3}`, the pointer `"$ref":"#/definitions/MofThree/not"` becomes invalid, which violates substitutability. Hence, in our presentation, we will use standard named variables instead of navigational references.

In the remaining part of this section, we introduce our algebraic presentation of JSON Schema, we define its semantics, and we prove its equivalence to standard JSON Schema.

3.1. The algebraic presentation of JSON Schema

The syntax. The operators of our succinct algebraic presentation correspond to those of JSON Schema, as illustrated in Figures 9 and 10, and they obey the grammar presented in Figure 7. 1.1, 1.2

In the grammar, n is always a natural number starting from 0, so that we use here indexes going from 1 to $n+1$ when at least one element is required. In $\mathsf{req}(k_1, \ldots, k_n)$, each k_i is a string. The other metavariables are listed in the first line of the grammar, where $\mathbb{R}$ indicates real numbers, $\mathbb{R}_{>0}$ denotes positive real numbers, $\mathbb{N}$ are the naturals (zero included). The sets $\mathbb{R}^{-\infty}$, $\mathbb{R}^{\infty}$, and $\mathbb{N}^{\infty}$ stand for the base set enriched with the extra symbols $-\infty$ or ∞. $JVal(*)$ is the set of all JSON terms. Apart from the syntax, this presentation reflects the operators of JSON Schema Draft-06 [10].

Each schema expresses properties of an *instance* which is a JSON value; the semantics of a schema S with respect to an environment E is, therefore, the set $[\![S]\!]_E$ of JSON instances that *satisfy* that schema, as specified in Figure 8. The environment E is a set of pairs $(x : S)$, that are introduced by the operator $D = S\ \mathsf{defs}(x_1 : S_1, \ldots, x_n : S_n)$ and are used to interpret variables x_i, as discussed below.

The operators. We provide an intuition for the operators, before discussing their formal semantics. The assertion $\mathsf{type}(T_1, \ldots, T_{n+1})$ is satisfied by any instance belonging to one of the listed predefined JSON types. $\mathsf{const}(J)$ is only satisfied by the instance J, and $\mathsf{enum}(J_1, \ldots, J_{n+1})$ is the same as $\mathsf{const}(J_1) \vee \ldots \vee \mathsf{const}(J_{n+1})$.

Our syntax includes the boolean operators ($\neg$, $\wedge$, $\vee$, $\Rightarrow$, $\Rightarrow\ |$, $\{S_1, \ldots, S_n\}$, and $①$). The operators $\mathbf{t}$, $\mathbf{f}$, $\neg$, $\vee$, $\Rightarrow$, $\wedge$ combine the results of their operands in the standard way, while the conditional $(S_1 \Rightarrow S_2 \mid S_3)$ stands for $(S_1 \wedge S_2) \vee ((\neg S_1) \wedge S_3)$, and is used for representing JSON Schema `"if"`- `"then"`- `"else"`. The exclusive or operator $①(S_1, \ldots, S_{n+1})$ is satisfied iff

$$
\begin{array}{lll}
\multicolumn{3}{l}{r \in \mathit{RegExp}, m \in \mathbb{R}^{-\infty}, M \in \mathbb{R}^{\infty}, l \in \mathbb{N}, j \in \mathbb{N}^{\infty}, q \in \mathbb{R}_{>0}, J \in \mathit{JVal}(*)} \\
T & ::= & \mathsf{Arr} \mid \mathsf{Obj} \mid \mathsf{Null} \mid \mathsf{Bool} \mid \mathsf{Str} \mid \mathsf{Num} \\
S & ::= & \mathsf{type}(T_1, \dots, T_{n+1}) \mid \mathsf{const}(J) \mid \mathsf{enum}(J_1, \dots, J_{n+1}) \\
 & & \mid \mathsf{len}_l^j \mid \mathsf{betw}_m^M \mid \mathsf{xBetw}_m^M \mid \mathsf{mulOf}(q) \mid \mathsf{pattern}(r) \\
 & & \mid \mathsf{props}(r_1 : S_1, \dots, r_n : S_n; S_{n+1}) \mid \mathsf{pro}_l^j \mid \mathsf{req}(k_1, \dots, k_n) \mid \mathsf{propNames}(S) \\
 & & \mid \mathsf{items}(S_1, \dots, S_n; S_{n+1}) \mid \mathsf{contains}(S) \mid \mathsf{ite}_l^j \mid \mathsf{uniqueItems} \\
 & & \mid x \mid \mathbf{t} \mid \mathbf{f} \mid \neg S \mid S_1 \vee S_2 \mid \textcircled{1}(S_1, \dots, S_{n+1}) \\
 & & \mid S_1 \Rightarrow S_2 \mid (S_1 \Rightarrow S_2 \mid S_3) \mid S_1 \wedge S_2 \mid \{S_1, \dots, S_n\} \\
E & ::= & x_1 : S_1, \dots, x_n : S_n \\
D & ::= & S\ \mathsf{defs}\ (E)
\end{array}
$$

Figure 7: Syntax of the algebra.

exactly one of the arguments holds, and represents "oneOf". $\{S_1, \dots, S_n\}$, finally, is the same as $S_1 \wedge \dots \wedge S_n$. 2.1

The operators from len_l^j to uniqueItems are called, in this paper, *implicative typed assertions* (ITAs), since their semantics has the following structure: "if the instance belongs to the type T, then ...". Every ITA is associated with a specific type; it discriminates inside that type, and it is satisfied by every element of any other type. $\mathsf{pattern}(r)$ means: *if* the instance is a string, *then* it matches r. len_l^j means: if the instance is a string, then its length is included between l and j. betw_m^M means: if the instance is a number, then it is included between m and M, extremes included. xBetw_m^M is the same with extremes excluded. $\mathsf{mulOf}(q)$ means: if the instance J is a number, then $J = i * q$, for some integer i.

An instance J satisfies the assertion $\mathsf{props}(r_1 : S_1, \dots, r_n : S_n; S)$ iff the following holds: if the instance J is an object, then for each pair $k : J'$ appearing at the top level of J and for every $r_i : S_i$ such that k matches r_i, it holds that J' satisfies S_i, and, when k does not match any pattern in $r_1, \dots, r_n$, then J' satisfies S; it combines the three JSON Schema operators `properties`, `patternProperties`, and `additionalProperties`, hence resolving the problem of the dependence of the third from the first two. 1.1

pro_l^j means: if the instance is an object, it has at least l and at most j properties. Assertion $\mathsf{req}(k_1, \dots, k_n)$ means: if the instance is an object, then, for each k_i, one of the names of the instance is equal to k_i. The assertion $\mathsf{propNames}(S)$ means that, if the instance is an object, then every member name of that object satisfies S.

An instance J satisfies $\mathsf{items}(S_1, \dots, S_n; S_{n+1})$ iff the following holds: if J is an array, then each of its elements at position $i \leq n$ satisfies S_i, while further elements satisfy S_{n+1}. Note that no constraint is posed over the length of J: if it is strictly shorter than n, or empty, that is not a problem (this operator combines the two JSON Schema operators `items` and `additionalItems`).To constrain the array length we have ite_l^j, satisfied by J when, if J is an array, its length is between l and j. Assertion $\mathsf{contains}(S)$ means: if the instance is an array, then it contains at least one element that satisfies S. The assertion uniqueItems means that, if the instance is an array, then all of its items are pairwise different.

The operator $D = S\ \mathsf{defs}(E)$, where $E = x_1 : S_1, \dots, x_n : S_n$, is used in our presentation of

JSON Schema to introduce variables: an instance J satisfies a *schema document* $D = S$ $\mathsf{defs}(E)$ iff it satisfies S in the environment E, which means that, when a variable x_i is met while checking whether J satisfies S, x_i is substituted by $E(x_i)$, that is, by S_i. This results in a cyclic definition when we have environments such as $\mathsf{defs}(x : x)$ or $\mathsf{defs}(x : \neg x)$. The JSON Schema standard rules out such cyclic environments by specifying that checking whether J satisfies S by expanding variables must never result in an infinite loop. We express this requirement as a guardedness condition on E, as follows.

Let us say that x_i *unguardedly depends* on x_j if the definition $E(x_i)$ of x_i contains one occurrence of x_j that is not in the scope of any typed operator (otherwise, we say that the occurrence is *guarded* by a typed operator): for instance, in $\mathsf{defs}(\ x : (\mathsf{items}(y; w) \wedge z)\ )$, x unguardedly depends on z, while y and w are guarded by $\mathsf{items}(;)$. Recursion is *guarded* if the *unguardedly depends* relation is acyclic: no pair (x, x) belongs to its transitive closure. Informally, guarded recursion requires that any cyclic dependency must traverse a *typed* operator, which ensures that, when a variable is unfolded for the second time, the resulting schema is applied to an instance that is strictly smaller than the one analyzed at the moment of the previous unfolding. This notion was introduced as *well-formedness* in related work [8, 9].

From now on, we will assume that, for any schema document S $\mathsf{defs}(E)$, recursion in E is guarded, and that E is closing for S, as defined below, and we will make the same assumption when discussing the semantics of a schema S with respect to an environment E.

Definition 1. 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 $x_1, \ldots, x_n$.

The semantics. We formalize JSON Schema semantics as follows. JSON Schema standard defines the semantics of variables by specifying that $x = E(x)$; unfortunately, this is not an inductive definition, since S_i is generally bigger than x_i. In order to make it inductive, we add a parameter $i \in \mathbb{N}$ to the semantic function $[\![S]\!]^i_E$, and we give the following definition for $[\![x]\!]^i_E$:

$$[\![x]\!]^0_E = \emptyset \qquad\qquad [\![x]\!]^{i+1}_E = [\![E(x)]\!]^i_E$$

All other operators are defined as in JSON Schema specification, and the full formal definition of $[\![S]\!]^i_E$ is shown in Figure 8, where $L(r)$ is the set of strings matched by r, $JVal(\mathsf{Obj})$ is the set of JSON values whose type is "object", and similarly for the other types; $JVal(*)$ is the set of all JSON values. We use these special curly braces "$\{\!|\ |\!\}$" for sets, in order to avoid confusion with standard braces "{}" that are used in JSON syntax. 1.2, 1.6

The definition of $[\![S]\!]^i_E$ in Figure 8 is inductive on the lexicographic pair $(i, |S|)$, and we have now to define a notion of "limit" for the $[\![S]\!]^i_E$ succession, in order to get rid of the i index. Because of negation, the sequence $[\![S]\!]^i_E$ not necessarily monotonic in i, but we can still define a forall-exists limit by stipulating that an instance J belongs to $[\![S]\!]_E$ if an i exists such that J belongs to every interpretation that comes after i:

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

The definition above ensures that every schema has a well defined interpretation.

Our presentation differs from standard JSON Schema in the syntax, in the fact that we group some operators together, and, most importantly, in how we deal with variables: while JSON

$$
\begin{array}{lcl}
\llbracket \mathsf{type}(\mathsf{Null}/\mathsf{Bool}/\mathsf{Str}) \rrbracket^i_E & = & \mathit{JVal}(\mathsf{Null})/\mathit{JVal}(\mathsf{Bool})/\mathit{JVal}(\mathsf{Str}) \\
\llbracket \mathsf{type}(\mathsf{Arr}/\mathsf{Obj}/\mathsf{Num}) \rrbracket^i_E & = & \mathit{JVal}(\mathsf{Arr})/\mathit{JVal}(\mathsf{Obj})/\mathit{JVal}(\mathsf{Num}) \\
\llbracket \mathsf{type}(T_1, \ldots, T_{n+1}) \rrbracket^i_E & = & \llbracket \mathsf{type}(T_1) \rrbracket^i_E \cup \ldots \cup \llbracket \mathsf{type}(T_{n+1}) \rrbracket^i_E \\
\llbracket \mathsf{mulOf}(q) \rrbracket^i_E & = & \{\!\{\, J \mid J \in \mathit{JVal}(\mathsf{Num}) \Rightarrow \exists k \textit{ integer with } J = k * q \,\}\!\} \\
\llbracket \mathsf{const}(J) \rrbracket^i_E & = & \{\!\{ J \}\!\} \\
\llbracket \mathsf{enum}(J_1, \ldots, J_{n+1}) \rrbracket^i_E & = & \{\!\{ J_1, \ldots, J_{n+1} \}\!\} \\
\llbracket \mathsf{len}^j_l \rrbracket^i_E & = & \{\!\{\, J \mid J \in \mathit{JVal}(\mathsf{Str}) \Rightarrow l \leq \mathit{length}(J) \leq j \,\}\!\} \\
\llbracket \mathsf{pattern}(r) \rrbracket^i_E & = & \{\!\{\, J \mid J \in \mathit{JVal}(\mathsf{Str}) \Rightarrow J \in L(r) \,\}\!\} \\
\llbracket \mathsf{betw}^M_m \rrbracket^i_E & = & \{\!\{\, J \mid J \in \mathit{JVal}(\mathsf{Num}) \Rightarrow m \leq J \leq M \,\}\!\} \\
\llbracket \mathsf{xBetw}^M_m \rrbracket^i_E & = & \{\!\{\, J \mid J \in \mathit{JVal}(\mathsf{Num}) \Rightarrow m < J < M \,\}\!\} \\
\llbracket \mathsf{pro}^j_l \rrbracket^i_E & = & \{\!\{\, J \mid J \in \mathit{JVal}(\mathsf{Obj}) \Rightarrow l \leq |J| \leq j \,\}\!\} \\
\llbracket \mathsf{req}(k_1, \ldots, k_n) \rrbracket^i_E & = & \{\!\{\, J \mid J \in \mathit{JVal}(\mathsf{Obj}) \Rightarrow \forall k \in \{\!\{k_1, \ldots, k_n\}\!\}. \exists J'.(k : J') \in J \,\}\!\} \\
\llbracket \mathsf{ite}^j_l \rrbracket^i_E & = & \{\!\{\, J \mid J \in \mathit{JVal}(\mathsf{Arr}) \Rightarrow l \leq |J| \leq j \,\}\!\} \\
\llbracket \mathsf{uniqueItems} \rrbracket^i_E & = & \{\!\{\, J \mid J = [J_1, \ldots, J_n] \Rightarrow \forall l, j \in \{\!\{1..n\}\!\}.\ l \neq j \ \Rightarrow\ J_l \neq J_j \,\}\!\} \\
\llbracket \mathbf{t} \rrbracket^i_E & = & \mathit{JVal}(*) \\
\llbracket \mathbf{f} \rrbracket^i_E & = & \emptyset \\
\llbracket \mathsf{propNames}(S) \rrbracket^i_E & = & \{\!\{\, J \mid J = \{k_1 : J_1, \ldots, k_m : J_m\}, l \in \{\!\{1..m\}\!\} \Rightarrow k_l \in \llbracket S \rrbracket^i_E \,\}\!\} \\
\llbracket \mathsf{props}(r_1 : S_1, \ldots, r_n : S_n; S) \rrbracket^i_E & = & \{\!\{\, J \mid J = \{k_1 : J_1, \ldots, k_m : J_m\}, l \in \{\!\{1..m\}\!\} \Rightarrow \\
 & & \qquad (\forall j \in \{\!\{1..n\}\!\}.\ k_l \in L(r_j) \Rightarrow J_l \in \llbracket S_j \rrbracket^i_E) \wedge \\
 & & \qquad (k_l \notin L((r_1 | \ldots | r_n)) \Rightarrow J_l \in \llbracket S \rrbracket^i_E) \,\}\!\} \\
\llbracket \mathsf{items}(S_1, \ldots, S_n; S_{n+1}) \rrbracket^i_E & = & \{\!\{\, J \mid J = [J_1, \ldots, J_m], l \in \{\!\{1..m\}\!\} \Rightarrow \\
 & & \qquad (\forall j \in \{\!\{1..n\}\!\}.\ l = j \Rightarrow J_l \in \llbracket S_j \rrbracket^i_E) \wedge \\
 & & \qquad (l > n \Rightarrow J_l \in \llbracket S_{n+1} \rrbracket^i_E) \,\}\!\} \\
\llbracket \mathsf{contains}(S) \rrbracket^i_E & = & \{\!\{\, J \mid J = [J_1, \ldots, J_m] \Rightarrow \exists l \in \{\!\{1..m\}\!\}.\ J_l \in \llbracket S \rrbracket^i_E \,\}\!\} \\
\llbracket S_1 \wedge S_2 \rrbracket^i_E & = & \llbracket S_1 \rrbracket^i_E \cap \llbracket S_2 \rrbracket^i_E \\
\llbracket \neg S \rrbracket^i_E & = & \mathit{JVal}(*) \setminus \llbracket S \rrbracket^i_E \\
\llbracket S_1 \vee S_2 \rrbracket^i_E & = & \llbracket S_1 \rrbracket^i_E \cup \llbracket S_2 \rrbracket^i_E \\
\llbracket S_1 \Rightarrow S_2 \rrbracket^i_E & = & \llbracket \neg S_1 \vee S_2 \rrbracket^i_E \\
\llbracket (S_1 \Rightarrow S_2 \mid S_3) \rrbracket^i_E & = & \llbracket (S_1 \wedge S_2) \vee ((\neg S_1) \wedge S_3) \rrbracket^i_E \\
\llbracket \textcircled{1}(S_1, \ldots, S_n) \rrbracket^i_E & = & \llbracket \bigvee_{1 \leq l \leq n} (\neg S_1 \wedge \ldots \wedge \neg S_{l-1} \wedge S_l \wedge \neg S_{l+1} \wedge \ldots \wedge \neg S_n) \rrbracket^i_E \\
\llbracket \{S_1, \ldots, S_n\} \rrbracket^i_E & = & \llbracket S_1 \wedge \ldots \wedge S_n \rrbracket^i_E \\
\llbracket x \rrbracket^0_E & = & \emptyset \quad \text{(any arbitrary set of JSON values could be used)} \\
\llbracket x \rrbracket^{i+1}_E & = & \llbracket E(x) \rrbracket^i_E \\
\llbracket S \rrbracket_E & = & \bigcup_{i \in \mathbb{N}} \bigcap_{j \geq i} \llbracket S \rrbracket^j_E \\
\llbracket S \ \mathsf{defs}(x_1 : S_1, \ldots, x_n : S_n) \rrbracket & = & \llbracket S \rrbracket_{x_1 : S_1, \ldots, x_n : S_n}
\end{array}
$$

Figure 8: Semantics of the algebraic presentation of JSON Schema.

Schema defines a variable meaning by ruling that $x = E(x)$, we adopt the inductive limit definition that we described above. 1.1, 1.6

In order to show that our presentation is equivalent to JSON Schema, in Section 3.2 we detail the syntactic relationship among our presentation and the original syntax, while in Section 3.3, we prove the semantic equivalence between our formalization of variables and JSON Schema definition.

3.2. Translating between JSON Schema and the algebraic presentation

Our algebraic rendition differs syntactically from actual JSON Schema in the use of a more compact functional syntax. Furthermore, we eliminate the context-dependency of operators such 2.5 that "`additionalProperties`" and "`additionalItems`" by combining them with the operators on which they depend. We also replace the "`$ref`" reference mechanism, that does not enjoy substitutability, with a standard variable mechanism. Finally, we collect pairs of min-max operators into a single interval operator, mostly for space reasons, as we do with "`minimum`"-"`maximum`" combined into betw_i^j. 2.6

We can define a formal translation from JSON Schema to the algebraic form, as follows.

We say that a reference "`$ref`": *path* is *normalized* when *path* is either (i) "`#`" or (ii) "`#/definitions/`"$\cdot x$, where x is a string that contains no /, and $k_1 \cdot k_2$ is string concatenation. In a first phase, for every subschema S that is referred by a non-normalized path, we copy S in the "`definitions`" section of the schema, under a name $f(path)$, where f transforms the path into a flat string, and we substitute all references "`$ref`": *path* with a normalized reference "`$ref`": "`#/definitions/`"$\cdot f(path)$. At this point, this reference-normalized document is translated as follows, where $\langle S \rangle$ is the translation of S, *xroot* is a fresh variable, and where each occurrence of ""`$ref`" : "`#/definitions/`"$\cdot x$" is translated as "x" and each occurrence of ""`$ref`": "`#`"" is translated as "*xroot*":

$$\langle \{ \quad k_1 : S_1, \ldots, k_n : S_n, \texttt{"definitions"} : \{x_1 : S'_1, \ldots, x_m : S'_m\} \ \} \rangle \ = \\ xroot \ \ \mathsf{defs}(xroot : \langle \{k_1 : S_1, \ldots, k_n : S_n\} \rangle, x_1 : \langle S'_1 \rangle, \ldots, x_n : \langle S'_n \rangle)$$

After definition normalization, we can translate any assertion *keyword* : S into the corresponding algebraic operator, as reported in Figure 9, where we also show how the non-algebraic combination `properties-patternProperties-additionalProperties` is merged into a props operator, and how the combination `items-additionalItems` is merged into a items. For the `items-additionalItems` we detail all possible combinations. For the props triple, a missing "`properties`" is treated as "`properties`": {}, a missing "`patternProperties`" as "`patternProperties`":{}, and, finally, a missing "`additionalProperties`" assertion as "`additionalProperties`": { }. In the translation of "`properties`", we use $\underline{k}$ to represent the pattern that only matches k, so that "`properties`" : { k : S } is translated as $\mathsf{props}(\underline{k} : \langle S \rangle; \mathbf{t})$.

Definition 2. Given a string k, we denote with $\underline{k}$ the pattern only matching k, i.e., $\underline{k}$ = "^"$\cdot k' \cdot$"$", where k' is obtained from k by escaping all special characters.

To illustrate equivalence between our presentation and JSON Schema, we also define a formal translation in the other direction, defined in Figure 10, by means of the $\langle\!\langle \_ \rangle\!\rangle$ mapping. Almost all cases in the figure are self-explaining. Observe that our interval operators betw_m^M admit trivial bounds ∞ and $-\infty$ that indicate a missing bound; in this case the corresponding "`minimum`" and "`maximum`" are not generated; the same applies to xBetw_m^M and to the upper bound of len_l^j, pro_l^j, and ite_l^j. 1.1, 1.2

Observe that the two mappings $\langle \_ \rangle$ and $\langle\!\langle \_ \rangle\!\rangle$ describe the correspondence between our syntax and the original JSON Schema syntax, and each of them maps a schema into one that is satisfied by the same values, but $\langle\!\langle \langle S \rangle \rangle\!\rangle$ is not syntactically identical to S, since our syntax is more compact, hence different JSON Schema terms are represented by the same term in our syntax.

This correspondence between JSON Schema and our presentation entails that every property that we prove about the expressive power of the algebraic operators holds for the corresponding operators of JSON Schema. For illustrative purposes, we will also provide JSON Schema examples for some formal properties in the following. 1.5

$\langle\{\ G_1, \ldots, G_n\ \}\ \rangle$	$\{\langle G_1\rangle, \ldots, \langle G_n\rangle\}$
$\langle$"`allOf`": $[\ S_1, \ldots S_n\ ]\rangle$	$\wedge(\langle S_1\rangle, \ldots \langle S_n\rangle)$
$\langle$"`anyOf`": $[\ S_1, \ldots S_n\ ]\ \rangle$	$\vee(\langle S_1\rangle, \ldots \langle S_n\rangle)$
$\langle$"`oneOf`": $[\ S_1, \ldots S_n\ ]\ \rangle$	$\textcircled{1}(\langle S_1\rangle, \ldots \langle S_n\rangle)$
$\langle$"`not`": $S\ \rangle$	$\neg\langle S\rangle$
$\langle$"`if`": S_1, "`then`" : S_2, "`else`" : $S_3\ \rangle$	$\langle S_1\rangle \Rightarrow \langle S_2\rangle \mid \langle S_3\rangle$
$\langle$"`const`": $J\rangle$	$\mathsf{const}(J)$
$\langle$"`enum`": $[J_1, \ldots, J_n]\rangle$	$\mathsf{enum}(J_1, \ldots, J_n)$
$\langle$"boolean"/ "null"/ "number"/ "string"/ "array"/ "object"$\rangle$	Bool/Null/Num/Str/Arr/Obj
$\langle$"`type`": Tp$\rangle$ with Tp $\neq$ "*integer*"	$\mathsf{type}(\langle \mathrm{Tp}\rangle)$
$\langle$"`type`": "integer"$\rangle$	$\mathsf{type}(\mathsf{Num}) \wedge \mathsf{mulOf}(1)$
$\langle$"`type`": $[\mathrm{Tp}_1, \ldots, \mathrm{Tp}_n]\ \rangle$ with every $\mathrm{Tp}_i \neq$ "*integer*"	$\mathsf{type}(\langle \mathrm{Tp}_1\rangle, \ldots, \langle \mathrm{Tp}_n\rangle)$
$\langle$"`type`": $[\mathrm{Tp}_1, \ldots, \mathrm{Tp}_n,$"integer"$]\ \rangle$	$\mathsf{type}(\langle \mathrm{Tp}_1\rangle, \ldots, \langle \mathrm{Tp}_n\rangle) \wedge \mathsf{mulOf}(1)$
$\langle$"`minimum`": m $\rangle$	betw_m^{∞}
$\langle$"`maximum`": M$\rangle$	$\mathsf{betw}_{-\infty}^{M}$
$\langle$"`exclusiveMinimum`": m $\rangle$	$\mathsf{xBetw}_m^{\infty}$
$\langle$"`exclusiveMaximum`": M$\rangle$	$\mathsf{xBetw}_{-\infty}^{M}$
$\langle$"`multipleOf`": q$\rangle$	$\mathsf{mulOf}(q)$
$\langle$"`minLength`": m $\rangle$	len_m^{∞}
$\langle$"`maxLength`": M $\rangle$	len_0^{M}
$\langle$"`pattern`": r $\rangle$	$\mathsf{pattern}(r)$
$\langle$"`uniqueItems`": "`true`"$\rangle$	uniqueItems
$\langle$"`uniqueItems`": "`false`"$\rangle$	**t**
$\langle$"`minItems`": m$\rangle$	ite_m^{∞}
$\langle$"`maxItems`": M$\rangle$	ite_0^{M}
$\langle$"`contains`": $S\ \rangle$	$\mathsf{contains}(\langle S\rangle)$
$\langle$"`items`": S, "`additionalItems`": $S'\rangle$	$\mathsf{items}(; \langle S\rangle)$
$\langle$"`items`": $S\rangle$	$\mathsf{items}(; \langle S\rangle)$
$\langle$"`items`": $[S_1, \ldots, S_n]$, "`additionalItems`": $S'\rangle$	$\mathsf{items}(\langle S_1\rangle, \ldots, \langle S_n\rangle; \langle S'\rangle)$
$\langle$"`items`": $[S_1, \ldots, S_n]\rangle$	$\mathsf{items}(\langle S_1\rangle, \ldots, \langle S_n\rangle; \mathbf{t})$
$\langle$"`additionalItems`": $S\rangle$	$\mathsf{items}(; \langle S\rangle)$
$\langle$"`minProperties`": m $\rangle$	pro_m^{∞}
$\langle$"`maxProperties`": M$\rangle$	pro_0^{M}
$\langle$"`propertyNames`": $S\rangle$	$\mathsf{propNames}(\langle S\rangle)$
$\langle$"`required`": $[\ k_1, \ldots, k_n\ ]\rangle$	$\mathsf{req}(k_1, \ldots, k_n)$
$\langle$"`properties`": $\{k_1 : S_1, \ldots, k_n : S_n\ \}$, "`patternProperties`": $\{r_1 : S'_1, \ldots, r_m : S'_m\ \}$, "`additionalProperties`": $S\ \rangle$	$\mathsf{props}(\underline{k_1} : \langle S_1\rangle, \ldots, \underline{k_n} : \langle S_n\rangle$ $r_1 : \langle S'_1\rangle, \ldots, r_m : \langle S'_m\rangle;$ $\langle S\rangle)$
$\langle$"`dependentSchemas`": $\{\ k_1 : S_1, \ldots, k_n : S_n\ \}\ \rangle$	$((\mathsf{type}(\mathsf{Obj}) \wedge \mathsf{req}(k_1)) \Rightarrow \langle S_1\rangle) \wedge$ $\ldots \wedge ((\mathsf{type}(\mathsf{Obj}) \wedge \mathsf{req}(k_n)) \Rightarrow \langle S_n\rangle)$
$\langle$"`dependentRequired`": $\{\ k_1 : [r^1_1 \ldots, r^1_{m_1}], \ldots, k_n : [r^n_1 \ldots, r^n_{m_n}]\ \}\ \rangle$	$(\mathsf{req}(k_1) \Rightarrow \mathsf{req}(r^1_1 \ldots, r^1_{m_1})) \wedge$ $\ldots \wedge (\mathsf{req}(k_n) \Rightarrow \mathsf{req}(r^n_1 \ldots, r^n_{m_n}))$
$\langle$"`dependencies`": *obj* $\rangle$	see two previous cases
$\langle k_1 : S_1, \ldots, k_m : S_m$, "`definitions`": $\{\ x_1 : S'_1, \ldots, x_n : S'_n\}\ \rangle$	$xroot\ \mathsf{defs}(xroot : \langle\{k_1 : S_1, \ldots, k_m : S_m\}\rangle,$ $x_1 : \langle S'_1\rangle, \ldots, x_n : \langle S'_n\rangle)$
$\langle$"`$ref`": "`#/definitions/`"$\cdot x\rangle$	x
$\langle$"`$ref`": "`#`"$\rangle$	$xroot$

Figure 9: Translation from JSON Schema to the algebra.

$\langle\!\langle \mathbf{t} \rangle\!\rangle$	`true`
$\langle\!\langle \mathbf{f} \rangle\!\rangle$	`false`
$\langle\!\langle \{S_1, \ldots, S_n\} \rangle\!\rangle$	$\{ \langle\!\langle S_1 \rangle\!\rangle, \ldots, \langle\!\langle S_n \rangle\!\rangle \}$
$\langle\!\langle \wedge(S_1, \ldots, S_n) \rangle\!\rangle$	"`allOf`": $[\langle\!\langle S_1 \rangle\!\rangle, \ldots, \langle\!\langle S_n \rangle\!\rangle]$
$\langle\!\langle \vee(S_1, \ldots, S_n) \rangle\!\rangle$	"`anyOf`": $[\langle\!\langle S_1 \rangle\!\rangle, \ldots, \langle\!\langle S_n \rangle\!\rangle]$
$\langle\!\langle \textcircled{1}(S_1, \ldots, S_n) \rangle\!\rangle$	"`oneOf`": $[\langle\!\langle S_1 \rangle\!\rangle, \ldots, \langle\!\langle S_n \rangle\!\rangle]$
$\langle\!\langle \neg S \rangle\!\rangle$	"`not`": $\langle\!\langle S \rangle\!\rangle$
$\langle\!\langle S_1 \Rightarrow S_2 \mid S_3 \rangle\!\rangle$	"`if`": $\langle\!\langle S_1 \rangle\!\rangle$, "`then`" : $\langle\!\langle S_2 \rangle\!\rangle$, "`else`" : $\langle\!\langle S_3 \rangle\!\rangle$
$\langle\!\langle \mathsf{const}(J) \rangle\!\rangle$	"`const`": $\langle\!\langle J \rangle\!\rangle$
$\langle\!\langle \mathsf{enum}(J_1, \ldots, J_n) \rangle\!\rangle$	$\langle$"`enum`": $[J_1, \ldots, J_n]\rangle$
$\langle\!\langle$Bool/Null/Num/ Str/Arr/Obj$\rangle\!\rangle$	"boolean"/ "null"/ "number"/ "string"/ "array"/ "object"
$\langle\!\langle \mathsf{type}(S_1, \ldots, S_n) \rangle\!\rangle$	"`type`": $[\langle\!\langle S_1 \rangle\!\rangle, \ldots, \langle\!\langle S_n \rangle\!\rangle]$
$\langle\!\langle \mathsf{betw}_m^M \rangle\!\rangle$	"`minimum`": m , "`maximum`": M
$\langle\!\langle \mathsf{xBetw}_m^M \rangle\!\rangle$	"`exclusiveMinimum`": m , "`exclusiveMaximum`": M
$\langle\!\langle \mathsf{mulOf}(q) \rangle\!\rangle$	"`multipleOf`": q
$\langle\!\langle \mathsf{len}_m^M \rangle\!\rangle$	"`minLength`": m , "`maxLength`": M
$\langle\!\langle \mathsf{pattern}(r) \rangle\!\rangle$	"`pattern`": r
$\langle\!\langle \mathsf{uniqueItems} \rangle\!\rangle$	"`uniqueItems`": "`true`"
$\langle\!\langle \mathsf{ite}_m^M \rangle\!\rangle$	"`minItems`": m , "`maxItems`": M
$\langle\!\langle \mathsf{contains}(S) \rangle\!\rangle$	"`contains`": $\langle\!\langle S \rangle\!\rangle$
$\langle\!\langle \mathsf{items}(S_1, \ldots, S_n; S') \rangle\!\rangle$	"`items`": $[\langle\!\langle S_1 \rangle\!\rangle, \ldots, \langle\!\langle S_n \rangle\!\rangle]$, "`additionalItems`": $\langle\!\langle S' \rangle\!\rangle$
$\langle\!\langle \mathsf{pro}_m^M \rangle\!\rangle$	"`minProperties`": m, "`maxProperties`": M
$\langle\!\langle \mathsf{propNames}(S) \rangle\!\rangle$	"`propertyNames`": $\langle\!\langle S \rangle\!\rangle$
$\langle\!\langle \mathsf{req}(k_1, \ldots, k_n) \rangle\!\rangle$	"`required`": $[k_1, \ldots, k_n]$
$\langle\!\langle \mathsf{props}(\underline{k_1} : S_1, \ldots, \underline{k_n} : S_n$ $r_1 : S'_1, \ldots, r_m : S'_m;$ $S) \rangle\!\rangle$	"`properties`": $\{k_1 : \langle\!\langle S_1 \rangle\!\rangle, \ldots, k_n : \langle\!\langle S_n \rangle\!\rangle \}$, "`patternProperties`": $\{r_1 : \langle\!\langle S'_1 \rangle\!\rangle, \ldots, r_m : \langle\!\langle S'_m \rangle\!\rangle \}$, "`additionalProperties`": $\langle\!\langle S \rangle\!\rangle$
$\langle\!\langle S\ \mathsf{defs}(x_1 : S'_1, \ldots, x_n : S'_n) \rangle\!\rangle$	$\langle\!\langle S \rangle\!\rangle$, "`definitions`": $\{ x_1 : \langle\!\langle S'_1 \rangle\!\rangle, \ldots, x_n : \langle\!\langle S'_n \rangle\!\rangle \}$
$\langle\!\langle x \rangle\!\rangle$	"`$ref`": "`#/definitions/`"$\cdot x$

"`minimum`" : m and "`exclusiveMinimum`" : m are not generated when $m = -\infty$
"`maximum`" : M and "`exclusiveMaximum`" : M are not generated when $M = \infty$
"`maxLength`" : M, "`maxItems`" : M, and "`maxProperties`" : M are not generated when $M = \infty$

Figure 10: Translation from the algebra to JSON Schema.

3.3. Semantic equivalence of the algebraic presentation and original JSON Schema

Our presentation differs from original JSON Schema in the syntax, which has however a strict correspondence as illustrated in the previous section, and in our adoption of an inductive stratified semantic to describe variables. In JSON Schema, it is specified that a variable is equivalent to its definition, that is that

$$[\![x]\!]_E = [\![E(x)]\!]_E.$$

This is a very natural property, but it is not an inductive definition, since in general $E(x)$ is bigger than x. For this reason, we adopted the definition of Figure 8, that allows us to conduct all our proofs by induction. In this section, however, we prove that the property $[\![x]\!]_E = [\![E(x)]\!]_E$ derives from our definition. Moreover, recall that we defined the interpretation of all operators by translating JSON Schema definitions into equations for $[\![S]\!]^i_E$, as in the following example: 1.1,1.6

$$[\![\mathsf{propNames}(S)]\!]^i_E \quad = \{\!\!\{\ J \mid \quad J = \{k_1 : J_1, \ldots, k_m : J_m\}, l \in \{\!\!\{1..m\}\!\!\} \Rightarrow k_l \in [\![S]\!]^i_E \quad \}\!\!\}.$$

In order to prove that our semantics $[\![S]\!]_E$ really corresponds to that of JSON Schema we have now to prove that these equations are preserved when the limit $[\![S]\!]_E$ of the $[\![S]\!]^i_E$ succession is computed, so that, for example, for $\mathsf{propNames}(S)$ we have: 1.1,1.6

$$[\![\mathsf{propNames}(S)]\!]_E \quad = \{\!\!\{\ J \mid \quad J = \{k_1 : J_1, \ldots, k_m : J_m\}, l \in \{\!\!\{1..m\}\!\!\} \Rightarrow k_l \in [\![S]\!]_E \quad \}\!\!\}.$$

These properties are proved in Theorem 1 below, after Lemma 1. Lemma 1 expresses the fact that, having fixed a specific JSON value J and a schema S, the question whether $J \in [\![S]\!]^j_E$ may have different answers for small values of j, but, after a certain index $I(J, S, E)$, the answer to this question converges to either *true* or *false*, even in presence of negative variables, provided that recursion is guarded. With this property, the proof of Theorem 1 becomes quite easy. 1.1,1.6

Lemma 1 (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:*

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

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

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

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

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

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

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 $S = \mathsf{items}(S_1, \ldots, S_n; S_{n+1})$. We want to prove that:
$(\forall j \geq I(J, S, E).\ J \in [\![\mathsf{items}(S_1, \ldots, S_n; S_{n+1})]\!]^j_E)$
$\vee\ (\forall j \geq I(J, S, E).\ J \notin [\![\mathsf{items}(S_1, \ldots, S_n; S_{n+1})]\!]^j_E)$
Consider the definition of $[\![\mathsf{items}(S_1, \ldots, S_n; S_{n+1})]\!]^j_E$:
$\{J \mid J = [J_1, \ldots, J_m], l \in \{\!\{1..m\}\!\} \Rightarrow ((\forall k \in \{\!\{1..n\}\!\}.\ l = k \Rightarrow J_l \in [\![S_k]\!]^j_E) \wedge (l > n \Rightarrow J_l \in [\![S_{n+1}]\!]^j_E)\ )\}$
The boolean value of the statement "$J \in [\![\mathsf{items}(S_1, \ldots, S_n; S_{n+1})]\!]^j_E$" converges after it converges for all pairs (J_l, S_k) and (J_l, S_{n+1}) that appear in the definition. Hence, we let $I(J, S, E)$ be the maximum among all the indexes $I(J_l, S_k, E)$ for the pairs $l = k$ and the indexes $I(J_l, S_{n+1}, E)$ for $l > n$, and we conclude by induction. The fact that all these indexes are well defined derives, by induction, from the fact that each J_l is a strict subterm of J. Observe that the fact that each J_l is *strictly* smaller than J is essential since, in general, the degree of each S_k may be bigger than the degree of S, since they are all in a guarded position. 2.3

Observe that, while we proceed by induction on S with the boolean operators, which apply smaller schemas on the same term, we use induction on J when working with typed operators. All operators of the language can be treated in the same way.

□

We can finally prove the semantics equivalence between our presentation and JSON Schema. 1.1, 1.6

Theorem 1 (Equivalence)**.** For any E guarded, the following equality holds:

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

Moreover, for each equivalence in Figure 8, the equivalence still holds if we substitute every occurrence of $[\![S]\!]^i_E$ with $[\![S]\!]_E$.

Proof. This is an immediate consequence of convergence (Lemma 1). Consider any equation such as:

$$[\![\mathsf{propNames}(S)]\!]^i_E \quad = \{\!\{ \; J \mid \quad J = \{k_1 : J_1, \ldots, k_m : J_m\}, l \in \{\!\{1..m\}\!\} \Rightarrow k_l \in [\![S]\!]^i_E \quad \}\!\}.$$

That is:

$$J \in [\![\mathsf{propNames}(S)]\!]^i_E \;\Leftrightarrow\; J = \{k_1 : J_1, \ldots, k_m : J_m\}, l \in \{\!\{1..m\}\!\} \Rightarrow k_l \in [\![S]\!]^i_E$$

Informally, if we consider any integer I that is bigger than $I(J, [\![\mathsf{propNames}(S)]\!]^i_E, E)$ and of every $I(k_l, S, E)$, then, if the equation holds for one index $i \geq I$, 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:

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

We use Lemma 1 and the fact that $I(J, x, E) = I(J, E(x), E) + 1$:
$J \in \bigcup_{i \in \mathbb{N}} \bigcap_{j \geq i} [\![x]\!]^j_E$
$\Leftrightarrow\; \forall j \geq I(J, x, E).\ J \in [\![x]\!]^j_E$
$\Leftrightarrow\; \forall j \geq I(J, E(x), E) + 1.\ J \in [\![x]\!]^j_E$
$\Leftrightarrow\; \forall j \geq I(J, E(x), E) + 1.\ J \in [\![E(x)]\!]^{j-1}_E$
$\Leftrightarrow\; \forall j \geq I(J, E(x), E).\ J \in [\![E(x)]\!]^j_E$
$\Leftrightarrow\; J \in \bigcup_{i \in \mathbb{N}} \bigcap_{j \geq i} [\![E(x)]\!]^j_E$

For the second property, the crucial case is that for $J \in [\![\neg S]\!]_E$, where we want to prove: $J \in [\![\neg S]\!]_E \;\Leftrightarrow J \in JVal(*) \setminus [\![S]\!]_E$.

We use Lemma 1 and the fact that $I(J, \neg S, E) = I(J, S, E)$:

$J \in [\![\neg S]\!]_E$
$\Leftrightarrow\; \forall j \geq I(J, \neg S, E).\ J \in [\![\neg S]\!]^j_E$
$\Leftrightarrow\; \forall j \geq I(J, S, E).\ J \in [\![\neg S]\!]^j_E$
$\Leftrightarrow\; \forall j \geq I(J, S, E).\ J \notin [\![S]\!]^j_E$
$\Leftrightarrow (*)\ \exists j \geq I(J, S, E).\ J \notin [\![S]\!]^j_E$
$\Leftrightarrow\; \neg(\forall j \geq I(J, S, E).\ J \in [\![S]\!]^j_E)$
$\Leftrightarrow\; J \notin [\![S]\!]_E$
$\Leftrightarrow\; J \in JVal(*) \setminus [\![S]\!]_E$

For the crucial $\Leftrightarrow$ $(*)$ step, the $\Rightarrow$ direction is immediate. The direction $\Leftarrow$ derives from the convergence property of $I(J, S, E)$, once $J \notin [\![S]\!]^j_E$ holds for an index bigger than $I(J, S, E)$: then it holds for any such index.

For all other cases, consider for example the case where $J \in [\![\mathsf{items}(S_1, \dots, S_n; S_{n+1})]\!]_E$. By Lemma 1, if we let $I = I(J, \mathsf{items}(S_1, \dots, S_n; S_{n+1}), E)$, we have the first equivalence below, from which other equivalence follow in order to prove the case.

$J \in [\![\mathsf{items}(S_1, \dots, S_n; S_{n+1})]\!]_E$
$\Leftrightarrow\ \forall j \geq I.\ J = [J_1, \dots, J_m], l \in \{\!\{1..m\}\!\} \Rightarrow\ \ (\forall k \in \{\!\{1..n\}\!\}.\ l = k \Rightarrow J_l \in [\![S_k]\!]^j_E) \wedge (l > n \Rightarrow J_l \in [\![S_{n+1}]\!]^j_E)\ )$
$\Leftrightarrow (*) J = [J_1, \dots, J_m], l \in \{\!\{1..m\}\!\} \Rightarrow$
$\forall j \geq I.(\ (\forall k \in \{\!\{1..n\}\!\}.\ l = k \Rightarrow J_l \in [\![S_k]\!]^j_E) \wedge (l > n \Rightarrow J_l \in [\![S_{n+1}]\!]^j_E)\ )$
$\Leftrightarrow (**)\ J = [J_1, \dots, J_m], l \in \{\!\{1..m\}\!\} \Rightarrow$
$\forall j \geq I.(\forall k \in \{\!\{1..n\}\!\}.\ l = k \Rightarrow J_l \in [\![S_k]\!]^j_E) \wedge\ \forall j \geq I.(l > n \Rightarrow J_l \in [\![S_{n+1}]\!]^j_E)\ )$
$\Leftrightarrow (*)\ J = [J_1, \dots, J_m], l \in \{\!\{1..m\}\!\} \Rightarrow$
$(\forall k \in \{\!\{1..n\}\!\}.\ l = k \Rightarrow \forall j \geq I. J_l \in [\![S_k]\!]^j_E) \wedge (l > n \Rightarrow \forall j \geq I. J_l \in [\![S_{n+1}]\!]^j_E)$
$\Leftrightarrow\ J = [J_1, \dots, J_m], l \in \{\!\{1..m\}\!\} \Rightarrow$
$(\forall k \in \{\!\{1..n\}\!\}.\ l = k \Rightarrow J_l \in [\![S_k]\!]_E) \wedge (l > n \Rightarrow J_l \in [\![S_{n+1}]\!]_E)$

The two $(*)$ steps are justified by the fact that premises not depending by the universally quantified i can be traversed by the universal quantification, while $(**)$ follows from the simple rule $\forall x.(P(x) \wedge Q(x)) \Leftrightarrow \forall x.P(x) \wedge \forall x.Q(x)$. The last equivalence follows from the fact that $I(J, \mathsf{items}(S_1, \dots, S_n; S_{n+1}), E)$ is greater than each $I(J, S_i, E)$, for $i \in \{\!\{1..n+1\}\!\}$ and Lemma 1.

All other cases follow easily from convergence. Consider for example the case where $J \in [\![\mathsf{contains}(S)]\!]_E$. We want to prove:

$$J \in [\![\mathsf{contains}(S)]\!]_E \ \Leftrightarrow\ (J = [J_1, \dots, J_n] \Rightarrow \exists l \in \{\!\{1..n\}\!\}.\ J_l \in [\![S]\!]_E)$$

If J is not an array, the double implication holds trivially. Consider now the case $J = [J_1, \dots, J_n]$; we exploit the fact that $I(J, \mathsf{contains}(S), E)$ is greater than $I(J_l, S, E)$ for every $l \in \{\!\{1..n\}\!\}$:

$J \in [\![\mathsf{contains}(S)]\!]_E$
$\Leftrightarrow\ \forall j \geq I(J, S, E).\ J \in [\![\mathsf{contains}(S)]\!]^j_E$
$\Leftrightarrow\ \forall j \geq I(J, S, E).\ \exists l \in \{\!\{1..n\}\!\}.\ J_l \in [\![S]\!]^j_E$
$\Leftrightarrow (*)\ \exists l \in \{\!\{1..n\}\!\}. \forall j \geq I(J, S, E).\ J_l \in [\![S]\!]^j_E$
$\Leftrightarrow\ \exists l \in \{\!\{1..n\}\!\}.\ J_l \in [\![S]\!]_E$

For the $\Leftrightarrow (*)$ step, the right-to-left direction is obvious. In the other direction, consider any j_0 with $j_0 \geq I(J, S, E)$, by hypothesis there exists l_0 such that $J_{l_0} \in [\![S]\!]^{j_0}_E$. Since $I(J_{l_0}, S, E) \leq I(J, S, E)$, we have that the value of the boolean assertion $J_{l_0} \in [\![S]\!]^{j_0}_E$ is fixed for every j_0 greater than $I(J, S, E)$, hence $\forall j \geq I(J, S, E).\ J_{l_0} \in [\![S]\!]^j_E$, hence $\exists l \in \{\!\{1..n\}\!\}. \forall j \geq I(J, S, E).\ J_l \in [\![S]\!]^j_E$.

All other cases are similar, or easier.

□

4. Failure of negation closure

In this section we will discuss the negation properties of JSON Schema and show that it is *almost closed* under negation, but there are assertions that cannot be expressed without negation, and we exactly characterize them.

4.1. Preliminaries

Definition 3 (Equivalence: $(S, E) \equiv (S', E')$, $S \equiv S'$). We say that (S, E) is equivalent to (S', E'), or $(S, E) \equiv (S', E')$, iff $[\![S]\!]_E = [\![S']\!]_{E'}$.

We say that S is equivalent to S', or $S \equiv S'$, iff they are defined over the same variables and, for any environment E that is guarded and is closing for both, $(S, E) \equiv (S', E)$.

4.2. Constraints and requirements

We divide object and array assertions into *constraints* and *requirements*, which are summarized in Table 1.

Table 1: Constraints and Requirements.

	Objects	Arrays
Constraints	pro_0^M $(M \neq \infty)$	ite_0^M $(M \neq \infty)$
	$\mathsf{props}(\underline{k_1} : S_1, \ldots, \underline{k_n} : S_n; S)$	$\mathsf{items}(S_1, \ldots, S_n; S)$
	$\mathsf{propNames}(S)$	$\mathsf{uniqueItems}$
Requirements	pro_m^∞ $(m \geq 1)$	ite_m^∞ $(m \geq 1)$
	$\mathsf{req}(k_1, \ldots, k_n)$	$\mathsf{contains}(S)$

An object assertion S is a *constraint* if it behaves as a universally quantification over all fields of an object: it is satisfied by the empty object and the following implication holds, when J^+ is an object obtained by adding a member to J:

$$J^+ \in [\![S]\!]_E \Rightarrow J \in [\![S]\!]_E$$

Hence, constraints can only be violated by *adding* a member: it *prevents* the addition of some specific members, but does not require the presence of a member. The object constraints are the assertions $\mathsf{props}(r_1 : S_1, \ldots, r_n : S_n; S)$, $\mathsf{propNames}(S)$ and pro_0^M with $M \neq \infty$.

Dually, an object assertion S is a *requirement* if it behaves as an existential quantification over the fields of an object: it is violated by the empty object and the following implication holds, when J^+ is an object obtained by adding a member to J:

$$J \in [\![S]\!]_E \Rightarrow J^+ \in [\![S]\!]_E$$

Hence, a requirement can only be violated by *removing* a member: it *requires* the presence of some specific members. The object requirements are the assertions $\mathsf{req}(k_1, \ldots, k_n)$ and pro_m^∞ with $m \geq 1$.

In the same way, we distinguish array constraints and requirements. An array constraint S is satisfied by the empty array and the following implication holds, when J^+ is an array obtained by adding an element to J:

$$J^+ \in [\![S]\!]_E \Rightarrow J \in [\![S]\!]_E$$

The array constraints are $\mathsf{items}(S_1, \ldots, S_n; S)$, $\mathsf{uniqueItems}$, and ite_0^M with $M \neq \infty$.

Dually, an array requirement is violated by the empty array and the following implication holds, when J^+ is an array obtained by adding an element to J:

$$J \in [\![S]\!]_E \Rightarrow J^+ \in [\![S]\!]_E$$

Hence, an array constraint may prevent the addition of an element, while a requirement may require the presence of some specific elements. The array requirements are the assertions $\mathsf{contains}(S)$ and $\mathsf{ite}^{\infty}_{m}$ with $m \geq 1$.

4.3. *JSON Schema is almost negation-closed, but not exactly*

We say that a logic is negation-closed if, for every formula, there exists an equivalent one where no negation operator appears. In our presentation of JSON Schema, negation operators include $S_1 \Rightarrow S_2$, $S_1 \Rightarrow S_2 \mid S_3$ and $①(S_1, \ldots, S_n)$, since $\neg S$ can be also expressed as $S \Rightarrow \mathbf{f}$ or as $①(S, \mathbf{t})$. Negation-closure is usually obtained by coupling each algebraic operator with a dual operator that is used to push negation inside the first one. We are going to prove here that negation-closure is "almost" true for JSON Schema but not completely, and we are going to exactly describe the situations where negation cannot be pushed through JSON Schema operators.

We first examine this issue for object-related assertions and we then move to arrays. 2.11

4.3.1. *Objects*

According to our collection of GitHub JSON Schema documents, the most common usage patterns for $\mathsf{props}(r_1 : S_1, \ldots, r_n : S_n; S_a)$ are those where each r_i is the pattern $\underline{k_i}$ that only matches the string k_i, generated by the use of the JSON Schema operator "`properties`", and where S_a is either $\mathbf{t}$ or $\mathbf{f}$.

When every pattern has the form $\underline{k_i}$ and S_a is $\mathbf{t}$, then negation can be easily pushed through props, as specified by Property 1.

Property 1 (Negation of the most common use case for props**).**

$$\neg\mathsf{props}(\underline{k_1} : S_1, \ldots, \underline{k_n} : S_n; \mathbf{t}) \quad \equiv \quad \mathsf{type}(\mathsf{Obj}) \wedge \bigvee_{i \in \{\!\{1..n\}\!\}} (\mathsf{req}(k_i) \wedge \mathsf{props}(\underline{k_i} : \neg S_i; \mathbf{t}))$$

Proof. By definition, $J \in [\![\mathsf{props}(\underline{k_1} : S_1, \ldots, \underline{k_n} : S_n; \mathbf{t})]\!]_E$ iff

$$\forall l.\ J = \{k'_1 : J_1, \ldots, k'_m : J_m\}, l \in \{\!\{1..m\}\!\} \Rightarrow \ (\forall i \in \{\!\{1..n\}\!\}.\ k'_l \in L(\underline{k_i}) \Rightarrow J_l \in [\![S_i]\!]_E)$$

Hence, $J \notin [\![\mathsf{props}(\underline{k_1} : S_1, \ldots, \underline{k_n} : S_n; \mathbf{t})]\!]_E$ iff

$$\begin{aligned}
&\exists l.\ J = \{k'_1 : J_1, \ldots, k'_m : J_m\} \wedge l \in \{\!\{1..m\}\!\} \wedge (\exists i \in \{\!\{1..n\}\!\}.\ k'_l = k_i \wedge J_l \notin [\![S_i]\!]_E) \\
\Leftrightarrow \quad & J = \{k'_1 : J_1, \ldots, k'_m : J_m\} \wedge \exists i \in \{\!\{1..n\}\!\}.\ \exists l \in \{\!\{1..m\}\!\}.\ k'_l = k_i \wedge J_l \in [\![\neg S_i]\!]_E) \\
\Leftrightarrow (*) \quad & J \in [\![\mathsf{type}(\mathsf{Obj}) \wedge \textstyle\bigvee_{i \in \{\!\{1..n\}\!\}} (\mathsf{req}(k_i) \wedge \mathsf{props}(\underline{k_i} : \neg S_i; \mathbf{t}))]\!]_E
\end{aligned}$$

In the last equivalence we exploit the fact that any name can only appear once in an object, hence saying that one field (k'_l, J_l) with $k'_l = k_i$ appears in the object such that $J_l \in [\![\neg S_i]\!]_E$ is equivalent to saying that one field with that name exists ($\mathsf{req}(k_i)$) and that *every* field with that name has a value in $\neg S$ ($\mathsf{props}(\underline{k_i} : \neg S_i; \mathbf{t})$). □

Property 1 shows that, in the most common case, req can be used to express negation of props in a quite natural way, although with a bit of complexity, since we need an extra "$\mathsf{type}(\mathsf{Obj}) \wedge \ldots$" to negate the implicative part of props semantics, a $\bigvee_{i \in \{\!\{1..n\}\!\}}$ to negate its implicit conjunction, and every $\mathsf{req}(k_i)$ must be paired with a $\mathsf{props}(\underline{k_i} : \neg S_i; \mathbf{t})$ in order to express both the fact that k_i is required, and the fact that its schema must violate S_i.

Example 2. Consider this fairly simple schema.

```
{ "not" : { "$ref" : "#/definitions/main"},
  "definitions" : { "main" : {
                             "properties":{"name": {"type" : "string"},
                                           "surname" : {"type" : "string"}},
                             "additionalProperties" : true}}}
```

This schema is satisfied by any value that does not satisfy the `main` definition, which only constrains the type of *name* and *surname* properties in objects.

According to Property 1, this schema is equivalent to the one shown below, illustrating how negation can be pushed down, inside `"properties"` assertions. 1.4

```
{ "type" : "object",
  "anyOf" : [{"required" : ["name"],
              "properties" : {"name" : {"not": {"type" : "string"}}},
              "additionalProperties" : true},
             {"required" : ["surname"],
              "properties" : {"surname" : {"not": {"type" : "string"}}},
              "additionalProperties" : true}]}
```

This schema is, hence, satisfied by any object having at least one property *name* or *surname* (or both) not of type string, such as:

```
{ "name": 1564, "surname" : "Galilei", "year" : 1642}
```

The situation becomes much more involved when the `"additionalProperties"` part of props is not **t** but is **f**, which is violated by the presence of a member name that does not belong to $\{\!\{k_1, \ldots, k_n\}\!\}$. We do not have a straightforward way, in JSON Schema, to express the requirement that an object contains a member whose name is not in a set $\{\!\{k_1, \ldots, k_n\}\!\}$, but this fact can be nevertheless expressed, by stating that, for some i, the object contains at least $i+1$ members, but at most i members whose names belong to $\{\!\{k_1, \ldots, k_n\}\!\}$.

We first present a way to express in JSON Schema the fact that at most i members whose names belong to $\{\!\{k_1, \ldots, k_n\}\!\}$ belong to an object, using a term whose size is quadratic in n. The encoding is not immediate, but the same technique will be reused in the paper for different problems.

We use $I(l, p)$ to denote the p-th interval of size 2^l starting from $[1, 2^l]$, so that $I(2, 1)$, $I(2, 2)$, $I(2, 3), \ldots$ stands for $[1, 4]$, $[5, 8]$, $[9, 12], \ldots$, and $I(0, p)$ is the singleton $[p, p]$. Every interval $I(l, p)$ at level l is the union of two smaller intervals at level $l-1$, as depicted in Figure 11, according to this equation:

$$I(l, p) = I(l-1, 2p-1) \cup I(l-1, 2p) \qquad (\dagger)$$

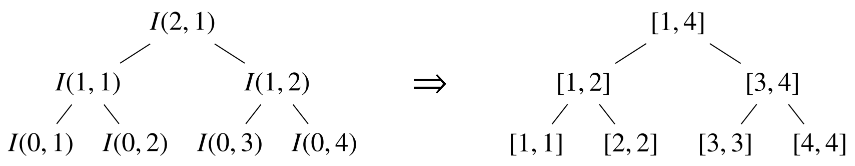

Figure 11: Structure of the tree of intervals.

Definition 4 ($I(l,p)$)**.**

$$I(l,p) \;=\; \{\!\{\; i \mid ((p-1)\times 2^l + 1) \le i \le (p \times 2^l) \;\}\!\}$$

Now, for each interval $I(l,p)$ included in $\{\!\{1..n\}\!\}$, we define a set of schema variables $U^u_{l,p}$, collected in an environment $\mathcal{E}(\{\!\{k_1,\ldots,k_n\}\!\})$, such that the objects in $[\![U^u_{l,p}]\!]_{\mathcal{E}}$ are all and only the objects with less than u names in $\{\!\{k_1,\ldots,k_n\}\!\}$ whose index belongs to $I(l,p)$, that is:

$$J \in [\![U^u_{l,p}]\!]_{\mathcal{E}} \;\Leftrightarrow\; |\,\{\!\{\, k_i \mid (k_i, \_) \in J \wedge i \in I(l,p)\,\}\!\}\,| \le u$$

Since, by (†), $I(l,p) = I(l-1,2p-1) \cup I(l-1,2p)$, we have that $J \in [\![U^u_{l,p}]\!]_{\mathcal{E}}$ iff a number i exists such that J satisfies both $U^i_{l-1,2p-1}$ and $U^{u-i}_{l-1,2p}$ — that is, u is an upper limit for the names of the object whose index belongs to the interval iff exists i such that i is an upper limit for the names whose index belongs to the first half of the interval and $u-i$ is an upper limit for the names whose index belongs to the second half of the interval: 2.7

$$U^u_{l,p} \quad : \quad \bigvee_{0\le i\le u}\,(\,U^i_{l-1,2p-1} \wedge U^{u-i}_{l-1,2p}\,) \qquad (\dagger\dagger)$$

Hence, we define the environment $\mathcal{E}(\{\!\{k_1,\ldots,k_n\}\!\})$ as follows. In the definition of $\mathcal{E}(\{\!\{k_1,\ldots,k_n\}\!\})$ we do not assume that n is a power of 2, hence we need the second line. The interesting equation is the third one, that we introduced above. Observe that the equation for $U^u_{l,p}$, in the case with $u = 2^l - 1$, requires that $U^i_{l-1,\_}$ is defined for i up to $2^l - 1$, which we ensure by adding, in line 4, a trivial definition for a generic $U^{u'}_{l',p'}$ for any u' up to $2^{l'+1} - 1$. Hence, when $u < 2^l$, the variable is defined by the (††) equation (line 3). When $u \ge 2^l$, then the set $I(l,p)$ contains less than u elements, hence any object trivially contains less than u elements whose names are indexed by $I(u,p)$, hence all variables $U^u_{l,p}$ with $2^l \le u$ are satisfied by any object. These trivial variables could be easily optimized away, at the price of expressing the third line is a way that would be slightly more complicated.

$$\begin{array}{llll}
\multicolumn{4}{l}{\mathcal{E}(\{\!\{k_1,\ldots,k_n\}\!\}) =} \\
\quad (& & & \\
\textbf{(1)}\ U^0_{0,p} & : & \mathsf{props}(k_p : \mathbf{f}; \mathbf{t}) & 1 \le p \le n \\
\textbf{(2)}\ U^0_{0,p} & : & \mathbf{t} & n+1 \le p \le 2^{\lceil log_2(n)\rceil} \\
\textbf{(3)}\ U^u_{l,p} & : & \bigvee_{0\le i\le u}(U^i_{l-1,2p-1} \wedge U^{u-i}_{l-1,2p}) & 1 \le l \le \lceil log_2(n)\rceil,\ 1 \le p \le 2^{\lceil log_2(n)\rceil - l},\quad 0 \le u < 2^l \\
\textbf{(4)}\ U^u_{l,p} & : & \mathbf{t} & 0 \le l \le \lceil log_2(n)\rceil,\ 1 \le p \le 2^{\lceil log_2(n)\rceil - l},\quad 2^l \le u < 2^{l+1} \\
\quad) & & &
\end{array}$$

The following property describes the semantics of these variables. Essentially, a variable $U^u_{l,p}$ denotes those objects that contain at most u names k_i among $\{\!\{k_1,\ldots,k_n\}\!\}$ whose index i belongs to $I(l,p)$; for example, an object in $[\![U^1_{2,2}]\!]_{E^+}$ contains at most 1 name whose index is in the interval $I(2,2)$. 2.7

Property 2. For a given set of names $\{\!\{k_1,\ldots,k_n\}\!\}$, the variables $U^u_{l,p}$ of the environment $E^+ = E \cup \mathcal{E}(\{\!\{k_1,\ldots,k_n\}\!\})$ enjoy the following property, for any object J:

$$J \in [\![U^u_{l,p}]\!]_{E^+} \;\Leftrightarrow\; |\,\{\!\{\, k_i \mid (k_i, \_) \in J \wedge i \in I(l,p)\,\}\!\}\,| \le u$$

Proof. We prove the property by induction on l.

Case $U^0_{0,p}$: $\mathsf{props}(k_p : \mathbf{f}; \mathbf{t})$ with $1 \leq p \leq n$: we want to prove that

$$J \in [\![\mathsf{props}(k_p : \mathbf{f}; \mathbf{t})]\!]_{E^+} \Leftrightarrow |\{\!\{ k_i \mid (k_i, \_) \in J \wedge i \in I(0, p) \}\!\}| \leq 0$$

which holds since $\mathsf{props}(k_p : \mathbf{f}; \mathbf{t})$ is satisfied by all and only the objects that do not contain the name k_p, and the singleton interval $I(0, p)$ only contains p.

Case $U^0_{0,p}$: $\mathbf{t}$ with $n + 1 \leq p \leq 2^{\lceil log_2(n) \rceil}$: we want to prove that

$$J \in [\![\mathbf{t}]\!]_{E^+} \Leftrightarrow |\{\!\{ k_i \mid (k_i, \_) \in J \wedge i \in I(0, p) \}\!\}| \leq 0 \text{ when } n + 1 \leq p$$

which holds trivially since, if $n + 1 \leq p$, then $\{\!\{ k_i \mid i \in I(0, p) \}\!\}$ is empty.

Case $U^u_{l,p}$: $\bigvee_{0 \leq i \leq u}(U^i_{l-1,2p-1} \wedge U^{u-i}_{l-1,2p})$ with $1 \leq l \leq \lceil log_2(n) \rceil$, $1 \leq p \leq 2^{\lceil log_2(n) \rceil - l}$, $0 \leq u < 2^l$. We observe that $|\{\!\{ k_i \mid (k_i, \_) \in J \wedge i \in I(l, p) \}\!\}| \leq u$ iff there exists i such that $0 \leq i \leq u$ and

$$|\{\!\{ k_i \mid (k_i, \_) \in J \wedge i \in I(l - 1, 2p - 1) \}\!\}| \leq i \ \wedge \ |\{\!\{ k_i \mid (k_i, \_) \in J \wedge i \in I(l - 1, 2p) \}\!\}| \leq u - i$$

and we conclude by induction. We also observe that the variable $U^i_{l,p}$ is defined for every triple (l, p, i) such that $0 \leq l \leq \lceil log_2(n) \rceil$, $1 \leq p \leq 2^{\lceil log_2(n) \rceil - l}$, $0 \leq u < 2^{l+1}$. This ensures that both $U^i_{l-1,2p-1}$ and $U^{u-i}_{l-1,2p}$ are defined when $1 \leq l \leq \lceil log_2(n) \rceil$, $1 \leq p \leq 2^{\lceil log_2(n) \rceil - l}$, $0 \leq u < 2^l$. Specifically, the biggest value required for i in $U^i_{l-1,2p-1}$ is $2^l - 1$, when $u = 2^l - 1$ and $i = u$, and $U^{u'}_{l-1,\_}$ is defined for values of u' up to $2^{(l-1)+1} - 1$, that is, $u' = 2^l - 1$. The same upper bound $2^l - 1$ holds for $u - i$ in $U^{u-i}_{l-1,2p}$, when $u = 2^l - 1$ and $i = 0$.

Finally, we have case $U^u_{l,p}$: $\mathbf{t}$ with $0 \leq l < \lceil log_2(n) \rceil$, $1 \leq p \leq 2^{\lceil log_2(n) \rceil - l}, 2^l \leq u < 2^{l+1}$: we want to prove that

$$J \in [\![\mathbf{t}]\!]_{E^+} \Leftrightarrow |\{\!\{ k_i \mid (k_i, \_) \in J \wedge i \in I(l, p) \}\!\}| \leq u$$

which holds trivially since $I(l, p)$ only contains 2^l indexes, and $2^l \leq u$.

□

Property 3. The number of symbols in $\mathcal{E}(\{\!\{k_1, \ldots, k_n\}\!\})$ grows like $O(n^2)$.

Proof. We only count the number of symbols in line 3, where the definition size grows as $O(u)$; the other lines are asymptotically dominated. Line 3 is defined for any l, p, and u such that $1 \leq l \leq \lceil log_2(n) \rceil$, $1 \leq p \leq 2^{\lceil log_2(n) \rceil - l}$, and $0 \leq u < 2^l$, and its size grows like u, hence its total size is:

$$\begin{aligned}
&\textstyle\sum_{l \in \{\!\{1..\lceil log_2(n) \rceil\}\!\}} \sum_{p \in \{\!\{1..2^{\lceil log_2(n) \rceil - l}\}\!\}} \sum_{u \in \{\!\{0..2^l - 1\}\!\}} (O(u)) \\
&= \textstyle\sum_{l \in \{\!\{1..\lceil log_2(n) \rceil\}\!\}} \sum_{p \in \{\!\{1..2^{\lceil log_2(n) \rceil - l}\}\!\}} O(2^{2l}) \\
&= \textstyle\sum_{l \in \{\!\{1..\lceil log_2(n) \rceil\}\!\}} (2^{\lceil log_2(n) \rceil - l}) \cdot O(2^{2l}) \\
&= \textstyle\sum_{l \in \{\!\{1..\lceil log_2(n) \rceil\}\!\}} O(2^{\lceil log_2(n) \rceil + l}) \\
&= O(2^{\lceil log_2(n) \rceil}) \times \textstyle\sum_{l \in \{\!\{1..\lceil log_2(n) \rceil\}\!\}} O(2^l) \\
&= O(2^{\lceil log_2(n) \rceil}) \times O(2^{\lceil log_2(n) \rceil}) = O(n^2)
\end{aligned}$$

□

Now that we know how to express the fact that $|\{\!\{ k_i \mid (k_i, \_) \in J \wedge i \in I(l, p) \}\!\}| \leq u$, we can express the negation of $\mathsf{props}(\underline{k_1} : \mathbf{t}, \ldots, \underline{k_n} : \mathbf{t}; \mathbf{f})$, as previously described: the instance is an object, and exists i such that the instance contains at most i fields whose names are in the set, but the instance contains at least $i + 1$ names (Property 4, case (1)). In Property 4, case (2) combines Property 1 and case (1): $\mathsf{props}(\underline{k_1} : S_1, \ldots, \underline{k_n} : S_n; \mathbf{f})$ is violated either by violating $\mathsf{props}(\underline{k_1} : S_1, \ldots, \underline{k_n} : S_n; \mathbf{t})$ or by violating the $\mathsf{props}(\ldots; \mathbf{f})$ part.

Property 4 (Negation of $\mathsf{props}(\ldots; \mathbf{f})$).

$$
\begin{array}{ll}
(1) & (\neg\mathsf{props}(\underline{k_1} : \mathbf{t}, \ldots, \underline{k_n} : \mathbf{t}; \mathbf{f}), E) \\
 & \quad = (\mathsf{type}(\mathsf{Obj}) \wedge (\bigvee_{0 \leq i \leq n}(U^i_{\lceil \log_2 n \rceil, 1} \wedge \mathsf{pro}^{\infty}_{i+1})), E \cup \mathcal{E}(\{\!\{k_1, \ldots, k_n\}\!\})) \\
(2) & (\neg\mathsf{props}(\underline{k_1} : S_1, \ldots, \underline{k_n} : S_n; \mathbf{f}), E) \\
 & \quad = (\neg\mathsf{props}(\underline{k_1} : S_1, \ldots, \underline{k_n} : S_n; \mathbf{t}) \ \vee \neg\mathsf{props}(\underline{k_1} : \mathbf{t}, \ldots, \underline{k_n} : \mathbf{t}; \mathbf{f}), E)
\end{array}
$$

Proof. (1) By definition, $J \notin [\![\mathsf{props}(\underline{k_1} : \mathbf{t}, \ldots, \underline{k_n} : \mathbf{t}; \mathbf{f})]\!]_E$ iff

$$
\begin{array}{l}
\neg(\forall l.\ (J = \{k'_1 : J_1, \ldots, k'_m : J_m\} \wedge l \in \{\!\{1..m\}\!\}) \Rightarrow \ (k'_l \notin L(\underline{k_1}| \ldots |\underline{k_n}) \Rightarrow J_l \in [\![\mathbf{f}]\!]_E)\ ) \\
\neg(\forall l.\ (J = \{k'_1 : J_1, \ldots, k'_m : J_m\} \wedge l \in \{\!\{1..m\}\!\}) \Rightarrow \ k'_l \in L(\underline{k_1}| \ldots |\underline{k_n})\ ) \\
\Leftrightarrow \quad J = \{k'_1 : J_1, \ldots, k'_m : J_m\} \ \wedge \exists l.\ l \in \{\!\{1..m\}\!\} \wedge \ k'_l \notin \{\!\{k_1, \ldots, k_n\}\!\}
\end{array}
$$

Consider now $J \in [\![\ \mathsf{type}(\mathsf{Obj}) \wedge (\bigvee_{0 \leq i \leq n}(U^i_{\lceil \log_2 n \rceil, 1} \wedge \mathsf{pro}^{\infty}_{i+1})\ )\ ]\!]_{E \cup \mathcal{E}(\{\!\{k_1, \ldots, k_n\}\!\})}$. By Property 2, it holds iff J is an object with at least $i + 1$ fields of which at most i belong to the set $\{\!\{k_1, \ldots, k_n\}\!\}$.

(2) By definition,

$$
\mathsf{props}(r_1 : S_1, \ldots, r_n : S_n; S_a) \equiv \mathsf{props}(r_1 : S_1, \ldots, r_n : S_n; \mathbf{t}) \ \wedge \mathsf{props}(r_1 : \mathbf{t}, \ldots, r_n : \mathbf{t}; S_a)
$$

The thesis follows immediately by setting $r_i = \underline{k_i}$ and $S_a = \mathbf{f}$, and by De Morgan rules. □

Example 3. Consider the following schema. 2.8

```
{ "not": { "properties": {"fullname": { "type": "string" },
           "additionalProperties": false
         }
}
```

This schema is satisfied by any value that does not satisfy the schema inside `"not"`, which constrains the type of the *fullname* property and forbids additional properties.

According to Property 4 case 2, this schema is equivalent to the one shown below. Observe that we only need here the variables $U^0_{0,1}$ and $U^1_{0,1}$, that correspond to lines (1) and (4) of the definition of $\mathcal{E}$. Line (2) generates no variable since $\lceil log_2(n) \rceil = 0$, hence $n + 1 > 2^{\lceil log_2(n) \rceil}$, and line (3) generates no variable since $1 > \lceil log_2(n) \rceil$.

```
 1 { "type": "object",
 2   "anyOf": [
 3      { "required": [ "fullname" ],
 4        "properties": { "fullname": { "not": { "type": "string" } } },
 5        "additionalProperties": true
 6      },
 7      { "anyOf": [
 8           { "$ref": "#/definitions/UpTo_0_In_0_1", "minProperties": 1 },
 9           { "$ref": "#/definitions/UpTo_1_In_0_1", "minProperties": 2 }
10        ]
11      }
12    ],
13    "definitions": {
14       "UpTo_0_In_0_1": { "properties": { "fullname": false },
15                          "additionalProperties": true
16                        },
17       "UpTo_1_In_0_1": true
18    }
19 }
```

This schema is, hence, satisfied by any object having at least one property *fullname* not of type string (lines 3-4), or having at least one property and no *fullname* property (lines 8 and 14-16), or having at least two properties (lines 9 and 17), such as:

```
{ "fullname": 1642 }
{ "year" : 1642 }
{ "fullname": "Galileo Galilei", "year" : 1642 }
```

A natural question is which other use cases can be expressed, maybe through more and more complex encodings. To answer this question, we first introduce some further notation.

Notation 1. Given a props assertion $S = \mathsf{props}(r_1 : S_1, \ldots, r_n : S_n; S_a)$ and a string k, the functions $[k]_S$ and $\mathcal{S}_S(k)$ are defined as follows.

1. $[k]_S = \{\!\{ k' \mid \forall i \in \{\!\{1..n\}\!\}.\ k \in L(r_i) \Leftrightarrow k' \in L(r_i) \}\!\}$: the set of strings that match exactly the same patterns found in S as k.
2. $\mathcal{S}_S(k)$: let $I = \{\!\{ i \mid k \in L(r_i) \}\!\}$; if $I = \emptyset$, then $\mathcal{S}_S(k) = S_a$ else $\mathcal{S}_S(k) = \wedge_{i \in I} S_i$: hence, $\mathcal{S}_S(k)$ is the conjunction of the schemas that must be satisfied by J' if $k : J'$ is a member of an object that satisfies S.

The function $\mathcal{S}_S$ is an alternative way of representing $\mathsf{props}(r_1 : S_1, \ldots, r_n : S_n; S_a)$, since it enjoys the following property.

Property 5. Given an assertion $S = \mathsf{props}(r_1 : S_1, \ldots, r_n : S_n; S_a)$, for any object J, J belongs to $\llbracket S \rrbracket_E$ iff, for every member $k : J'$ of J, we have that $J' \in \llbracket \mathcal{S}_S(k) \rrbracket_E$.

Proof. An object J belongs to $\llbracket \mathsf{props}(r_1 : S_1, \ldots, r_n : S_n; S_a) \rrbracket_E$ iff the following two conditions hold, where we use I_k to denote the set $\{\!\{ i \mid k \in L(r_i) \}\!\}$:

1. for any field $(k : J')$ of J with $I_k = \emptyset$, we have that $J' \in \llbracket S_a \rrbracket_E$;
2. for any field $(k : J')$ of J where $I_k \neq \emptyset$, we have that $J' \in \llbracket S_i \rrbracket_E$ for each $i \in I_k$.

In both cases, by definition of $\mathcal{S}_S(k)$, we are just asking that $J' \in \llbracket \mathcal{S}_S(k) \rrbracket_E$. □

We now prove that Property 4 exhausts all cases where $\neg\mathsf{props}(r_1 : S_1, \ldots, r_n : S_n; S_a)$ can be expressed without negation. When we say "(S, E) can be expressed without negation", we mean "there exists a pair (S', E') that contains no negation such that $(S, E) \equiv (S', E')$".

Theorem 2. Given $S = \mathsf{props}(r_1 : S_1, \ldots, r_n : S_n; S_a)$ and a closing environment E, if exist k_1, k_2 such that (1) $[k_1]_S$ and $[k_2]_S$ are both infinite, and (2) exist both J_1^+ and J_2^- such that $J_1^+ \in [\![\mathcal{S}_S(k_1)]\!]_E$ and $J_2^- \notin [\![\mathcal{S}_S(k_2)]\!]_E$, then $(\neg S, E)$ cannot be expressed without negation.

Proof. By Property 5, $\neg\mathsf{props}(r_1 : S_1, \ldots, r_n : S_n; S_a)$ is satisfied by every instance that is an object that contains at least one member $k : J'$ such that $J' \notin [\![\mathcal{S}_S(k)]\!]_E$. 2.9

Assume that k_1, k_2, J_1^+, J_2^- exist, and assume that a positive $D = S_0\ \mathsf{defs}(E')$ with $E' = x_1 : S'_1, \ldots, x_n : S'_n$ expresses $(\neg S, E)$, where $S = \mathsf{props}(r_1 : S_1, \ldots, r_n : S_n; S_a)$, in order to reach a contradiction. Consider a name $k \in [k_2]_S$ that does not appear in any req operator that is in D: since $[k_2]_S$ is infinite, such k exists. Let mm be a number that is bigger than every lower bound m that appears in any pro_m^M in D and strictly bigger than the number of members of any object found inside any const or enum operator in D. Consider a set of mm different names $\{\!\{k'_1, \ldots, k'_{mm}\}\!\}$ that belong to $[k_1]_S$ — such a set exists since $[k_1]_S$ is infinite. Now consider the following two objects:

$$\begin{array}{ll} \{\ k'_1 : J_1^+, \ldots, k'_{mm} : J_1^+, k : J_2^-\ \} & \qquad O_1 \\ \{\ k'_1 : J_1^+, \ldots, k'_{mm} : J_1^+\ \} & \qquad O_2 \end{array}$$

Observe that O_1 violates the constraint S, thanks to the $k : J_2^-$ member, while O_2 satisfies S, hence O_1 satisfies the requirement $\neg S$, thanks to the same member, while O_2 does not satisfy $\neg S$: we have that $(O_1 \in [\![\neg S]\!]_E \wedge \neg(O_2 \in [\![\neg S]\!]_E))$. Hence, we can prove that D does not express $\neg S$ by showing that it enjoys the opposite property: $(\neg(O_1 \in [\![D]\!]_{E'}) \vee O_2 \in [\![D]\!]_{E'})$, that is, $O_1 \in [\![D]\!]_{E'} \Rightarrow O_2 \in [\![D]\!]_{E'}$.

We say that a schema S' satisfies *NonDifferentiating* if $O_1 \in [\![S']\!]^i_{E'} \Rightarrow O_2 \in [\![S']\!]^i_{E'}$. We now prove that every assertion S' inside D satisfies *NonDifferentiating*, by induction on the lexicographic pair $(i, |S'|)$, where $|S'|$ is the size of S'. In this way, we prove that D satisfies *NonDifferentiating*, which means that it is not equivalent to $\neg S$. 2.10

Every assertion that cannot distinguish two objects (number assertions, for example) satisfies *NonDifferentiating*. const and enum assertions in D do not contain O_1 or O_2 since these are too big, by construction, hence they do not distinguish O_1 from O_2. The props and $\mathsf{propNames}$ constraints can only fail because of the presence of a member, never for its absence, hence they satisfy *NonDifferentiating*. The name k does not appear in any req requirement in D, hence all req requirements in D satisfy *NonDifferentiating*. If $O_1 \in [\![\mathsf{pro}_m^M]\!]^i_{E'}$, then O_2 satisfies the upper bound since O_2 is shorter than O_1, and it satisfies the lower bound since it has mm members, and $mm \geq m$ by construction. If $O_1 \in [\![S_1 \wedge S_2]\!]^i_{E'}$, then $O_1 \in [\![S_1]\!]^i_{E'}$ and $O_1 \in [\![S_2]\!]^i_{E'}$, hence the same holds for O_2 by induction on the size of S, hence $O_2 \in [\![S_1 \wedge S_2]\!]^i_{E'}$. The same holds if we exchange $\wedge$ with $\vee$ and *and* with *or*. For variables, the thesis follows by induction on i, since $[\![x_j]\!]^{i+1}_{E'} = [\![S_j]\!]^i_{E'}$ and S_j is a subterm of D, and, finally, *NonDifferentiating* holds trivially when $i = 0$. Hence, D itself is *NonDifferentiating*, hence D does not express $(\neg S, E)$. □

Corollary 1. Let $S = \mathsf{props}(r_1 : S_1, \ldots, r_n : S_n; S_a)$. The assertion $(\neg S, E)$ can be expressed without negation only if (S, E) can be expressed either as $(\mathsf{props}(\underline{k_1} : S_1, \ldots, \underline{k_n} : S_n; \mathbf{t}), E)$ or as $(\mathsf{props}(\underline{k_1} : S_1, \ldots, \underline{k_n} : S_n; \mathbf{f}), E)$.

Proof. Assume that $(\neg S, E)$ can be expressed without negation. By Theorem 2, it is not the case that exist k_1 and k_2 such that (1) $[k_1]_S$ and $[k_2]_S$ are both infinite, and (2) exist both J_1^+ and J_2^- such that $J_1^+ \in [\![\mathcal{S}_S(k_1)]\!]_E$ and $J_2^- \notin [\![\mathcal{S}_S(k_2)]\!]_E$. Hence, either for every k such that $[k]_S$ is infinite there exists no J^+ such that $J^+ \in [\![\mathcal{S}_S(k)]\!]_E$, hence $[\![\mathcal{S}_S(k)]\!]_E = [\![\mathbf{f}]\!]_E$, or for every k such that $[k]_S$ is infinite there exists no J^- such that $J^- \notin [\![\mathcal{S}_S(k)]\!]_E$, hence $[\![\mathcal{S}_S(k)]\!]_E = [\![\mathbf{t}]\!]_E$. In the first case, $\mathsf{props}(r_1 : S_1, \ldots, r_n : S_n; S_a)$ can be expressed as $\mathsf{props}(\underline{k_1} : S'_1, \ldots, \underline{k_m} : S'_m; \mathbf{f})$, as follows: every k_f such that $[k_f]_S$ is finite, so that $[k_f]_S = \{\!\{k'_1, \ldots, k'_l\}\!\}$, is transformed into a finite set of simple constraints $\underline{k'_1} : \mathcal{S}_S(k_f), \ldots, \underline{k'_l} : \mathcal{S}_S(k_f)$, and the additional constraint $\mathbf{f}$ expresses the fact that every name k such that $[k]_S$ is infinite must satisfy the assertion $\mathbf{f}$. In the second case, we reason in the same way to prove that the schema can be expressed as $(\mathsf{props}(\underline{k_1} : S'_1, \ldots, \underline{k_m} : S'_m; \mathbf{t}), E)$. □

Theorem 2 gives an abstract characterization of the schemas whose negation cannot be expressed. Observe that k_1 and k_2 may coincide, as long as $\mathcal{S}_S(k_1)$ is not trivial, where (S, E) *is trivial* when either $[\![S]\!]_E = [\![\mathbf{t}]\!]_E$, or $[\![S]\!]_E = [\![\mathbf{f}]\!]_E$. Corollary 1 rephrases Theorem 2 and shows that Properties 1 and 4 are exhaustive: negation cannot be pushed through props unless the schema is equivalent to one of those presented in Properties 1 and 4. In terms of the original `patternProperties` and `additionalProperties` operators, Corollary 1 shows that the negation-free complement of a schema that contains `patternProperties` at the top level is only expressible when the schema can be rewritten into one where `patternProperties` is not used. For a schema S that contains `additionalProperties` at the top level, its complement has a negation-free expression only if S can be rewritten into one where `additionalProperties` is associated to a trivial schema.

A last Corollary of Theorem 2 regards expressibility of negation of $\mathsf{propNames}(S)$.

Corollary 2. $(\neg\mathsf{propNames}(S), E)$ *can be expressed without negation if, and only if, either* $[\![\mathsf{type(Str)} \wedge S]\!]_E$ *is finite or* $[\![\mathsf{type(Str)} \wedge \neg S]\!]_E$ *is finite.*

Proof. ($\Leftarrow$) If $[\![\mathsf{type(Str)} \wedge S]\!]_E$ is finite and equal to $\{\!\{k_1, \ldots, k_n\}\!\}$, then $(\mathsf{propNames}(S), E)$ is equivalent to $\mathsf{props}(\underline{k_1} : \mathbf{t}, \ldots, \underline{k_n} : \mathbf{t}; \mathbf{f})$, whose negation can be expressed by Property 4. If the complement of $[\![\mathsf{type(Str)} \wedge S]\!]_E$ is finite and equal to $\{\!\{k_1, \ldots, k_n\}\!\}$, then $(\mathsf{propNames}(S), E)$ is equivalent to $\mathsf{props}(\underline{k_1} : \mathbf{f}, \ldots, \underline{k_n} : \mathbf{f}; \mathbf{t})$, whose negation can be expressed by Property 1.

($\Rightarrow$) If both $[\![\mathsf{type(Str)} \wedge S]\!]_E$ and its complement are infinite, then any string of $[\![\mathsf{type(Str)} \wedge S]\!]_E$ satisfies the conditions for k_1 in Theorem 2, and any string in the complement satisfies the conditions for k_2, hence no positive schema can express $(\neg\mathsf{propNames}(S), E)$.

□

While $\mathsf{propNames}(S)$ is a constraint that specifies that "every name in J belongs to S", $\neg\mathsf{propNames}(S)$ is a requirement that specifies that *there exists* a name that satisfies $\neg S$; Corollary 2 specifies that $\neg\mathsf{propNames}(S)$ has a negation-free expression only in the finitary cases when either the allowed names, or the forbidden names, form a finite set, so that propNames is another operator that does not admit, in general, the negation-free expression of its negation dual in JSON Schema.

4.3.2. Arrays

JSON Schema changes its expressive power when moving from objects to arrays. When objects are described, the negated schema $\neg\mathsf{props}(;S)$ is used to require the presence of one object member whose value satisfies S, independently of its name, and Corollary 1 specifies that this assertion cannot be expressed in the negation-free fragment of JSON Schema, for any non-trivial S. 2.11, 2.12

Consider now arrays; arrays can be described as objects where the member names are natural numbers, with the extra constraint that, whenever the name $n+1$ is present, with $n \geq 1$, then n must be present as well. However, arrays have a positive requirement operator $\mathsf{contains}(S)$ that forces the presence of at least one element that satisfies any S, while the inexpressibility of $\neg\mathsf{props}(;S)$ implies that objects have no negation-free way of requiring the presence of a member that satisfies a non trivial S: hence, non-negative JSON Schema is more expressive for arrays than for objects.

Despite this crucial difference, the overall situation, as far as negation closure is concerned, is quite similar: for arrays, as it happens for objects, we will show that the negation of the fundamental items operator can be expressed without negation in the most common cases, but not in general.

From our analysis of GitHub schemas, we have verified that the most common use cases for arrays are the following three, in this order:

1. homogeneous arrays `"items"`: S_a, that we indicate with $\mathsf{items}(;S_a)$;
2. open tuple arrays `"items"`: $[S_1,\ldots,S_n]$, and no `"additionalItems"`, or `"additionalItems"`: `true`, that we indicate with $\mathsf{items}(S_1,\ldots,S_n;\mathbf{t})$;
3. closed tuple arrays `"items"`: $[S_1,\ldots,S_n]$ with `"additionalItems"`: `false`, that we indicate with $\mathsf{items}(S_1,\ldots,S_n;\mathbf{f})$.

We first show, in Property 6, how negation of items can be expressed in these three most common cases, where case (2) below generalizes the second case in the above list.

Observe that case (3) includes the case when any $[\![S_i]\!]_E = [\![\mathbf{f}]\!]_E$ for some S_i since, in that case, $[\![\mathsf{items}(S_1,\ldots,S_n;S_a)]\!]_E$ is equivalent to $[\![\mathsf{items}(S_1,\ldots,S_{i-1};\mathbf{f})]\!]_E$.

Property 6 (Negation of common use cases for items).

(1) $\neg\mathsf{items}(;S_a) \quad\equiv\quad \mathsf{type}(\mathsf{Arr}) \wedge \mathsf{contains}(\neg S_a)$

(2) if for each i, $[\![S_i]\!]_E \subseteq [\![S_a]\!]_E$:

$$\neg\mathsf{items}(S_1,\ldots,S_n;S_a) \equiv$$
$$\mathsf{type}(\mathsf{Arr}) \wedge (\textstyle\bigvee_{i\in\{\!\{1..n\}\!\}}(\mathsf{items}(\mathbf{t}_1,\ldots,\mathbf{t}_{i-1},\neg S_i;\mathbf{t}) \wedge \mathsf{ite}_i^{\infty}) \vee \mathsf{contains}(\neg S_a))$$

(3) $\neg\mathsf{items}(S_1,\ldots,S_n;\mathbf{f}) \quad\equiv\quad \mathsf{type}(\mathsf{Arr}) \wedge (\bigvee_{i\in\{\!\{1..n\}\!\}}(\mathsf{items}(\mathbf{t}_1,\ldots,\mathbf{t}_{i-1},\neg S_i;\mathbf{t}) \wedge \mathsf{ite}_i^{\infty}) \vee \ \mathsf{ite}_{n+1}^{\infty})$

Proof.
We recall the definition of $[\![\mathsf{items}(S_1,\ldots,S_n;S_{n+1})]\!]_E$:

$$\begin{array}{rcl}
[\![\mathsf{items}(S_1,\ldots,S_n;S_{n+1})]\!]_E & = & \{\!\{\ J \mid J=[J_1,\ldots,J_m], l\in\{\!\{1..m\}\!\} \Rightarrow \\
& & \quad (\forall j\in\{\!\{1..n\}\!\}.\ l=j \Rightarrow J_l \in [\![S_j]\!]_E)\wedge \\
& & \quad (l>n \Rightarrow J_l \in [\![S_{n+1}]\!]_E)\ \}\!\}
\end{array}$$

(1) We apply De Morgan rules to the definition of $[\![\mathsf{items}(;S_a)]\!]_E$:

$$\begin{array}{rcl}
J \notin [\![\mathsf{items}(;S_a)]\!]_E & \Leftrightarrow & J=[J_1,\ldots,J_m] \wedge \exists l.\ l\in\{\!\{1..m\}\!\} \wedge l>0 \wedge\ J_l \notin [\![S_a]\!]_E \\
& \Leftrightarrow & J \in [\![\mathsf{type}(\mathsf{Arr}) \wedge \mathsf{contains}(\neg S_a)]\!]_E
\end{array}$$

(2) We apply De Morgan rules to rewrite $J \notin [\![\mathsf{items}(S_1, \ldots, S_n; S_a)]\!]_E$ as (*) below:

$$\begin{aligned}
& J \notin [\![\mathsf{items}(S_1, \ldots, S_n; S_a)]\!]_E \\
\Leftrightarrow\ & J = [J_1, \ldots, J_m] \wedge \exists i \in \{\!\{1..m\}\!\}. (\exists j \in \{\!\{1..n\}\!\}. i = j \wedge J_i \notin [\![S_j]\!]_E) \vee (i > n \wedge J_i \notin [\![S_a]\!]_E) \\
\Leftrightarrow\ & J = [J_1, \ldots, J_m] \wedge (\exists i \in \{\!\{1..\min(m,n)\}\!\}. J_i \notin [\![S_i]\!]_E) \vee (\exists i.\ n < i \leq m \wedge J_i \notin [\![S_a]\!]_E) \quad (*)
\end{aligned}$$

We now prove that

$$(*) \Leftrightarrow J \in [\![\mathsf{type}(\mathsf{Arr}) \wedge (\bigvee_{i \in \{\!\{1..n\}\!\}} (\mathsf{items}(\mathbf{t}_1, \ldots, \mathbf{t}_{i-1}, \neg S_i; \mathbf{t}) \wedge \mathsf{ite}^{\infty}_i) \vee \mathsf{contains}(\neg S_a))]\!]_E$$

($\Rightarrow$) Let us assume that J and i are such that $i \in \{\!\{1..\min(m,n)\}\!\} \wedge J_i \notin [\![S_i]\!]_E$; in this case, $J \in [\![\mathsf{type}(\mathsf{Arr}) \wedge \mathsf{items}(\mathbf{t}_1, \ldots, \mathbf{t}_{i-1}, \neg S_i; \mathbf{t}) \wedge \mathsf{ite}^{\infty}_i]\!]_E$. Otherwise, by (*), we have that exists i such that $n < i \leq m \wedge J_i \notin [\![S_a]\!]_E$; hence, $J \in [\![\mathsf{type}(\mathsf{Arr}) \wedge \mathsf{contains}(\neg S_a)]\!]_E$.
($\Leftarrow$) In the other direction, assume that $J \in [\![\mathsf{type}(\mathsf{Arr}) \wedge (\bigvee_{i \in \{\!\{1..n\}\!\}} (\mathsf{items}(\mathbf{t}_1, \ldots, \mathbf{t}_{i-1}, \neg S_i; \mathbf{t}) \wedge \mathsf{ite}^{\infty}_i) \vee \mathsf{contains}(\neg S_a))]\!]_E$; hence, this implies that J is an array $[J_1, \ldots, J_m]$ and either $J \in [\![\bigvee_{i \in \{\!\{1..n\}\!\}} \mathsf{items}(\mathbf{t}_1, \ldots, \mathbf{t}_{i-1}, \neg S_i; \mathbf{t}) \wedge \mathsf{ite}^{\infty}_i]\!]_E$ or $J \in [\![\mathsf{contains}(\neg S_a)]\!]_E$. In the first case, let i be the smallest index i such that $J \in [\![\mathsf{items}(\mathbf{t}_1, \ldots, \mathbf{t}_{i-1}, \neg S_i; \mathbf{t}) \wedge \mathsf{ite}^{\infty}_i]\!]_E$. By $J \in [\![\mathsf{ite}^{\infty}_i]\!]_E$ we have that $i \leq m$, hence we have that $\exists i \in \{\!\{1..\min(m,n)\}\!\}. J_i \notin [\![S_i]\!]_E$. In the second case, $J \in [\![\mathsf{contains}(\neg S_a)]\!]_E$ implies that $\exists i.\ i \leq m \wedge J_i \notin [\![S_a]\!]_E$. If $i > n$, we have that $\exists i.\ n < i \leq m \wedge J_i \notin [\![S_a]\!]_E$, hence (*) holds. If $i \leq n$, we observe that, by $[\![S_i]\!]_E \subseteq [\![S_a]\!]_E$, we have that $J_i \notin [\![S_a]\!]_E$ implies $J_i \notin [\![S_i]\!]_E$, hence $\exists i \in \{\!\{1..\min(m,n)\}\!\}. J_i \notin [\![S_i]\!]_E$, hence (*) holds.
(3) We first rewrite $J \notin [\![\mathsf{items}(S_1, \ldots, S_n; \mathbf{f})]\!]_E$ as (*) below:

$$\begin{aligned}
& J \notin [\![\mathsf{items}(S_1, \ldots, S_n; \mathbf{f})]\!]_E \\
\Leftrightarrow\ & J = [J_1, \ldots, J_m] \wedge ((m > n) \vee \exists i \in \{\!\{1..\min(m,n)\}\!\}. J_i \notin [\![S_i]\!]_E) \quad (*)
\end{aligned}$$

We now prove that $(*) \Rightarrow J \in [\![\mathsf{type}(\mathsf{Arr}) \wedge (\bigvee_{i \in \{\!\{1..n\}\!\}} (\mathsf{items}(\mathbf{t}_1, \ldots, \mathbf{t}_{i-1}, \neg S_i; \mathbf{f}) \wedge \mathsf{ite}^{\infty}_i) \vee \mathsf{ite}^{\infty}_{n+1})]\!]_E$. Let us assume that J and i are such that $i \in \{\!\{1..\min(m,n)\}\!\} \wedge J_i \notin [\![S_i]\!]_E$; in this case, $J \in [\![\mathsf{type}(\mathsf{Arr}) \wedge \mathsf{items}(\mathbf{t}_1, \ldots, \mathbf{t}_{i-1}, \neg S_i; \mathbf{f}) \wedge \mathsf{ite}^{\infty}_i]\!]_E$. Otherwise, by (*), we have that $m > n$, hence, $J \in [\![\mathsf{type}(\mathsf{Arr}) \wedge \mathsf{ite}^{\infty}_{n+1}]\!]_E$.

In the other direction, assume that $J \in [\![\mathsf{type}(\mathsf{Arr}) \wedge (\bigvee_{i \in \{\!\{1..n\}\!\}} (\mathsf{items}(\mathbf{t}_1, \ldots, \mathbf{t}_{i-1}, \neg S_i; \mathbf{f}) \wedge \mathsf{ite}^{\infty}_i) \vee \mathsf{ite}^{\infty}_{n+1})]\!]_E$, hence J is an array $[J_1, \ldots, J_m]$ and either $J \in [\![\bigvee_{i \in \{\!\{1..n\}\!\}} \mathsf{items}(\mathbf{t}_1, \ldots, \mathbf{t}_{i-1}, \neg S_i; \mathbf{f}) \wedge \mathsf{ite}^{\infty}_i]\!]_E$ or $J \in [\![\mathsf{ite}^{\infty}_{n+1}]\!]_E$. In the first case, we reason as in (2) and have that $\exists i \in \{\!\{1..\min(m,n)\}\!\}. J_i \notin [\![S_i]\!]_E$. In the second case, we have that $m > n$. □

Example 4. Consider the following array schema.

```
{ "not" : { "items": { "type" : "object",
                       "required" : [ "name" , "surname" ] } } }
```

This schema is satisfied by any array that contains at least an item which is not an object with a *name* and a *surname* property. According to Property 6(1), this schema is equivalent to the one shown below.

```
{ "type" : "array",
  "contains" : { "not" : { "type" : "object",
                           "required" : [ "name" , "surname" ] } }
}
```

This schema is therefore satisfied by arrays like the one shown below.

```
[1564, {"surname": "Galilei", "name": "Galileo"}, 1642]
```

The three cases of Property 6 include the quasi-totality of the items assertions that we found in our collection. The only case that is not covered by the three cases above is when $n > 0$, there exists an i where $[\![S_i]\!]_E \not\subseteq [\![S_a]\!]_E$, for all i $[\![S_i]\!]_E \neq [\![\mathbf{f}]\!]_E$, and $[\![S_a]\!]_E \neq [\![\mathbf{f}]\!]_E$. In this specific case, we prove that negation cannot be expressed.

Theorem 3. Given $(\neg\mathsf{items}(S_1, \ldots, S_n; S_a), E)$, if $n \neq 0$, all schemas $S_1, \ldots, S_n$, S_a, are non-empty in E, and there exists an i where $[\![S_i]\!]_E \not\subseteq [\![S_a]\!]_E$, then the algebra without negation cannot express it. 2.13

Proof. Assume that a positive document $D = x_j\ \mathsf{defs}(x_1 : S'_1, \ldots, x_m : S'_m)$ expresses the assertion $S = \neg\mathsf{items}(S_1, \ldots, S_n; S_a), E$, when $n \neq 0$, all schemas $S_1, \ldots, S_n$, S_a, are non-empty in E, and there exists an i where $[\![S_i]\!]_E \not\subseteq [\![S_a]\!]_E$, that is, there exist i and J^-_i such that $J^-_i \in [\![S_i \wedge \neg S_a]\!]_E$.

$\neg\mathsf{items}(S_1, \ldots, S_n; S_a)$ is satisfied by any J that is an array and has either an element at a position $j \leq n$ that satisfies $\neg S_j$, or an element after position $n+1$ (included) that satisfies $\neg S_a$. Let nn be a number greater (i.e., $\geq$) than n and greater than all the lower bounds of any ite^M_m in D and greater than the lengths of all the arrays that appear in const and enum assertions in D. Consider the following two arrays:

$$\begin{array}{ll} [J_1, \ldots, J_{i-1}, J^-_i, J_{i+1}, \ldots, J_{nn}, J^+, J^-_i] & A_1 \\ [J_1, \ldots, J_{i-1}, J^-_i, J_{i+1}, \ldots, J_{nn}, J^+] & A_2 \end{array}$$

In these arrays, $J^+ \in [\![S_a]\!]_E$, $J^-_i \in [\![S_i \wedge \neg S_a]\!]_E$, and all other elements $J_1, \ldots, J_{nn}$ are chosen to satisfy the corresponding S_j, if their position j is before n, or S_a otherwise, which is possible since all these schemas are not empty.

A generic S' satisfies *NonDifferentiating* if $A_1 \in [\![S']\!]^i_{E'} \Rightarrow A_2 \in [\![S']\!]^i_{E'}$. We now prove that every assertion S' inside D satisfies *NonDifferentiating*, by induction on the lexicographic pair $(i, |S'|)$, where $|S'|$ is the size of S'. In this way, we prove that D satisfies *NonDifferentiating*, which is a contradiction since $\neg S$ is satisfied by A_1, thanks to the last element J^-_i, and is not satisfied by A_2, since A_2 satisfies S. 2.10, 2.14

For variables, boolean expressions, and non-array typed operators we reason as in the proof of Theorem 2. const and enum assertions in D do not contain neither A_1 nor A_2 since these are too big, by construction. If S' is $\mathsf{contains}(S'')$, then it is either satisfied by both A_1 or A_2 or by none, since they contain the same elements. The lower bound of ite^M_m is satisfied by both A_1 or A_2 since $nn \geq m$ by construction. Finally, uniqueItems, $\mathsf{items}(S'_1, \ldots, S'_m; S'_a)$, and the upper bound of ite^M_m are constraints and not requirements, hence they all satisfy *NonDifferentiating* since A_2 is an initial segment of A_1. Hence D is not equivalent to $\neg\mathsf{items}(S_1, \ldots, S_n; S_a)$. □

Corollary 3. *$\neg\mathsf{items}(S_1, \ldots, S_n; S_a)$ can be expressed without negation only if it is equivalent to one of the three cases whose negation is expressed in Property 6.*

Proof. By Theorem 3, if the positive algebra can express $\neg\mathsf{items}(S_1, \ldots, S_n; S), E$, then either $n = 0$, or exists one schema among $S_1, \ldots, S_n$, S_a that is empty in E, or, for each i in $\{\!\{1..n\}\!\}$, $[\![S_i \wedge \neg S_a]\!]_E$ is empty. These three situations correspond to, respectively, cases (1), (3), and (2), of Property 6. □

Hence, we are again in a situation where negation can be pushed through items in almost all cases of practical interest, but not always. Moreover, as in the object case, we have an exact characterization of what can be expressed without negation, and what cannot.

We finally observe that, while on one side the requirements req and $\mathsf{contains}$ can express the negation of the constraints props and items in most cases, but not always, on the other side the constraints props and items can *always* express the negation of the requirements req and $\mathsf{contains}$, in a quite simple and direct way.

Property 7 (Full negation for req and $\mathsf{contains}$).

$$\begin{array}{lll}
(1) \quad \neg\mathsf{req}(k_1,\ldots,k_n) & \equiv & \mathsf{type}(\mathsf{Obj}) \wedge (\mathsf{props}(\underline{k_1} : \mathbf{f}; \mathbf{t}) \vee \ldots \vee \mathsf{props}(\underline{k_n} : \mathbf{f}; \mathbf{t})) \\
(2) \quad \neg\mathsf{contains}(S) & \equiv & \mathsf{type}(\mathsf{Arr}) \wedge \mathsf{items}(; \neg S)
\end{array}$$

Proof. (1) $J \in [\![\mathsf{req}(k_1,\ldots,k_n)]\!]_E \Leftrightarrow (J \in JVal(\mathsf{Obj}) \Rightarrow \forall k \in \{\!\{k_1,\ldots,k_n\}\!\}.\ \exists J'.\ (k : J') \in J)$, hence $J \notin [\![\mathsf{req}(k_1,\ldots,k_n)]\!]_E$ iff $J \in [\![\mathsf{type}(\mathsf{Obj})]\!]_E$ and $\exists k \in \{\!\{k_1,\ldots,k_n\}\!\}.\ \forall J'.\ (k_i : J') \notin J$. The property $\forall J'.\ (k_i : J') \notin J$ is enjoyed by an object J iff it belongs to $\mathsf{props}(\underline{k_i} : \mathbf{f}; \mathbf{t})$.

(2) $J \in [\![\mathsf{contains}(S)]\!]_E \Leftrightarrow (J = [J_1,\ldots,J_m] \Rightarrow \exists l \in \{\!\{1..m\}\!\}.\ J_l \in [\![S]\!]_E)$, hence $J \notin [\![\mathsf{contains}(S)]\!]_E$ iff $J = [J_1,\ldots,J_m] \wedge \forall l \in \{\!\{1..m\}\!\}.\ J_l \notin [\![S]\!]_E$, that is, iff $J \in [\![\mathsf{type}(\mathsf{Arr}) \wedge \mathsf{items}(; \neg S)]\!]_E$. □

This quasi-duality can be synthesized as follows: in JSON Schema we have two constraint/requirement pairs, $\mathsf{items}/\mathsf{contains}$ for arrays and $\mathsf{props}/\mathsf{req}$ for objects. In both cases, the requirement is somehow less expressive than the negation of its constraint companion: req lacks the ability to describe infinite sets of names and the associated schemas, $\mathsf{contains}$ does not express the distinction between a head-part and a tail-part in an array schema.

Example 5. Consider the following schema, from a question posted on StackOverflow [22].

```
{   "type": "object",
    "properties": {
        "x": { "type": "integer" }
    },
    "required": [ "x" ],
    "not": { "required": [ "z" ] }
}
```

This schema describes objects having a mandatory x property of type integer, but without any z property. The fact that z is actually forbidden is not immediately clear, and, indeed, this was the actual topic debated in the StackOverflow post. However, by exploiting Property 7, the previous schema can be transformed as follows.

```
{   "type": "object",
    "properties": {
        "x": { "type": "integer" },
        "z": false
    },
    "required": [ "x" ]
}
```

This new version of the schema clearly states that z is forbidden.

To conclude our remarks about arrays, we observe that the negation of uniqueItems cannot be expressed in the language without negation.

Theorem 4. ($\neg$uniqueItems, E) cannot be expressed in the algebra without negation.

Proof. Assume that a positive (D, E') expresses $\neg$uniqueItems. Choose an integer N strictly greater than any l that appears as lower bound in a ite_l^j in D and also greater than the length of any array that appears in a const or enum assertion in D. Define the following two arrays, the first one ending with a repetition of N.

$$A_1 = [1, 2, \ldots, N-1, N, N] \qquad\qquad A_2 = [1, 2, \ldots, N-1, N]$$

An assertion S' satisfies *NonDifferentiating* if $A_1 \in [\![S']\!]^i_{E'} \Rightarrow A_2 \in [\![S']\!]^i_{E'}$. We now prove that every assertion S' inside D satisfies *NonDifferentiating*, by induction on the lexicographic pair $(i, |S'|)$, where $|S'|$ is the size of S'. In this way, we prove that D satisfies *NonDifferentiating*, which is a contradiction since $\neg$uniqueItems is satisfied by A_1, thanks to the repetition of the last element N, while A_2 satisfies uniqueItems, hence it does not satisfy $\neg$uniqueItems. 2.10

We prove that every assertion S' inside D satisfies *NonDifferentiating* by induction on the lexicographic pair $(i, |S'|)$. When $S = x$, we prove that by induction on i: in the base case, $[\![x]\!]^i_{E'} = \emptyset$ does not contain A_1, and when $i = i+1$ we have that $[\![x]\!]^{i+1}_{E'} = [\![E'(x)]\!]^i_{E'}$, and we conclude by induction on i. Non-array typed assertions accept both A_1 and A_2. All const and enum assertions refuse both, since the arrays that they enumerate are shorter than N. Since A_2 is an initial subarray of A_1, any constraint $\mathsf{items}(S_1, \ldots, S_n; S)$ and uniqueItems, satisfies *NonDifferentiating*. Since the length N is greater than any lower bound l, and A_2 is shorter than A_1, any ite_l^j that accepts A_1 also accepts A_2. A requirement $\mathsf{contains}(S)$ does not distinguish the two since they contain the same elements. For $S_1 \wedge S_2$ we conclude by induction on the size, since implication of satisfaction is preserved by $\wedge$, and similarly for $\vee$. Since A_1 belongs to $[\![D]\!]$ by assumption, A_2 belongs to $[\![D]\!]$, which contradicts the hypothesis, since A_2 satisfies uniqueItems. □

4.3.3. Other types: mulOf(q)

We conclude this section with mulOf(q), whose negation cannot be expressed in the language without negation. In the next section, we will show that this completes the list, hence that negation can be pushed through all the other operators in the language. 2.12

Theorem 5. The pair ($\neg$mulOf(q), E), for any $q > 0$ cannot be expressed in the algebra without negation.

Proof. Assume towards a contradiction that a positive document $D = S_0\ \mathsf{defs}(E')$ with $E' = x_1 : S_1, \ldots, x_n : S_n$ expresses $\neg$mulOf(q). Let us choose a number N such that $N > q$ (hence, $N > 0$), and $N > M$ and $N > m$ for any bound m and M, different from ∞, that is found in any assertion betw_m^M, xBetw_m^M inside D.

We say that a generic S' is Full Or Finite (FOF) for i over the closed interval $[N, 2N]$, if $[N, 2N] \cap [\![S]\!]^i_{E'}$ is either equal to $[N, 2N]$, or is finite (where the empty set is also regarded as finite). We prove that any subexpression S' of D is FOF over $[N, 2N]$ for any i, by induction on the lexicographic pair $(i, |S'|)$. For the variables, in the case $i = 0$ the empty set is finite, and the inductive step is immediate since $[\![x_j]\!]^{i+1}_{E'} = [\![S_j]\!]^i_{E'}$ and S_j is a subterm of D. Typed operators whose type is not Num accept all numbers, hence are Full. An interval operator whose bounds are both smaller than N has empty intersection with $[N, 2N]$, and is Full when $M = \infty$. The positive $\mathsf{mulOf}(q)$ operator has a finite intersection with $[N, 2N]$. Union or intersection of two subsets of $[N, 2N]$ which are either finite or full is either finite or full. Hence, D is FOF over $[N, 2N]$ for any i. The limit $\bigcup_{i\in\mathbb{N}} \bigcap_{j\geq i} [\![D]\!]^j$ can be infinite only if exists i such that $[\![D]\!]^j$ is infinite for $j \geq i$, hence the limit is full or finite as well. However, $\neg\mathsf{mulOf}(q)$ is not FOF: it is not full on $[N, 2N]$ since the interval contains at least one multiple of q, by $N > q$, and its intersection with $[N, 2N]$ is not finite. □

5. Completing the language

5.1. The missing operators

As we have seen, JSON Schema does not enjoy negation-closure, but is endowed with constraint-requirement pairs props/req and items/contains that exhibit an imperfect duality. We now define a more regular algebra by adding some *negative* operators, to obtain a closed algebra where each operator has a real negation dual, and negation can be fully eliminated. This negation-closed algebra is the one that we implemented in our tools and, in our experience, it is practical both to reason about JSON Schema and to implement tools for JSON Schema analysis.

The closed algebra completes JSON Schema with the following four dual operators, defined below: pattReq, contAfter, notMulOf, repeatedItems. All these operators can be expressed in the algebra using negation but none of them, by the theorems we presented, can be expressed *without* negation. The semantics of these operators is defined as follows (where the notation $\mathbf{t}^1, \ldots, \mathbf{t}^n$ indicates a sequence of n copies of $\mathbf{t}$).

$$\begin{array}{lcl}
\mathsf{pattReq}(r_1 : S_1, \ldots, r_n : S_n) & = & \mathsf{type}(\mathsf{Obj}) \Rightarrow \bigwedge_{i\in\{\!\{1..n\}\!\}} \neg\mathsf{props}(r_i : \neg S_i; \mathbf{t}) \\
\mathsf{contAfter}(n : S) & = & \mathsf{type}(\mathsf{Arr}) \Rightarrow \neg\mathsf{items}(\mathbf{t}^1, \ldots, \mathbf{t}^n; \neg S) \\
\mathsf{notMulOf}(q) & = & \mathsf{type}(\mathsf{Num}) \Rightarrow \neg\mathsf{mulOf}(q) \\
\mathsf{repeatedItems} & = & \mathsf{type}(\mathsf{Arr}) \Rightarrow \neg\mathsf{uniqueItems}
\end{array}$$

The operator $\mathsf{pattReq}(r_1 : S_1, \ldots, r_n : S_n)$ specifies that, if the instance is an object, then, for each $i \in \{\!\{1..n\}\!\}$, it must possess a member whose name matches r_i and whose value satisfies S_i. It is strictly more expressive than req, since it allows one to require a name that belongs to an infinite set $L(r_i)$, and it associates a schema S_i to each required pattern r_i. In the closed algebra, we regard req as an abbreviated form of pattReq where every pattern has the shape $\underline{k}$ and every associated schema is $\mathbf{t}$.

$\mathsf{contAfter}(n : S)$ specifies that, if the instance is an array, it must contain at least one element that satisfies S in a position that is strictly greater than n. This operator has an expressive power that is slightly greater than the $\mathsf{contains}(S)$ operator, since it can distinguish between the head and the tail of the array. In the closed algebra, we regard $\mathsf{contains}(S)$ as an abbreviation for $\mathsf{contAfter}(0 : S)$.

The operators $\mathsf{notMulOf}(q)$ and repeatedItems are just the duals of $\mathsf{mulOf}(q)$ and uniqueItems. In the next section we prove that these four operators are all that we need to make JSON Schema negation-closed.

5.2. Proving negation closure: the not-elimination algorithm

We prove negation closure through the definition of a not-elimination algorithm, which eliminates any instance of negation from any expression in the closed algebra.

$$
\begin{array}{lcll}
\mathsf{enum}(J_1, \ldots, J_n) & = & \mathsf{const}(J_1) \vee \ldots \vee \mathsf{const}(J_n) & \\
\mathsf{const}(\texttt{null}) & = & \mathsf{type}(\mathsf{Null}) & \\
\mathsf{const}(n) & = & \mathsf{type}(\mathsf{Num}) \wedge \mathsf{betw}^n_n & n \in \mathsf{Num} \\
\mathsf{const}(s) & = & \mathsf{type}(\mathsf{Str}) \wedge \mathsf{pattern}(\underline{s}) & s \in \mathsf{Str} \\
\mathsf{const}([J_1, \ldots, J_n]) & = & \mathsf{type}(\mathsf{Arr}) \wedge \mathsf{ite}^n_n \wedge \mathsf{itemAt}(1 : \mathsf{const}(J_1)), \ldots, \mathsf{itemAt}(n : \mathsf{const}(J_n)) & \\
\mathsf{const}(\{k_1 : J_1, \ldots, k_n : J_n\}) & = & \mathsf{type}(\mathsf{Obj}) \wedge \mathsf{req}(k_1, \ldots, k_n) \wedge \mathsf{pro}^n_0 & \\
 & & \quad \wedge\, \mathsf{props}(\underline{k_1} : \mathsf{const}(J_1); \mathbf{t}) \wedge \ldots \wedge \mathsf{props}(\underline{k_n} : \mathsf{const}(J_n); \mathbf{t}) &
\end{array}
$$

Figure 12: Elimination of enum and const.

Not elimination exploits a complement operator $\overline{r}$ that denotes the pattern that matches any string that is not matched by r, and the operator $r_1 \sqcap r_2$ that denotes a pattern whose language is $L(r_1) \cap L(r_2)$. While this can be regarded as a metanotation, since regular languages are closed under complement and intersection, in our implementation we actually extended JSON Schema regular expressions with these operators, for complexity reasons. We delay the discussion of the choice to Section 5.5.

The not-elimination algorithm starts with a simplification phase, where we use the following derived operators, similar to those used in JSL for arrays [9]:

$$
\begin{aligned}
\mathsf{itemAt}(i : S) &= \mathsf{items}(\mathbf{t}^1, \ldots, \mathbf{t}^{i-1}, S; \mathbf{t}) \\
\mathsf{itemsAfter}(i : S) &= \mathsf{items}(\mathbf{t}^1, \ldots, \mathbf{t}^i; S)
\end{aligned}
$$

These are the simplification steps.

1. items and props simplification: we rewrite each $\mathsf{items}(S_1, \ldots, S_n; S_a)$ as $\mathsf{itemAt}(1 : S_1) \wedge \ldots \wedge \mathsf{itemAt}(n : S_n) \wedge \mathsf{itemsAfter}(n : S_a)$ and each $\mathsf{props}(r_1 : S_1, \ldots, r_n : S_n; S_a)$ as $\mathsf{props}(r_1 : S_1; \mathbf{t}) \wedge \ldots \wedge \mathsf{props}(r_n : S_n; \mathbf{t}) \wedge \mathsf{props}(\overline{(r_1 | \ldots | r_n)} : S_a; \mathbf{t})$.
2. Type simplification: we rewrite each $\mathsf{type}(T_1, \ldots, T_n)$ as $\mathsf{type}(T_1) \vee \ldots \vee \mathsf{type}(T_n)$.
3. Const-elimination: we rewrite every instance of const and of enum, with the only exceptions of $\mathsf{const}(\texttt{true})$ and $\mathsf{const}(\texttt{false})$, through the repeated application of the rules shown in Figure 12, as also done by Habib et al. in [2].
4. propNames elimination: we rewrite every instance of $\mathsf{propNames}(S)$ using $\mathsf{props}(r_S : \mathbf{f}; \mathbf{t})$, as specified in Section 5.3.
5. Not-explicitation: we rewrite every instance of $S_1 \Rightarrow S_2$, $(S_1 \Rightarrow S_2 \,|\, S_3)$ and $\textcircled{1}(S_1, \ldots, S_n)$, according to their definition; the only remaining boolean operators are $\neg$, $\wedge$, $\vee$, $\mathbf{t}$, $\mathbf{f}$.

Naïf not-explicitation may exponentially increase the size of the input schema, since the translation of $\textcircled{1}(S_1, \ldots, S_n)$ takes n copies of each argument, and that of $(S_1 \Rightarrow S_2 \,|\, S_3)$ takes two copies of S_1. This explosion can be easily avoided, by substituting each duplicated argument of these two operators with a fresh variable, so that the not-explicitation phase would only multiply the number of occurrences of these variables, but not the entire subschemas represented. Moreover, the obvious encoding of $\textcircled{1}(x_1, \ldots, x_n)$ produces an expression whose size is in $O(n^2)$, but there exists an alternative encoding with linear size, that we present in Section 5.4. Hence, not-explicitation can be implemented in such a way that its output size is linear in the input size, and the same holds for the other phases of simplification.

On this simplified form, we now apply the following two fundamental steps. 2.15

1. Not-completion of variables: this is a key technical step, since not-elimination needs to deal with the presence of recursive variables in negative positions. In this step, for every variable $x_n : S_n$ we define a complement variable $not\_x_n : \neg S_n$, which will then be used to eliminate negation applied to x_n.
2. Not-pushing: given a not-completed pair (S, E) we repeatedly push negation inside every $\neg S'$ expression until negation reaches the leaves and is removed.

Not-completion of variables. Not-completion of variables is a key step that allows us to deal with the combined presence of unrestricted negation and recursive variables. In particular, not-completion transforms a set of definitions as follows.

Definition 5 (Not-completion of variables). Not completion of a pair (S, E) is defined as following, where *not_* is any string that does not appear at the beginning of any variable name among $x_1, \ldots, x_n$:

$$\begin{array}{l}\text{not-completion}(S \ \mathsf{defs}(x_1 : S_1 \ldots, x_n : S_n)) = \\ \ S \ \mathsf{defs}(x_1 : S_1, \ldots, x_n : S_n, not\_x_1 : \neg S_1, \ldots, not\_x_n : \neg S_n)\end{array}$$

As a result, every variable x has a complement variable $co(x)$ defined in the natural way: $co(x_i) = not\_x_i$ and $co(not\_x_i) = x_i$. Variable $co(x)$ will later be used for not-elimination.

Property 8. Let $(x_1 : S_1 \ldots, x_n : S_n)$ be a closing environment. Then, for every variable x_i with $i \in \{\!\{1..n\}\!\}$, we have:

$$\begin{array}{l}[\![\neg x_i \ \mathsf{defs}(x_1 : S_1, \ldots, x_n : S_n)]\!] \ = \\ [\![not\_x_i \ \mathsf{defs}(x_1 : S_1, \ldots, x_n : S_n, not\_x_1 : \neg S_1, \ldots, not\_x_n : \neg S_n)]\!]\end{array}$$

Proof. Immediate, by Theorem 1. □

The not-pushing phase. The not-pushing phase pushes negation down any algebraic expression up to its complete elimination. Not-pushing is defined by the rules in Figure 13. Observe that the negation of each conditional operation asserts the corresponding type, while the negation of const is actually conditional: if the value is a boolean, then it is equal to `false`/`true`.

Not-pushing over len_0^M or len_i^∞ generates one satisfiable bound and one that is actually illegal (len_0^{-1} or $\mathsf{len}_{\infty+1}^\infty$). Rather than splitting the rule in three cases, we just assume that the illegal term is rewritten as **f** in the resulting disjunction. An analogous assumption is made for the betw, xBetw, pro, ite operators.

Not-elimination preserves the semantics of the schema.

Property 9. The not-elimination procedure preserves the semantics of the schema.

Proof. The simplification steps preserve the semantics by construction. Not-completion only adds variables, hence it does not affect the semantics. The not-pushing rules should be analyzed one by one. We just analyze the most interesting ones.

- $\neg(\mathsf{type}(T)) \ = \ \bigvee(\mathsf{type}(T') \mid T' \neq T)$

  The rule for $\neg(\mathsf{type}(T))$ expresses the closed nature of our model — every value belongs to one of the six base types. If we wanted to build a tool that reasons in an open model, we should reconsider this rule by adding a seventh category, for all the values that do not belong to the closed model.

$$
\begin{aligned}
\neg\mathbf{t} &= \mathbf{f}\\
\neg\mathbf{f} &= \mathbf{t}\\
\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\\
\neg(\mathsf{type}(T)) &= \bigvee(\mathsf{type}(T') \mid T' \neq T)\\
\neg(\mathsf{const}(\mathtt{true})) &= \bigvee(\mathsf{type}(T) \mid T \neq \mathsf{Bool}) \vee \mathsf{const}(\mathtt{false})\\
\neg(\mathsf{const}(\mathtt{false})) &= \bigvee(\mathsf{type}(T) \mid T \neq \mathsf{Bool}) \vee \mathsf{const}(\mathtt{true})\\
\neg(\mathsf{len}_i^j) &= \mathsf{type}(\mathsf{Str}) \wedge (\mathsf{len}_0^{i-1} \vee \mathsf{len}_{j+1}^{\infty})\\
\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})^2\\
\neg(\mathsf{xBetw}_m^M) &= \mathsf{type}(\mathsf{Num}) \wedge (\mathsf{betw}_{-\infty}^m \vee \mathsf{betw}_M^{\infty})^2\\
\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{ite}_i^j) &= \mathsf{type}(\mathsf{Arr}) \wedge (\mathsf{ite}_0^{i-1} \vee \mathsf{ite}_{j+1}^{\infty})^1\\
\neg(\mathsf{uniqueItems}) &= \mathsf{type}(\mathsf{Arr}) \wedge \mathsf{repeatedItems}\\
\neg(\mathsf{repeatedItems}) &= \mathsf{type}(\mathsf{Arr}) \wedge \mathsf{uniqueItems}\\
\neg(\mathsf{itemAt}(i : S)) &= \mathsf{type}(\mathsf{Arr}) \wedge \mathsf{itemAt}(i : \neg S_i) \wedge \mathsf{ite}_i^{\infty}\\
\neg(\mathsf{itemsAfter}(n : S)) &= \mathsf{type}(\mathsf{Arr}) \wedge \mathsf{contAfter}(n : \neg S)\\
\neg(\mathsf{contAfter}(n : S)) &= \mathsf{type}(\mathsf{Arr}) \wedge \mathsf{itemsAfter}(n : \neg S)\\
\neg(\mathsf{pro}_i^j) &= \mathsf{type}(\mathsf{Obj}) \wedge (\mathsf{pro}_0^{i-1} \vee \mathsf{pro}_{j+1}^{\infty})^1\\
\neg(\mathsf{props}(r : S; \mathbf{t})) &= \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; \mathbf{t})\\
\neg(x) &= co(x)
\end{aligned}
$$

[1]Terms with an upper bound -1 or a lower bound $\infty + 1$ are rewritten as **f**
[2]Terms with an upper bound $-\infty$ or a lower bound ∞ are rewritten as **f**

Figure 13: Not-pushing rules.

- $\neg(\mathsf{mulOf}(q)) = \mathsf{type}(\mathsf{Num}) \wedge \mathsf{notMulOf}(q)$ 1.3

  By definition we have $\mathsf{notMulOf}(q) = \mathsf{type}(\mathsf{Num}) \Rightarrow \neg\mathsf{mulOf}(q)$. We have that $J \in [\![\mathsf{mulOf}(q)]\!]_E$ iff $J \in JVal(\mathsf{Num}) \Rightarrow \exists k$ *integer with* $J = k * q$. So $J \notin [\![\mathsf{mulOf}(q)]\!]_E$ iff $J \in JVal(\mathsf{Num}) \wedge \nexists k$ *integer with* $J = k * q$, that is, $J \in [\![\mathsf{type}(\mathsf{Num}) \wedge \mathsf{notMulOf}(q)]\!]_E$.

- $\neg(\mathsf{itemAt}(i : S)) = \mathsf{type}(\mathsf{Arr}) \wedge \mathsf{itemAt}(i : \neg S_i) \wedge \mathsf{ite}_i^{\infty}$

  By definition, $J \in [\![\mathsf{itemAt}(i : S)]\!]_E$ iff $(J = [J_1, \ldots, J_m] \wedge m \geq i) \Rightarrow J_i \in [\![S]\!]_E$, hence $J \notin [\![\mathsf{itemAt}(i : S)]\!]_E$ iff $J = [J_1, \ldots, J_m] \wedge m \geq i \wedge \neg(J_i \in [\![S]\!]_E)$, that is, $J \in [\![\mathsf{type}(\mathsf{Arr}) \wedge \mathsf{itemAt}(i : \neg S_i) \wedge \mathsf{ite}_i^{\infty}]\!]_E$.

- $\neg(\mathsf{itemsAfter}(n : S)) = \mathsf{type}(\mathsf{Arr}) \wedge \mathsf{contAfter}(n : \neg S)$

  By definition, $J \in [\![\mathsf{itemsAfter}(n : S)]\!]_E$ iff $J = [J_1, \ldots, J_m] \Rightarrow \forall i.\ n < i \leq m \Rightarrow J_i \in [\![S]\!]_E$, hence $J \notin [\![\mathsf{itemsAfter}(n : S)]\!]_E$ iff $J = [J_1, \ldots, J_m] \wedge \exists i.\ n < i \leq m \wedge \neg(J_i \in [\![S]\!]_E)$, that is, $J \in [\![\mathsf{type}(\mathsf{Arr}) \wedge \mathsf{contAfter}(n : \neg S)]\!]_E$.

- $\neg(\mathsf{contAfter}(n : S)) \;=\; \mathsf{type}(\mathsf{Arr}) \wedge \mathsf{itemsAfter}(n : \neg S)$

  By definition, $J \in [\![\mathsf{contAfter}(n : S)]\!]_E$ iff $J = [J_1, \ldots, J_m] \Rightarrow \exists i.\ n < i \leq m \wedge J_i \in [\![S]\!]_E$, hence $J \notin [\![\mathsf{contAfter}(n : S)]\!]_E$ iff $J = [J_1, \ldots, J_m] \wedge (\forall i.\ n < i \leq m \Rightarrow \neg(J_i \in [\![S]\!]_E))$, that is, $J \in [\![\mathsf{type}(\mathsf{Arr}) \wedge \mathsf{itemsAfter}(n : \neg S)]\!]_E$.

- $\neg(x) \;=\; co(x)$

  This is a consequence of Property 8.

- $\neg(\mathsf{props}(r : S); \mathbf{t}) = \mathsf{type}(\mathsf{Obj}) \wedge \mathsf{pattReq}(r : \neg S)$ 1.3

  By definition, $J \in [\![(\mathsf{props}(r : S); \mathbf{t})]\!]_E$ iff $J = \{k_1 : J_1, \ldots, k_m : J_m\} \Rightarrow \forall l \in \{\!\{1..m\}\!\}.\ k_l \in L(r) \Rightarrow J_l \in [\![S]\!]_E$. Hence $J \notin [\![(\mathsf{props}(r : S); \mathbf{t})]\!]_E$ iff $J = \{k_1 : J_1, \ldots, k_m : J_m\} \wedge \exists l \in \{\!\{1..m\}\!\}.\ k_l \in L(r) \wedge \neg(J \in [\![S]\!]_E)$, that is $J \in [\![\mathsf{type}(\mathsf{Obj}) \wedge \mathsf{pattReq}(r : \neg S)]\!]_E$

- $\neg(\mathsf{pattReq}(r : S)) = \mathsf{type}(\mathsf{Obj}) \wedge \mathsf{props}(r : \neg S; \mathbf{t})$

  By definition we have $\mathsf{pattReq}(r : S) = \mathsf{type}(\mathsf{Obj}) \Rightarrow \neg\mathsf{props}(r : \neg S; \mathbf{t})$, hence $\mathsf{pattReq}(r : S) = \neg\mathsf{type}(\mathsf{Obj}) \vee \neg\mathsf{props}(r : \neg S; \mathbf{t})$. By negating both sides and by De Morgan rules we obtain $\neg\mathsf{pattReq}(r : S) = \mathsf{type}(\mathsf{Obj}) \wedge \mathsf{props}(r : \neg S; \mathbf{t})$.

- $\neg\mathbf{t} = \mathbf{f},\ \ \neg\mathbf{f} = \mathbf{t},\ \ \neg(S_1 \wedge S_2) = (\neg S_1) \vee (\neg S_2),\ \ \neg(S_1 \vee S_2) = (\neg S_1) \wedge (\neg S_2),\ \ \neg(\neg S) = S$ 1.3

  These are the standard De Morgan rules for the boolean algebra.

- $\neg(\mathsf{const}(\texttt{true})) = \bigvee(\mathsf{type}(T) \mid T \neq \mathsf{Bool}) \vee \mathsf{const}(\texttt{false})$,
  $\neg(\mathsf{const}(\texttt{false})) = \bigvee(\mathsf{type}(T) \mid T \neq \mathsf{Bool}) \vee \mathsf{const}(\texttt{true})$

  A value is different from `true` if it either belongs to a type other than Bool, or if it is the boolean value `false`, and similarly for `false`.

- $\neg(\mathsf{len}_i^j) = \mathsf{type}(\mathsf{Str}) \wedge (\mathsf{len}_0^{i-1} \vee \mathsf{len}_{j+1}^{\infty})$,
  $\neg(\mathsf{ite}_i^j) = \mathsf{type}(\mathsf{Arr}) \wedge (\mathsf{ite}_0^{i-1} \vee \mathsf{ite}_{j+1}^{\infty})$,
  $\neg(\mathsf{pro}_i^j) = \mathsf{type}(\mathsf{Obj}) \wedge (\mathsf{pro}_0^{i-1} \vee \mathsf{pro}_{j+1}^{\infty})$

  A value violates len_i^j if it is a string and is either strictly shorter than i or strictly longer than j; when $i = 0$, then len_0^{-1} is substituted with $\mathbf{f}$ and we have $\neg(\mathsf{len}_0^j) = \mathsf{type}(\mathsf{Str}) \wedge (\mathbf{f} \vee \mathsf{len}_{j+1}^{\infty})$, which is correct since a string violates len_0^j if, and only if, is strictly longer then j. Similarly, we have $\neg(\mathsf{len}_i^{\infty}) = \mathsf{type}(\mathsf{Str}) \wedge (\mathsf{len}_0^{i-1} \vee \mathbf{f})$, which is correct since a string violates len_i^{∞} when is strictly shorter than i. The cases for ite_i^j and pro_i^j are analogous.

- $\neg(\mathsf{betw}_m^M) = \mathsf{type}(\mathsf{Num}) \wedge (\mathsf{xBetw}_{-\infty}^m \vee \mathsf{xBetw}_M^{\infty})$,
  $\neg(\mathsf{xBetw}_m^M) = \mathsf{type}(\mathsf{Num}) \wedge (\mathsf{betw}_{-\infty}^m \vee \mathsf{betw}_M^{\infty})$

  A value violates betw_m^M if it is a number and is either strictly smaller than m or strictly greater than M; when $m = -\infty$, then $\mathsf{xBetw}_{-\infty}^{-\infty}$ is substituted with $\mathbf{f}$ and we have $\neg(\mathsf{betw}_{-\infty}^M) = \mathsf{type}(\mathsf{Num}) \wedge (\mathbf{f} \vee \mathsf{xBetw}_M^{\infty})$, which is correct since a number violates $\mathsf{betw}_{-\infty}^M$ if, and only if, is strictly greater than M. Similarly, we have $\neg(\mathsf{betw}_m^{\infty}) = \mathsf{type}(\mathsf{Num}) \wedge (\mathsf{xBetw}_{-\infty}^m \vee \mathbf{f})$, since a number violates betw_m^{∞} when is strictly smaller than m. The case for xBetw_m^M is analogous.

- $\neg(\mathsf{pattern}(r)) = \mathsf{type}(\mathsf{Str}) \wedge \mathsf{pattern}(\overline{r})$,
  $\neg(\mathsf{notMulOf}(q)) = \mathsf{type}(\mathsf{Num}) \wedge \mathsf{mulOf}(q)$,
  $\neg(\mathsf{uniqueItems}) = \mathsf{type}(\mathsf{Arr}) \wedge \mathsf{repeatedItems}$,
  $\neg(\mathsf{repeatedItems}) = \mathsf{type}(\mathsf{Arr}) \wedge \mathsf{uniqueItems}$

  A value violates $\mathsf{pattern}(r)$ if, and only if, it is a string, and this string does not belong to $L(r)$; the fact that the value is a string is expressed by $\mathsf{type}(\mathsf{Str})$ and the fact that it does not belong to $L(r)$ is expressed by $\mathsf{pattern}(\overline{r})$, where $\overline{r}$ is the pattern that matches the complement of $L(r)$; the term $\mathsf{type}(\mathsf{Str})$ is necessary because values that are not strings trivially satisfy $\mathsf{pattern}(\overline{r})$, but they do not violate $\mathsf{pattern}(r)$. The proofs for $\mathsf{notMulOf}(q)$, $\mathsf{uniqueItems}$, and $\mathsf{repeatedItems}$ are analogous.

□

Example 6. Consider again the JSON Schema document of Section 2.3:

```
{ "properties": {"a": {"not": {"$ref": "#"}}} }
```

We demonstrate how to eliminate negation in order to clarify its meaning. We write it in our algebra as follows. 2.17

$$x\ \mathsf{defs}(x : \mathsf{props}(\underline{a} : \neg x; \mathbf{t}))$$

By applying not-completion, we get the following definition (for readability, we omit the trivial ";**t**" at the end of the props operator).

$$x\ \mathsf{defs}(x : \mathsf{props}(\underline{a} : \neg x), \mathit{not\_x} : \neg\mathsf{props}(\underline{a} : \neg x))$$

This is how not-elimination would now proceed within our algebra (we push ¬ through props using Property 1 and use $\{S_1, S_2\}$ for conjunction):

$$\begin{aligned}
&\mathsf{defs}(\quad x : \mathsf{props}(\underline{a} : \neg x),\ \ \mathit{not\_x} : \neg\mathsf{props}(\underline{a} : \neg x)\ \ ) \rightarrow \\
&\mathsf{defs}(\quad x : \mathsf{props}(\underline{a} : co(x)),\ \ \mathit{not\_x} : \{\mathsf{type}(\mathsf{Obj}), \mathsf{req}(a), \mathsf{props}(\underline{a} : \neg\neg x)\}\ \ ) \rightarrow \\
&\mathsf{defs}(\quad x : \mathsf{props}(\underline{a} : \mathit{not\_x}),\ \ \mathit{not\_x} : \{\mathsf{type}(\mathsf{Obj}), \mathsf{req}(a), \mathsf{props}(\underline{a} : x)\}\ \ )
\end{aligned}$$

We now substitute *not_x* with its definition, and obtain a much clearer schema: *if* the instance is an object with an *a* member, then the value of that member *must* be an object with an *a* member, whose value satisfies the same specification:

$$x\ \mathsf{defs}(\ \ x : \mathsf{props}(\underline{a} : \{\mathsf{type}(\mathsf{Obj}), \mathsf{req}(a), \mathsf{props}(\underline{a} : x)\})\ \ )$$

These are some examples of values that match that schema:

$$1,\quad \{"b" : 2\},\quad \{"a" : \{"a" : "foo"\}\},\quad \{"a" : \{"a" : \{"a" : \{"a" : \mathtt{null}\}\}\}\}$$

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

Figure 14: Definition of $\mathit{PattOfS}(S, E)$.

5.3. $\mathsf{propNames}(S)$ *encoded through* $\mathit{PattOfS}(S, E)$

The assertion $\mathsf{propNames}(S)$ in an environment E requires that, if the instance is an object, every member name belongs to $[\![S]\!]_E$, which is equivalent to saying that no member name exists that is not in $[\![S]\!]_E$. Hence, if we translate every S into a pattern $r = \mathit{PattOfS}(S, E)$ that exactly describes the strings that satisfy $[\![S]\!]_E$, we can translate $\mathsf{propNames}(S)$ into $\mathsf{props}(\mathit{PattOfS}(\neg S, E) : \mathbf{f}; \mathbf{t})$, which means: if the instance is an object, it cannot contain any member whose name matches the complement of $\mathit{PattOfS}(S)$.

We now show how to transform every schema S into a pattern $\mathit{PattOfS}(S, E)$ such that the following equivalences hold, where we write $S \equiv_E S'$ as a more readable notation for $(S, E) \equiv (S', E)$

$$
\begin{array}{lcl}
\mathsf{type}(\mathsf{Str}) \wedge S & \equiv_E & \mathsf{type}(\mathsf{Str}) \wedge \mathsf{pattern}(\mathit{PattOfS}(S, E)) \\
\mathsf{propNames}(S) & \equiv_E & \mathsf{props}(\mathit{PattOfS}(\neg S, E) : \mathbf{f}; \mathbf{t})
\end{array}
$$

For all the implicative typed assertions S whose type is not Str, such as $\mathsf{mulOf}(q)$, we define $\mathit{PattOfS}(S, E) = .*$, since they are satisfied by any string. For the other operators, $\mathit{PattOfS}(S, E)$ is defined as shown in Figure 14. Observe that, while $\mathit{PattOfS}(\mathsf{mulOf}(q), E) = .*$ since $\mathsf{mulOf}(q)$ is an implicative typed assertion, $\mathit{PattOfS}(\mathsf{type}(\mathsf{Num}), E) = \overline{.*}$, since $\mathsf{type}(\mathsf{Num})$ is not conditional, and is not satisfied by any string; observe that $.*$ matches any string and $\overline{.*}$ matches none. Since $\mathit{PattOfS}(S, E)$ definition does not depend on the schemas that are nested inside typed operators, this definition is well-founded in presence of guarded recursion: after we have expanded a variable x once, in the result of any further expansion x will always be guarded, hence we will not need to expand it again. For the operators not cited, such as enum and the derived boolean operators, we first translate them as in the simplification phase before not-elimination, and then we apply the rules of Figure 14.[1]

It is easy to prove that we have the following equivalences.

Property 10. For any assertion S and for any guarded environment E, the following equiva-

[1]We translate $S_1 \vee S_2$ using $\overline{\overline{\mathit{PattOfS}(S_1, E)} \sqcap \overline{\mathit{PattOfS}(S_2, E)}}$, rather than $\mathit{PattOfS}(S_1, E)|\mathit{PattOfS}(S_2, E)$, since $\mathit{PattOfS}(S_1, E)$ may contain negation and intersection, and in our extended regular expressions we apply intersection and negation only outside the standard regular expression operators. This is just a minor technical choice.

lences hold.

$$\begin{array}{lcl} \mathsf{type}(\mathsf{Str}) \wedge S & \equiv_E & \mathsf{type}(\mathsf{Str}) \wedge \mathsf{pattern}(\mathit{PattOfS}(S, E)) \\ \mathsf{propNames}(S) & \equiv_E & \mathsf{props}(\mathit{PattOfS}(\neg S, E) : \mathbf{f}; \mathbf{t}) \end{array}$$

Proof. $[\![\mathsf{type}(\mathsf{Str}) \wedge S]\!]^i_E = [\![\mathsf{type}(\mathsf{Str}) \wedge \mathsf{pattern}(\mathit{PattOfS}(S, E))]\!]^i_E$ can be proved by induction on i and $|S|$, using induction on i for $S = \mathit{PattOfS}(x, E)$ and on $|S|$ in all the other cases.

$J \in [\![\mathsf{propNames}(S)]\!]_E$ iff, when J is an object, then any name k in J satisfies $k \in [\![S]\!]_E$. $J \in [\![\mathsf{props}(\mathit{PattOfS}(\neg S, E) : \mathbf{f}; \mathbf{t})]\!]_E$ iff, when J is an object, then no name k in J satisfies $k \in [\![\mathsf{pattern}(\mathit{PattOfS}(\neg S, E))]\!]_E$, which, by the previous property, means that no name k in J satisfies $k \in [\![\neg S]\!]_E$, hence, every name satisfies $k \in [\![S]\!]_E$. □

This encoding creates, in general, an exponential blowup, even if one extends regular expres- 2.18
sions with complement and intersection, because of the equation $\mathit{PattOfS}(x, E) = \mathit{PattOfS}(E(x), E)$. This could be avoided either by representing regular expressions by alternating automata, or by extending regular expressions with systems of equations, as discussed in [23], or by avoid- 2.19
ing to encode propNames by extending the language with an operator reqPropName dual to propNames, as discussed in Section 5.5.

5.4. *Linear encoding of oneOf*

We describe here a linear-size encoding of $①(x_1, \ldots, x_m)$; we assume that $log_2 m$ is an integer, which can always be obtained by adding irrelevant **f** arguments to the ① operator. We use $I(l, p)$ to denote the p-th interval of size 2^l, with $0 \leq l \leq log_2 m$, as we did in the construction for Property 2. Since these intervals form a complete balanced binary tree with m leaves, we have $2m - 1$ such intervals.

Given a set of variables $x_1, \ldots, x_n$, for each interval $I(l, p)$, we define two variables $Z_{l,p}$, and $O_{l,p}$ such that:

1. $Z_{l,p}$ is equivalent to the conjunction of $\neg(x_i)$ for all $i \in I(l, p)$, hence $Z_{l,p}$ (Z for Zero) is satisfied iff none of these variables is satisfied;

2. $O_{l,p}$ is satisfied iff one and only one of the variables indexed by an $i \in I(l, p)$ is satisfied (O stands for One).

The environment for these two sets of variables is defined as follows, exploiting the equality $I_{l+1,p} = I_{l,2p-1} \cup I_{l,2p}$, for $0 \leq l \leq (log_2 m - 1)$ and $1 \leq p \leq m/(2^l)$.

$$\begin{array}{lcl@{\qquad}lcl} Z_{0,p} & : & \neg x_p & O_{0,p} & : & x_p \\ Z_{l+1,p} & : & Z_{l,2p-1} \wedge Z_{l,2p} & O_{l+1,p} & : & (O_{l,2p-1} \wedge Z_{l,2p}) \vee (Z_{l,2p-1} \wedge O_{l,2p}) \end{array}$$

The size of this environment is linear in m, and the variable $O_{log_2 m,1}$ encodes $①(x_1, \ldots, x_m)$.

5.5. *About regular expressions*

The negation of $\mathsf{pattern}(r)$ can be expressed in JSON Schema, since we have the following equivalence:

$$\neg\mathsf{pattern}(r) \equiv \mathsf{type}(\mathsf{Str}) \wedge \mathsf{pattern}(\overline{r})$$

However, the simplicity of this equivalence hides a problem. If we regard $\overline{r}$ as a notation that indicates a regular expression whose language is the complement of $L(r)$, then we must consider the fact that the size of that expression is, in the worst case, doubly exponential in $|r|$ [24], hence the result of not-elimination could be very big, and unreadable. In our implementation, we added an operator $\overline{r}$ at the outermost level of regular expressions, which keeps the size of the result of not-elimination linear in the input size, and is very convenient for a theoretical treatment, but we would not suggest its addition to the JSON Schema standard, as its impact would be too big.

In practice, the problem of negation-closure for $\mathsf{pattern}(r)$ would be better solved by adding a dedicated dual operator $\mathsf{notPattern}$ defined by:

$$[\![\mathsf{notPattern}(r)]\!]_E = [\![\mathsf{typeStr} \Rightarrow \neg\mathsf{pattern}(r)]\!]_E$$

This operator is characterized by the natural not-pushing rules:

$$\begin{array}{lcl} \neg\mathsf{pattern}(r) & \equiv & \mathsf{type}(\mathsf{Str}) \wedge \mathsf{notPattern}(r) \\ \neg\mathsf{notPattern}(r) & \equiv & \mathsf{type}(\mathsf{Str}) \wedge \mathsf{pattern}(r) \end{array}$$

Similarly, in Section 5.3 we have proved that $\mathsf{propNames}(S)$ can be encoded in terms of $\mathsf{props}(\mathit{PattOfS}(\neg S, E) : \mathbf{f}; \mathbf{t})$, which reduces its negation closure to the negation closure of props, at the price of adding a tree of $\overline{r}$ and $\sqcap$ operators around the language of regular expressions. This is the approach that we used in our implementation of not-elimination, but, from a language-design viewpoint, it would make much more sense to just add a direct dual $\mathsf{reqPropName}(S)$ that requires at least one name satisfying S, with the natural semantics:

$$[\![\mathsf{reqPropName}(S)]\!]_E = [\![\mathsf{typeObj} \Rightarrow \neg\mathsf{propNames}(\neg S)]\!]_E$$

This operator is characterized by the natural not-pushing rules:

$$\begin{array}{lcl} \neg\mathsf{reqPropName}(S) & \equiv & \mathsf{type}(\mathsf{Obj}) \wedge \mathsf{propNames}(\neg S) \\ \neg\mathsf{propNames}(S) & \equiv & \mathsf{type}(\mathsf{Obj}) \wedge \mathsf{reqPropName}(\neg S) \end{array}$$

Finally, at the beginning of not-elimination (Section 5.2) we translate $\mathsf{props}(r_1 : S_1, \ldots, r_n : S_n; S_a)$ as $\mathsf{props}(r_1 : S_1; \mathbf{t}) \wedge \ldots \wedge \mathsf{props}(r_n : S_n; \mathbf{t}) \wedge \mathsf{props}(\overline{(r_1|\ldots|r_n)} : S_a; \mathbf{t})$, again using $\overline{r}$. Then, not-elimination, when applied to $\mathsf{props}(\overline{(r_1|\ldots|r_n)} : S_a; \mathbf{t})$, produces an instance of $\mathsf{pattReq}(\overline{(r_1|\ldots|r_n)} : \neg S_a)$. This could be avoided at the price of a new algebraic operator $\mathsf{addPropReq}(r_1, \ldots, r_n; S_a)$, which, at the JSON Schema level, could be expressed as `"additionalPropertyRequired"` : S_a, getting its meaning from the surrounding operators as currently happens for `"additionalProperties"`, and in the algebra it would have the obvious semantics:

$$[\![\mathsf{addPropReq}(r_1, \ldots, r_n; S_a)]\!]_E \equiv [\![\mathsf{pattReq}(\overline{(r_1|\ldots|r_n)} : S_a)]\!]_E$$

At this point, we would simplify $\mathsf{props}(r_1 : S_1, \ldots, r_n : S_n; S_a)$ as $\mathsf{props}(r_1 : S_1; \mathbf{t}) \wedge \ldots \wedge \mathsf{props}(r_n : S_n; \mathbf{t}) \wedge \mathsf{props}(r_1 : \mathbf{t}, \ldots, r_n, \mathbf{t}; S_a)$, and we would use the following not-pushing rules, with no need for the $\overline{r}$ operator:

$$\begin{array}{lcl} \neg\mathsf{props}(r : S; \mathbf{t}) & \equiv & \mathsf{type}(\mathsf{Obj}) \wedge \mathsf{pattReq}(r : \neg S) \\ \neg\mathsf{pattReq}(r : S) & \equiv & \mathsf{type}(\mathsf{Obj}) \wedge \mathsf{props}(r : \neg S; \mathbf{t}) \\ \neg\mathsf{props}(r_1 : \mathbf{t}, \ldots, r_n : \mathbf{t}; S_a) & \equiv & \mathsf{type}(\mathsf{Obj}) \wedge \mathsf{addPropReq}(r_1, \ldots, r_n; \neg S_a)) \\ \neg\mathsf{addPropReq}(r_1, \ldots, r_n; S_a) & \equiv & \mathsf{type}(\mathsf{Obj}) \wedge \mathsf{props}(r_1 : \mathbf{t}, \ldots, r_n : \mathbf{t}; \neg S_a) \end{array}$$

To sum up, when studying negation closure of JSON Schema, the issue raised by the negation of pattern-based operators such as $\neg$pattern(r) is not trivial, and the best answer depends on the applications we are envisioning for not-elimination. Since regular expressions are closed under complement, but rewriting $\neg r$ into a positive expression has double exponential complexity, if we are looking for compact not-elimination meant for machine manipulation, which was our central motivation, then the extension of regular expressions is a reasonable choice. If, instead, one wants to design a language where every operator has a dual, in order to make it easier to reason about negation, then we think that one may leave regular expressions untouched and add explicit dual operators such as notPattern, reqPropName, and addPropReq.

6. The "`minContains`" and "`maxContains`" operators (Draft 2019-09)

6.1. The $\#_i^j S$ operator

Draft 2019-09 introduced the new operators "`minContains`": n and "`maxContains`": n, where n is a natural number. To understand their semantics, consider a schema that contains, at the top level, the three assertions "`contains`": S, "`minContains`": m, "`maxContains`": M. An instance J satisfies this combination iff: if it is an array, then it contains at least m and at most M elements that satisfy S. When `minContains` is missing, its value defaults to 1, while a missing `maxContains` means no upper limit. The three operators are better interpreted as specifying the parameters of a unique operator; in particular, when "`minContains`": m is present, one cannot consider "`contains`": S as an independent operator, since an adjacent "`minContains`": 0 has the effect that no item that satisfies S is required any more: the usual effect of "`contains`": S is "deactivated". These operators have an interesting relationship with the problem of negation closure, hence we discuss them here.

We represent the "`contains`" : S, "`minContains`" : l, "`maxContains`" : j combination by adding an operator $\#_l^j S$ to the algebra, with $l \in \mathbb{N}$ and $j \in \mathbb{N}^\infty$. The semantics of $\#_l^j S$ is defined as follows.

$$[\![\#_l^j S]\!]_E^i \ = \ \{\!\{ \ J \ | \ J = [J_1, \ldots, J_n] \Rightarrow l \ \le \ |\{\!\{ \, o \ | \ o \in \{\!\{1..n\}\!\} \wedge J_o \in [\![S]\!]_E^i \ \}\!\}| \ \le \ j \ \}\!\}$$

Similarly to ite$_l^j$, the operator $\#_l^j S$ combines a requirement $\#_l^\infty S$ and a constraint $\#_0^j S$.

The operator $\#_l^j S$ cannot be expressed in the algebra (Theorem 6).

Theorem 6. The pair $(\#_l^j S, E)$ with $l \le j$ cannot be expressed in the algebra if (S, E) is not trivial and either $l \ge 2$ or $0 < j < \infty$.

Proof. Assume that (S, E) is not trivial and that $D = S_0$ defs(E') expresses $(\#_l^j S, E)$ with $l \ge 2$ or $0 < j < \infty$. We first consider the case when $l = 2$, and we then consider the cases for $l > 2$ and for $0 < j < \infty$.

Consider $J^+ \in [\![S]\!]_E$ and $J^- \notin [\![S]\!]_E$. If we say that n is the head-length of an operator items$(S_1, \ldots, S_n; S')$, let N be the maximum among the head-lengths of all instances of this operator inside D and the lengths of all arrays that appear inside the arguments of enum and const in D.

Consider the following arrays of length $N + 3$, starting with $N + 1$ copies of J^-: 2.21

$[J^-, \ldots, J^-, J^+, J^+]$	A_1
$[J^-, \ldots, J^-, J^+, J^-]$	A_2

An assertion S' satisfies *BothOrNone* if $A_1 \in [\![S']\!]^i_E \Leftrightarrow A_2 \in [\![S']\!]^i_E$. We now prove that every assertion S' inside D satisfies *BothOrNone*, by induction on the lexicographic pair $(i, |S'|)$. In this way, we prove that D satisfies *BothOrNone*, which is a contradiction since $(\#^j_2 S, E)$ is satisfied by A_1 but not by A_2, since A_1 has two copies of J^+ and A_2 only one.

Observe that *BothOrNone* is a stronger property than *NonDifferentiating* that we used in the previous proofs. In this case we cannot rely on *NonDifferentiating* since we want to prove that $(\#^j_2 S, E)$ cannot be expressed in a language that includes negation, and the weaker invariant *NonDifferentiating* is not preserved by negation.

We now prove the invariant. When $S' = x$, if $i = 0$, then both arrays are rejected (i.e., are not *accepted*), and when $i > 0$ the result follows by induction on i. Any implicative typed assertion that is unrelated to arrays accepts both arrays, while $\mathsf{uniqueItems}$, and all const and enum in D, reject both of them, by construction. Since they have the same length, ite^l_j will not distinguish the two. Consider any $S' = \mathsf{items}(S_1, \ldots, S_n; S'')$ and assume that it accepts A_1. This means that each S_i accepts J^- and that S'' accepts both J^- and J^+, since N is bigger than n by construction, hence $A_2 \in [\![S']\!]^i_{E'}$ as well. In the same way we prove that $A_2 \in [\![S']\!]^i_{E'} \Rightarrow A_1 \in [\![S']\!]^i_{E'}$.

If $S' = \mathsf{contains}(S'')$, then it cannot distinguish the two arrays since they contain the same elements. If $S' = S_1 \wedge S_2$, or $S' = S_1 \vee S_2$, or $S' = \neg S_1$, we know, by induction, that S_1 is not able to distinguish A_1 from A_2 and the same holds for S_2, hence no boolean combination of S_1 and S_2 may distinguish A_1 from A_2.

To sum up, for every subterm S' of D we have that $[\![S']\!]^i_{E'}$ does not distinguish the two arrays, and hence the limit $[\![S']\!]_{E'}$ does not distinguish them; therefore, $[\![D]\!]$ cannot be equivalent to $\#^\infty_2 S$.

The case $l > 2$ can be proved in the same way, by having l copies of J^+ in the tail of A_1 and $l - 1$ copies in the tail of A_2. Finally, in the case when $0 < j < \infty$, we put exactly j copies of J^+ in the tail of A_1 and $j + 1$ copies in the tail of A_2, so that A_2 violates the upper bound while A_1 does not, and we reason in the same way. $\square$

6.2. $\#^j_i S$: *negation closure of array operators*

We have seen that full negation closure for the constraint $\mathsf{items}(S_1, \ldots, S_n; S_a)$ can be obtained through the combination of the requirements ite^∞_i and $\mathsf{contAfter}(n : S)$ (Property 9), but the latter cannot be expressed in the algebra without negation (Theorem 3). When we enrich the algebra with the $\#^j_i S$ operator, the situation changes completely. First of all, $\#^j_i S$ immediately subsumes the two operators $\mathsf{contains}(S)$ and ite^j_i:

$$\mathsf{contains}(S) \;=\; \#^\infty_1 S \qquad\qquad \mathsf{ite}^j_i \;=\; \#^j_i \mathbf{t}$$

More interestingly, $\#^j_i S$ also allows one to encode the $\mathsf{contAfter}(n : S)$ operator, as follows. To encode $\mathsf{contAfter}(n : S)$, we define the intervals $I(l, p)$ as in Section 4, and we define a set of variables $U^u_{l,p}$, such that:

$$[J_1, \ldots, J_m] \in [\![U^u_{l,p}]\!]_E \;\Leftrightarrow\; |\, \{\!|\, j \mid 1 \leq j \leq \min(m, n) \wedge j \in I(l, p) \wedge J_j \in [\![S]\!]_E \,|\!\}\, | \leq u$$

i.e., $J \in [\![U^u_{l,p}]\!]_E$ implies that the number of elements J_j of J whose position is in $I(l, p)$ but is not in the tail, and such that $J_j \in [\![S]\!]_E$, is less than u. In the definition above, we count those j that are less than n since $U^u_{l,p}$ counts only the elements that are in the head, and we impose that $j \leq m$ since J_j is not defined past m, hence the upper bound for j in the definition is $\min(m, n)$.

The environment $\mathcal{E}(S,n)$ is defined as follows. In the first two lines we deal with intervals of length $2^0 = 1$ where $u = 0$ — the second line ensures that all positions greater than $n+1$ will be ignored. The third lines splits an interval $I(l,p)$ in two halves, and uses the same technique as in Section 4 to express $U^u_{l,p}$ in terms of $U_{l-1,2p-1,i}$ and $U_{l-1,2p,u-i}$. The fourth line introduces variables for the trivial case when $2^l \leq u$, when the interval $I(l,p)$ contains less than u elements.

$$
\begin{array}{llll}
\multicolumn{4}{l}{\mathcal{E}(S,n) =} \\
(U^0_{0,p} & : & \mathsf{itemAt}(p : \neg S) & 1 \leq p \leq n \\
\ U^0_{0,p} & : & \mathbf{t} & n+1 \leq p \leq 2^{\lceil log_2(n)\rceil} \\
\ U^u_{l,p} & : & \bigvee_{0\leq i\leq u}(U^i_{l-1,2p-1} \wedge U^{u-i}_{l-1,2p}) & 1 \leq l \leq \lceil log_2(n)\rceil,\ 1 \leq p \leq 2^{\lceil log_2(n)\rceil - l},\quad 0 \leq u < 2^l \\
\ U^u_{l,p} & : & \mathbf{t} & 0 \leq l \leq \lceil log_2(n)\rceil,\ 1 \leq p \leq 2^{\lceil log_2(n)\rceil - l},\quad 2^l \leq u < 2^{l+1} \\
)
\end{array}
$$

Property 11.

$$
(\,\mathsf{contAfter}(n : S)\,,\ E\,) \ \equiv\ (\,\bigvee_{0\leq i\leq n} (U^i_{\lceil log_2(n)\rceil,1} \wedge \#^{\infty}_{i+1} S)\,,\ E \cup \mathcal{E}(S,n)\,)
$$

Proof. We first prove that

$$
\begin{array}{l}
J = [J_1,\ldots,J_m] \Rightarrow \\
\quad J \in [\![U^u_{l,p}]\!]_{E^+} \ \Leftrightarrow\ |\{\!\{\, j \mid 1 \leq j \leq \min(m,n) \wedge j \in I(l,p) \wedge J_j \in [\![S]\!]_E \,\}\!\}| \leq u
\end{array}
$$

where $E^+ = E \cup \mathcal{E}(S,n)$, by induction on l.

Case $U^0_{0,p}$: $\mathsf{itemAt}(p : \neg S)$ with $1 \leq p \leq n$: we want to prove that

$$
J \in [\![\mathsf{itemAt}(p : \neg S)]\!]_{E^+} \ \Leftrightarrow\ |\{\!\{\, j \mid 1 \leq j \leq \min(m,n) \wedge j \in I(0,p) \wedge J_j \in [\![S]\!]_E \,\}\!\}| \leq 0
$$

Observe that $I(0,p) = \{\!\{p\}\!\}$. We distinguish the cases where $p > m$ and $p \leq m$. If $p > m$, then $j \leq m$ is incompatible with $j \in I(0,p)$, hence $\{\!\{\, j \mid \ldots \}\!\}$ is empty, hence $|\{\!\{\, j \mid \ldots \}\!\}| \leq 0$ holds, and $J \in [\![\mathsf{itemAt}(p : \neg S)]\!]_{E^+}$ also holds trivially, since the array J has no element at position p. If $p \leq m$, then both the left hand side and the right hand side express the fact that J_p does not belong to $[\![S]\!]_E$, hence they are equivalent.

Case $U^0_{0,p}$: $\mathbf{t}$ with $n+1 \leq p \leq 2^{\lceil log_2(n)\rceil}$: we want to prove that

$$
J \in [\![\mathbf{t}]\!]_{E^+} \ \Leftrightarrow\ |\{\!\{\, j \mid 1 \leq j \leq \min(m,n) \wedge j \in I(0,p) \wedge J_j \in [\![S]\!]_E \,\}\!\}| \leq 0 \text{ when } n+1 \leq p
$$

which holds trivially since, if $n+1 \leq p$, then $\{\!\{\, j \mid 1 \leq j \leq \min(m,n) \wedge j \in I(0,p) \,\}\!\}$ is empty.

Case $U^u_{l,p}$: $\bigvee_{0\leq i\leq u}(U^i_{l-1,2p-1} \wedge U^{u-i}_{l-1,2p})$ with $1 \leq l \leq \lceil log_2(n)\rceil,\ 1 \leq p \leq 2^{\lceil log_2(n)\rceil - l},\ 0 \leq u < 2^l$. We observe that $|\{\!\{\, j \mid 1 \leq j \leq \min(m,n) \wedge j \in I(l,p) \wedge J_j \in [\![S]\!]_E \,\}\!\}| \leq u$ iff exists i such that $0 \leq i \leq u$ and

$$
\begin{array}{l}
|\{\!\{\, j \mid 1 \leq j \leq \min(m,n) \wedge j \in I(l-1,2p-1) \wedge J_j \in [\![S]\!]_E \,\}\!\}| \leq i \\
\wedge\, |\{\!\{\, j \mid 1 \leq j \leq \min(m,n) \wedge j \in I(l-1,2p) \wedge J_j \in [\![S]\!]_E \,\}\!\}| \leq u-i
\end{array}
$$

and we conclude by induction. We also observe that both $U^i_{l-1,2p-1}$ and $U^{u-i}_{l-1,2p}$ are defined when $1 \leq l \leq \lceil log_2(n)\rceil,\ 1 \leq p \leq 2^{\lceil log_2(n)\rceil - l},\ 0 \leq u < 2^l$, reasoning as in the proof of Property 2.

Finally, we have case $U^{u}_{l,p}$: $\mathbf{t}$ with $0 \leq l \leq \lceil log_2(n) \rceil$, $1 \leq p \leq 2^{\lceil log_2(n) \rceil - l}, 2^l \leq u < 2^{l+1}$: we want to prove that

$$J \in [\![\mathbf{t}]\!]_{E^+} \Leftrightarrow |\{\!\{ j \mid 1 \leq j \leq \min(m,n) \wedge j \in I(l,p) \wedge J_j \in [\![S]\!]_E \}\!\}| \leq u$$

which holds trivially since $I(l,p)$ only contains 2^l indexes, and $2^l \leq u$.

At this point, we have proved that

$$J = [J_1, \ldots, J_m] \Rightarrow$$
$$J \in [\![U^{i}_{\lceil log_2(n) \rceil,1}]\!]_{E \cup \mathcal{E}(S,n)} \Leftrightarrow |\{\!\{ j \mid 1 \leq j \leq \min(m,n) \wedge j \in I(\lceil log_2(n) \rceil, 1) \wedge J_j \in [\![S]\!]_E \}\!\}| \leq i$$

Hence, $U^{i}_{\lceil log_2(n) \rceil,1}$ is satisfied by an array iff it has less than i elements in its head positions that satisfy S. We conclude by observing that an array J satisfies contAfter(n : S) iff there exists an i such that J contains at least $i+1$ elements that satisfy S, but at most i of these elements are in the head positions, hence:

$$(\,\mathsf{contAfter}(n : S)\,,\ E\,) \equiv (\bigvee_{0 \leq i \leq n} (\#^{\infty}_{i+1} S \wedge U^{i}_{\lceil log_2(n) \rceil,1})\,,\ E \cup \mathcal{E}(S,n)\,)$$

□

Hence, $\#^{j}_{i} S$ is expressive enough to express contAfter(n : S), and thus to express negation of items, although at the cost of a complex encoding (of size $O(n^2)$).

Finally, we observe that $\#^{j}_{i} S$ is self-dual, so that, while it solves the problem of not-elimination for the items operator, it does not introduce any new not-elimination issue. The self-duality of $\#^{j}_{i} S$ is expressed by the following equation, where $\#^{i-1}_{0} S$ is just **f** when $i = 0$, and $\#^{\infty}_{j+1} S$ is just **f** when $j = \infty$.

$$\neg(\#^{j}_{i} S) = \mathsf{type(Arr)} \wedge (\#^{i-1}_{0} S \vee \#^{\infty}_{j+1} S)$$

While contAfter(n : S) is strictly less expressive than $\#^{j}_{i} S$ (by Theorem 6), it seems to be more compact, in the sense that we could not find any way to express contAfter(n : S) using $\#^{j}_{i} S$ with an expression of size $O(log(n))$.

To summarize, JSON Schema Draft-06 needs four new operators — pattReq, notMulOf, repeatedItems, and contAfter — in order to become closed under negation; if we add $\#^{l}_{j} S$, then we only need three of them.

7. Experiments

We implemented our not-elimination algorithm as part of a more general tool that can be used to verify equivalence and inclusion of JSON Schema documents, to decide satisfiability of a schema, and to generate witnesses from satisfiable schemas, supporting Draft-06 extended with `minContains` and `maxContains`, as described in [23]. An interactive version has been presented as a demo [11, 12]. This Java implementation, that exploits the Brics automaton library [25] for handling regular expressions, comprises about 110K lines of code. 2.22

In this section we report the results of two experiments, validating our claims of practical feasibility of our not-elimination approach. In these experiments we analyze a corpus of schemas and, for each schema S, we measure the size of its internal representation before and after not-elimination, as well as the time required to apply not-elimination to S. A reproduction package is available on GitHub: `https://github.com/sdbs-uni-p/json-schema-not-elimination`.

Table 2: Summary information about the corpus.

Size (Bytes)			
min	max	avg	median
25	747,779	3,984.37	1,524

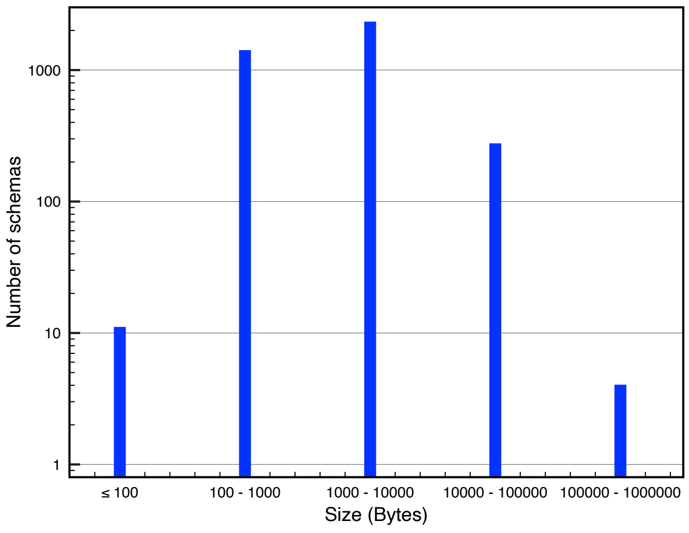

Figure 15: Size distribution of schemas.

7.1. *Datasets*

To run our experiments, we retrieved virtually every accessible, open source-licensed file from GitHub that presents the features of JSON Schema, based on a BigQuery search on the GitHub public dataset (Google hosts a snapshot of all open source-licensed GitHub repositories). We performed duplicate-elimination and data cleaning (better described in [23]), arriving at a corpus of 4,000 real-world schemas. These schemas come from multiple application domains, and are representative of schemas used in real-world scenarios; they cover almost every aspects of the language, apart from `uniqueItems` and some optional constructs, like `format`, whose treatment in our tool is still under development.

In Table 2, we reported the minimum, maximum, average, and median size of these schemas, while Figure 15 illustrates their size distribution.

7.2. *Tests*

We performed two different experiments on our schemas. We first processed them as they are, to analyze the behavior and the effects of not-elimination in a real-world setting, where, as reported in [15], negation is usually applied to relatively simple schemas; it should be observed, however, that negation comes into play also with the translation of `oneOf` and `if-then-else`.

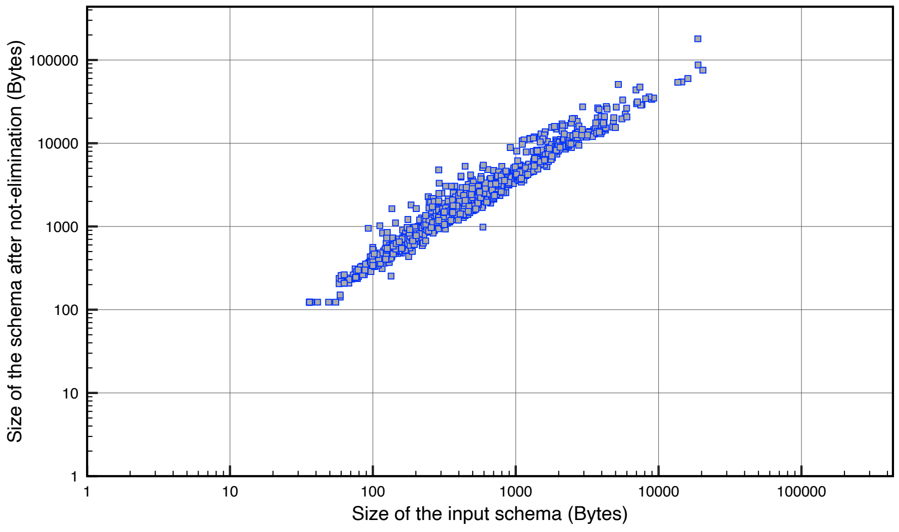

Figure 16: Size increase after not-elimination for original schemas.

Table 3: Summary of experimental results for original schemas.

runtime (ms)				size ratio			
avg	median	max	min	avg	median	max	min
144.158	1	178,107	<1	4.43	4.14	47.14	1.66

To further stress our not-elimination algorithm, we repeated our tests by injecting negation above the schema root node. By doing so, we obtained schemas where negation is applied to very complex and deeply nested schema expressions.

7.3. *Experimental setup*

We performed our experiments on an M1 Pro 8-14 machine with 16 GBs of unified memory, running macOS 12.4. As our tool has been implemented in Java, we used JRE 17.0.3 2022-04-19 LTS for our experiments.

7.4. *Results*

Results of our experiments on unmodified schemas are shown in Figures 16 and 17, where we used for both axes the log-scale. Figure 16 shows that the schema size after not-elimination grows polynomially with the schema size. Figure 17 indicates that in a vast majority of cases running time is below 1s. 2.23

Figures 18 and 19 depict the results we obtained when negation has been injected above the schema root node; we used again the log-scale for both axes. As in the previous case, the schema size grows polynomially and, in most cases, the running time is below 1s.

In Tables 3 and 4 we summarized our findings.

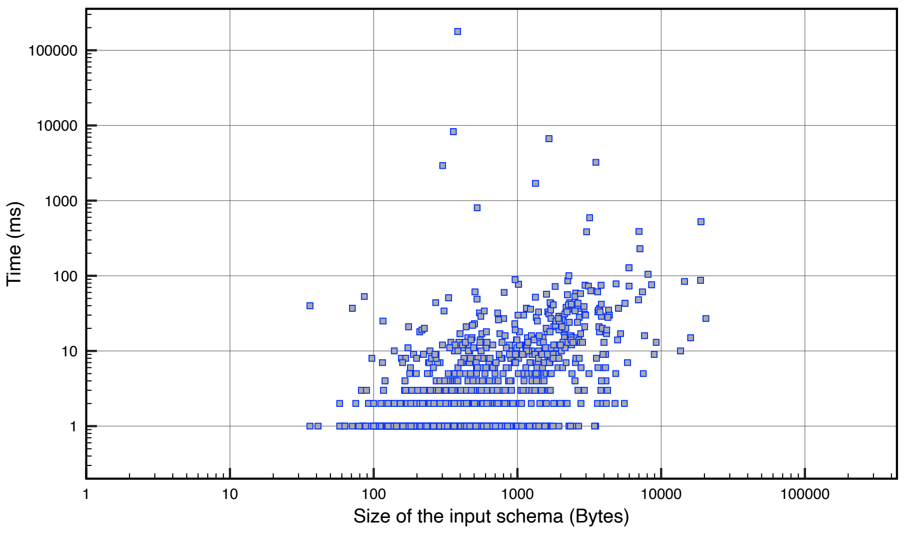

Figure 17: Not-elimination runtime for original schemas.

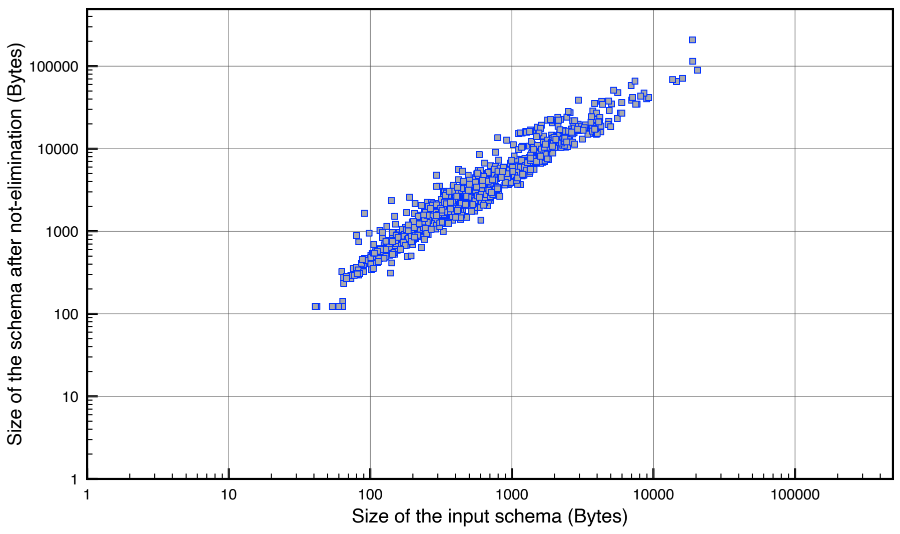

Figure 18: Size increase after not-elimination for negation-injected schemas.

Table 4: Summary of experimental results for negation-injected schemas.

runtime (ms)				size ratio			
avg	median	max	min	avg	median	max	min
49.082	1	24,733	<1	5.499	5.190	45.241	1.809

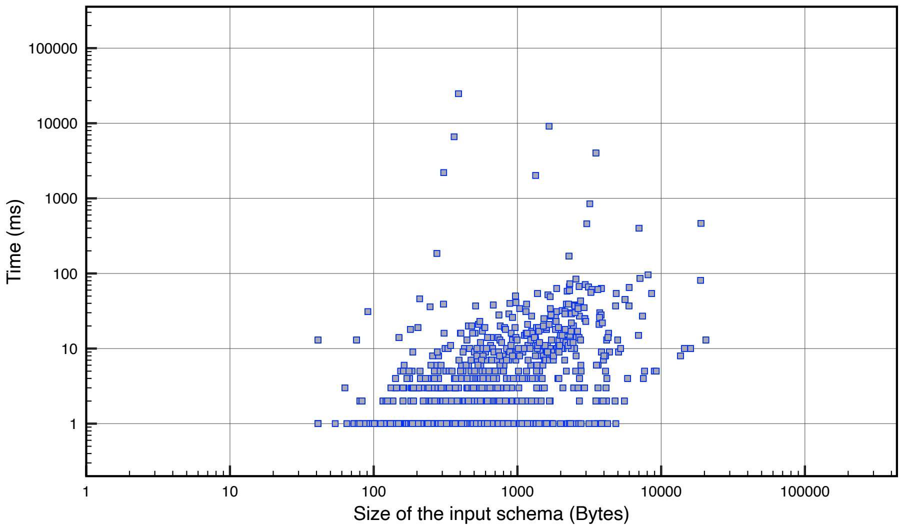

Figure 19: Not-elimination runtime for negation-injected schemas.

Discussion. Apart from a very limited number of schemas, translation to the algebraic representation, combined with not-elimination, is in the sub-second range, which we consider acceptable. On average, not-elimination increases the size by a factor below 5.5 in both schema sets. The maximum size ratio is caused by not-elimination over enumerations with over 200 items. Nevertheless, we observe polynomial growth.

Our experiments show that not-elimination is indeed feasible on real-world JSON Schema documents. While our prototype cannot yet handle all language constructs, the current limitations are merely technical. One unique selling point is that our approach fully supports negation and recursion (even in combination), which is often a conceptual limitation of algorithms and tools designed for JSON Schema processing (e.g., [26]).

8. Related work

In an empirical study over the collection of real-world schemas used in our experiments, we analyzed usage patterns of the negation operator [15, 27]. While we find occurrences of `not` to be rare, there is anecdotal evidence that this operator can be subtle, and often difficult to understand. This contributed some motivation for the work presented here. 2.24

The problem of negation closure of JSON Schema, that is, the precise study of the duality among couples of structural operators, does not seem to have been studied before. Not-elimination for JSON Schema, which is not our central aim but is a contribution of this study, has been partially studied by others, as discussed next.

Habib et al. [2] study schema inclusion for JSON Schema. Their algorithm comprises a preliminary step of canonicalization and simplification, where a limited form of not-elimination is also applied. More in detail, negation is pushed down into boolean connectives, and it is eliminated from string and boolean schemas. However, their approach is not able to deal with negated numbers, objects, and arrays. Furthermore, their technique is not able to process recursive schemas as well as schemas with unions over objects and arrays. This is in line with their intended use case, checking interfaces of operators in machine learning pipelines [3], where recursion does not play a role. However, in general-purpose schemas, recursion does occur [28].

Indeed, our not-elimination algorithm for JSON Schema is the first one to deal with the combination of negation and recursive variables [26], where we use a *not-completion* technique that we believe to be original. The combination of negation and recursion has been deeply studied in the context of logic languages, but these results cannot be easily transferred to JSON Schema, because of the different nature of these languages. For example, languages in the Prolog/Datalog family describe relations, while JSON Schema describes sets. Moreover, variables in relational languages denote elements of the domain, while, in JSON Schema, a variable denotes a set, like in Monadic Second Order logic (MSO). However, in MSO, variables are subject to quantification, while here, variables are only used to express recursion. A logic language where variables denote sets, and are used for recursion rather than for quantification, is the μ-calculus [29], which has been used to interpret JSON Schema [9]. However, classical μ-calculus techniques cannot be immediately transferred to this context, since μ-calculus does not allow the presence of negative recursive variables, that is, recursive variables below an odd number of negations, but negative recursive variables are allowed by the JSON Schema standard, if recursion is guarded. Alternating automata are used in the study of μ-calculus and, in that case, the complement of an alternating automaton is computed by switching final and non-final states, which is a technique that is reminiscent of our not-completion approach [8, 30]. 2.25

The first effort to formalize JSON Schema semantics was by Pezoa et al. [8], who laid the formal foundations of the JSON Schema proposal by studying its expressive power and the complexity of the validation problem. The authors showed that JSON Schema cannot be captured by MSO or tree automata thanks to the uniqueItems constraints. While they focused their attention 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, and, hence, schema satisfiability is EXPTIME-hard.

In [9] Bourhis et al. refined the analysis of Pezoa et al. They mapped JSON Schema onto an equivalent modal logic, called 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, finally, it is in 2EXPTIME for recursive schemas with uniqueItems. The semantics that we provide is not that different from that of Bourhis et al., who rely on JSL modal logic. The main difference is that Bourhis et al. translate JSON Schema into an elegant mathematical formalism that is a bit removed from JSON Schema syntax, and which indeed enjoys negation-completeness. On the contrary, we work with an algebraic rendition of JSON Schema, since we want to study JSON Schema at the source level, in order to build tools that manipulate JSON Schema as it is, with its peculiarities and limits. 2.26

9. Conclusions

We have shown that JSON Schema is "almost" negation-closed and we have provided an exact characterization of the schemas that cannot be expressed without negation. We have proposed two different sets of operators whose addition would make JSON Schema completely negation-closed, one set based on pattern complement $\overline{r}$ and intersection, and a richer set that avoids this problematic extension.

We have studied the impact of the new operators `minContains` and `maxContains` introduced in Draft 2019-09, and we have shown that they make the array operator items negation-closed, at the price of a non-trivial encoding.

We have introduced an algebraic rendition of JSON Schema syntax that is amenable for automated manipulation.

We contributed a not-elimination algorithm that we are using as a building block for our algorithm for witness generation, satisfiability checking, and inclusion verification [11]. As shown in Example 6, not-elimination can also be useful to improve the readability of some JSON Schema documents. Our not-elimination algorithm is the first for JSON Schema that deals with negative recursive variables, and it uses an original, and very simple, technique to do so.

To check the completeness of our approach, we have implemented the translation from JSON Schema to the algebra and back, as well as a version of the not-elimination algorithm, and we tested them on 4,000 schemas collected on the Web, as well as on their negation-injected counterparts. Our experiments confirm that we deal with every aspect of the language, and that the 2.27
not-elimination result has a size that grows polynomially with the input size.

10. Acknowledgments.

We would like to thank Cristiano Landi and Francesco Falleni who implemented the original algorithms. We would also like to thank Lyes Attouche for his terrific implementation, debugging, and testing efforts. We further thank Luca Escher for helping with first tests, Michael Fruth for helping with data preparation, Christoph Köhnen for handling JSON Schema patterns, and Stefan Klessinger for his help with building a fully automated reproduction package.

Funding. Giorgio Ghelli's contribution was supported by MIUR [project PRIN 2017FTXR7S "IT-MaTTerS" (Methods and Tools for Trustworthy Smart Systems)]. Stefanie Scherzinger's contribution was supported by *Deutsche Forschungsgemeinschaft* (DFG, German Research Foundation) [grant #385808805].

References

[1] json-schema org, JSON Schema, 2021. Available at `https://json-schema.org`.

[2] A. Habib, A. Shinnar, M. Hirzel, M. Pradel, Finding data compatibility bugs with JSON subschema checking, in: C. Cadar, X. Zhang (Eds.), ISSTA '21: 30th ACM SIGSOFT International Symposium on Software Testing and Analysis, Virtual Event, Denmark, July 11-17, 2021, ACM, 2021, pp. 620–632. URL: `https://doi.org/10.1145/3460319.3464796`. doi:`10.1145/3460319.3464796`.

[3] G. Baudart, M. Hirzel, K. Kate, P. Ram, A. Shinnar, Lale: Consistent automated machine learning, in: KDD Workshop on Automation in Machine Learning, 2020. `arXiv:2007.01977`, `https://arxiv.org/abs/2007.01977`.

[4] T. Makota, B. Maguire, D. Gagne, R. Chakrabarti, Scalable Data Streaming with Amazon Kinesis: Design and secure highly available, cost-effective data streaming applications with Amazon Kinesis, Packt Publishing, 2021. URL: `https://books.google.de/books?id=GekmEAAAQBAJ`.

[5] D. Poccia, AWS Lambda in Action: Event-driven serverless applications, Manning, 2016. URL: `https://books.google.de/books?id=8jozEAAAQBAJ`.

[6] MongoDB, Inc., MongoDB Manual: $jsonSchema (Version 4.4), 2021. `https://docs.mongodb.com/manual/reference/operator/query/jsonSchema/`.

[7] V. Lakshmanan, J. Tigani, Google BigQuery: The Definitive Guide: Data Warehousing, Analytics, and Machine Learning at Scale, O'Reilly Media, 2019. URL: `https://books.google.de/books?id=9pq4DwAAQBAJ`.

[8] F. Pezoa, J. L. Reutter, F. Suarez, M. Ugarte, D. Vrgoč, Foundations of JSON Schema, in: Proceedings of the 25th International Conference on World Wide Web (WWW), 2016, pp. 263–273.

[9] P. Bourhis, J. L. Reutter, F. Suárez, D. Vrgoc, JSON: Data model, Query languages and Schema specification, in: 36th ACM SIGMOD-SIGACT-SIGAI Symposium on Principles of Database Systems (PODS), 2017, pp. 123–135.

[10] A. Wright, G. Luff, H. Andrews, JSON Schema Validation: A Vocabulary for Structural Validation of JSON - draft-wright-json-schema-validation-01, Technical Report, Internet Engineering Task Force, 2017. URL: `https://tools.ietf.org/html/draft-wright-json-schema-validation-01`.

[11] L. Attouche, M. A. Baazizi, D. Colazzo, F. Falleni, G. Ghelli, C. Landi, C. Sartiani, S. Scherzinger, A Tool for JSON Schema Witness Generation, in: Proceedings of the 24th International Conference on Extending Database Technology, EDBT 2021, 2021, pp. 694–697. doi:10.5441/002/edbt.2021.86.

[12] L. Attouche, M. A. Baazizi, D. Colazzo, Y. Ding, M. Fruth, G. Ghelli, C. Sartiani, S. Scherzinger, A test suite for JSON schema containment, in: R. Lukyanenko, B. M. Samuel, A. Sturm (Eds.), Proceedings of the ER Demos and Posters 2021 co-located with 40th International Conference on Conceptual Modeling (ER 2021), St. John's, NL, Canada, October 18-21, 2021, volume 2958 of *CEUR Workshop Proceedings*, CEUR-WS.org, 2021, pp. 19–24. URL: http://ceur-ws.org/Vol-2958/paper4.pdf.

[13] J. Blackler, Json generator, 2022. Available at https://github.com/jimblackler/jsongenerator.

[14] J. Dolby, J. Tsay, M. Hirzel, Automatically debugging automl pipelines using maro: ML automated remediation oracle (extended version), CoRR abs/2205.01311 (2022).

[15] M. A. Baazizi, D. Colazzo, G. Ghelli, C. Sartiani, S. Scherzinger, An empirical study on the "usage of not" in real-world JSON schema documents, in: A. K. Ghose, J. Horkoff, V. E. S. Souza, J. Parsons, J. Evermann (Eds.), Conceptual Modeling - 40th International Conference, ER 2021, Virtual Event, October 18-21, 2021, Proceedings, volume 13011 of *Lecture Notes in Computer Science*, Springer, 2021, pp. 102–112. URL: https://doi.org/10.1007/978-3-030-89022-3_9. doi:10.1007/978-3-030-89022-3_9.

[16] Security Guidance for the Use of JavaScript Object Notation (JSON) and JSON Schema, Technical Report, National Security Agency, 2018. URL: https://apps.nsa.gov/iaarchive/library/reports/security_guidance_for_json.cfm.

[17] K. Zyp, G. Court, A JSON Media Type for Describing the Structure and Meaning of JSON Documents - draft-zyp-json-schema-03, Technical Report, Internet Engineering Task Force, 2010. URL: https://tools.ietf.org/html/draft-zyp-json-schema-03.

[18] F. Galiegue, K. Zyp, JSON Schema: interactive and non interactive validation - draft-fge-json-schema-validation-00, Technical Report, Internet Engineering Task Force, 2013. URL: https://tools.ietf.org/html/draft-fge-json-schema-validation-00.

[19] A. Wright, G. Luff, H. Andrews, JSON Schema Validation: A Vocabulary for Structural Validation of JSON - draft-handrews-json-schema-02, Technical Report, Internet Engineering Task Force, 2019. URL: https://datatracker.ietf.org/doc/html/draft-handrews-json-schema-02.

[20] A. Wright, H. Andrews, B. Hutton, G. Dennis, JSON Schema: A Media Type for Describing JSON Documents - draft-bhutton-json-schema-00, Technical Report, Internet Engineering Task Force, 2020. URL: https://json-schema.org/draft/2020-12/json-schema-core.html.

[21] M. A. Baazizi, D. Colazzo, G. Ghelli, C. Sartiani, S. Scherzinger, A JSON Schema Corpus, 2021. doi:10.5281/zenodo.5141199.

[22] JSON Schema - valid if object does *not* contain a particular property, Available at:https://stackoverflow.com/questions/30515253/json-schema-valid-if-object-does-not-contain-a-particular-property, 2015.

[23] L. Attouche, M.-A. Baazizi, D. Colazzo, G. Ghelli, C. Sartiani, S. Scherzinger, Witness generation for json schema, 2022. arXiv:2202.12849.

[24] W. Gelade, F. Neven, Succinctness of the complement and intersection of regular expressions, ACM Trans. Comput. Log. 13 (2012) 4:1–4:19. URL: https://doi.org/10.1145/2071368.2071372. doi:10.1145/2071368.2071372.

[25] A. Møller, dk.brics.automaton – Finite-State Automata and Regular Expressions for Java, 2017. http://www.brics.dk/automaton/, version 1.12-1.

[26] M. Fruth, M.-A. Baazizi, D. Colazzo, G. Ghelli, C. Sartiani, S. Scherzinger, Challenges in Checking JSON Schema Containment over Evolving Real-World Schemas, in: Advances in Conceptual Modeling - ER 2020 Workshops CMAI, CMLS, CMOMM4FAIR, CoMoNoS, EmpER, 2020, pp. 220–230.

[27] M. A. Baazizi, D. Colazzo, G. Ghelli, C. Sartiani, S. Scherzinger, An Empirical Study on the "Usage of Not" in Real-World JSON Schema Documents (Long Version), CoRR abs/2107.08677 (2021). URL: https://arxiv.org/abs/2107.08677. arXiv:2107.08677.

[28] B. Maiwald, B. Riedle, S. Scherzinger, What Are Real JSON Schemas Like?, in: Advances in Conceptual Modeling - ER 2019 Workshops FAIR, MREBA, EmpER, MoBiD, OntoCom, and ER Doctoral Symposium Papers, 2019, pp. 95–105.

[29] D. Kozen, Results on the propositional mu-calculus, Theor. Comput. Sci. 27 (1983) 333–354. URL: https://doi.org/10.1016/0304-3975(82)90125-6. doi:10.1016/0304-3975(82)90125-6.

[30] H. Comon, M. Dauchet, R. Gilleron, F. Jacquemard, D. Lugiez, C. Löding, S. Tison, M. Tommasi, Tree Automata Techniques and Applications, 2008. URL: https://hal.inria.fr/hal-03367725.